Volume 26 Number 3

ACTA CYBERNETICA

Szeged, 2024

ACTA CYBERNETICA

Information for authors. Acta Cybernetica publishes only original papers in the field of Computer Science. Manuscripts must be written in good English. Contributions are accepted for review with the understanding that the same work has not been published elsewhere. Papers previously published in conference proceedings, digests, preprints are eligible for consideration provided that the author informs the Editor at the time of submission and that the papers have undergone substantial revision. If authors have used their own previously published material as a basis for a new submission, they are required to cite the previous work(s) and very clearly indicate how the new submission offers substantively novel or different contributions beyond those of the previously published work(s). There are no page charges. An electronic version of the published paper is provided for the authors in PDF format.

Manuscript Formatting Requirements. All submissions must include a title page with the following elements: title of the paper; author name(s) and affiliation; name, address and email of the corresponding author; an abstract clearly stating the nature and significance of the paper. Abstracts must not include mathematical expressions or bibliographic references.

References should appear in a separate bibliography at the end of the paper, with items in alphabetical order referred to by numerals in square brackets. Please prepare your submission as one single PostScript or PDF file including all elements of the manuscript (title page, main text, illustrations, bibliography, etc.).

When your paper is accepted for publication, you will be asked to upload the complete electronic version of your manuscript. For technical reasons we can only accept files in LaTeX format. It is advisable to prepare the manuscript following the guidelines described in the author kit available at https://cyber.bibl.u-szeged.hu/index.php/actcybern/about/submissions even at an early stage.

Submission and Review. Manuscripts must be submitted online using the editorial management system at https://cyber.bibl.u-szeged.hu/index.php/actcybern/submission/wizard. Each submission is peer-reviewed by at least two referees. The length of the review process depends on many factors such as the availability of an Editor and the time it takes to locate qualified reviewers. Usually, a review process takes 6 months to be completed.

Subscription Information. Acta Cybernetica is published by the Institute of Informatics, University of Szeged, Hungary. Each volume consists of four issues, two issues are published in a calendar year. From 2024, issues are published online only, and articles are made available as soon as they are accepted and copyedited. The content is available free of charge.

Contact information. Acta Cybernetica, Institute of Informatics, University of Szeged. P.O. Box 652, H-6701 Szeged, Hungary. Tel: +36 62 546 396, Fax: +36 62 546 397, Email: acta@inf.u-szeged.hu.

Web access. The above information along with the contents of past and current issues are available at the Acta Cybernetica homepage https://cyber.bibl.u-szeged.hu/.

Zoltan Kato
Department of Image Processing and Computer Graphics, University of Szeged, Hungary
kato@inf.u-szeged.hu

Dragan Kukolj
RT-RK Institute of Computer Based Systems, Novi Sad, Serbia
dragan.kukolj@rt-rk.com

László Lovász
Department of Computer Science, Eötvös Loránd University, Budapest, Hungary
lovasz@cs.elte.hu

Kálmán Palágyi
Department of Image Processing and Computer Graphics, University of Szeged, Hungary
palagyi@inf.u-szeged.hu

Dana Petcu
Department of Computer Science, West University of Timisoara, Romania
petcu@info.uvt.ro

Andreas Rauh
School II – Department of Computing Science, Group Distributed Control in Interconnected Systems, Carl von Ossietzky Universität Oldenburg, Germany
andreas.rauh@uni-oldenburg.de

György Vaszil
Department of Computer Science, Faculty of Informatics, University of Debrecen, Hungary
vaszil.gyorgy@inf.unideb.hu

Conference of PhD Students in Computer Science

Guest Editor:

Judit Jász

University of Szeged, Hungary
jasy@inf.u-szeged.hu

Preface

The *13th Conference of PhD Students in Computer Science (CSCS)* was organized by the Institute of Informatics of the University of Szeged (SZTE) and held in Szeged, Hungary, between June 29 – July 1, 2022.

The members of the *Scientific Committee* were the following representatives of the Hungarian doctoral schools in Computer Science: János Csirik (Co-Chair, SZTE), Lajos Rónyai (Co-Chair, SZTAKI, BME), András Benczúr (ELTE), András Benczúr ifj. (SZTAKI), Tibor Csendes (SZTE), László Cser (BCE), Erzsébet Csuhaj-Varjú (ELTE), József Dombi (SZTE), Zoltán Fülöp (SZTE), Aurél Galántai (ÓE), Zoltán Gingl (SZTE), Tibor Gyimóthy (SZTE), Katalin Hangos (MTA), Zoltán Horváth (ELTE), Márk Jelasity (SZTE), Tibor Jordán (ELTE), Zoltán Kása (Sapientia EMTE), Zoltán Kató (SZTE), László Kóczy (SZE), Andrea Kő (Corvinus), Miklós Kuczmann (SZE), János Levendovszki (BME), Gyöngyvér Márton (Sapientia EMTE), Branko Milosavljevic (UNS), Valerie Novitzka (TUKE), László Nyúl (SZTE), Marius Otesteanu (UPT), Attila Pethő (DE), Sándor Radeleczki (ME), András Recski (BME), Sándor Szabó (PTE), Gábor Szederkényi (PPKE), János Sztrik (DE), János Tapolcai (BME), János Tar (ÓE), Gyula Vastag (Corvinus), and János Végh (ME).

The members of the *Organizing Committee* were: Judit Jász, Balázs Bánhelyi, Tamás Gergely, Melinda Katona, and Zoltán Kincses.

There were more than 50 participants and 42 talks in several fields of computer science and its applications (12 sessions). The talks were going in sections in Education, Healthcare, Computer Graphics, Image Processing, Network, Graph, Interpolation, Analysis Methods, Program Analysis, Machine Learning,Verification, and AI&Testing.

The talks of the students were completed by 3 plenary talks of leading scientists: Herbert Edelsbrunner (IST, Austria), András Benczúr (SZTAKI, Hungary), and Gergely Röst (SZTE, Hungary).

The open-access scientific journal Acta Cybernetica offered PhD students to publish the paper version of their presentations after a careful selection and review process. Altogether 25 manuscripts were submitted for review, out of which 17 were accepted for publication in the present special issue of Acta Cybernetica.

The full program of the conference, the collection of the abstracts and further information can be found at https://www.inf.u-szeged.hu/~cscs/.

Judit Jász
Guest Editor

Acta Cybernetica **26** (2024) 297–322.

Single and Combined Algorithms for Open Set Classification on Image Datasets

Modafar Al-Shouha[ab] and Gábor Szűcs[ac]

Abstract

Generally, classification models have closed nature, and they are constrained by the number of classes in the training data. Hence, classifying ”unknown” – OOD (out-of-distribution) – samples is challenging, especially in the so called ”open set” problem. We propose and investigate different solutions – single and combined algorithms – to tackle this task, where we use and expand a K-classifier to be able to identify $K + 1$ classes. They do not require any retraining or modification on the K-classifier architecture. We show their strengths when avoiding type I or type II errors is fundamental. We also present a mathematical representation for the task to estimate the $K + 1$ classification accuracy, and an inequality that defines its boundaries. Additionally, we introduce a formula to calculate the exact $K+1$ classification accuracy.

Keywords: binary classification, multi-class classification, GAN, out-of-distribution, open set classification

1 Introduction

In the field of computer vision, classification is one of the earliest and most common tasks that are challenged by deep neural networks [38]. With the availability of large, well maintained training datasets, and the advancement of convolutional neural networks (CNNs) [21, 22], neural networks could achieve remarkable results in performing this task. However, their classification ability is bounded by the training data features and attributes [3].

Majority of these neural networks apply SoftMax [14] function on the last layer, that outputs the probability of each of the K training classes, and as a result the most likely class is chosen accordingly. One main limitation is the inability of classifying an instance correctly in case it is not presented during training, i.e. OOD (out-of-distribution) or ”unknown” class. The task to overcome this limitation is

[a]Department of Telecommunications and Media Informatics, Budapest University of Technology and Economics, Műegyetem rkp. 3., H-1111, Budapest, Hungary

[b]E-mail: modafar.alshouha@tmit.bme.hu, ORCID: 0000-0003-2051-4036

[c]E-mail: szucs@tmit.bme.hu, ORCID: 0000-0002-5781-1088

DOI: 10.14232/actacyb.298356

called open set recognition, open set classification, or as we call it $K + 1$ classification. As a solution for such challenge, data can be produced or gathered for the $K+1$ class, and a classifier could be trained on $K+1$ classes instead of K. However, this solution remains insufficient and constrained by the ambiguity of defining the "unknown" class while covering its wide features and possibilities.

Another way to address $K + 1$ classification is by adjusting K-classifier to be able to solve the task. Most of the available approaches require retraining for the original K-classifier or altering its architecture [43, 46]. In this paper, we propose and study several solutions, that avoid the need for defining the "unknown" data explicitly or retraining the original K-classifier. The first group of solutions consists of two single algorithms. One of them relies on the K classes confidence when classifying an instance. The other takes advantage of GANs [15] to learn the representation of the training data. A GAN consists of two parts, a generator and a discriminator, and there is competition between them. The generative network generates candidates while the discriminative network evaluates them. We use the discriminator block as a binary classifier to distinguish between "known" and "unknown" instances, before performing K classification. As for the second group we propose more robust solutions, by joining the strengths of various individual algorithms; namely, the discriminator-based algorithm from the first group with a threshold-based algorithm.

Moreover, we suggest a formula that represents mathematically the $K + 1$ algorithm classification accuracy when following our approach. Based on this formula, we define an inequality that sets the boundaries for the $K + 1$ algorithm classification accuracy. We validate those formulas empirically, and show that the test results confirm their applicability.

In the next chapter we present some solutions that try to tackle the $K + 1$ classification task. Then we detail the proposed algorithms in two groups; single and combined ones. In the same chapter, we introduce and prove the constructed formula and inequality. After detailing the examination approach and presenting the used models, datasets and metrics, we show and discuss the experimental results. Lastly, we conclude the paper and review the limitations and future work possibilities.

2 Related work

Supervised learning methods hold an assumption about the excessive similarity between training and testing data. With the presence of "open set" data, the performance of such models might degrade hugely, and it could be worse than random guessing [9]. Many solutions were proposed to address this challenge focusing on enhancing the supervised learning pipeline [33].

In computer vision related tasks, learning the feature representation is the first component of the pipeline, where the aim is to achieve a proper generalization on unseen target domain instances (images), i.e. images under different circumstances. Some methods try to learn the disentangled and casual feature representation of the data [4, 27], in order to assess the model generalization ability over OOD data during the learning process [17, 20, 42]. DANN (domain-adversarial neural network) [12, 13], CIAN (conditional invariant adversarial network) [24], and some others follow domain adversarial learning approach to catch the domain invariant features during training and inference. Another approach for representation learning is to increase and decrease the distances between different and similar domain instances, i.e. domain alignment [23, 36, 39].

Other works focus on the training strategy. Under this category, Finn et al. [11] and later improvements aim to achieve model domain generalization, this is done with the help of meta-learning [19]. Works [32, 40, 45] follow ensemble learning approach, by combining group of models from different domains' knowledge. Papers [26, 44] adopt semi-supervised and unsupervised approaches. Zhang et al. [43] combine the original classifier with a discriminative classifier. Later, they train the model (end-to-end) based on the latent feature space of the train data. They propose a flow-based model (OpenHybrid), without facing a common issue of assigning larger likelihood to the OOD data.

Closer to our work, ODIN (Out-of-DIstribution detector for Neural networks) [25] does not require model retraining, but it involves temperature scaling and input preprocessing inspired by other papers [16, 18]. Additionally, they introduce a detector which catches the OOD data after combining the preprocessing components. On the other hand, paper [46] integrates a GAN network from an AC-GAN [28], where the discriminator is used as a K+1 classifier for HSIs (hyper-spectral images).

In this work, Double Probability Model (DPM) [30] is used in constructing some of the combined algorithms. DPM relies on the likelihoods of a classifier with the assumption that the training data is accessible. The K classifier cumulative distribution function (CDF) and its inverse (inverse-CDF) are calculated. After obtaining the K classifier output for an instance, and using CDF and inverse-CDF, the probability of the i^{th} and $K+1$ classes are calculated, P_{C_i} and $P_{C_{K+1}}$ respectively. Lastly, the condition described by Formula 1 is checked, and if it is true, then $K+1$ class is assigned to the instance, otherwise, the K classifier predicted label. Thus, the K classifier is extended by the OOD class; $K+1$.

$$P_{C_{K+1}} > max_i\{P_{C_i}\} \tag{1}$$

3 Proposed method

In this paper, we propose multiple algorithms with various combinations to tackle $K+1$ classification task. In contrast to prior work, these algorithms are model agnostic (where the model is a K-classifier), and they do not require K-classifier retraining or any modification on its architecture. The task becomes more difficult in scenarios where the training data is not accessible. Our aim is to tackle these challenges while maintaining the immunity against several uncertainties, including but not limited to: OOD data characteristic and amount. We rely on two assumptions; (1) models tend to assign lower likelihoods to OOD (out-of-distribution) than

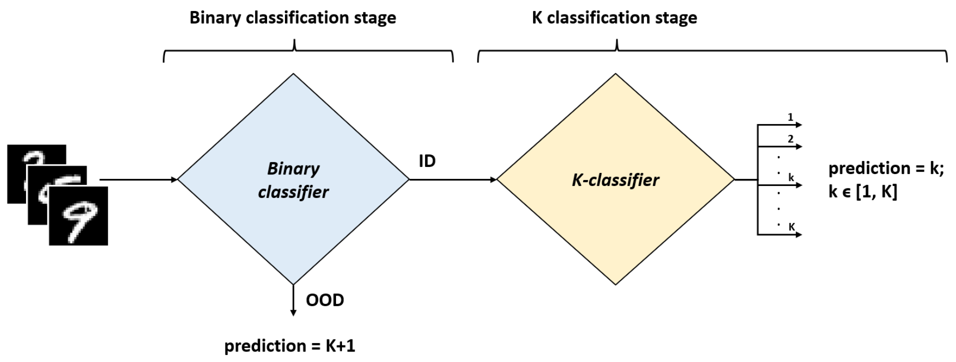


Figure 1: $K + 1$ classification task stages; (1) binary classification, and (2) K classification.

ID (in-distribution) instances [5], and (2) the two distributions (ID and OOD) are different [1].

Moreover, $K + 1$ classification task can be divided into two steps, (1) binary classification, and (2) K classification (Figure 1). Firstly, in the binary classification stage, the instance is categorized as either ID or OOD. If it is determined as OOD instance, $K + 1$ label is assigned to it. Otherwise, it is directed to the K-classifier to get one of the K labels. Therefore, our approach is to construct the binary classification component, and let the original K-classifier to handle the other task. To do so, we design two groups of algorithms; i.e. single [2] and combined. In the combined algorithms, the binary classification task is performed jointly by two methods; while in the single algorithm, it is an individual decision. Furthermore, we formulate the overall accuracy of the $K + 1$ classification task, by connecting the test accuracy scores of the individual components (binary and K classification), and the ratio of the OOD data in the test set. Consequently, we define an inequality for the $K + 1$ classification accuracy based on those factors.

3.1 Single algorithms

We propose two single algorithms, one of them is the Threshold algorithm "Thr" (Algorithm 1) that relies on the prediction of the K-classifier. The prediction output of the K-classifier is a vector that contains values that can be considered as probabilities of an instance belonging to the K classes, i.e. higher probability means more confidence. The Threshold algorithm checks the highest confidence level for a prediction, and if it is lower than the threshold value, the instance is assigned an $K + 1$ label, otherwise, the label is one of the K classes (K-classifier prediction). The threshold value β ($\beta \in [0, 1]$) represents the aggressiveness of the algorithm, higher threshold means that the algorithm is more strict in considering the ID decision from the classifier. Despite that a similar idea was presented earlier, e.g. by [43] and [30], those approaches require either an end-to-end retraining or access to the training data, to obtain a proper threshold value. In contrast, our proposed "Thr" algorithm does not assume that the training ID data is accessible, and does not require any data preprocessing or additional model training.

Algorithm 1 Threshold algorithm

```
1: Obtain $Preds$, $\beta$ $(Threshold)$
2: $y_i = ArgMax(Preds)$
3: $P_i = Max(Preds)$                          ▷ Prediction confidence
4: if $P_i < \beta$ then
5:     $Y_i = not\,in\,class$                  ▷ K+1 label
6: else
7:     $Y_i = y_i$                             ▷ $K$-classifier label
8: end if
```

The other proposed algorithm is the Discriminator algorithm "Disc"; which, unlike "Thr", requires access to ID training data. We construct a $K + 1$-classifier by using a discriminator block from a GAN as a binary classifier, then cascade it by the pre-trained K-classifier. We built a simple GAN using two convolution and two deconvolution layers for its discriminator and generator, respectively. In contrast to some of the papers which are listed in the Related Work [11, 19, 24, 28], neither the GAN nor the Discriminator component has a special architecture. Additionally, unlike [25, 40, 45] it does not require knowledge or assumption about the OOD data distribution, and it does not need any preprocessing. Furthermore, in contrast to the mentioned papers [12, 13, 17, 20, 23, 32, 36, 39, 43], all the proposed methods do not need any access, modification or retraining for the K-classifier. Later, we trained the GAN on the available ID data only. The discriminator's job is to distinguish between "real" and "fake" instances based on the knowledge it gains about the data during the GAN training process. We use this discriminator to catch the OOD instances before deciding if the K-classifier prediction is considered or not. If the discriminator defines an instance as "fake", $K + 1$ label is assigned to it, otherwise, its label is the one that is offered by the K-classifier. While this algorithm requires an extra training step with access to the ID training data, there is no need to retrain the original K-classifier. Additionally, it is considered as a generic method, and there are no specific characteristics defined for the GAN network or any of its components (Algorithm 2).

3.2 Combined algorithm

Although the simplicity of the single algorithms is a big advantage, stability issues in the performance might appear. For instance, the decision about β value is crucial and influences "Thr" algorithm performance. Also, relying on the K-classifier Softmax confidence might be misleading in our task [31]. In order to utilize their strengths, we combine them together in different variations. The general framework stays the same; at first, if the instance is OOD, $K + 1$ label is assigned to

Algorithm 2 Discriminator algorithm

```
1: Obtain $Preds$, $Disc\_Pred$
2: $y_i = ArgMax(Preds)$
3: if $Disc\_Pred = fake$ then
4:     $Y_i = not\, in\, class$                    ▷ K+1 label
5: else
6:     $Y_i = y_i$                                  ▷ K-classifier label
7: end if
```

it, otherwise, K-classifier decision is considered. The combination happens in the binary classification level; at the stage where an instance is allocated either to ID or OOD. Unlike in the single algorithms, the decision is jointly made by two individual algorithms.

The combination has two main aspects; (1) selecting the individual methods to combine, and (2) defining the logical relation between their decisions. We join our proposed "Disc" method with a threshold based method. For the latter, we use either our proposed "Thr" algorithm or "DPM" (Double Probability Model) [30]. "DPM" [30] is a threshold based approach, which unlike "Thr" algorithm demands access on training data, and its threshold value is dynamically set. Regarding the other aspect, logical "OR" and "AND" are used to combine the decisions of the individual methods (Table 1). As a result, the combined algorithm has four different variations: "ThrAndDisc", "ThrOrDisc", "DpmAndDisc" and "DpmOrDisc".

Table 1: Truth table for combined algorithms. The first main column shows two individual methods ("A" and "B") decisions, while the other represents their decisions combined by logical "OR" and "AND".

individual decision		combined decision	
method A	method B	OR	AND
ID	ID	ID	ID
ID	OOD	OOD	ID
OOD	ID	OOD	ID
OOD	OOD	OOD	OOD

3.3 Formula for estimating the open set classification accuracy

Our approach to solve the $K+1$ classification task goes through two stages; (1) binary classification to opt out the OOD instances, and (2) K classification for the instances which are ruled to be ID (Figure 1). Accordingly, the accuracy of the algorithm can be estimated by combining high level information about the binary and K classification tasks. This can be accomplished by incorporating (1) the ratio of OOD data in the test set, (2) the original model K classification accuracy, and (3) the algorithm binary classification accuracy. This estimation does not require more details about the task, such as: access to the confusion matrix, or the true positive and negative ratios. In Formula 2, the first and second terms represent the binary and the K classification tasks, respectively. Another form for Formula 2 is Formula 3.

$$\hat{A}_{K+1} = A_{bin} \cdot OOD + A_{bin} \cdot \hat{A}_K \cdot ID \tag{2}$$

$$\hat{A}_{K+1} = A_{bin} + A_{bin} \cdot ID \cdot (\hat{A}_K - 1) \tag{3}$$

Where:

$\hat{A}_{K+1}$: estimated $K+1$ classification accuracy

$\hat{A}_K$: K classification accuracy of the original model

A_{bin}: binary classification accuracy

OOD: OOD data ratio

ID: ID data ratio

Proof.

1. Let us denote the number of instances by N. The number of ID instances and OOD instances are $N \cdot ID$ and $N \cdot OOD$ respectively. The decision between ID and OOD leads to a binary classification task. The approximate number of correct decisions among the OOD (TP_{OOD} or TN) and ID instances (TP_{ID} or TP) is expressed by Equations 4 and 5, receptively.

$$TP_{OOD} \approx N \cdot OOD \cdot A_{bin} \tag{4}$$

$$TP_{ID} \approx N \cdot ID \cdot A_{bin} \tag{5}$$

2. The correctly classified ID instances are passed to the K-classifier. The original accuracy of the K-classifier ($\hat{A}_K$) is not more than the original model test accuracy (test set consists of ID data only). The number of the true positives in the K-classification task is approximated by multiplying $\hat{A}_K$ by the all number of the instances in the K-classification task (TP_{ID}).

$$TP_K \approx N \cdot ID \cdot A_{bin} \cdot \hat{A}_K \tag{6}$$

3. The estimated $K+1$ classification accuracy ($\hat{A}_{K+1}$) is the ratio of the correct decisions.

$$\hat{A}_{K+1} = \frac{N \cdot ID \cdot A_{bin} \cdot \hat{A}_K + N \cdot OOD \cdot A_{bin}}{N} \tag{7}$$

After using $OOD = 1 - ID$ and simplifying Equation 7, we get Formula 3.

□

Additionally, we prove that the estimated $K+1$ classification accuracy is necessarily larger than w and cannot exceed the algorithm binary classification accuracy (A_{bin}); where $w = A_{bin} \cdot \hat{A}_K$ (Formula 9) and the conditions (Formulas 10, 11 and 12) hold. To do so, we rewrite Formula 2 as Equation 8 by using $OOD = 1-ID$.

$$OOD = \frac{\hat{A}_{K+1} - w}{A_{bin} - w} \tag{8}$$

Where:

$$w = A_{bin} \cdot \hat{A}_K \tag{9}$$
$$0 < A_{bin} \leq 1 \tag{10}$$
$$0 \leq \hat{A}_K < 1 \tag{11}$$
$$0 \leq OOD \leq 1 \tag{12}$$

We use Formula 8 to derive the $K+1$ classification accuracy inequality defined in Formula 13.

$$w \leq \hat{A}_{K+1} \leq A_{bin} \tag{13}$$

Proof.

1. Using proof by contradiction, we will prove that the denominator in Formula 8 is always positive ($A_{bin} - w > 0$). Let us suppose the opposite of this statement.

$$A_{bin} - w \leq 0 \tag{14}$$

   a) if $A_{bin} - w = 0$, then OOD is *undefined*, which contradicts Formula 12. Thus,

$$A_{bin} - w \neq 0 \tag{15}$$

b) Let us suppose the following

$$A_{bin} - w < 0 \tag{16}$$
$$A_{bin} < w \tag{17}$$
$$A_{bin} < A_{bin} \cdot \hat{A}_K \tag{18}$$
$$1 < \hat{A}_K \tag{19}$$

but this contradicts Formula 11.
Therefore,

$$A_{bin} - w > 0 \tag{20}$$

2. Using the left side of the inequality (Formula 12) and Formula 20, the numerator cannot be negative ($\hat{A}_{K+1} - w \geq 0$).

$$OOD \geq 0 \tag{21}$$
$$\frac{\hat{A}_{K+1} - w}{A_{bin} - w} \geq 0 \tag{22}$$

but $A_{bin} - w > 0$, then,

$$\hat{A}_{K+1} - w \geq 0 \tag{23}$$
$$\hat{A}_{K+1} \geq w \tag{24}$$

3. Using the right side of the inequality (Formula 12) and Equation 8

$$OOD \leq 1 \tag{25}$$
$$\frac{\hat{A}_{K+1} - w}{A_{bin} - w} \leq 1 \tag{26}$$
$$\hat{A}_{K+1} - w \leq A_{bin} - w \tag{27}$$
$$\hat{A}_{K+1} \leq A_{bin} \tag{28}$$

4. Finally, combining Formulas 24 and 28

$$w \leq \hat{A}_{K+1} \leq A_{bin} \tag{29}$$

□

3.4 Formula for calculating the exact open set classification accuracy

The exact accuracy of the algorithm can be calculated by using Formula 30. It is important to highlight that A_K is the actual K-classification accuracy. It is calculated for the ID instances in the test set and it depends on the actual scenario. In general, it can be assumed that A_K is close to $\hat{A}_K$, in scenarios where the ID data has similar distribution as the original K-classifier test set. Additionally, if the data has only OOD instances ($ID = 0$), Formula 30 cannot be applied, since R_{ID} is undefined ($R_{ID} = \frac{0}{0}$).

$$A_{K+1} = A_{bin} + R_{ID} \cdot ID \cdot (A_K - 1) \tag{30}$$

Where:

A_{K+1}: exact $K+1$ classification accuracy

A_K: K classification accuracy of the actual model

A_{bin}: binary classification accuracy

R_{ID}: binary True Positive Ratio ($TPR = \frac{TP_{ID}}{TP_{ID}+FN_{ID}}$)

OOD: OOD data ratio

ID: ID data ratio

Proof.

1. Let us denote the number of instances by N. The exact number of correct decisions among the OOD (TP_{OOD}) and ID instances (TP_{ID}) is expressed by Equations 31 and 32, receptively.

$$TP_{OOD} = N \cdot OOD \cdot R_{OOD} \tag{31}$$

$$TP_{ID} = N \cdot ID \cdot R_{ID} \tag{32}$$

2. The accuracy of binary classification task A_{bin} comes from the sum of correct decisions divided by the all decisions (Equation 33).

$$A_{bin} = \frac{N \cdot ID \cdot R_{ID} + N \cdot (1 - ID) \cdot R_{OOD}}{N} \tag{33}$$

Using Equation 33, R_{OOD} can be expressed by Equation 34.

$$R_{OOD} = \frac{A_{bin} - ID \cdot R_{ID}}{(1 - ID)} \tag{34}$$

3. The diagonal entries of the confusion matrix contains the true positive instances in the K-classification task, and the sum of them gives the number of correct decisions, which is equal to accuracy A_K multiplied with all instances in the K classification task (Equation 35).

$$TP_K = N \cdot ID \cdot R_{ID} \cdot A_K \tag{35}$$

4. The exact $K+1$ classification accuracy (A_{K+1}) is the ratio of the correct decisions.

$$A_{K+1} = \frac{N \cdot ID \cdot R_{ID} \cdot A_K \; + \; N \cdot (1 - ID) \cdot R_{OOD}}{N} \tag{36}$$

5. After substituting the R_{OOD} in Equation 36, and simplification, Equation 36 can be written as Formula 30

□

To determine A_{K+1}, R_{ID} and A_K have to be known. Whereas the estimation ($\hat{A}_{K+1}$) is calculated using $\hat{A}_K$ without the need for R_{ID}. The difference between the exact and the estimated algorithm accuracy is the correction factor (ΔA_{K+1}). It is expressed by Formula 38, which is derived by plugging Formulas 3 and 30 in Equation 37. If $A_K = \hat{A_K}$ and $R_{ID} = A_{bin}$, then $\Delta A_{K+1} = 0$.

$$\Delta A_{K+1} = \Big| A_{K+1} \; - \; \hat{A}_{K+1} \Big| \tag{37}$$

$$\Delta A_{K+1} = \Big| ID \cdot \Big[R_{ID} \cdot (A_K - 1) \; - \; A_{bin} \cdot (\hat{A}_K - 1) \Big] \Big| \tag{38}$$

4 Experiments

In the experiments, we examined and evaluated the four variations of the combined algorithm; "ThrAndDisc", "ThrOrDisc", "DpmAndDisc" and "DpmOrDisc". Also, we studied the three single algorithms – "Thr", "Disc" and "DPM" – individually to show their drawbacks and set them as a baseline. One main variable is the threshold value (β) for the "Thr" algorithm. Hence, we picked and tested different threshold values (with 0.01 step) out of the infinitely many possibilities; $\beta \in [0, 1]$. The other variable is the OOD percentage in the test set, since it has a direct relation with the overall algorithm performance (Formulas 2 and 30). Those two variables; i.e. β and OOD ratio, affect the algorithms robustness when dealing with different scenarios. For proper generalization, we used multiple datasets, models (classifiers) and metrics.

4.1 Datasets

We defined ID and OOD datasets; ID data includes instances of K classes, while OOD data is the test set from other datasets. For ID data we used two sets separately, creating two main experiment groups. The first is Extended-mnist (E-MNIST - by merge) dataset [7], where the numbers are excluded, hence, it contains 37 letter categories ($K = 37$). The other is Arabic handwritten characters set (Arab-L) [10], that contains 13440 and 3360 grey-scaled 32x32 pixel images for train and test splits respectively. Those images are distributed evenly over 28 classes, hence, $K = 28$. We augmented the data in order to expand its size. After resizing it into 28x28 pixel, random scaling and rotation was applied resulting in 40 000 train and 10 000 test images. The train sets were used to train two different K-classifiers, and the experiments were conducted with the test sets (Table 2).

Table 2: Datasets size details. E-MNIST and Arab-L datasets are the ID data. K is the number of classes in each dataset.

Dataset	train	test	K
E-MNIST	410 000	10 000	37
Arab-L	40 000	10 000	28

In order to provide better generalization and represent diverse levels of similarity with respect to the ID data, we used six different types for OOD data (Figure 2). We chose the test set of four classical datasets: MNIST dataset [8], Fashion-mnist (F-MNIST) [41], Kuzushiji-mnist (Ku-MNIST) [6] and B-MNIST (we performed binarization for MNIST test set). Additionally, we generated two datasets with Gaussian distribution. R-gauss (28x28 pixel random Gaussian data with 128 mean and 12.8 standard deviation), and I-gauss (ID-based Gaussian data). For the I-gauss we calculated the training data (ID) mean and standard deviation and used them as the distribution parameters. Since we have two ID sets, separate I-gauss data was generated for each of them.

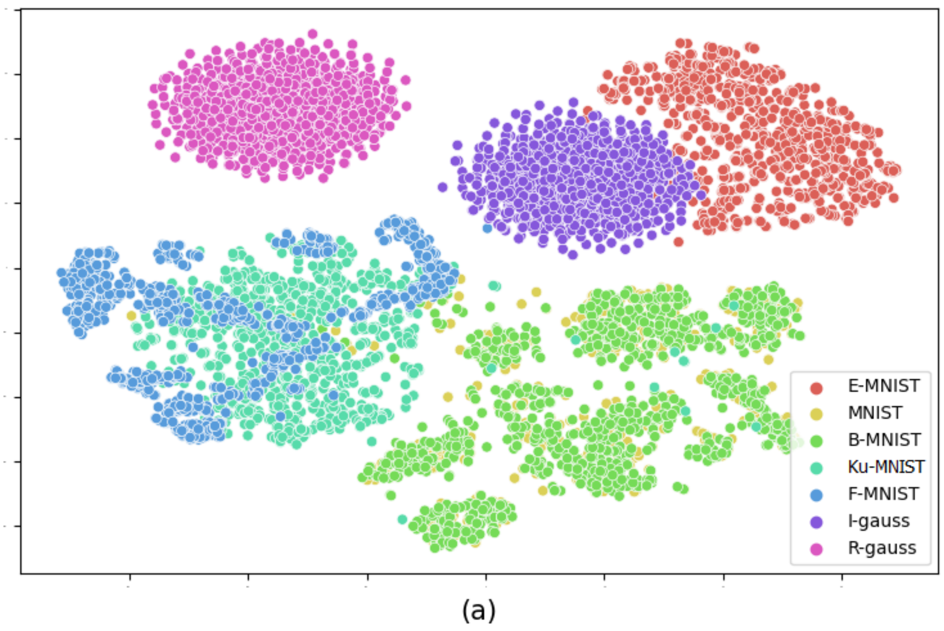
We executed the experiments on mixed test sets, each containing in total 10000 ID and OOD instances. For the test set of each experiment, we selected one ID set and one OOD set. Hence, we chose either E-MNIST or Arab-L data as ID, and combined it with one of the six OOD sets. The percentage of OOD instances was defined by setting the OOD ratio, which varies from 0% to 100% with 5% step.

4.2 Classifiers

For K classification we trained two different classifiers on the two different ID sets. For the first classifier, we used VGG16 [34] architecture with dropout layers. The classifier was trained on E-MNIST data (without numbers data). Since the training data has 37 categories ($K = 37$), it is called "Classifier-37". Similarly, we trained AlexNet [21] on Arab-L dataset. Following the same fashion, this classifier is called "Classifier-28"; $K = 28$. Additionally, we trained a simple GAN on the two ID training sets separately, then we extracted the discriminators to be used as a binary classifier in the proposed "Disc" algorithm accordingly. As mentioned earlier, any arbitrary classifier can fit this purpose.

4.3 Metrics

The first component of our approach is the binary classification, where the suspected OOD instances are filtered out. In case the instance is from ID, it is either correctly classified (TP) or results in type II error (FN). While if it is from OOD,



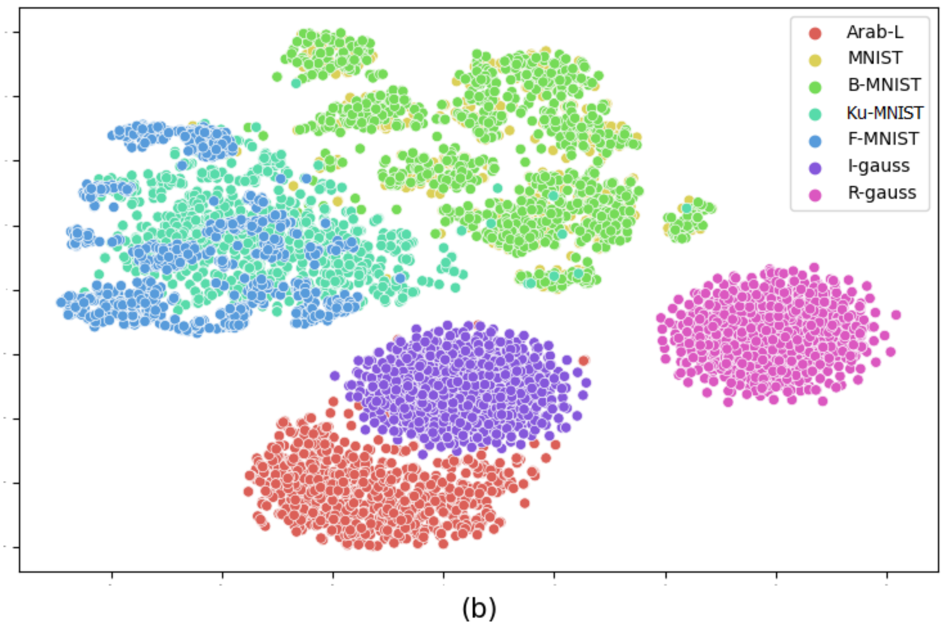


Figure 2: OOD with (a) E-MNIST and (b) Arab-L datasets after dimension reduction using t-SNE [37]. The number of components for t-SNE is 2, and it uses PCA initialization with 50 components.

it is either TN or FP (type I error). Therefore, to evaluate the first step we used sensitivity (TP rate) and specificity (TN rate). Moreover, we used the accuracy (Equation 39) and the balanced accuracy[1] (Equation 40) to measure the performance of the overall classification task [35], because unbalanced data might mislead the conclusion [29]. Lastly, we used mean squared error for evaluating the goodness of Formula 2 using Formula 37 (ΔA_{K+1}), and aggregating these results of all the experiments.

$$Accuracy = \frac{1}{n_{samples}} \sum_{i=1}^{n_{samples}} 1\left(\hat{y}_i = y_i\right) \tag{39}$$

$$Balanced\ Accuracy = \frac{1}{\sum \hat{w}_i} \sum_{i=1}^{n_{samples}} \hat{w}_i\left(\hat{y}_i = y_i\right) \tag{40}$$

5 Results and discussion

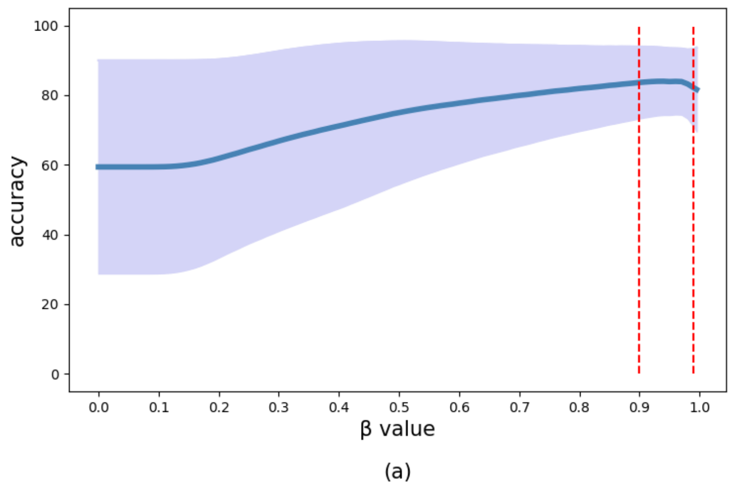
When performing the experiments, the two main variables to deal with are the OOD ratio in the test set, and the β value. Keeping in mind that β value is only relevant for the methods that include "Thr" algorithm, and it is not arbitrarily chosen. Instead, we defined a range $[0.90, 0.99]$, where the resulted $K+1$ accuracy is the highest with the smallest variance as shown in Figure 3. The aim is to avoid any misleading intuitions that might be caused by relying on K-classifier Softmax confidence [31]. Additionally, we created three groups. In the first group we fixed β value to 0.99, and checked the results over the OOD ratio between 5% and 95%. The other two groups simulate two extreme scenarios; we fixed OOD ratio to 5% and 95% over a range of β values (Table 3).

Table 3: Groups parameters

Parameter	Group 1	Group 2	Group 3
β	0.99	$\in [0.90, 0.99]$	$\in [0.90, 0.99]$
OOD ratio	$\in [0.05, 0.95]$	0.05	0.95

First, we evaluated the proposed algorithms' ability to execute the binary classification task. In Table 4 it can be seen that "And" methods, i.e. "ThrAndDisc" and "DpmAndDisc", achieve the highest sensitivity score, alongside with the single "Disc" algorithm. In other words, they can lead the ID instances to the next stage successfully and attain the highest TP rate. They are well suited in scenarios where avoiding type I error (FP) is vital. In contrast, Table 5 shows that "OR" methods, i.e. "ThrOrDisc" and "DpmOrDisc", are better fit in scenarios where committing type II error (FN) is more harmful. Sensitivity and specificity results

[1] $\hat{w}_i$ is the sample weight adjusted according to its true class inverse prevalence.



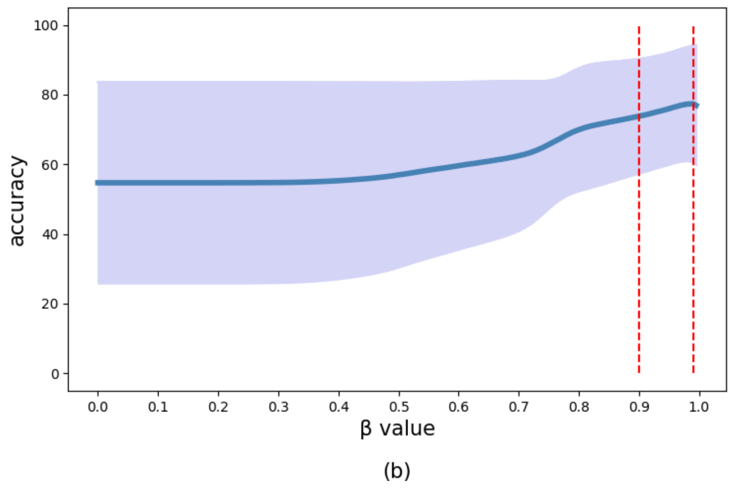


Figure 3: The average $K+1$ accuracy (in dark blue) and standard deviation range (in light blue) from all experiments that include "Thr" at β value in range [0, 1]. (a) and (b) plots are for the experiments that use Classifier-37 and Classifier-28, respectively. Red lines define the range of β value where the accuracy is the highest and the standard deviation is the lowest.

(Tables 4 and 5) also highlight some interesting points: (1) despite its simplicity, single "Disc" algorithm executes the binary classification task adequately; (2) the ability to catch OOD instances is in an acceptable range regardless of the algorithm; (3) the methods are more stable in terms of sensitivity (TP rate) compared to specificity (TN rate).

Next, we used the three groups to assess the algorithms capability in accomplishing the $K + 1$ classification task, based on their average accuracy results. The experiment results are consistent regardless of the used classifier and the ID data type. Furthermore, second and third group results in Tables 6 and 7 confirm our previous findings w.r.t. sensitivity and specificity. For instance, when OOD ratio is low (Group 2) "And" methods perform the best, whereas "OR" methods excel with high OOD ratio (Group 3). Also, the first group demonstrates the overall superiority of "DpmAndDisc" method in terms of achieved accuracy and stability.

Table 4: Average sensitivity of binary task among all OOD datasets (using Classifier-37 & Classifier-28). In Table 3, Group 1 shows the corresponding experiment parameters. The results are in the form of mean and standard deviation. The highest three sensitivity scores in every group are in bold.

Algorithm	Classifier-37		Classifier-28	
	mean	sd	mean	sd
Disc	**85.81**	0.27	**76.73**	2.42
DPM	65.14	0.45	37.46	2.28
DpmAndDisc	**95.59**	0.24	**84.97**	2.28
DpmOrDisc	55.36	0.48	29.22	2.39
Thr	66.44	0.41	69.98	2.35
ThrAndDisc	**95.54**	0.13	**91.90**	1.23
ThrOrDisc	56.72	0.57	54.81	3.33

Table 5: Average specificity of binary task among all OOD datasets (using Classifier-37 & Classifier-28). In Table 3, Group 1 shows the corresponding experiment parameters. The results are in the form of mean and standard deviation. The highest three specificity scores in every group are in bold.

Algorithm	Classifier-37		Classifier-28	
	mean	sd	mean	sd
Disc	**93.84**	13.70	**100**	0.00
DPM	80.61	12.13	93.16	13.25
DpmAndDisc	75.40	16.00	93.16	13.25
DpmOrDisc	**99.04**	2.12	**100**	0.00
Thr	91.63	9.89	82.53	27.48
ThrAndDisc	85.47	13.53	82.53	27.48
ThrOrDisc	**100**	0.00	**100**	0.00

Additionally, Tables 6 and 7 highlight the instability of single "Thr" algorithm. It is very sensitive to β value, which is reflected in Group 2 and 3 standard deviation results. Thus, albeit having a high mean value in Group 2, it cannot be concluded that it outperforms the others. For instance, in this scenario single ”Disc” algorithm might be a better choice.

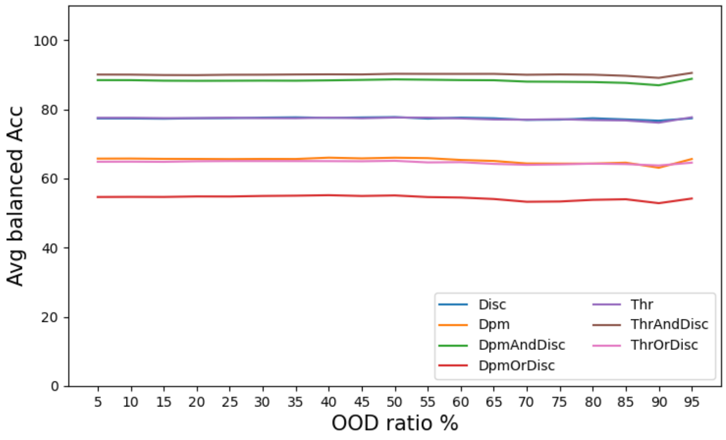
Table 6: Average A_{K+1} (using Classifier-37). The experiments' parameters are detailed in Table 3. The results are in the form of mean and standard deviation. The highest three accuracy scores in every group are in bold.

Algorithm	Group 1		Group 2		Group 3	
	mean	sd	mean	sd	mean	sd
Disc	**87.39**	8.90	81.57	0.78	**93.15**	14.33
DPM	72.53	8.58	65.65	0.67	78.71	13.73
DpmAndDisc	**83.00**	10.80	**90.11**	0.86	75.09	17.34
DpmOrDisc	76.93	13.46	57.11	0.06	**96.77**	2.48
Thr	78.70	9.68	**86.46**	9.54	48.44	33.75
ThrAndDisc	**87.93**	8.29	**91.26**	1.60	46.06	32.42
ThrOrDisc	78.17	13.29	76.76	8.92	**95.53**	9.19

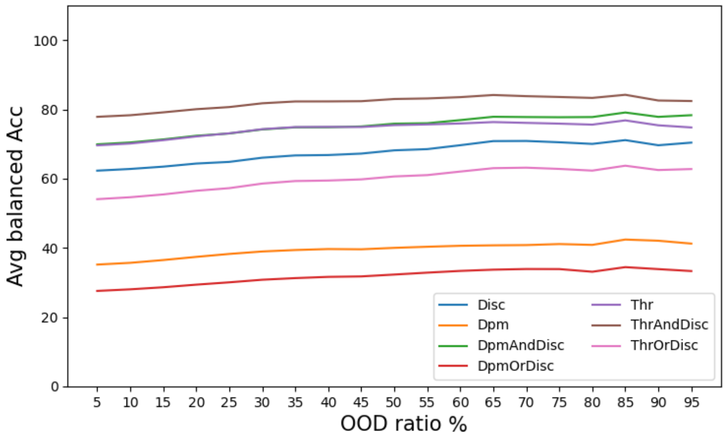
Table 7: Average A_{K+1} (using Classifier-28). The experiments' parameters are detailed in Table 3. The results are in the form of mean standard deviation. The highest three accuracy scores in every group are in bold.

Algorithm	Group 1		Group 2		Group 3	
	mean	sd	mean	sd	mean	sd
Disc	**82.08**	12.17	62.87	0.01	**98.52**	0.00
DPM	64.39	19.94	36.12	0.64	90.38	13.94
DpmAndDisc	**82.81**	0.82	**70.28**	0.64	92.25	13.94
DpmOrDisc	63.66	23.06	28.72	0.02	**96.64**	0.00
Thr	73.83	17.34	**74.29**	7.96	28.68	31.52
ThrAndDisc	**80.71**	16.26	**77.52**	1.73	28.84	31.73
ThrOrDisc	75.20	16.49	59.64	6.59	**98.36**	0.36

We evaluated the algorithms further, by investigating a more general scenario. Figure 4 shows their average $K + 1$ balanced accuracy, given that $\beta \in [0.90, 0.99]$. Using balanced accuracy eliminates the effect of OOD ratio and provides broader insight about the algorithms' performance. This figure gives another evidence of the ”And” methods general effectiveness. In this case, ”ThrAndDisc” algorithm outperforms the others, followed closely by ”DpmAndDisc”. Since the two algorithms include ”Disc” component, they both require access to the training data. Therefore, the main advantage of ”Thr” algorithm, i.e. train data independence, vanishes.



(a)



(b)

Figure 4: The average $K+1$ balanced accuracy for all OOD sets, at OOD ratio value in range [0.05, 0.95] for the experiments using (a) Classifier-37 and (b) Classifier-28.

Moreover, Figure 5 shows the algorithms' average $K + 1$ accuracy results with respect to the OOD set. All the algorithms, regardless of the used K-classifier, performed the best when OOD data was random (R-gauss). With other OOD sets, "Disc" algorithm performance was independent from the OOD data. This behaviour was reflected also on the "OR" methods.

Lastly, Table 8 demonstrates an empirical evidence of our proposed Formula 2. We executed an extensive amount of experiments ($\sim$76 000 experiments[2]) and

[2]total number of experiments = 2 ID sets (classifiers) * 6 OOD sets * 20 OOD ratios * [4 algorithms without "Thr" + 3 algorithms with "Thr" * 100 β values].

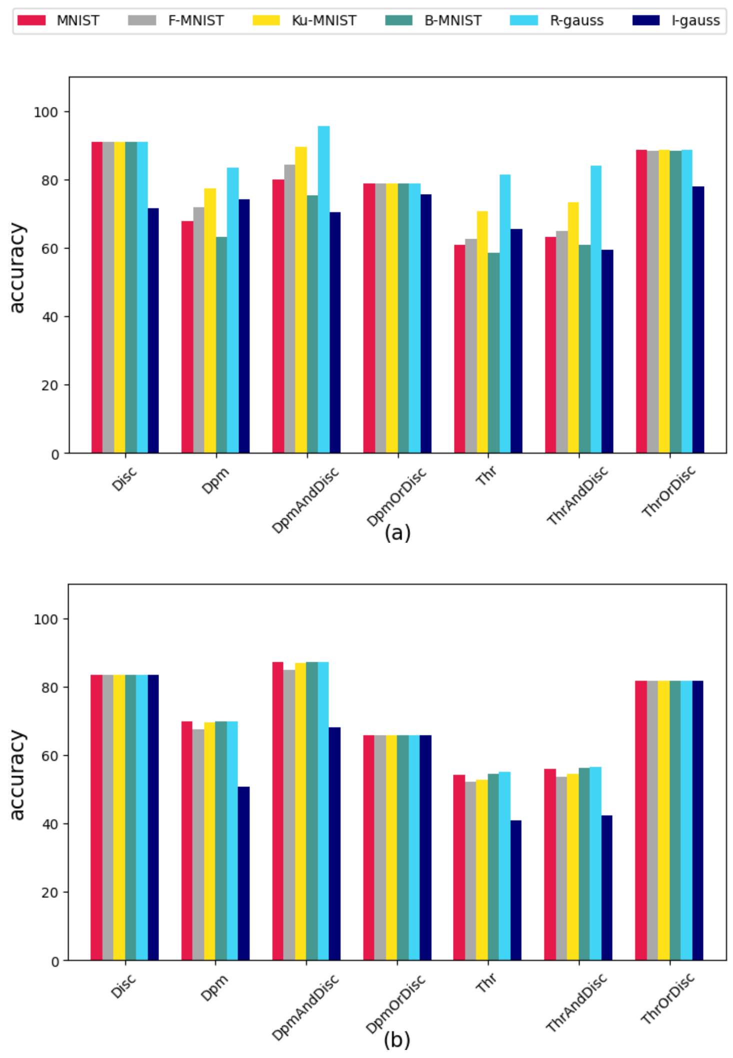
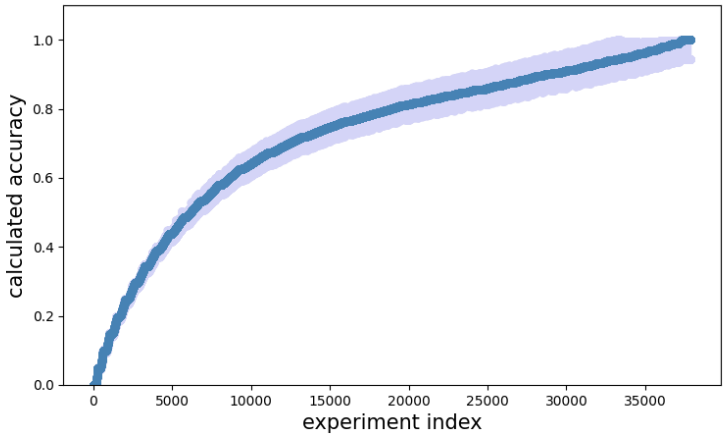


Figure 5: The average $K+1$ accuracy for all algorithms with respect to the selected OOD set for the experiments using (a) Classifier-37 and (b) Classifier-28.

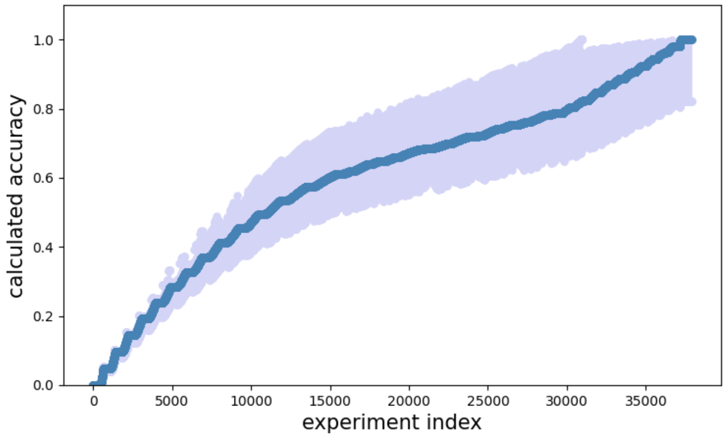
calculated the mean squared error between the actual and estimated $K+1$ accuracy. Additionally, we validated that the proposed inequality (Formula 13) holds in all cases (Figure 6). The estimated $K+1$ accuracy ($\hat{A}_{K+1}$) of the algorithm is higher than or equal to w ($\hat{A}_K \cdot A_{bin}$), but it can not exceed A_{bin}. The inequality highlights that the binary classifier is the vital segment in this architecture (Figure 1). Failing to distinguish ID from OOD data degrades the overall algorithm performance.

Table 8: MSE scores between the actual (by experiment) and the estimated (by formula) $K+1$ accuracy for the algorithms (using Classifier-37 & Classifier-28).

	Classifier-37	Classifier-28
MSE	$1.02 \cdot 10^{-4}$	$6.31 \cdot 10^{-4}$



(a)



(b)

Figure 6: The inequality empirical results sorted by the calculated accuracy value. The dark blue data is the calculated $K+1$ accuracy ($\hat{A}_{K+1}$), that lies between the lower (w) and upper (A_{bin}) bounds in light blue for all the experiments using (a) Classifier-37 and (b) Classifier-28. A plot was used instead of a table, because of the large number of experiments (more than 76 000).

6 Conclusion

In this paper we proposed various approaches to solve the open set classification task for image datasets. By proposing a flexible methodology, we overcome the need for retraining a pretrained K-classifier or altering its architecture. As a result, our proposed methods can adapt to any available classifier.

We interpreted $K+1$ classification task as two consecutive steps: (1) Binary classification; i.e. ID or OOD, followed by (2) K classification. Our proposal handles the first task and lets the original K-classifier to solve the other. We grouped our proposed algorithms based on the decision technique. The first is the single algorithms, where we proposed threshold-based "Thr" and discriminator-based "Disc" methods. The second is the combined algorithms, where we built the final judgment based on a collective decision between "Disc" and a threshold-based method, i.e. "Thr" or "DPM". Their outcomes are joined either by logical "OR" or "AND". As a result, we proposed four variations; "ThrAndDisc", "ThrOrDisc", "DpmAndDisc" and "DpmOrDisc". After evaluating all methods, the results show that "DpmAndDisc" and "ThrAndDisc" algorithms are an excellent general solutions. Additionally, "And" algorithms are good fit when the priority is to avoid committing type I error (FP), while "OR" algorithms are more suitable in dealing with higher percentage of OOD instances; avoiding type II error (FN).

Furthermore, we presented mathematical formulas to calculate the exact and estimated $K+1$ accuracy of the algorithm, and used the latter to define an inequality for $\hat{A}_{K+1}$. We proved mathematically and empirically that $\hat{A}_{K+1}$ is equal to or larger than w $(\hat{A}_K A_{bin})$, but it is lower than A_{bin}.

7 Future work

We evaluated our proposal to tackle open set classification task for image datasets from multiple aspects. However, the proposal ability to solve the task for other data types, e.g. text (document) classification, can be shown. Another direction is to investigate the influence of the ID and OOD data characteristic on the proposed solutions performance. For instance, the task is expected to be more challenging with higher similarity between ID and OOD data distribution. Additionally, more experiments can be conducted to analyze how the hyper-parameters (β) tunning is affected by multiple factors, such as the ID and OOD data characteristic and the K-classifier performance (A_K).

References

[1] Adila, D. and Kang, D. Understanding out-of-distribution: A perspective of data dynamics. In *I (Still) Can't Believe It's Not Better! Workshop at NeurIPS 2021*, pages 1–8. PMLR, 2022. DOI: 10.48550/arXiv.2111.14730.

[2] Al-Shouha, M. Two algorithms for not-in-class classification task on image datasets. *13th Conference of PhD Students in Computer Science (CSCS)*, pages 130–134, 2022. URL: https://www.inf.u-szeged.hu/~cscs/pdf/cscs2022.pdf.

[3] Bendale, A. and Boult, T. E. Towards open set deep networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1563–1572, 2016. DOI: 10.1109/cvpr.2016.173.

[4] Bengio, Y., Courville, A., and Vincent, P. Representation learning: A review and new perspectives. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(8):1798–1828, 2013. DOI: 10.1109/TPAMI.2013.50.

[5] Bishop, C. M. Novelty detection and neural network validation. *IEE Proceedings — Vision, Image and Signal Processing*, 141(4):217–222, 1994. DOI: 10.1049/ip-vis:19941330.

[6] Clanuwat, T., Bober-Irizar, M., Kitamoto, A., Lamb, A., Yamamoto, K., and Ha, D. Deep learning for classical japanese literature. *arXiv preprint arXiv:1812.01718*, 2018. DOI: 10.48550/arXiv.1812.01718.

[7] Cohen, G., Afshar, S., Tapson, J., and Van Schaik, A. EMNIST: Extending MNIST to handwritten letters. In *International Joint Conference on Neural Networks*, pages 2921–2926. IEEE, 2017. DOI: 10.1109/IJCNN.2017.7966217.

[8] Deng, L. The mnist database of handwritten digit images for machine learning research [best of the web]. *IEEE signal processing magazine*, 29(6):141–142, 2012. DOI: 10.1109/MSP.2012.2211477.

[9] Duchi, J. and Namkoong, H. Learning models with uniform performance via distributionally robust optimization. *arXiv preprint arXiv:1810.08750*, 2018. DOI: 10.48550/arXiv.1810.08750.

[10] El-Sawy, A., El-Bakry, H., and Loey, M. CNN for handwritten Arabic digits recognition based on LeNet-5. In *International Conference on Advanced Intelligent Systems and Informatics*, pages 566–575. Springer, 2016. DOI: 10.1007/978-3-319-48308-5_54.

[11] Finn, C., Abbeel, P., and Levine, S. Model-agnostic meta-learning for fast adaptation of deep networks. In *International Conference on Machine Learning*, Volume 70, pages 1126–1135. Proceedings of Machine Learning Research, 2017. DOI: 10.48550/arXiv.1703.03400.

[12] Ganin, Y. and Lempitsky, V. Unsupervised domain adaptation by backpropagation. In *International Conference on Machine Learning*, pages 1180–1189. Proceedings of Machine Learning Research, 2015. DOI: 10.48550/arXiv.1409.7495.

[13] Ganin, Y., Ustinova, E., Ajakan, H., Germain, P., Larochelle, H., Laviolette, F., Marchand, M., and Lempitsky, V. Domain-adversarial training of neural networks. *The Journal of Machine Learning Research*, 17(1):2096–2030, 2016. DOI: 10.48550/arXiv.1505.07818.

[14] Goodfellow, I., Bengio, Y., and Courville, A. *Softmax Units for Multinoulli Output Distributions.* In *Deep Learning*, chapter 6.2.2.3. MIT Press Cambridge, MA, USA, 2016. URL: http://www.deeplearningbook.org.

[15] Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. Generative adversarial nets. *Communications of the ACM*, 63(11):139–144, 2020. DOI: 10.1145/3422622.

[16] Goodfellow, I. J., Shlens, J., and Szegedy, C. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014. DOI: 10.48550/arXiv.1412.6572.

[17] Higgins, I., Matthey, L., Pal, A., Burgess, C., Glorot, X., Botvinick, M., Mohamed, S., and Lerchner, A. beta-VAE: Learning basic visual concepts with a constrained variational framework. In *International Conference on Learning Representations*, 2017. URL: https://openreview.net/forum?id=Sy2fzU9gl.

[18] Hinton, G., Vinyals, O., Dean, J., et al. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02530*, 2(7), 2015. DOI: 10.48550/arXiv.1503.02531.

[19] Hospedales, T., Antoniou, A., Micaelli, P., and Storkey, A. Meta-learning in neural networks: A survey. *arXiv preprint arXiv:2004.05439*, 2020. DOI: 10.48550/arXiv.2004.05439.

[20] Kim, H. and Mnih, A. Disentangling by factorising. In *International Conference on Machine Learning*, Volume 80, pages 2649–2658. Proceedings of Machine Learning Research, 2018. DOI: 10.48550/arXiv.1802.05983, URL: https://proceedings.mlr.press/v80/kim18b.html.

[21] Krizhevsky, A., Sutskever, I., and Hinton, G. E. ImageNet classification with deep convolutional neural networks. In Pereira, F., Burges, C., Bottou, L., and Weinberger, K., editors, *Advances in Neural Information Processing Systems*, Volume 25. Curran Associates, Inc., 2012. URL: https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

[22] LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998. DOI: 10.1109/5.726791.

[23] Li, H., Pan, S. J., Wang, S., and Kot, A. C. Domain generalization with adversarial feature learning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 5400–5409, 2018. DOI: 10.1109/CVPR.2018.00566.

[24] Li, Y., Tian, X., Gong, M., Liu, Y., Liu, T., Zhang, K., and Tao, D. Deep domain generalization via conditional invariant adversarial networks. In *Proceedings of the European Conference on Computer Vision*, pages 624–639, 2018. DOI: 10.1007/978-3-030-01267-0_38.

[25] Liang, S., Li, Y., and Srikant, R. Enhancing the reliability of out-of-distribution image detection in neural networks. *arXiv preprint arXiv:1706.02690*, 2017. DOI: 10.48550/arXiv.1706.02690.

[26] Liao, Y., Huang, R., Li, J., Chen, Z., and Li, W. Deep semisupervised domain generalization network for rotary machinery fault diagnosis under variable speed. *IEEE Transactions on Instrumentation and Measurement*, 69(10):8064–8075, 2020. DOI: 10.1109/TIM.2020.2992829.

[27] Locatello, F., Bauer, S., Lucic, M., Raetsch, G., Gelly, S., Schölkopf, B., and Bachem, O. Challenging common assumptions in the unsupervised learning of disentangled representations. In *International Conference on Machine Learning*, pages 4114–4124. PMLR, 2019. DOI: 10.48550/arXiv.1811.12359.

[28] Odena, A. Semi-supervised learning with generative adversarial networks. *arXiv preprint arXiv:1606.01583*, 2016. DOI: 10.48550/arXiv.1606.01583.

[29] Papp, D. and Szűcs, G. Balanced active learning method for image classification. *Acta Cybernetica*, 23(2):645–658, 2017. DOI: 10.14232/actacyb.23.2.2017.13.

[30] Papp, D. and Szűcs, G. Double probability model for open set problem at image classification. *Informatica*, 29(2):353–369, 2018. DOI: 10.15388/Informatica.2018.171.

[31] Pearce, T., Brintrup, A., and Zhu, J. Understanding softmax confidence and uncertainty. *arXiv preprint arXiv:2106.04972*, 2021. DOI: 10.48550/arXiv.2106.04972.

[32] Segu, M., Tonioni, A., and Tombari, F. Batch normalization embeddings for deep domain generalization. *arXiv preprint arXiv:2011.12672*, 2020. DOI: 10.48550/arXiv.2011.12672.

[33] Shen, Z., Liu, J., He, Y., Zhang, X., Xu, R., Yu, H., and Cui, P. Towards out-of-distribution generalization: A survey. *arXiv preprint arXiv:2108.13624*, 2021. DOI: 10.48550/arXiv.2108.13624.

[34] Simonyan, K. and Zisserman, A. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014. DOI: 10.48550/arXiv.1409.1556.

[35] Sokolova, M. and Lapalme, G. A systematic analysis of performance measures for classification tasks. *Information processing & management*, 45(4):427–437, 2009. DOI: 10.1016/j.ipm.2009.03.002.

[36] Tzeng, E., Hoffman, J., Zhang, N., Saenko, K., and Darrell, T. Deep domain confusion: Maximizing for domain invariance. *arXiv preprint arXiv:1412.3474*, 2014. DOI: 10.48550/arXiv.1412.3474.

[37] van der Maaten, L. and Hinton, G. Visualizing high-dimensional data using t-SNE. *Journal of Machine Learning Research*, 9(2579-2605), 2008. URL: https://jmlr.org/papers/volume9/vandermaaten08a/vandermaaten08a.pdf.

[38] Viola, P. and Jones, M. Rapid object detection using a boosted cascade of simple features. In *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, Volume 1. IEEE, 2001. DOI: 10.1109/CVPR.2001.990517.

[39] Wang, J., Feng, W., Chen, Y., Yu, H., Huang, M., and Yu, P. S. Visual domain adaptation with manifold embedded distribution alignment. In *Proceedings of the 26th ACM International Conference on Multimedia*, pages 402–410, 2018. DOI: 10.1145/3240508.3240512.

[40] Wang, S., Yu, L., Li, K., Yang, X., Fu, C.-W., and Heng, P.-A. Dofe: Domain-oriented feature embedding for generalizable fundus image segmentation on unseen datasets. *IEEE Transactions on Medical Imaging*, 39(12):4237–4248, 2020. DOI: 10.1109/TMI.2020.3015224.

[41] Xiao, H., Rasul, K., and Vollgraf, R. Fashion-MNIST: A novel image dataset for benchmarking machine learning algorithms. *arXiv preprint arXiv:1708.07747*, 2017. DOI: 10.48550/arXiv.1708.07747.

[42] Yang, M., Liu, F., Chen, Z., Shen, X., Hao, J., and Wang, J. CausalVAE: Disentangled representation learning via neural structural causal models. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 9588–9597, 2021. DOI: 10.1109/CVPR46437.2021.00947.

[43] Zhang, H., Li, A., Guo, J., and Guo, Y. Hybrid models for open set recognition. In *European Conference on Computer Vision*, pages 102–117. Springer, 2020. DOI: 10.1109/JPROC.2021.3052449.

[44] Zhang, X., Zhou, L., Xu, R., Cui, P., Shen, Z., and Liu, H. Domain-irrelevant representation learning for unsupervised domain generalization. *arXiv preprint arXiv:2107.06219*, 2021. DOI: 10.48550/arXiv.2107.06219.

[45] Zhou, F., Jiang, Z., Shui, C., Wang, B., and Chaib-draa, B. Domain generalization with optimal transport and metric learning. *arXiv preprint arXiv:2007.10573*, 2020. DOI: 10.48550/arXiv.2007.10573.

[46] Zhu, L., Chen, Y., Ghamisi, P., and Benediktsson, J. A. Generative adversarial networks for hyperspectral image classification. *IEEE Transactions on Geoscience and Remote Sensing*, 56(9):5046–5063, 2018. DOI: 10.1109/TGRS.2018.2805286.

Acta Cybernetica **26** (2024) 323–371.

Towards a Block-Level ML-Based Python Vulnerability Detection Tool[*]

Amirreza Bagheri[ab] and Péter Hegedűs[ac]

Abstract

Computer software is driving our everyday life, therefore their security is pivotal. Unfortunately, security flaws are common in software systems, which can result in a variety of serious repercussions, including data loss, secret information disclosure, manipulation, or system failure. Although techniques for detecting vulnerable code exist, the improvement of their accuracy and effectiveness to a practically applicable level remains a challenge. Many existing methods require a substantial amount of human expert labor to develop attributes that indicate vulnerabilities. In previous work, we have shown that machine learning is suitable for solving the issue automatically by learning features from a vast collection of real-world code and predicting vulnerable code locations. Applying a BERT-based code embedding, LSTM models with the best hyperparameters were able to identify seven different security flaws in Python source code with high precision (average of 91%) and recall (average of 83%). Upon the encouraging first empirical results, we go beyond this paper and discuss the challenges of applying these models in practice and outlining a method that solves these issues. Our goal is to develop a hands-on tool for developers that they can use to pinpoint potentially vulnerable spots in their code.

Keywords: deep learning, vulnerability detection, source code embedding, data mining

1 Introduction

In today's applications, security bugs (i.e., vulnerabilities) in software are becoming increasingly difficult to detect, allowing hackers and attackers to profit from

[*]The research was supported by the Ministry of Innovation and Technology NRDI Office within the framework of the Artificial Intelligence National Laboratory Program (RRF-2.3.1-21-2022-00004) and by project TKP2021-NVA-09, implemented with the support provided by the Ministry of Innovation and Technology of Hungary from the National Research, Development and Innovation Fund, financed under the TKP2021-NVA funding scheme.

[a]Institute of Informatics, University of Szeged, Hungary

[b]E-mail: bagheri@inf.u-szeged.hu, ORCID: 0000-0001-9691-7937

[c]E-mail: hpeter@inf.u-szeged.hu, ORCID: 0000-0003-4592-6504

DOI: 10.14232/actacyb.299667

their exploits. Tens of thousands of such flaws are discovered and fixed each year. Manually auditing source code and discovering vulnerabilities is time-consuming at best, if not impossible.

In our previous work [2], we have shown that machine learning is suitable for solving the issue automatically by learning features from a vast collection of real-world code and predicting vulnerable code locations. The dataset was gathered from GitHub and contains Python code with a variety of vulnerabilities that is embedded into a vector space using one of three embedding models (word2vec, fasttext, BERT) [5, 16, 8]. Individual code tokens and their context are extracted from the source code of the vulnerable files to provide data samples for fine-grained analysis. Then, we trained various machine learning (ML) models to see how effective they were at identifying vulnerable code parts.

The entire training process can be divided into two parts: first, an embedding model is trained using its parameters, such as min-count (how frequently a token must appear in the training corpus to be assigned a vector representation), and second, the system is trained using its parameters, such as min-count or iterations. After that, the code blocks can only be encoded in their vector representations. We found that LSTM models were the most suitable, thus we used them and trained them with different hyperparameters, such as the number of neurons or dropout, in the second stage. Applying a BERT-based code embedding, LSTM models performed the best, they were able to identify seven different security flaws in Python source code with high precision (average of 91%) and recall (average of 83%).

Following a successful empirical evaluation, the results must be implemented in practice. However, there are several difficulties in putting the above-described method into practice and making it available as a developer tool. The training data samples are code snippets (extracted from vulnerability fixing commits), but when we use vulnerability identification in practice, we use the entire program as input. To use code embedding and model prediction, we need a method for efficiently locating code blocks within the program. Furthermore, because these code blocks may overlap, we require a method for aggregating block-level predictions.

In this paper, we focus on overcoming these challenges and outline a potential developer tool that developers can use. We apply a small focus area and a sliding context window to divide the code into blocks. The focus area moves through the code, and with each step, the model gathers surrounding information, generates a prediction based on that context as input, and uses that prediction to determine the vulnerability rating of the focus area. In a developer tool, the different classification confidence levels may be represented by different colors. To summarize our contributions, we provide a block-level vulnerability prediction method that is practically applicable to Python code, in contrast to the majority of other research initiatives, which are primarily focused on Java, C, C++, or PHP and do not provide guidance on practical application. Furthermore, existing vulnerability prediction approaches provide predictions at higher abstraction levels, such as methods, classes, or files, whereas we aim for finer-grained, smaller block-level predictions.

2 Related Work

This section describes previous works in finding vulnerabilities and also attempts a classification, although there are many different criteria under which approaches can be compared. The advantages and disadvantages of the previous approaches are described.

2.1 Vulnerability Prediction Based on Software Metrics

What characteristics should be considered while determining whether or not code is vulnerable? For a long time, the most popular features were software and developer metrics. Code churn, developer activity, coupling, amount of dependencies, and legacy metrics are examples [22]. Such metrics are widely employed as features in fault prediction models [13], and they are extremely important in the field of software quality and reliability assurance. Nagappan et al. [23], for example, use organizational measures to predict software failures.

Although it appears that those data may be employed in vulnerability prediction, there are significant issues with this. For starters, two pieces of code could have the same characteristics but completely different behaviors, resulting in a distinct risk of being exposed. They also tend not to transfer well from one software project to the next. The most serious complaint is that such measurements fail to capture the semantics of the code [30], and that this method ignores the source code, program behavior, and data flow. The method effectively applies a presumption that particular meta characteristics will be linked to security issues, which is not always accurate [15].

Many vulnerabilities can, for example, be found in extremely simple programs. The simplest or most direct solution to an algorithmic problem frequently lacks the safeguards and measures necessary to prevent attacks, which is precisely why software engineers working under time constraints or with little familiarity with security issues have difficulty. Code complexity isn't a perfect indicator of security problems, and there are analogous arguments and counterexamples for the other measures as well. However, it must be accepted that software metrics can provide at least some insight. This is demonstrated in the following studies, which employ machine learning techniques and code metrics to anticipate the occurrence of software security problems. Shin et al. [30] estimate vulnerabilities in JavaScript projects using nine complexity measures, with a low false positive rate but a large false negative rate. Using linear discriminant analysis and Bayesian networks, the authors used code complexity, code churn, and developer metrics to identify vulnerabilities in a later paper [29], attaining 80 percent recall and 25 percent false positives. Chowdhury et al. [4] use complexity, coupling, and cohesion metrics to try to anticipate software vulnerabilities using methodologies that have previously been used for fault detection. They conduct a study on Mozilla Firefox releases and anticipate vulnerabilities using decision trees, random forest, logistic regression, and naive Bayes models, with precision and recall of roughly 70% and 70%, respectively. Zimmerman et al. contributed to the list by looking into code churn, code complexity, code coverage, organizational metrics, and actual dependencies [40].

They discovered a weak but statistically significant link between the indicators studied and utilized logistic regression to identify vulnerabilities based on them, with an emphasis on Windows Vista's proprietary code. The measures were able to anticipate vulnerabilities with a median precision of 60%, but a recall of 40%, which was disappointing. When using support vector machines to anticipate vulnerabilities using import statements, Neuhaus et al. [24] found an average precision of 70% and recall of 40% when using import statements in the Mozilla project. Yu et al. [38] include a variety of factors, including software metrics like the number of subclasses or methods in a file, as well as crash features and code tokens with associated tf-idf scores. As a result, their strategy incorporates a variety of perspectives. They can forecast vulnerabilities at the file level and get very good results in reducing the amount of code that needs to be reviewed by humans to detect a vulnerability.

Other researchers have made predictions based just on commit messages. Zhou et al. [39] use a K-fold stacking technique to examine commit messages to forecast whether or not a commit contains vulnerabilities. In contrast, Russel et al. [26] discovered that humans and Machine Learning systems both struggled to identify build failures or defects based just on commit messages. Our approach, the suggested method, does not use external code measurements and instead learns characteristics directly from the source code.

2.2 Anomaly Detection Approaches for Finding Vulnerabilities

The task of characterizing normal and anticipated behavior and finding deviations from it is known as anomaly detection. The assumption is that code that does not follow the indicated criteria is frequently the source of a bug. To evaluate source code and uncover normal coding patterns, data mining techniques were applied. For instance, Li et al. [19] developed PR-Miner, a tool that can discover code patterns in any programming language and has shown to be highly beneficial. Their method, which is based on associating programming patterns that are used in tandem, is independent of the language used, and the violations detected by their tool have been confirmed as flaws in Linux, PostgreSQL, and Apache. However, a basic issue is that faults that are themselves normal patterns are routinely neglected, resulting in common vulnerabilities going undetected [36].

Rare programming patterns or API usages, on the other hand, may be labeled as false positives simply because they are uncommon. Several anomaly detection methods have a high risk of false positives [10]. Anomaly detection in code is not a simple way for detecting security vulnerabilities, because it is difficult to tell when a violation of typical code patterns has a security implication and when it does not. The method utilized in this study varies from traditional anomaly detection in that explicit labels are used to train a model on both vulnerable and secure code, avoiding the dubious assumption that ”normal” = ”right.” It falls under the heading of susceptible code pattern analysis.

2.3 Vulnerable Code Pattern Analysis and Similarity Analysis

In comparison to learning about abstract metrics or the notion of proper code, it seems almost natural to just try to answer the question: What does vulnerable code often look like? Vulnerable code pattern analysis and similarity analysis are two slightly different approaches to answering that question. The name suggests that similarity analysis achieves exactly that. The purpose is to locate the most comparable code fragments to a susceptible code snippet, presuming that they are at risk of sharing the vulnerability. This method works well for identical or almost identical code clones in which the compared code fragments' inherent structure is quite close [18], a circumstance that occurs frequently, especially in the open-source community.

Susceptible code pattern analysis examines vulnerable code segments using data mining and machine-learning techniques to discover common characteristics. These characteristics are patterns that can be used to detect vulnerabilities in new code portions. As detailed by Ghaffarian et al. [10], most of the work in this area gathers a huge dataset, analyzes it to extract feature vectors, and then applies machine-learning techniques to it. Both methodologies are often used to analyze source code without running it, which is known as a static analysis, while some academics also use a dynamic analysis. The bottom line is that, unlike 'conventional' static analysis, the characteristics are generated automatically or semi-automatically, removing the need for subjective human specialists. An unbiased model can be developed by learning directly from a dataset of code what susceptible code includes.

In many cases, those approaches use a coarse granularity, classifying entire programs [11], files [29], components [24], or functions [37], making pinpointing the specific position of a vulnerability hard. Some researchers, such as Li et al. [18] and Russell et al. [26], employ a finer representation of the code. Furthermore, the approaches differ in several ways, including the language used, the data source, the dataset size, the labeling process, the granularity level of the analysis (whole files down to code tokens), the machine learning model used, the types of vulnerabilities examined, and whether the model can be used in cross-project predictions or only on the project it was created for. To begin, some fundamental approaches utilizing various machine learning techniques will be discussed.

Following that, deep learning-based techniques are investigated in further depth. Morrison et al. [22] look at security flaws in Windows 7 and Windows 8. 8 using a variety of machine learning approaches such as logistic regression, naive Bayes, and others With very unsatisfactory results, support vector machines, and random forest classifiers. As a result, the precision and recall levels were quite poor. Pang et al. [25] extract labels from an internet database in a fairly basic manner. To classify the entire Java code, utilize a combination of feature selection and n-gram analysis. susceptible or not vulnerable classes. They use a simple n-gram model in combination with feature selection methods to integrate related features and limit the number of irrelevant features taken into account on a relatively small dataset of four Java android applications. After that, they use support vector machines as their learning algorithm, with 92 percent accuracy, 96 percent precision, and 87 percent recall inside the same project, and 65 percent in cross-project prediction (training on one project and trying to classify vulnerable files in another one).

Shar et al. [28] use machine learning to detect XSS and SQLI vulnerabilities in PHP code and reduce false positives. They manually select specific code attributes before training a multi-layer perceptron to augment static analysis tools. They discovered fewer vulnerabilities than a static analysis tool, but with reduced false positive rates, resulting in a satisfactory outcome. They adopt a hybrid technique with dynamic analysis in their later work [28], which significantly improves their earlier results, as tested on six large PHP projects. They also try unsupervised predictors, which are less accurate but still a fascinating study topic.

Raw source code is analyzed as text by Hovsepyan et al. [15]. They used an Android email client built in Java as an example, focusing on evaluating the source code as if it were a natural language and processing files as a whole. They convert files into feature vectors made up of Java tokens with their respective counts in the file after filtering out comments. These feature vectors are classed as susceptible or clean in a binary approach. Finally, a support vector machine classifier is trained to determine whether a file is vulnerable. This classifier has an accuracy of 87 percent, with 85 percent precision and 88 percent recall. Their success demonstrates that evaluating source code as natural text and gaining insight without sophisticated models of code representation is possible. Unfortunately, the application on a single software repository limits their work. For a comparable job, they later utilized decision trees, k-nearest-neighbor, naive Bayes, random forest, and support vector machines [27].

2.3.1 Deep Learning for Vulnerability Prediction

Several papers have successfully used deep learning models to automatically learn features for fault prediction [35]. The following works show how this approach can be applied to vulnerability detection. Russell et al. [26] employ recurrent and convolutional neural networks to scrape a large codebase of C projects from GitHub, the Debian Linux distribution, and synthetic examples from the SATE IV Juliet test suite, resulting in a database of over 12 million functions. They produce the binary labels 'vulnerable' and 'not vulnerable' for the routines using three separate static tools, as well as a randomly initialized one-hot embedding for lexing. Convolutional and recurrent neural networks are used for feature extraction at the core of their research, followed by a random forest classifier. The best results came from convolutional neural networks, which allowed for fine-tuning of precision and recall against each other.

Russel et al. are not only among the first researchers to use deep representation learning directly on source code from a large codebase, but they are also able to use a convolutional feature activation map to highlight suspicious parts of the code, rather than simply classifying a whole function as vulnerable, with their work. The work of Liu et al. [20] is based on the notion that violations that are consistently remedied are genuine positives, whereas violations that are disregarded are likely to be either not important or false positives. They look into changes in 730 Java projects, use the static bug detection tool Findbugs to find changes that are fixing a violation reported by that tool, then follow the violations across versions to see if they are addressed or ignored. They can use this information to determine which tool-reported infractions are consistently disregarded over several revisions and which are addressed almost immediately. They collect the code patterns that correlate to infractions using an abstract syntax tree as a representation.

Liu et al employ an unsupervised learning approach to extract features of code, focusing primarily on patches to learn fix patterns, rather than building a binary classifier on 'vulnerable' or 'not vulnerable.' As a result, their method may be characterized as a type of similarity analysis. The discovered coding patterns are encoded into a vector space using an embedding layer, the discriminating features are learned using a convolutional neural network, and violations with learned features are clustered using an X-means clustering technique. They discovered that, while security-related breaches are uncommon, they are common in 30 percent of the projects. Furthermore, the research shows that only a small percentage of breaches are corrected. Liu et al. discovered that for 90% of fixed breaches, a chunk of merely 10 lines of code or fewer is adequate to capture the relevant context. The CNN produces patterns that are nearly identical to the tool's violation description and are used to build fixed patterns. One of the top five suggested fix patterns can fix roughly one-third of a test set of violations. Liu et al. also chose 10 open-source Java projects to offer proposals to based on the modifications proposed by their program, with 67 of the 116 suggestions being accepted right away. Of course, their technology can only suggest patches that match previously discovered fix patterns.

2.3.2 Long-short Term Memory Networks

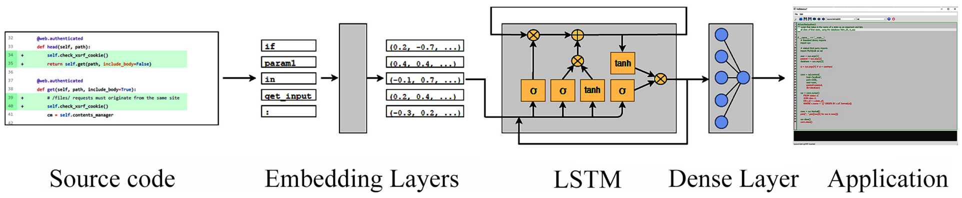
Although Gupta et al. [12] and Dam et al. [6] have demonstrated that extended short-term memory networks are well suited to modeling source code and correcting faults in C code, the latter was likely the first to do so. to learn features automatically for anticipating security vulnerabilities [7]. They Extract the code from a publicly available dataset including 18 Java applications. utilizing Java Abstract Syntax Tree to replace all methods in the source file Some tokens are available in generic versions. They then employ LSTM to train syntactic and semantic skills.

A random forest classifier and semantic characteristics. They got over 91 percent precision for within-project vulnerability prediction, and after training a model on one project, it got over 80 percent precision and recall in at least four of the other 17 projects. VulDeePecker [18] is a deep learning-based vulnerability detection method. The authors propose the first dataset of vulnerabilities targeted for deep learning algorithms, which is derived from the National Vulnerability Database and the Software Assurance Reference Dataset maintained by the NIST and come from popular C and C++ open-source products.

Li et al. want to design a tool that doesn't rely on humans to determine features but still has a low rate of false positives and false negatives. They divided files into code gadgets, which are semantically related lines of code that are grouped, focusing on critical areas of library and function API calls in a very sophisticated manner. Only two sorts of vulnerabilities are evaluated: buffer errors and resource management problems. On different subsets of their data, Li et colleagues chose bidirectional long short-term memory networks, attaining a precision of roughly 87 percent, with better results if the network is trained on manually selected function calls. Harer et al. [14] used LSTM networks to detect and resolve C vulnerabilities in the synthetic SATE IV code base. They were able to use a sequence-to-sequence strategy to develop solutions for discovered vulnerabilities, albeit measuring and comparing their success is difficult. Similarly, Gupta et al. [12] employ RNNs in a sequence-to-sequence configuration to remedy flawed C code, while not focusing on security vulnerabilities, fixing 27 percent of their applications completely and 19 percent partially.

3 Approach

Our approach to vulnerability detection is to analyze code tokens and their surrounding tokens to determine the context in which they exist. Using embedding layer models, the code is embedded into semantically meaningful numerical vectors. After that, an LSTM network is used to recognize vulnerable code features and categorize code as vulnerable or not vulnerable. The overview of the approach is shown in Figure 1.



```
    @web.authenticated
    def head(self, path):
+       self.check_xsrf_cookie()
+       return self.get(path, include_body=False)

    @web.authenticated
    def get(self, path, include_body=True):
+       # /files/ requests must originate from the same site
+       self.check_xsrf_cookie()
        cm = self.contents_manager
```

Figure 1: Overview of the approach

3.1 Data Source

In prior research, the researchers found that applying their model to code from the same project that it was trained on yielded better results in detecting vulnerabilities [25]. Cross-project prediction resulted in a significant reduction in precision and recall. In the works of Russel et al. [26] and Li et al. [18], the best results were obtained when dealing with a synthetic data set rather than code from real applications. Nonetheless, because such a vulnerability detection tool appears to be the most desirable and final result, our strategy attempts to leverage a huge dataset of real-life source code to train a model that can be applied to any code, not just one project.

For numerous reasons, the entire dataset was compiled from publicly available GitHub projects: First, because GitHub is the world's largest repository of source code, the amount of meaningful data accessible is unlikely to be insufficient for this application. Second, unlike synthetic code bases, nearly all GitHub projects contain 'natural' source code in the sense that they are real-world projects. Third, the data is open, making it easier to re-examine and reproduce the work, which is difficult in studies that focus on proprietary code, for example. Because GitHub is primarily a version control system, it is centered on commits, and as Zhou et al. [39] explains, it is possible to detect vulnerabilities by looking at commits. Patches are commits that remedy a defect or vulnerability and consist of two versions, one buggy and one updated and correct. Vulnerable code patterns can be discovered by evaluating the differences between the old and new versions.

3.2 The Data Collection Process

Commits that fixed numerous vulnerabilities were gathered. Each vulnerability needed its dataset after the data had been collected and filtered. Table 1 summarizes the fundamental data of the dataset, such as the number of repositories and commits that comprise it, the number of modified files that contain known vulnerabilities, the number of lines of code, the number of distinct functions they contain, and the total number of characters.The following sections will demonstrate the suitability of this dataset by using it to train the model.

Table 1: Vulnerability Dataset

Vulnerability	Repo.	Commits	Files	Functions	LOC	Chars
SQL Injection	457	582	721	7452	102558	5960074
XSS	52	89	102	983	18916	1236587
Command injection	125	225	354	3561	48031	2740339
XSRF	112	189	384	6418	76198	3368206
Remote code execution	71	88	186	4198	40591	1955087
Path disclosure	175	204	332	4596	62303	2814413

3.2.1 Python Vulnerabilities

Injection attacks - An injection attack is based on user input that causes unexpected or harmful behavior when processed or executed. A user can sometimes access or change data without permission by exploiting an injection vulnerability, which usually allows the user to have the interpreter (such as the server or the operating system) execute arbitrary commands. Injection attacks can be avoided by vetting all user input and employing so-called "sanitization" techniques that convert harmful to harmless inputs, such as filtering out special characters.

SQL injection - The OWASP foundation lists SQL injections as one of the top security flaws, ranking them among the most prevalent and dangerous flaws affecting web applications. The Common Weakness Enumeration defines SQL injection as ”when a SQL command is provided to a downstream component, the software generates all or a portion of it using externally influenced input from an upstream component, but fails to neutralize or does so in a way that may cause it to change the intended SQL command.” When user-controllable input contains SQL syntax that has not been removed, it can be misinterpreted and executed as a SQL statement. This can be used to alter searches, such as accessing files that should not be accessible or adding new statements that can alter or destroy databases. If its sanitization is not thorough, any form of a database-driven website could end up being the subject of such an exploit.

Command injection - According to the Common Weakness Enumeration, the software constructs all or a portion of a command using externally affected input from an upstream component, but it fails to neutralize or does so incorrectly specific aspects that could change the intended command when it is sent to a downstream component. This is another instance of untrusted data being executed, but instead of being directed at a SQL database, it is directed at a command run by the system being attacked, such as the server shell. An attacker could then read, modify, or delete files that they shouldn’t have access to.

Remote code execution - The primary distinction between command injection and remote code execution is that command injection executes an OS system command, whereas remote code execution executes actual programming code on the target machine. It is also sometimes used to define a hacking goal rather than a vulnerability, in the sense that an attacker can execute arbitrary commands on a system by exploiting a vulnerability.

Various types of session hijacking - The main goal of session hijacking is to allow an attacker to enter a client’s connection with a server, typically by obtaining or guessing a valid session token and then posing as a trusted client.To connect to a client using a maliciously set session ID by a third party, a user must be tricked into clicking a link that contains the session ID as a parameter. Because the malicious third party now has access to the session token, the active session can be accessed. An attacker could even gain access to a logged-in account. By using cross-site scripting to obtain a session token, the attacker can hijack the shared session between the client and server.

Man-in-the-middle attacks are also included in session hijacking. An attacker pretends to be the connection partner on both sides of a conversation between two systems, possibly a client and a server. Because they are effectively acting as a proxy, the attacker can view and occasionally change the content of the communication. By utilizing appropriate encryption and certifications, man-in-the-middle attacks are avoided. The term ”replay attack” refers to an attack in which the attacker, posing as the original originator of the transmission, records a legitimate portion of communication between two parties (such as a client and a server) and sends it again later. The attacker can access features and data that were only intended to be accessible to the original sender if suitable protective measures are not in place (primarily secret one-time session IDs).

Cross-site scripting - Cross-site scripting, also known as XSS, is one of the most serious flaws in web applications. It frequently appears on OWASP’s top ten list of vulnerabilities. Unsanitized data is also central to the cross-site scripting problem. This time, a user adds custom code to a website or URL before passing it on to other users, who will see the code as part of the page and run it in their browser. The CWE defines cross-site scripting as follows: The program either does not neutralize user-controllable input at all or neutralizes it incorrectly before including it in output that is used to create a web page that is served to other users.

A guest book that accepts arbitrary input is a simple example of stored cross-site scripting. A visitor may post plain text or Javascript, which is saved on the website permanently and distributed to other users, who will receive and run the Javascript code. Of course, changes can be made, such as using different input methods and producing executable code in languages such as Flash. Another example is an email that contains a link to another website, but the URL contains malicious Javascript that, when clicked, executes the malicious code. To prevent XSS attacks, user-generated content should be sanitized with tools such as HTML escape and others.

Cross-Site request forgery - The CWE defines cross-site request forgery as follows: The web application does not, or is unable to, thoroughly verify whether the request was submitted by a well-formed, legitimate, consistent user. This is further explained below: If a web server is designed to accept requests from clients without any means of verifying that they were made voluntarily, an attacker may be able to trick a client into sending an unintended request to the website that would be treated as a legitimate request. This can expose data or result in accidental code execution and can be accomplished via a URL, image load, XMLHttpRequest, or other means.

Directory traversal/path disclosure -When a user changes the input in such a way that paths of a file system that were not intended to be accessed are exposed, this is known as a path traversal or directory traversal vulnerability. According to the CWE, the software does not properly neutralize special elements in the pathname that could cause it to resolve to a location outside of the restricted directory. The software generates a pathname from external input to identify a file or directory that is located beneath a restricted parent directory. A common example of this vulnerability is a website that displays a file whose path is specified in a URL parameter. The attacker can explore the file system and possibly show files that weren't intended to be accessible by altering this parameter to contain some "../../..".

3.2.2 Scraping GitHub

The first step is to build a dataset, or more precisely, to find a large number of commits that address a security issue. Because the goal is to cover a wide range of vulnerabilities, each vulnerability type requires multiple examples. Commits are the main topic of interest in our work because the process of patching a flaw indicates the presence of the flaw in the first place and provides the basis for labeling the data afterward. The GitHub search API can only handle certain types of requests, and the number of results for each request is limited to 1000. Filters cannot be implemented in the search API, unlike the regular search available to users, so filtering for only the programming language is not possible. As a result, after obtaining the results and selecting the few relevant and useful ones from among them, this filtering must be done manually. As a result, the approach taken here is to write a script that searches the Github API for contributions containing various security-related search phrases, then filters out everything that isn't relevant, such as code written in a different programming language or configuration files. The script is included in the repository and authenticates with an API token.

Initially, a lengthy list of security-related terms was used. These terms are based on prior research citezhou2017automated, the CVE database, and the OWASP foundation's list of security risks. To collect the data, a script was written that connected to the GitHub API via the requests library. The keyword list must be supplied at the beginning of the script. This first set of keywords will be combined with a second set of keywords related to improvements, repairs, or modifications in order to consider every possible combination of the first and second set elements. Because the second set of terms denotes a problem or a solution, the combinations should be useful (but not sufficient) in distinguishing genuine security improvements from numerous other mentions of vulnerabilities, such as examples in showcase projects for educational purposes.

However, it quickly became clear that only a few of those keyword combinations were truly relevant to the task at hand. Some, like 'vuln', 'XXE','malicious', or 'CVE', were overly broad and yielded a wide range of results; others, like 'dos' (as an abbreviation for denial of service), yielded completely unrelated results due to overlap of meanings (in this case, 'dos' referring to an old Windows operating system, and, even more frequently, the very common Portuguese As a result, the available options were significantly reduced. After combining every keyword from the revised first set with every keyword from the second set, a search request is sent to Github for each of the combinations. It should be noted that this only means that the names (and thus URLs) of commits and repositories are gathered; no actual source code or even a diff file is downloaded at this time. After combining every keyword from the revised first set with every keyword from the second set, a search request for each combination is submitted to Github. It should be noted that this only collects the names (and thus URLs) of commits and repositories: no actual source code or even a diff file has been retrieved.

3.2.3 Filtering the Results

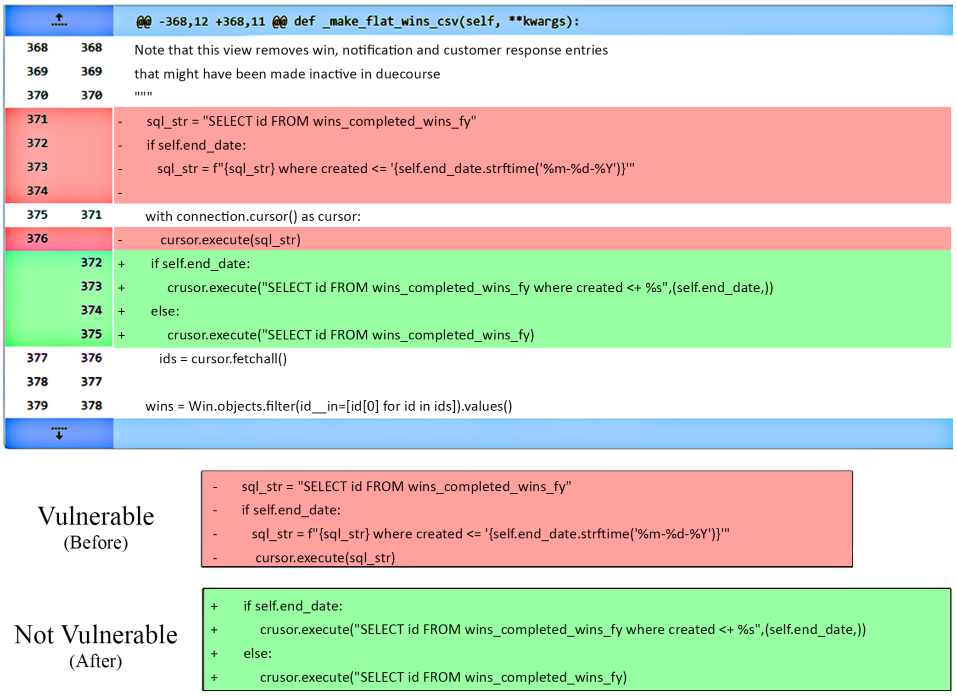
The second priority was to find projects that display security flaws, exhibit exploits, or serve as tools for attacking or preventing exploits. While those works frequently include useful examples of vulnerabilities, they rarely include commits that repair them, but rather commits that introduce them into the codebase on purpose. Furthermore, they run counter to the work's methodological assumptions, as the goal is to learn about vulnerable code as it appears in real-world projects where developers make legitimate mistakes. As a result, an attempt is made to screen such projects out.

With the -b parameter, the script could include a list of keywords to indicate projects that should be avoided. The repository names were searched for the following keywords: "offensive", "pentest", "vulnerab", "security", "hack", "exploit", "ctf", "capture the flag", "attack". The README files for the remaining projects are then downloaded from Github. The next step is to obtain the diff files. In the GNU diff model and similar representations used by GitHub, a diff is a text file that represents the changes made in a commit. It contains some metadata (such as the filename and change line number), the modified lines, and three lines of code before and after. A '+' at the beginning of a line denotes a new and fixed line, whereas a '-' denotes a line that was eliminated in favor of the repair [20]. A commit on Github might include modifications to multiple files at once.

A single HTTP request can be used to download the diff for a commit URL. This is a far easier way than cloning the entire repository and selecting individual files from a certain point in the project's history, which appears to be impossible at this time due to the size of the dataset and computational and temporal constraints. The preceding phase produces a large number of code diffs that can be used to recreate important lines of code in the state before and after the modification. The diff from GitHub includes the modified lines as well as three lines before the first change and three lines after the last change for each changed file, so there isn't much context for the change. However, the vast number of changes that can be mined with this method may more than compensate for the comparatively limited context offered for each change. The time it took to run all of those queries was over 80 hours.

To build the dataset, we obtained only the diff files and recreated the 'before version' and 'after version' of the required code snippet, each with the modified lines and three lines above and below them. The goal was to classify the first version as vulnerable and the second version as 'not vulnerable', which yielded some pleasing results. The classifier that had learned from the training set was able to accurately classify the validation set samples and determine whether they belonged in the 'previous/vulnerable' or 'after/fixed' categories. When the model was applied to a new file containing source code, it went through several parts of it and tried to identify them, and the problem became evident.

That endeavor resulted in an astonishing number of false positives. The reason behind this is that the dataset had the same number of (actual) positives and negatives, whereas in reality, Figure 2: In between numerous lines of 'clean' code, retrieving the snippet in the state before and after the commit from a git diff, the old vulnerable version in red, the new vulnerable version in green The dataset does not accurately reflect the class unbalanced nature of the data to which the classifier should be applied. Of course, this was clear from the start, but owing to the aforementioned time and processing resource constraints, it appeared that collecting the diffs was simply the best technique that could be done at all. This was not accurate, and a better solution may be found.



```
@@ -368,12 +368,11 @@ def _make_flat_wins_csv(self, **kwargs):
   Note that this view removes win, notification and customer response entries
   that might have been made inactive in duecourse
   """
-     sql_str = "SELECT id FROM wins_completed_wins_fy"
-     if self.end_date:
-        sql_str = f"{sql_str} where created <= '{self.end_date.strftime('%m-%d-%Y')}'"
-
      with connection.cursor() as cursor:
-         cursor.execute(sql_str)
+      if self.end_date:
+           crusor.execute("SELECT id FROM wins_completed_wins_fy where created <+ %s",(self.end_date,))
+      else:
+           crusor.execute("SELECT id FROM wins_completed_wins_fy)
          ids = cursor.fetchall()

      wins = Win.objects.filter(id__in=[id[0] for id in ids]).values()
```

Vulnerable
(Before)

```
-     sql_str = "SELECT id FROM wins_completed_wins_fy"
-     if self.end_date:
-        sql_str = f"{sql_str} where created <= '{self.end_date.strftime('%m-%d-%Y')}'"
-          cursor.execute(sql_str)
```

Not Vulnerable
(After)

```
+      if self.end_date:
+           crusor.execute("SELECT id FROM wins_completed_wins_fy where created <+ %s",(self.end_date,))
+      else:
+           crusor.execute("SELECT id FROM wins_completed_wins_fy)
```

Figure 2: Vulnerable and Not Vulnerable parts selection

3.2.4 Downloading the Dataset

We noticed that downloading the source code in a reasonable amount of time was possible if all of the filterings were done beforehand in a clever way to keep the number of downloaded repositories to a bare minimum. First, the commit is examined to see if it contains keywords related to the vulnerability. The diff file is then examined to see if any files with the code language of choice are affected. If this is not the case, the commit can be ignored because only commits that change specific language source code files are taken into account. The commit is then compared to the previously downloaded commits. By definition, many open-source repositories are forks or clones of one another, or they contain the commit history of other projects. Duplicates are excluded. The distinction is then thoroughly examined.Each change in the commit has an effect on a specific file. The filename is reviewed for each modification to see if it contains terms that indicate it is a showcase project - a file called ”SQL exploit” is more likely to be part of a project exhibiting vulnerabilities than a patch that fixes an inadvertent vulnerability.

The body of the diff file is then processed. If HTML tags or the keywords’sage’ are used, the diff is no longer considered. Although HTML code is sometimes embedded in some files, the vulnerabilities in those files are almost never in the same code. Sage is an open-source mathematics system, and some commits include parameters and variables that are useful to it but not relevant to this project. Finally, the change is checked to see if any lines of code have been removed or replaced. If there are only minor changes, the algorithm will struggle to determine which lines are vulnerable. Finally, after much deliberation, it is determined which commits are truly worth downloading.

Pydriller [32], a tool for downloading repositories containing intriguing commits and traversing their commitments to identify all the matches with the commits that remain in the collection of interesting ones, is only now being used. Some checks are performed anew for each commit. The commit is skipped if the prior file is empty. The commit is also avoided if the previous file is longer than 30.000 characters. Similar to the file name, the commit message is reviewed for suspicious terms. Finally, the source code for the dataset is downloaded and saved.

3.2.5 Flaws in the Data

When we dug deeper into the data, we discovered that the process of collecting vulnerability samples based on commit messages is far from perfect. There were still some (albeit minor) commits that contained exploit implementations rather than fixes, such as setups for capturing the flag, attack demonstrations, or cyber security tools like Burp Suite. Some commit messages, for example, read 'fix remote code execution,' and this vulnerability is repaired somewhere, but the same commit also contains, for example, eight other files with minor and significant changes that may or may not be related to the issue indicated in the commit message. It's difficult to tell whether modifications are related to the commit message's stated goal without human supervision or predetermined knowledge.

The answers for several keywords were just unspecific. There were many results for the phrase brute force in which a brute force strategy was utilized to solve a problem rather than a defense against a brute force attack. As a result, the findings were not particularly useful. A similar issue arose with the phrase tampering, which was used seldom and for a variety of reasons (including DNS tampering, but also game data manipulation for cheating purposes). The term "keyword hijacking" was frequently used in a figurative sense, for example, to describe a person or application that inserted undesirable but authorized material, or to describe data fields or entries that were used by the developers for other purposes as intended. Many fixes and changes relating to developers traversing their file structures, not an attacker attempting to do so, were found using the phrase directory traversal.

Changes were occasionally overly convoluted and spanned numerous files, including those that were not written in Python. The more complex the modifications and the more lines changed, the more difficult it is to model and learn from the sample. Another issue is that many vulnerabilities are defined by the lack of specific defense mechanisms, such as XSRF tokens or nonces/counters that prevent replay attacks. Fixing those vulnerabilities sometimes does not alter or remove a susceptible section of code, resulting in insights into what vulnerable code looks like, but instead adds a few extra lines. In other circumstances, those lines can be positioned in a variety of ways, with a variety of ways to provide the needed functionality. Learning to notice the lack of something vague that is required is far more difficult than learning to recognize a very explicit erroneous piece of code that is there.

Commits using replay attacks typically had both of the aforementioned issues: They're dispersed throughout a lot of files, and they usually add new lines rather than alter an existing, broken code segment. As a result, this kind of vulnerability had to be ruled out. There were just a few results for man-in-the-middle attacks that were trying to harden an application against them rather than performing them. And the defense systems were so specialized that they yielded little usable information. The majority of the unauthorized commits were likewise not related to fixing susceptible code segments, but instead invoked methods or handled errors that were not particularly tied to a vulnerability.

There were simply too many applications outside the realm of security and vulnerabilities that were only concerned with pretty formatting of outputs rather than preventing vulnerabilities exploiting format strings, and there were simply too many applications outside the realm of security and vulnerabilities that were only concerned with pretty formatting of outputs rather than preventing vulnerabilities exploiting format strings. Other types of vulnerabilities, such as cross-site scripting, command injection, cross-site request forgery, path disclosure, remote code execution, open redirect vulnerabilities, SQL injection, and so on, did provide excellent learning opportunities.

3.2.6 Filtering the Data

Individual samples were subjected to specific constraints to improve the dataset's quality. Only files with a length of fewer than 10,000 characters were considered. This offers some advantages: Long portions of comments, docstrings, and manually specified variables are common features of very long files. Furthermore, they act as a form of 'long tail' in terms of computing costs, requiring a significant amount of time to analyze for very small advantages. Finally, certain manual examinations revealed that they do not appear to contain the best quality code. Commits that removed or changed a file in more than 10 different locations were removed from the sample to improve the dataset's quality even more. Such bulk modifications are likely to affect several different concerns at once, rather than just one. Of course, such steps lowered the number of samples. In the case of SQL injections, for example, the dataset was reduced from 842 repositories and 903 commits affecting 2354 files totaling 212913 lines of code to 457 repositories, 582 commits, 721 files, and 102558 lines of code. The quality of the data did not suffer as a result of the reduction, as a test of the final model with the non-trimmed dataset yielded no better results.

A severe flaw in the code was introduced at this time, which was only discovered and repaired late in the process. After identifying which lines of code in the diff file were susceptible, they were removed from the source code and labeled as such. The rest of the file was then divided into even blocks of the same length as the vulnerable code snippets on average, and tagged as 'not vulnerable'. Notice how the splitting occurs initially, followed by the separation of vulnerable and non-vulnerable individuals. The issue with this method was that it regarded susceptible code areas differently than non-vulnerable code areas. The procedure of constructing a code block differed: vulnerable blocks were extracted directly from the source code, whereas clean blocks were constructed using the block-splitting algorithm. As a result, the vulnerable blocks developed some specific characteristics that the trained classifier could easily recognize.

Some of the susceptible areas were probably very long (with entire functions removed, for example) Since the majority were relatively brief (one or two lines modified), resulting in an average of medium length, thus the clean code was divided into medium-length blocks, which doubled as a proxy for their vulnerability status. When the classifier was applied to a new source code file cut into even blocks and should determine which were vulnerable, the outcome was excessively high precision and recall numbers, as well as poor performance.

3.3 Labeling

The data is tagged using information from the commit context, similar to Li et al. [18]. The bits of code that were altered or deleted in such a commit can be labeled as vulnerable, and the version after the fix, as well as all the data around the affected component, can be labeled as not susceptible. Of course, there are times when a repair fails to cure an issue, when many vulnerabilities exist at the same time, or when a new vulnerability is introduced. This strategy ignores all of it because the key goal is simple automation without the requirement for human expert oversight. Furthermore, everything marked as 'not-vulnerable' should be regarded as 'at least not demonstrated to be vulnerable'.

3.4 Representation of the Source Code

Simple techniques like a bag of words representations have previously shown unsatisfactory results and are unable to capture the semantic context of code by design. They may be promptly rejected. Others, such as Russell et al. [26] and Hovsepyan et al. [15], claim that an AST representation is required to mine patterns from code, while others, such as Liu et al. [20], argue that this is not the case. Furthermore, Dam et al. [7] claim that, in addition to human-engineered features and software metrics, ASTs may be unable to capture the semantics buried deep within source code. Code is sequential data akin to natural text, and long short-term memory networks are created specifically for modeling such data, with excellent results.

Given all of this, our technique is built to work directly on source code as text. Because code snippets are used as samples, the method could be compared to an n-gram technique, however, the snippets are far longer than those used in n-grams. To account for the code's locality aspect [34], the context surrounding each code token will be emphasized for learning features.

3.4.1 Choosing Granularity

As Morrison et al. [22] explain, binary-level predictions and analysis on the level of whole files provide little insight because developers often already know which files are vulnerable to security issues, and developers prefer, if possible, a much finer approach at the level of lines or instructions. Dam et al. [7] start their paper with some persuasive examples, suggesting that there are files with comparable metrics, structure, and even almost identical tokens, one of which may be clean and the other vulnerable, despite the same metrics. A technique that 'zooms in' to study small bits of code individually may be more promising than a top-down approach that looks at entire files. Our method employs a fine-grained approach, examining each character in the code as well as its context. Only in this way can the specific position of the vulnerability be pinpointed.

3.4.2 Preprocessing the Source Code

Tokens at the source-code level in languages like Python include identifiers, keywords, separators, operators, literals, and comments. While some researchers [25] omit separators and operators, others [37] remove a large number of tokens and keep only API nodes or function calls. Comments are removed from this work because they do not affect the program's behavior. Even if they have some predictive value for vulnerability status, this is not the type of data that should be learned by the model, which is designed to discover vulnerable code. Otherwise, the source code remains unchanged. Hovsepyan et al [15] take a similar strategy. Generic names are not used to substitute variables or literals; everything is taken exactly as it is represented. Because neural networks work with numerical vectors of uniform size, it's vital to represent code tokens as vectors that keep the semantic and syntactic information from the code. Furthermore, the vector's variables must be chosen in such a way that the vectors are manageable in size.

Li et al. [18] apply carefully constructed code gadgets, Hovsepyan et al. [15] use a simple bag-of-words strategy, Russell et al. [26] train a randomly initialized one-hot-embedding, and Liu et al. [20] use word2vec. A naive one-hot encoding is one possibility, but it is utterly oblivious to the semantic meaning of tokens. An embedding layer, on the other hand, uses vectors with high cosine similarity to represent semantically comparable code elements. A code snippet is turned into a list of representations of its tokens in our method. Language keywords, identifiers such as function names and variables, integers, operators, and even whitespaces, brackets, and indentations are examples of these. Every one of the tokens must be embedded, or represented by a numeric vector.

As a result, a complete portion of the code is converted into a vector of vectors of numbers. All the embedding layers have previously been used successfully for similar projects [20]. Aside from the conceptual advantages over a one-hot encoding, it also requires significantly smaller vector sizes, making it computationally less expensive. It was picked as the best embedding method for our strategy. Because no pre-trained language model for Python code is currently available, embedding layers must be trained first. A corpus of high-quality Python code is obtained for this purpose, once again from GitHub. The embedding layer model is trained on this corpus to prepare it for the task of encoding Python code tokens as vectors.

The vulnerable and non-vulnerable components had to be treated the same the entire way up to the labeling stage to properly analyze the data. The data was divided into equal chunks, which were then tagged as vulnerable if they overlapped with one of the vulnerable code segments, otherwise as clean. The technique of breaking down source code into blocks has been given in a simplified manner thus far. Initially, the comments are filtered out of the code, similar to the work of Hovsepyan et al. [15] and many others, because they are unlikely to alter the vulnerability of a file. A small focus window iterates over the entire source code in n-steps. To avoid tokens being split in half, the focus window always starts and stops at a character that represents the end of a token in Python, such as a colon, bracket, or whitespace. The surrounding context of roughly length m, starting and stopping at the border of code tokens, is determined for this focus window, with $m > n$. The context will largely lie behind the focus window if it is at the beginning of the file, and if it is in the middle, the surrounding context will span a snippet that spans equally before and after the focus window.

As a result, there are a lot of overlapping blocks. It is labeled as vulnerable if the entire block contains partially vulnerable code, otherwise, it is labeled as clean. This ensures that code snippets that contain a vulnerability are identified. The parameters n and m will be optimized, and their ideal values will be found through experimentation. According to Liu et al. [20], a portion of merely 10 lines of code is usually enough to capture the important context for a vulnerability. The next step is to convert those code blocks, which are simply lists of Python characters, into numerical vector lists.

3.5 Embedding Layers

A suitable embedding layer model trained on Python source code is required to encapsulate the code tokens in a numerical vector. A substantial training base of code, ideally made up of clean, working Python code, is necessary to train this model. This research follows the heuristic that popular code projects are of high quality, similar to Bhoopchand et al. [3] and Allamanis et al. [1]. It is worth noting that those repositories are likely to include minimal security flaws and defects in general. We propose our recommended embedding layers in our previous paper and test them with different hyperparameters to see the effectiveness of each of them [2].

3.6 Selecting the Machine Learning Model

Many machine learning models and methodologies have been applied to vulnerability detection, with inconsistent results, including SVMs, decision trees, random forest, and naive Bayes models. However, not all of those models are equipped with the needed features. Our technique aims to construct a model that can learn vulnerability aspects from code token sequences. Source code is sequential data by definition, as the effect of each line is highly dependent on the effects of the instructions around it. This will result in many false positives when trying to detect a vulnerability. Rather, the idea is to discover that a token is 'bad' when used in a specific way, with tokens that have come before it.

Deep learning-based models, particularly RNNs, are particularly well-suited to representing code locality while also being able to capture far more context than ngrams [6]. Deep neural networks, particularly recurrent neural networks and long short-term memory networks, have numerous advantages. To recap, such networks may describe sequential data by using an internal state as a "memory" to keep track of prior inputs and contextualize data. RNNs, on the other hand, suffer from vanishing or exploding gradients, making it difficult to train them on longer sequences since the distance between the occurrence of a piece of information and the point at which it becomes relevant exceeds the RNN's capabilities. LSTM, on the other hand, were created to cope with problems like this since they can learn how long information should be preserved. They have been effectively employed in modeling code and are designed for the type of task required in Our method. As a result, an LSTM is used as the model in this study. We decided to utilize Bi-LSTM for the final version of the tool, but because we started with LSTM, we will explain it first.

3.7 Preparing the Data for Classification

The information gathered is still in the form of code snippets that are vulnerable and not vulnerable.The snippets are translated into a list of tokens, and each token is replaced with its vector representation based on the chosen embedding layer model. Each vector has a binary label, with '0' indicating vulnerable and '1' indicating not vulnerable or unknown status. The data is divided into three sets: training, validation, and testing. 70% of the data is chosen at random as a training set, 15% is chosen as a test set for validation, and 15% is kept aside for a final evaluation after the experiments. Dam et al. [7] utilized the same ratios, Russell et al. [26] split their dataset into 80 percent training, 10% validation, and 10% final test set, and Li et al. [18] used an 80-20 split between train and test set.

It is worth noting that the validation set is not used to learn parameters; instead, it is used to assess the model's performance after it has learned its parameters on the training set. This evaluation is taken into account while adjusting the model's hyperparameters, and all findings are finally presented using the final test set, which the model has never seen before. To obtain an equal length of vectors for each sample, the lists of vectors are shortened and padded.

3.8 Training the LSTM

3.8.1 Architecture of the Model

A sequential model is built using the Keras package. The most crucial aspect comes first: the LSTM layer. The goal of this layer is to discover features that are linked to a code snippet's vulnerability state. There is no need for a separate dropout layer because the LSTM layer is susceptible to numerous hyperparameters, including dropout and recurrent dropout. The classes should be weighed appropriately because the data is inherently class imbalanced (there are far more clean code blocks in the training data than vulnerable ones). This ensures that, even though there are many more instances of clean code, the examples of vulnerable code are handled appropriately in training. The class weights are determined automatically using the scikit-learn library's class weight function. The activation layer, a dense output layer with a single neuron, follows the LSTM layer. Because the purpose is to produce a forecast between 0 and 1 for the two classes of non-vulnerable and vulnerable code, the activation function utilized here is a sigmoid activation function.

3.8.2 Selecting Hyperparameters for the LSTM

Many choices must be taken when it comes to the LSTM hyperparameters. The hyperparameters are adjusted and tested empirically to identify the ideal configurations after calculating some plausible beginning values based on other research and common sense. Technically, the metric and loss functions are hyperparameters Figure 3. Because our approach prefers a fair balance of false positives and false negatives and the classes are already weighed, the F1 metric appears to be particularly well suited to evaluate overall performance. As a result, the F1 score is chosen as the LSTM model's optimization criterion. In the scripts, the F1 metric and its accompanying loss function are custom defined. The number of neurons is, of course, a key hyperparameter that defines the model. It has an impact on learning ability. More neurons let the model learn a more complex structure, but training the model takes longer. The dimensionality of the output space is likewise determined by the number of neurons.

The batch size specifies how many samples are displayed to the network before the weights are changed again. As a result, when making a prediction later, the model should not be trained with a batch size smaller than the number of samples used at the time. To compare the outcomes, a range of various batch sizes are used. A batch size of one single sample or a batch size of the entire training set is the most extreme value. Batch sizes of 32, 64, and 128 samples are commonly employed in the middle of the two. LSTM, like many other models, can be overfitted with training data, lowering their predictive effectiveness. Dropout is a regularization strategy in which input and recurrent connections to LSTM units are occasionally randomly omitted from the next step of the training, preventing the network from updating its weights. This decreases the risk of the network overfitting by depending too heavily on a few inputs.

There are two types of dropout in LSTM: the standard dropout describes the proportion of units that are dropped from the inputs; the cheval dropout describes the fraction of units that are dropped from the inputs. The recurrent dropout is the percentage of units that leave the recurrent condition. A normal dropout rate is between 10% and 50%. Experimentation will establish the ideal dropout. Finally, the number of epochs, or the number of times the learning algorithm will run through the entire training data set, must be changed. In the literature, epochs are commonly referred to as 10, 100, 500, or even 1000. Several optimizers from the adam family will be tried to discover which produces the best results. A model with optimal configurations can be calculated when all of those hyperparameters have been modified.

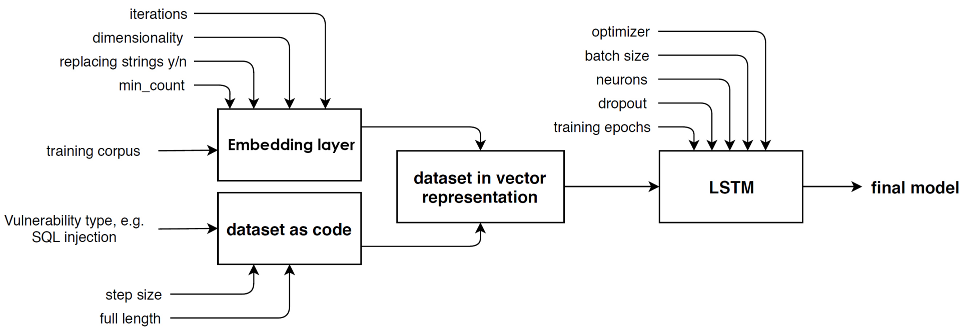


Figure 3: Creating models with different hyperparameters

3.8.3 Selecting the Optimizer

The objective for the F1 score was chosen as a criterion for the model's performance because our strategy is to attain high precision and recall at the same time. To determine how 'wrong' the forecasts are at a given point, a loss function based on the F1 score will be employed. The optimizer must update the model parameters until the global minimum is found to minimize the loss function. Simply remove the gradient of the loss concerning the weights multiplied by a modest amount called the 'learning rate' from the weights to be improved. With each iteration of the optimization, the gradient is calculated for a distinct sub-sample of the data and is thus subject to statistical fluctuation, which is why this approach is called Gradient Descent" (SGD). However, if the loss function is not convex or there are ill-conditioned regions, SGD can become stuck in a local minimum. This can be changed by lowering the learning rate. A slow learning rate, on the other hand, suggests that the network will not learn rapidly enough. What factors should be considered when determining the learning rate?

Fortunately, the learning rate does not have to be set in stone and may be dynamically adjusted. The adam optimizer dynamically selects a learning rate. It was first published in 2014 [17], and it is built primarily for deep neural networks, where it produces excellent results quickly and is frequently used as a go-to optimization approach for a variety of issues. It considers prior updates as well as the first and second moments of the gradient, which are defined as the expected value of that variable to the power of one or two, respectively - the mean and centered variance of the gradient. It combines the advantages of the Adaptive Gradient Algorithm (adagrad) [9] and Root Mean Square Propagation (RMSprop) [33], according to the authors. Adagrad adjusts the learning rate for different characteristics and performs exceptionally well on sparse datasets with a large number of missing samples. Its disadvantage is that it has a very slow learning rate. RMSprop, a variant of adagrad, adjusts learning rates based on recent gradient magnitudes for the weight and works well on both online and non-stationary issues.

When calculating momentum, it only considers gradients in a fixed window. Other optimizers, such as adadelta, nadam, adamax, NAG, and others, will not be discussed in length here. Since the chosen loss function, the F1 score is not convex, SGD will probably not converge towards the optimal solution. According to IBM, the adam family of optimizers (which includes RMSprop, adagrad, and others) should converge under certain conditions. Li et al. [18], Russell et al. [26], and Dam et al. [7] employ adamax, Russell et al. [26] use the conventional adam optimizer, and Dam et al. [7] utilize RMSprop, however, the applicability depends heavily on the dataset's peculiarities. The Adam optimizer is utilized as a starting point for our technique, and it may be empirically compared to other optimizers to see which provides the best results in practice.

3.9 Evaluation

True positives, true negatives, false positives, and false negatives are frequently the basis for evaluation when it comes to prediction and categorization. They have been referenced before, but they will be adequately clarified here. Positive and negative refer to the prediction, so a prediction of 'vulnerable' would be positive, and a prediction of 'not vulnerable' would be negative in this work. True and false refer to whether the forecast matches the actual value or an external evaluation. As a result, a false positive is a piece of clean code that the classifier incorrectly labels as vulnerable, a true positive is a vulnerability that was correctly identified, a false negative is an actual vulnerability that was not classified as such, and a true negative is a piece of code that was classified as 'not vulnerable' and is free from vulnerabilities. Precision and recall are two metrics that are directly derived from those four numbers.

The rate of genuine positives within all positives is the precision. It assesses how accurate the model is in terms of how many of the predicted positives are true positives, or, to put it another way, how much trust can be placed in the positive categorization and how many false alarms are generated. The recall, also known as sensitivity, is a metric that compares the percentage of correctly detected positives to the total number of positives. It could be interpreted as a measure of how diligently the classifier looks for all positives - or how much is missed.

When the data set is class imbalanced, meaning there are many more positives than negatives or vice versa, accuracy does not provide much insight. When it comes to vulnerability detection, the majority of code fragments will be clean, and vulnerabilities will be uncommon. For example, Morrison et al. [22] discovered that only 0.003 percent of their Windows code was vulnerable, while Shin et al. [31] found that 3% of their Firefox files were vulnerable. When genuine positives are few and true negatives are common, a classifier can attain high accuracy ratings even though it misses the majority of the positives because the many true negatives make the total result appear to be extremely accurate. As a result, the accuracy alone is insufficient for this application. The F1 score is a balanced score that considers precision and memory. The F1 score is better suitable for class-imbalanced data sets since it is less easily influenced by a large number of true negatives.

In an ideal, perfect world, the model would have a near-zero percent rate of false positives and false negatives, implying that precision and recall, as well as accuracy and F1 score, are all close to one. The accuracy, precision, recall, and F1 score will be used to evaluate the model in this study, although many previous studies on similar themes only use the first three of those four variables. Precision and recall values of 70% are feasible for prediction models, according to some studies [31], [22], however, current techniques have shown some more astonishing outcomes. Precision and recall of more than 65% seem like a good target for this project.

4 Study Results

A significant amount of contributions that addressed vulnerabilities were gathered. Each vulnerability needed its dataset after the data had been collected and filtered. The table below provides a summary of their basic information, including the number of repositories and commits that make up the dataset, the number of modified files that contain security holes, the number of lines of code, the number of distinct functions they contain, and the total number of characters. By using it to train the model, the next parts will show that this dataset is appropriate. Since some configurations must be used as a starting point, even though their hyperparameters are not optimum, they can be used to show how alternative hyperparameters lead to better or worse results. The ideal combination of all parameters can be found after going through each hyperparameter and describing how it impacts performance.

The baseline model analyzes the dataset for SQL injections using a focus region step size n of 5 and a context length m of 200. It has 30 neurons and is trained using the Adam optimizer for 10 epochs with a dropout and recurrent dropout of 20% and a batch size of 200. Even though training a model for more epochs would almost surely produce superior results, this was not possible due to the need to test numerous combinations, which would have taken more than an hour. As a result, only the resultant "best" model is trained for more epochs. The classification performance of the resulting LSTM model's F1 score, which offers a balanced score that considers precision and recall, is used to compare results. It should be noted that the same model can be trained on the same data two times, one right after the other, and the resulting scores for precision, accuracy, recall, etc. can deviate by roughly 1-3% due to the nondeterministic character of the entire process. As a result, all of the results in the following tables are only estimates and could vary somewhat.

4.1 Hyperparameters of the Embedding Layers

Are our models useful as an embedding, and how do their hyperparameters affect the overall outcomes? The tests that follow look into this. The training corpus has 69,517,343 (almost 70 million) unique tokens that were extracted from multiple Python projects. In various configurations, the hyperparameters vector length, min count, and training iterations are tested. The results of retaining strings as-is versus replacing them with generic string tokens are also compared. Since the baseline model is employed, all hyperparameters are, unless otherwise provided, selected using this default setup. In general, the method entails training an embedding layer model, using it to embed the data, and then training an LSTM model on it. Since the embedding layer itself cannot be evaluated by any kind of number, the quality of the embedding is assessed using the performance of the LSTM model. Its ability to be applied in the situation for which it was designed determines how effective it is. A poor embedding will produce a poor LSTM model that is unable to interpret the data that is given to it. However, a functional LSTM model demonstrates that the embedding layers were appropriate.

4.1.1 Vector Dimensionality

The code tokens are transformed into numerical vectors of a specific length or dimensionality when utilizing embedding layers. The more distinct "axes" there are for relating words to one another, the longer those vectors are, and the better the models can capture more intricate connections. It is doubtful that a vector with a size of under 100 can represent Python's semantics well understood, judging by comparisons to jobs involving natural language, where 200-point vector sizes are usual. The minimal count of a token to occur in the vector is used to compare different vector lengths. The models' training iterations are set at 100 and their vocabulary to 1000.

4.1.2 String Replacement

As some other researchers have done, strings found in the Python training file can either be replaced with a generic "string" token or left alone. It is difficult to predict which option will perform better in advance. Replacing them could lessen the level of detail in the model while maintaining them could focus too much on the particular content of string tokens. The embedding vectors are set to have a length of 200 to compare the two methods. The comparison is made between a min count of 10, 100, and 5000 with training iterations between 1 and 300. The Average F1 score for the embedding layers encoding that retains strings is indicated in Table 2 by the value before the slash (/), while the score for the model that substitutes a generic string token for strings is indicated by the value after the slash (/). These outcomes demonstrate that the variant without string substitution consistently produces better outcomes.

Table 2: F1 score for different min-count and iteration number w and w/o string

Minimum count	1 Iter.	10 Iter.	100 Iter.	300 Iter.
10	64% / 58%	78% / 72%	82% / 72%	84% / 74%
100	58% / 49%	75% / 66%	73% / 69%	82% / 74%
5000	50% / 49%	67% / 64%	75% / 73%	76% / 73%

4.1.3 Minimum Count

The minimum count specifies the minimum number of times a token must appear in the training corpus before a vector representation is given to it. Less frequently occurring tokens will not be encoded and will instead be skipped over later when entire lists of tokens are transformed into lists of vectors. This largely serves to ignore illegitimate identifiers such as uncommon variable names, strings, and other identifiers. To train the embedding layer model, strings are left unchanged for 100 iterations with a 200-vector training set. It could have appeared logical to believe that disregarding unusual tokens would enhance performance, but this was not the case. When tokens are seldom disregarded, the model performs better, Figure 4.

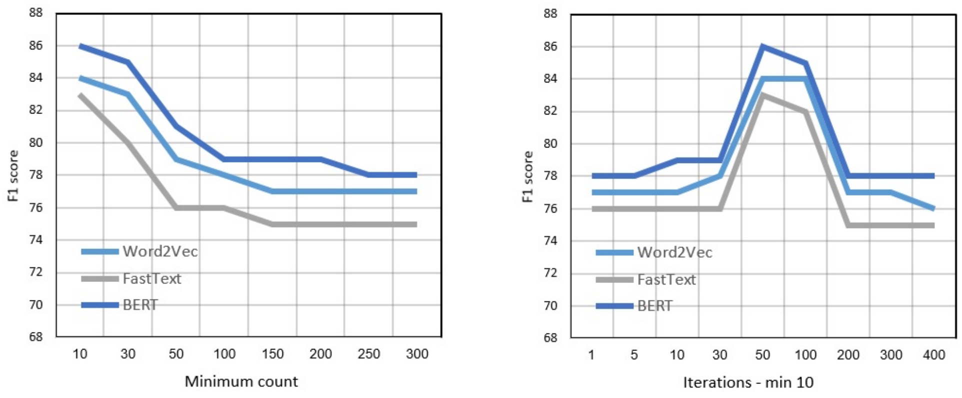


Figure 4: Iterations and minimum count in the different models

4.1.4 Iterations

The quantity of repetitions in training is determined by the number of iterations. It is reasonable to anticipate that there will be no additional advantage from additional training after a certain number of iterations. Using the same parameters as before—a corpus of original strings, a dimensionality of 200, and a min count of 10—the model is trained. Up until 50 or 100 iterations, it seems that more iterations improve the model's performance. There is no need to increase the iterations to 300 because doing so decreases rather than improves model performance and necessitates a significant increase in training time. It should be noted that the overall trend for greater performance is a smaller min count.

The LSTM model, which uses different embedding layers, performs noticeably differently depending on the hyperparameters, as shown by the tables above, with a difference between the best and worst parameters in the LSTM's F1 score of almost 25 percentage points. Therefore, careful evaluation of the hyperparameter values was not a waste of effort, as the final model's ability to learn features is influenced by the quality of the embedding. The final model will require a min count of merely 10 for tokens to be included, encode code tokens in 200-dimensional numerical vectors, not alter any strings, and be trained for 100 iterations, Figure 4.

4.2 Parameters in Creating the Dataset

The collection is made up of samples, each of which is a brief section of code built around a single token. Different step sizes n can be selected while shifting the focus point through the source code. Higher total samples and more sample overlap result from a smaller step size. The second argument, the complete length of a code sample m, determines the size of the context window surrounding the token in focus. Characters are used to measure both. The default settings are applied to all hyperparameters of the LSTM model, and the previously established ideal model is employed. Consistently lower outcomes follow bigger n. This is most likely because there will not be much overlap between the focus points' surrounding context, and the moving window that contains the code snippets if the gaps between them are wide.

A single token will appear multiple times if the emphasis shifts in very short steps because the code snippets have a lot of overlap. For example, a token can appear at the end of one snippet, in the middle of the next, and at the beginning of the one after that. This implies that there are samples that demonstrate the pertinent code with more information before and after it for each vulnerability, Making it somewhat simpler for the model to figure out which component is the real source of the vulnerability With a longer whole length m of the code snippet constituting one sample, the model performs better. Again, a bigger m results in more overlap. The drawback of this is that the prediction may become less accurate, as a significant portion of text around a token may be identified as vulnerable because it is located within a snippet of length m. However, a bigger m also has the benefit that more token context may be taken into account, which is precisely why the LSTM was initially chosen. The samples, which were already rather numerous and relatively huge in size, continued to grow for a whole length of more than 200, outpacing the machines' computing power. Moving forward, the settings for building the training set were fixed to a step length of n=5 and a full context window length of m=200.

4.3 Hyperparameters and Performance of the LSTM model

It is necessary to choose appropriate hyperparameters for the LSTM model to respond to the study question, "How effective is our technique in finding vulnerabilities as measured with accuracy, precision, and recall?" All additional LSTM hyperparameters are evaluated using the baseline model with the following values: n=5, m=200, 30 neurons, 10 epochs, dropout 20%, and Adam optimizer. The code examples are embedded using the embedding layer models with the optimal configuration predetermined.

4.3.1 Number of Neurons

The model can represent more complicated structures with a larger number of neurons, but training takes longer. In general, a model performs better with more neurons, with diminishing results beyond 50 to 70 neurons. When all other factors are held constant, the training time nearly doubles from 1 neuron to 100 neurons, then again from 100 neurons to 250 neurons. The machines the models are trained on reached their limits after more epochs and bigger datasets, sometimes terminating the operation. The optimal arrangement is therefore determined to be 100 neurons, Figure 5.

4.3.2 Batch Size

The following outcomes (Figure 5) were achieved using the baseline model with standard batch sizes (32, 64, and 128) as well as some very small and very large batch sizes: The size of the batch does not appear to have a significant impact on the model's overall performance. Only very large batch sizes of above 1000 result in performance degradation. On the other hand, the batch size had a big impact on how long it took to train the model. While training with a batch size of 5000 took 45 seconds each epoch, a batch size of 200 took 130 seconds, a batch size of 64 required 270 seconds, and the smallest practicable batch size required roughly 370 seconds. The model had to be trained for more than twenty minutes with a batch size of 10, hence the training was stopped. Conclusion: It can be said that for batch sizes less than 64, no improvement in accuracy and recall would warrant spending the additional time required for training with such little chunks of samples. From now on, a batch size of 128 will be regarded as ideal.

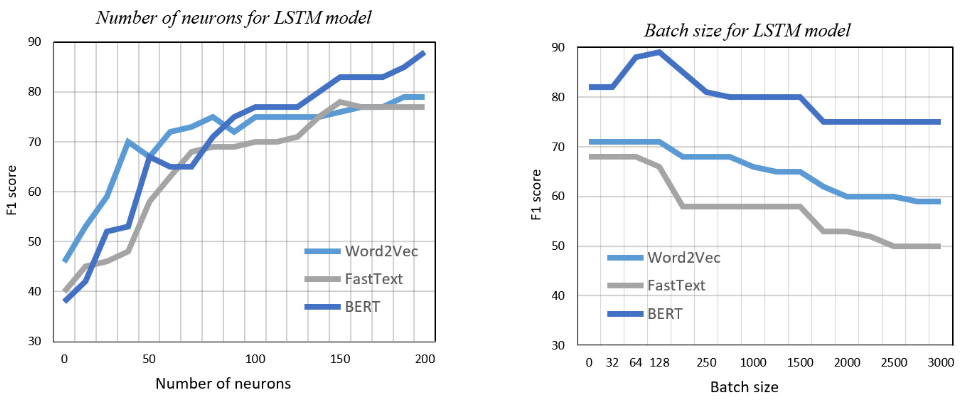


Figure 5: Hyperparameters for the LSTM model

4.3.3 Optimizer

In addition to the common Adam optimizer, the Keras model also provides similar optimizers like RMSprop and Adagrad, as well as NAdam and Adamax. To assess each person's performance, they are all put to the test. Due to their suitability for online issues, Adam, NAdam, and RMSprop appear to perform slightly better than Adagrad and Adamax. The SGD's performance is significantly worse. It takes around three hours to train each of the top three optimizers, therefore they are compared once more with 50 epochs. Adam has been selected as the preferred standard optimizer since it was a very close call. All things being equal, this optimizer is more likely to be employed in other studies, making comparisons easier.

4.3.4 Dropout

The terms dropout and repeat dropout are combined. The baseline model is trained once more but for 30 epochs this time. A fluctuation of about 2 percent points can still be accounted for by a few remaining variances in the outcome. The model functions well up to a 25% dropout. Performance gradually declines as there is a greater random loss of neurons. Therefore, setting the default dropout at 20% seems like a sensible decision, minimizing overfitting while yet allowing for adequate model performance, Figure 6.

4.3.5 Number of Training Epochs

Up to a certain point, training the model for additional epochs improves performance (Figure 6). 100 neurons were used in the model's training. Keep in mind that the performance on the validation set is used to calculate the accuracy, precision, recall, and F1 score. Additionally, the model has a 20% dropout, which should help avoid overfitting. Naturally, using more epochs lengthens the time required to train the entire model. Lengthier training sessions result in noticeable benefits.

However, beyond 100 epochs, there is not much to be gained, hence 100 epochs are selected for the model.

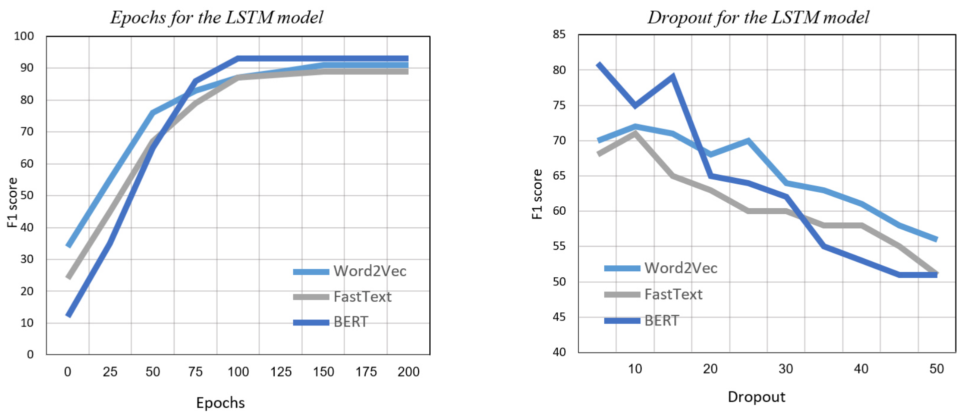


Figure 6: Hyperparameters for the LSTM model

4.3.6 Optimal Configuration

Given the dataset, limitations on processing capacity, and storage space, the recommended hyperparameter settings are:

- 100 neurons
- Training for 100 epochs
- About 20% of dropouts and repeat dropouts
- Batch size 128
- Utilizing the Adam Optimizer

These hyperparameters enable the model to be trained on all vulnerabilities for the best outcomes.

4.4 Performance for Subsets of Vulnerabilities

To respond to our study question, what categories of vulnerabilities are detectable, we looked at each vulnerability group separately. Several of the initial considerations for vulnerabilities have to be eliminated. There were relatively few results for the keywords cross-origin, buffer overflow, function injection, clickjack, eval injection, cache overflow, smurf, and denial of service, and no dataset of any size could be produced. Numerous commits that were unrelated to security vulnerabilities were produced by the keywords brute force, tampering, directory traversal, hijacking, replay attack, man-in-the-middle, format string, unauthorized, and sanitize.

A manual review of a few randomly chosen samples revealed that the majority of those commits dealt with other problems unrelated to thwarting an exploit.

Table 3: LSTM+word2vec results for each vulnerability categories

Vulnerability	Accuracy	Precision	Recall	F1
SQL Injection	92.5%	86.2%	86.0%	86.1%
XSS	91.2%	87.9%	80.8%	84.2%
Command injection	90.3%	88.0%	82.3%	84.0%
XSRF	90.1%	87.6%	84.4%	85.9%
Remote code execution	90.0%	86.0%	85.1%	85.8%
Path disclosure	89.3%	89.0%	86.4%	86.1%
Average	91.0%	88.2%	86.1%	85.6%

Therefore, it was unable to produce a high-quality dataset for those vulnerabilities. Seven vulnerabilities are left for which a dataset might be produced. The LSTM model is trained on the training sets using the determined ideal hyperparameters, with the optimizers set to minimize the F1 scores. Finally, the performance of the model is assessed, this time using the final test dataset that the models have never "seen". The findings are shown in Tables 3, 4 and 5. It seems that while the optimizer is attempting to reduce the F1 score, it is more straightforward to do so by increasing precision while the recall is a little lower. Figure 7 displays the exact meanings of the colors. In the sections that follow, one example for each vulnerability is also provided.

Table 4: LSTM+fastText results for each vulnerability categories

Vunlnerability	Accuracy	Precision	Recall	F1
SQL Injection	91.2%	82.2%	88.0%	85.1%
XSS	92.8%	83.8%	80.8%	82.2%
Command injection	91.2%	89.0%	87.3%	88.1%
XSRF	92.3%	82.7%	81.3%	81.9%
Remote code execution	90.2%	86.0%	82.8%	83.7%
Path disclosure	89.8%	82.0%	81.1%	81.5%
Average	91.8%	86.4%	85.1%	84.0%

4.4.1 SQL Injection

With 96041 samples for training and 20581 samples for testing, the data for the SQL injection vulnerability was divided into a training set and a test set. 10.9% or so of those code fragments have some susceptible code in them.

Table 5: LSTM+BERT results for each vulnerability categories

Vunlnerability	Accuracy	Precision	Recall	F1
SQL Injection	92.5%	82.2%	78.0%	80.1%
XSS	93.8%	91.9%	80.8%	86.0%
Command injection	95.8%	94.0%	87.2%	90.5%
XSRF	92.2%	92.9%	85.4%	89.0%
Remote code execution	91.1%	96.0%	82.6%	88.8%
Path disclosure	91.3%	92.0%	84.4%	88.1%
Average	93.8%	91.4%	83.2%	87.1%

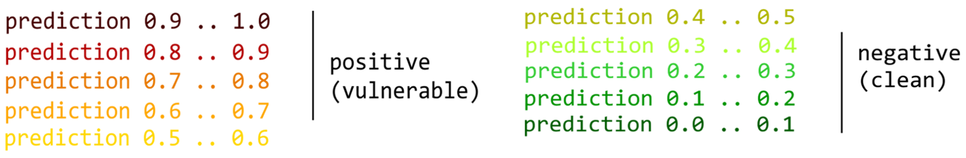
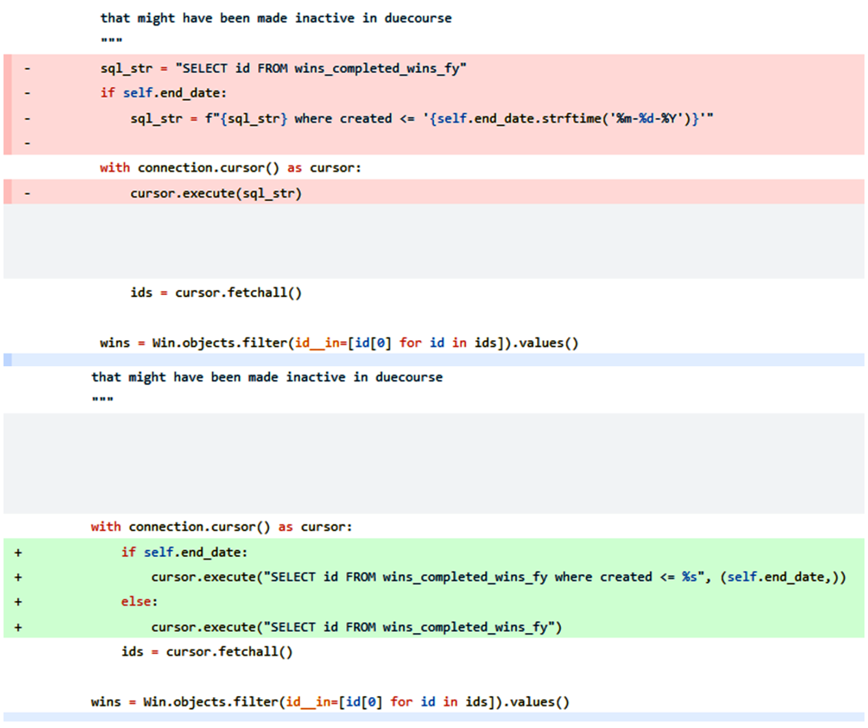


Figure 7: Color codes and confidence levels

With the aforementioned hyperparameters, the LSTM model trained for 100 iterations on the training set, yielding accuracy, precision, recall, and F1 scores of 92.5%, 82.2%, and 78.0%, 83.5% respectively within the test set. Figure 8 shows a tiny example of a SQL injection repair on GitHub. The SQL query stored in the variable SQL str, which is formed by directly concatenating other variables into a string, is executed by the instruction cursor.execute in the exposed code snippet. Figure 9 shows the detection of this vulnerability with help of our model.

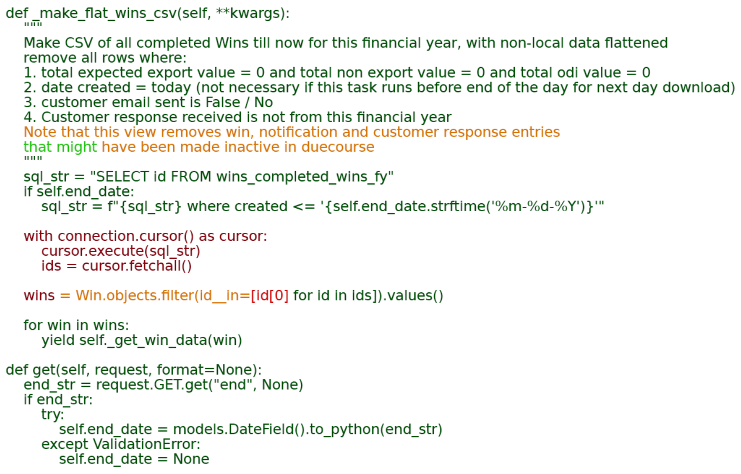
4.4.2 Cross-site Scripting

A rate of 8.9% vulnerable samples was obtained after splitting and processing the data for cross-site scripting, producing 17010 training samples and 3645 test samples. Following training on the training set, the model performed on the test set with accuracy, precision, recall, and F1 score of 97.7%, 91.9%,80.8%, and 86.0%. For an illustration of how the model finds an XSS vulnerability, see Figure 10. The variable `self.content` is used to create dynamically generated HTML code for a comment area. This code needs to be escaped to prevent script injection. Figure 11 shows the detection on the source code.



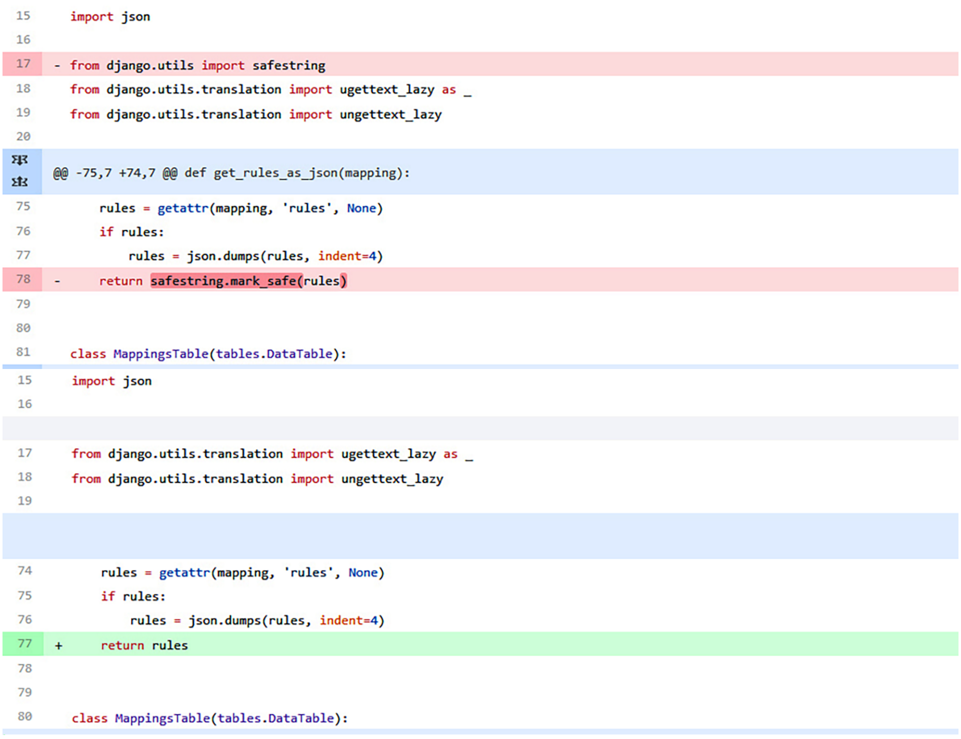
```
        that might have been made inactive in duecourse
        """
-       sql_str = "SELECT id FROM wins_completed_wins_fy"
-       if self.end_date:
-           sql_str = f"{sql_str} where created <= '{self.end_date.strftime('%m-%d-%Y')}'"
-
        with connection.cursor() as cursor:
-           cursor.execute(sql_str)
            ids = cursor.fetchall()

        wins = Win.objects.filter(id__in=[id[0] for id in ids]).values()

        that might have been made inactive in duecourse
        """

        with connection.cursor() as cursor:
+           if self.end_date:
+               cursor.execute("SELECT id FROM wins_completed_wins_fy where created <= %s", (self.end_date,))
+           else:
+               cursor.execute("SELECT id FROM wins_completed_wins_fy")
            ids = cursor.fetchall()

        wins = Win.objects.filter(id__in=[id[0] for id in ids]).values()
```

Figure 8: Commit for vulnerability (SQL injection)



```
def _make_flat_wins_csv(self, **kwargs):
    """
    Make CSV of all completed Wins till now for this financial year, with non-local data flattened
    remove all rows where:
    1. total expected export value = 0 and total non export value = 0 and total odi value = 0
    2. date created = today (not necessary if this task runs before end of the day for next day download)
    3. customer email sent is False / No
    4. Customer response received is not from this financial year
    Note that this view removes win, notification and customer response entries
    that might have been made inactive in duecourse
    """
    sql_str = "SELECT id FROM wins_completed_wins_fy"
    if self.end_date:
        sql_str = f"{sql_str} where created <= '{self.end_date.strftime('%m-%d-%Y')}'"

    with connection.cursor() as cursor:
        cursor.execute(sql_str)
        ids = cursor.fetchall()

    wins = Win.objects.filter(id__in=[id[0] for id in ids]).values()

    for win in wins:
        yield self._get_win_data(win)

def get(self, request, format=None):
    end_str = request.GET.get("end", None)
    if end_str:
        try:
            self.end_date = models.DateField().to_python(end_str)
        except ValidationError:
            self.end_date = None
```

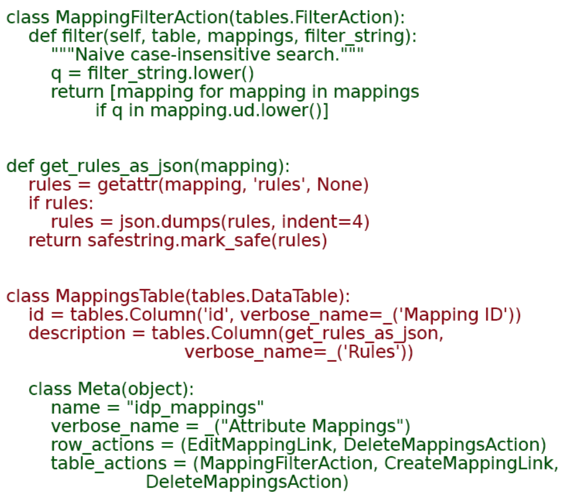
Figure 9: Detection of vulnerability (SQL injection)



```
  import json

- from django.utils import safestring
  from django.utils.translation import ugettext_lazy as _
  from django.utils.translation import ungettext_lazy

@@ -75,7 +74,7 @@ def get_rules_as_json(mapping):
      rules = getattr(mapping, 'rules', None)
      if rules:
          rules = json.dumps(rules, indent=4)
-     return safestring.mark_safe(rules)


  class MappingsTable(tables.DataTable):
```

```
  import json

  from django.utils.translation import ugettext_lazy as _
  from django.utils.translation import ungettext_lazy

      rules = getattr(mapping, 'rules', None)
      if rules:
          rules = json.dumps(rules, indent=4)
+     return rules


  class MappingsTable(tables.DataTable):
```

Figure 10: Commit for vulnerability (Cross-site scripting)

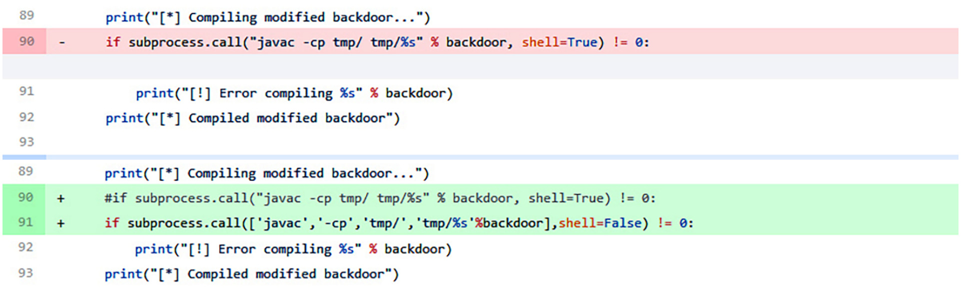


```
class MappingFilterAction(tables.FilterAction):
    def filter(self, table, mappings, filter_string):
        """Naive case-insensitive search."""
        q = filter_string.lower()
        return [mapping for mapping in mappings
                if q in mapping.ud.lower()]


def get_rules_as_json(mapping):
    rules = getattr(mapping, 'rules', None)
    if rules:
        rules = json.dumps(rules, indent=4)
    return safestring.mark_safe(rules)


class MappingsTable(tables.DataTable):
    id = tables.Column('id', verbose_name=_('Mapping ID'))
    description = tables.Column(get_rules_as_json,
                                verbose_name=_('Rules'))

    class Meta(object):
        name = "idp_mappings"
        verbose_name = _("Attribute Mappings")
        row_actions = (EditMappingLink, DeleteMappingsAction)
        table_actions = (MappingFilterAction, CreateMappingLink,
                         DeleteMappingsAction)
```

Figure 11: Detection of vulnerability (Cross-site scripting)

4.4.3 Command Injection

The accuracy, precision, recall, and F1 score of the command injection model's performance on the test set were 97.8%, 94.0%, 87.2%, and 90.5%, respectively. With a rate of 4.6% samples containing a vulnerability, 51763 training samples, and 11073 test samples were generated from the dataset. One illustration can be found in Figure 12. Here is an example of some code that uses `subprocess.call` to run the Java compiler when given a command with the detection part in Figure 13. Extra items can be handled as additional arguments to the shell because the command is passed to it as a string and with the option "shell=True," which enables the injection of other commands.



```
        print("[*] Compiling modified backdoor...")
-       if subprocess.call("javac -cp tmp/ tmp/%s" % backdoor, shell=True) != 0:

            print("[!] Error compiling %s" % backdoor)
        print("[*] Compiled modified backdoor")

        print("[*] Compiling modified backdoor...")
+       #if subprocess.call("javac -cp tmp/ tmp/%s" % backdoor, shell=True) != 0:
+       if subprocess.call(['javac','-cp','tmp/','tmp/%s'%backdoor],shell=False) != 0:
            print("[!] Error compiling %s" % backdoor)
        print("[*] Compiled modified backdoor")
```

Figure 12: Commit for vulnerability (Command injection)

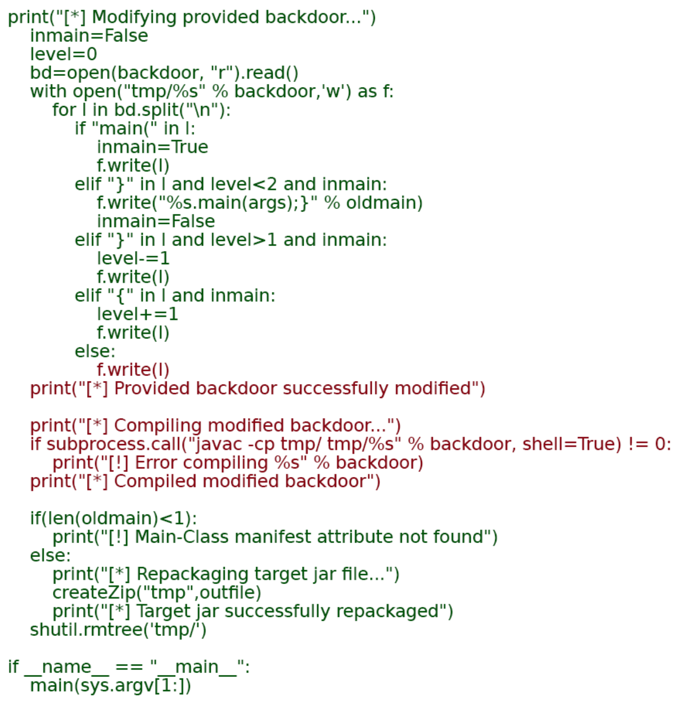
4.4.4 Cross-site Request Forgery

68434 training samples and 14665 test samples were used to process the data, and 5.9% of the samples contained susceptible code. The model performed quite well on the test data set for XSRF, achieving an accuracy of 97.2%, a precision of 92.9%, a recall of 85.4%, and an F1 score of 89.0%. Figure 14 shows an example of an XSRF vulnerability and Figure 15 shows the detection done by our approach. In this instance, an XSRF attack prevention check for proper XSRF cookies was merely absent.

4.4.5 Remote Code Execution

There were 9797 test samples and 45723 training samples in the data for remote code. 5.3% or so of the samples were vulnerable, the remainder were uncontaminated. The model was performed on the final test set with an accuracy of 98.1%, a precision of 96.0%, a recall of 82.6%, and an F1 score of 88.8% after being trained on the training set. Similar to the previous vulnerabilities, a specific illustration is provided here. Figure 16 illustrates a situation in which a command created by concatenating strings is executed using a call to `os.system`.

It is preferable to give the command as a sequence rather since just the first element of the sequence will be considered as a program to run. Figure 17 shows



```
print("[*] Modifying provided backdoor...")
    inmain=False
    level=0
    bd=open(backdoor, "r").read()
    with open("tmp/%s" % backdoor,'w') as f:
        for l in bd.split("\n"):
            if "main(" in l:
                inmain=True
                f.write(l)
            elif "}" in l and level<2 and inmain:
                f.write("%s.main(args);}" % oldmain)
                inmain=False
            elif "}" in l and level>1 and inmain:
                level-=1
                f.write(l)
            elif "{" in l and inmain:
                level+=1
                f.write(l)
            else:
                f.write(l)
    print("[*] Provided backdoor successfully modified")

    print("[*] Compiling modified backdoor...")
    if subprocess.call("javac -cp tmp/ tmp/%s" % backdoor, shell=True) != 0:
        print("[!] Error compiling %s" % backdoor)
    print("[*] Compiled modified backdoor")

    if(len(oldmain)<1):
        print("[!] Main-Class manifest attribute not found")
    else:
        print("[*] Repackaging target jar file...")
        createZip("tmp",outfile)
        print("[*] Target jar successfully repackaged")
    shutil.rmtree('tmp/')

if __name__ == "__main__":
    main(sys.argv[1:])
```

Figure 13: Detection of vulnerability (Command injection)

the detection of this vulnerability with help of our model.

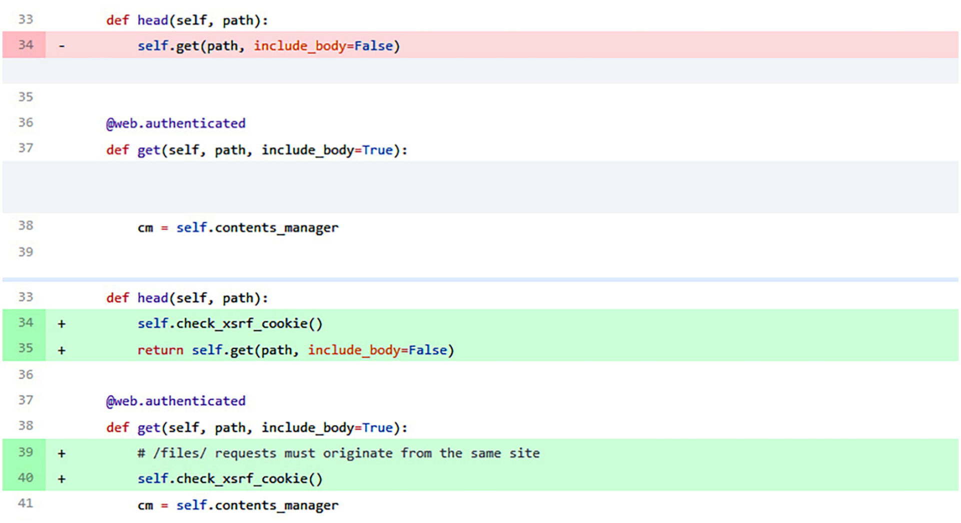
4.4.6 Path Disclosure

With 11802 test samples and 55072 training samples, this vulnerability had a rate of 7.13% vulnerable samples. The model's performance on the test set was 97.3% accurate, 92.0% precise, 84.4% recall, and 88.0% overall F1 score. An example is shown in Figure 18 and the detection example in Figure 19. Using the `commonprefix` function to determine whether the requested path is located inside the web root directory prevented a path disclosure in the example.

4.4.7 Open Redirect

There were 38189 training samples and 8184 test samples after the data had been processed. 6.4% of the samples have a vulnerability in them. An accuracy of 96.8%, a precision of 91.0%, a recall of 83.9%, and an F1 score of 87.3% were attained for this last vulnerability.

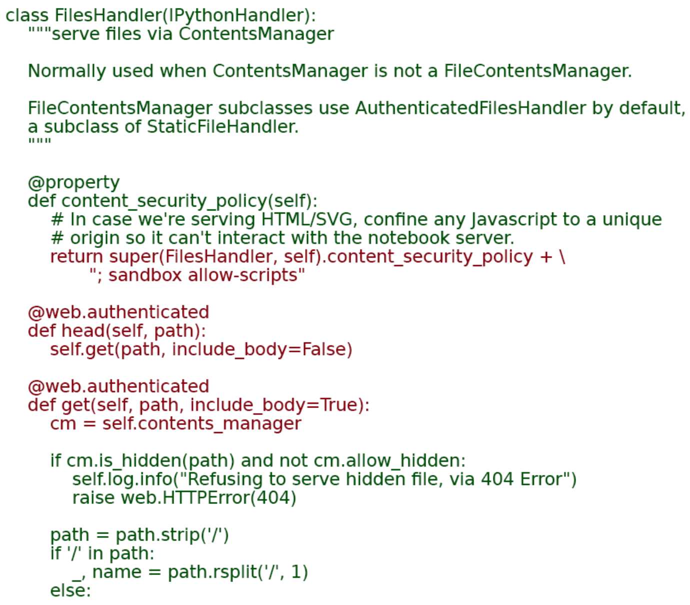
Figure 20 shows a common and simple case in which the session's next URL is requested without being sanitized, allowing untrusted URL strings to contain redi-



```
    def head(self, path):
-       self.get(path, include_body=False)


    @web.authenticated
    def get(self, path, include_body=True):


        cm = self.contents_manager

```

```
    def head(self, path):
+       self.check_xsrf_cookie()
+       return self.get(path, include_body=False)

    @web.authenticated
    def get(self, path, include_body=True):
+       # /files/ requests must originate from the same site
+       self.check_xsrf_cookie()
        cm = self.contents_manager
```

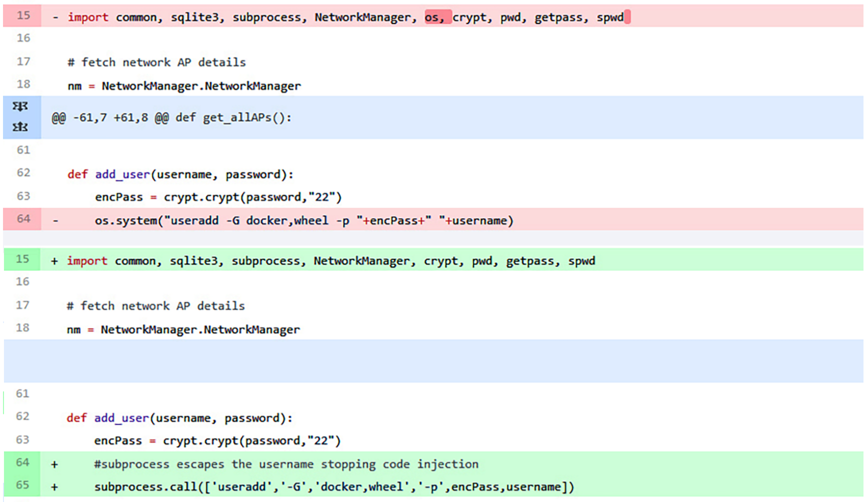
Figure 14: Commit for vulnerability (Cross-site request forgery)



```
class FilesHandler(IPythonHandler):
    """serve files via ContentsManager

    Normally used when ContentsManager is not a FileContentsManager.

    FileContentsManager subclasses use AuthenticatedFilesHandler by default,
    a subclass of StaticFileHandler.
    """

    @property
    def content_security_policy(self):
        # In case we're serving HTML/SVG, confine any Javascript to a unique
        # origin so it can't interact with the notebook server.
        return super(FilesHandler, self).content_security_policy + \
               "; sandbox allow-scripts"

    @web.authenticated
    def head(self, path):
        self.get(path, include_body=False)

    @web.authenticated
    def get(self, path, include_body=True):
        cm = self.contents_manager

        if cm.is_hidden(path) and not cm.allow_hidden:
            self.log.info("Refusing to serve hidden file, via 404 Error")
            raise web.HTTPError(404)

        path = path.strip('/')
        if '/' in path:
            _, name = path.rsplit('/', 1)
        else:
```

Figure 15: Detection of vulnerability (Cross-site request forgery)

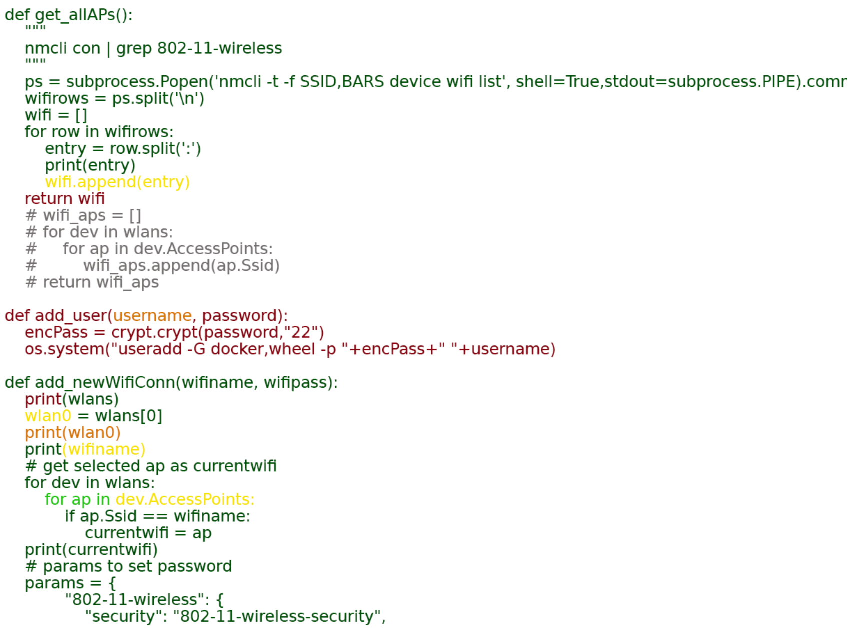
rect parameters that route users to pages other than the ones they were supposed to see and detection sample in Figure 21.



```
- import common, sqlite3, subprocess, NetworkManager, os, crypt, pwd, getpass, spwd

  # fetch network AP details
  nm = NetworkManager.NetworkManager
@@ -61,7 +61,8 @@ def get_allAPs():

  def add_user(username, password):
      encPass = crypt.crypt(password,"22")
-     os.system("useradd -G docker,wheel -p "+encPass+" "+username)
```

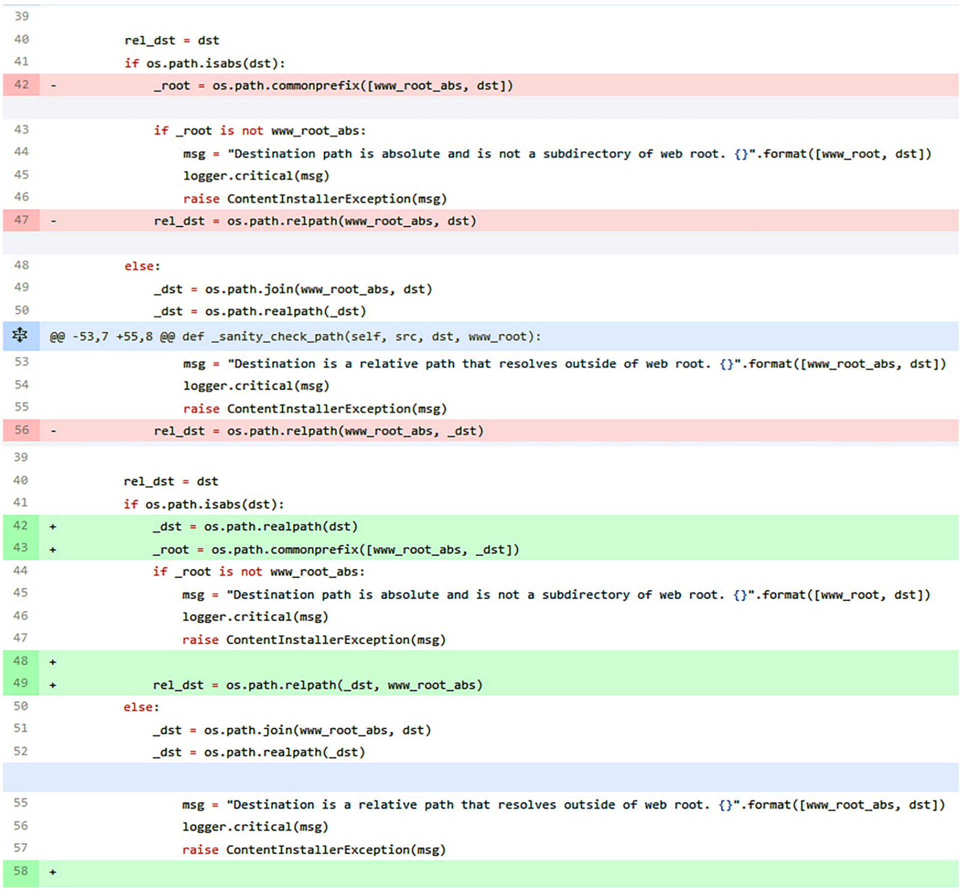
```
+ import common, sqlite3, subprocess, NetworkManager, crypt, pwd, getpass, spwd

  # fetch network AP details
  nm = NetworkManager.NetworkManager


  def add_user(username, password):
      encPass = crypt.crypt(password,"22")
+     #subprocess escapes the username stopping code injection
+     subprocess.call(['useradd','-G','docker,wheel','-p',encPass,username])
```

Figure 16: Commit for vulnerability (Remote code execution)



```
def get_allAPs():
    """
    nmcli con | grep 802-11-wireless
    """
    ps = subprocess.Popen('nmcli -t -f SSID,BARS device wifi list', shell=True,stdout=subprocess.PIPE).comr
    wifirows = ps.split('\n')
    wifi = []
    for row in wifirows:
        entry = row.split(':')
        print(entry)
        wifi.append(entry)
    return wifi
    # wifi_aps = []
    # for dev in wlans:
    #     for ap in dev.AccessPoints:
    #         wifi_aps.append(ap.Ssid)
    # return wifi_aps

def add_user(username, password):
    encPass = crypt.crypt(password,"22")
    os.system("useradd -G docker,wheel -p "+encPass+" "+username)

def add_newWifiConn(wifiname, wifipass):
    print(wlans)
    wlan0 = wlans[0]
    print(wlan0)
    print(wifiname)
    # get selected ap as currentwifi
    for dev in wlans:
        for ap in dev.AccessPoints:
            if ap.Ssid == wifiname:
                currentwifi = ap
    print(currentwifi)
    # params to set password
    params = {
            "802-11-wireless": {
                "security": "802-11-wireless-security",
```

Figure 17: Detection of vulnerability (Remote code execution)



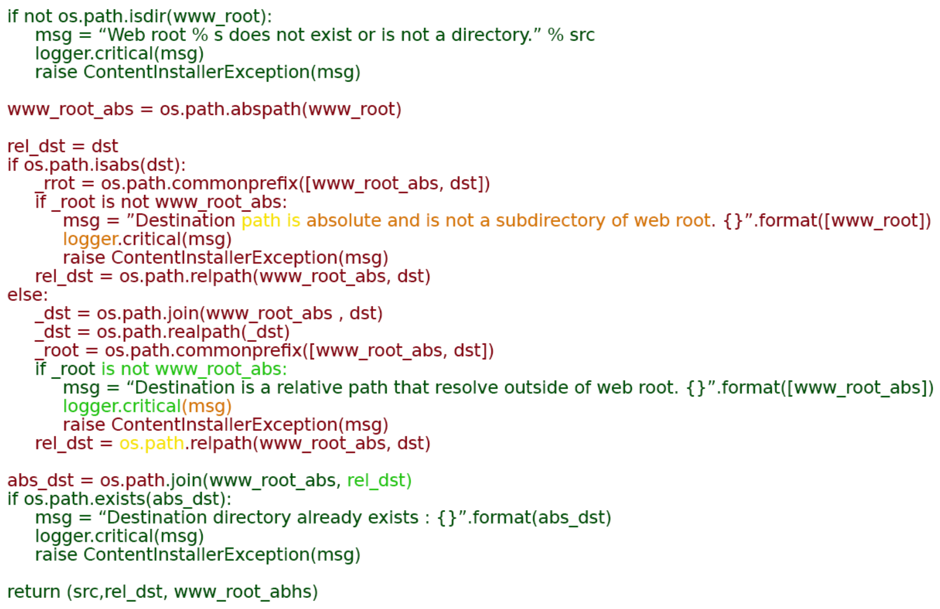
```
        rel_dst = dst
        if os.path.isabs(dst):
-           _root = os.path.commonprefix([www_root_abs, dst])

            if _root is not www_root_abs:
                msg = "Destination path is absolute and is not a subdirectory of web root. {}".format([www_root, dst])
                logger.critical(msg)
                raise ContentInstallerException(msg)
-           rel_dst = os.path.relpath(www_root_abs, dst)

        else:
            _dst = os.path.join(www_root_abs, dst)
            _dst = os.path.realpath(_dst)
@@ -53,7 +55,8 @@ def _sanity_check_path(self, src, dst, www_root):
                msg = "Destination is a relative path that resolves outside of web root. {}".format([www_root_abs, dst])
                logger.critical(msg)
                raise ContentInstallerException(msg)
-           rel_dst = os.path.relpath(www_root_abs, _dst)

        rel_dst = dst
        if os.path.isabs(dst):
+           _dst = os.path.realpath(dst)
+           _root = os.path.commonprefix([www_root_abs, _dst])
            if _root is not www_root_abs:
                msg = "Destination path is absolute and is not a subdirectory of web root. {}".format([www_root, dst])
                logger.critical(msg)
                raise ContentInstallerException(msg)
+
+           rel_dst = os.path.relpath(_dst, www_root_abs)
        else:
            _dst = os.path.join(www_root_abs, dst)
            _dst = os.path.realpath(_dst)

                msg = "Destination is a relative path that resolves outside of web root. {}".format([www_root_abs, dst])
                logger.critical(msg)
                raise ContentInstallerException(msg)
+
```

Figure 18: Commit for vulnerability (Path disclosure)

4.5 Application on Source Code

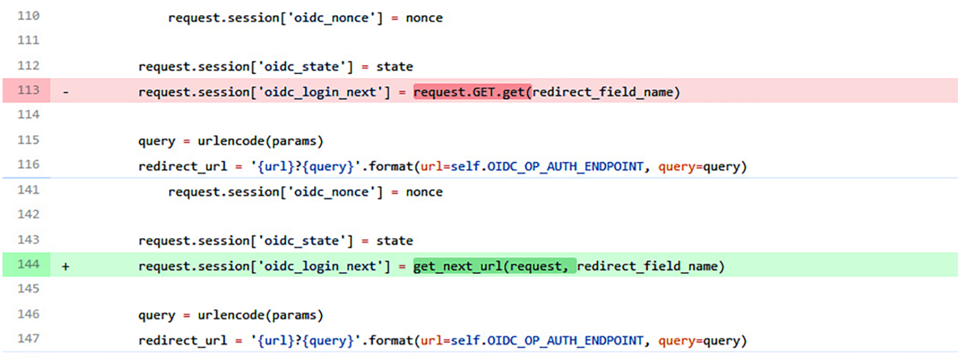
Our approach expands the work in the area of vulnerable code pattern analysis. A large dataset of source code written in Python is collected from Github, filtered, preprocessed, and labeled based on the information from commits. Several different types of vulnerabilities are taken into consideration, and source code from many different projects is collected. The resulting dataset of natural code containing vulnerabilities is made available for further research. Samples are generated by dividing the code into overlapping snippets that capture the immediate context of some tokens. The samples are embedded in numerical vectors using different embedding layers.

A long short-term memory network is trained to extract features and then applied to classify code that was not used in training, highlighting the exact locations within the code that are potentially vulnerable. We combine all of the



```
if not os.path.isdir(www_root):
    msg = “Web root % s does not exist or is not a directory.” % src
    logger.critical(msg)
    raise ContentInstallerException(msg)

www_root_abs = os.path.abspath(www_root)

rel_dst = dst
if os.path.isabs(dst):
    _rrot = os.path.commonprefix([www_root_abs, dst])
    if _root is not www_root_abs:
        msg = ”Destination path is absolute and is not a subdirectory of web root. {}”.format([www_root])
        logger.critical(msg)
        raise ContentInstallerException(msg)
    rel_dst = os.path.relpath(www_root_abs, dst)
else:
    _dst = os.path.join(www_root_abs , dst)
    _dst = os.path.realpath(_dst)
    _root = os.path.commonprefix([www_root_abs, dst])
    if _root is not www_root_abs:
        msg = “Destination is a relative path that resolve outside of web root. {}”.format([www_root_abs])
        logger.critical(msg)
        raise ContentInstallerException(msg)
    rel_dst = os.path.relpath(www_root_abs, dst)

abs_dst = os.path.join(www_root_abs, rel_dst)
if os.path.exists(abs_dst):
    msg = “Destination directory already exists : {}”.format(abs_dst)
    logger.critical(msg)
    raise ContentInstallerException(msg)

return (src,rel_dst, www_root_abhs)
```

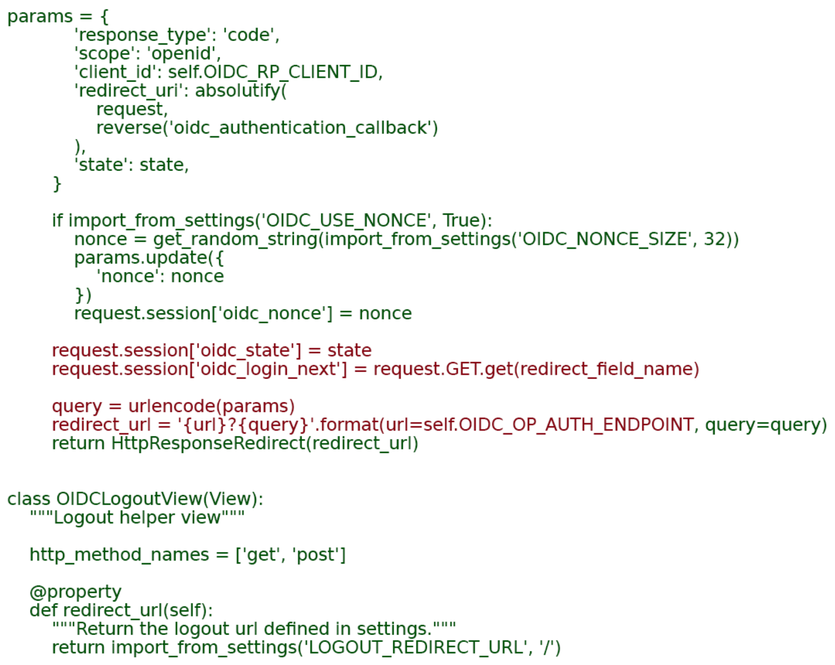
Figure 19: Detection of vulnerability (Path disclosure)



```
            request.session['oidc_nonce'] = nonce

        request.session['oidc_state'] = state
-       request.session['oidc_login_next'] = request.GET.get(redirect_field_name)

        query = urlencode(params)
        redirect_url = '{url}?{query}'.format(url=self.OIDC_OP_AUTH_ENDPOINT, query=query)
            request.session['oidc_nonce'] = nonce

        request.session['oidc_state'] = state
+       request.session['oidc_login_next'] = get_next_url(request, redirect_field_name)

        query = urlencode(params)
        redirect_url = '{url}?{query}'.format(url=self.OIDC_OP_AUTH_ENDPOINT, query=query)
```

Figure 20: Commit for vulnerability (Open redirect)

trained models into a single, straightforward text editor, called VulDetective, and test them using a variety of features, including which embedding layer and which vulnerabilities they are vulnerable to. Additionally, the tool displays the content color coded, Figure 22, including gray for comments, green for not vulnerable, and red for vulnerable. We aim to keep it as straightforward as we can because the tool's goal is better detection; as a result, we spend a lot of time training various models and experimenting with various embedding layers and hyperparameters.

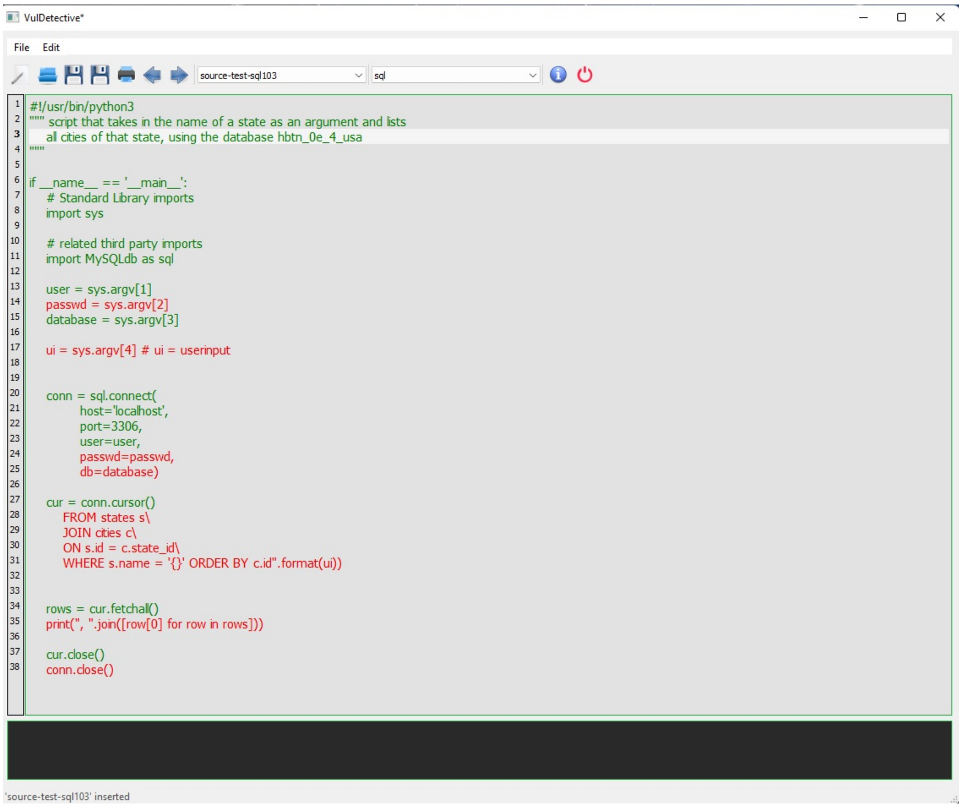


```
params = {
        'response_type': 'code',
        'scope': 'openid',
        'client_id': self.OIDC_RP_CLIENT_ID,
        'redirect_uri': absolutify(
            request,
            reverse('oidc_authentication_callback')
        ),
        'state': state,
    }

    if import_from_settings('OIDC_USE_NONCE', True):
        nonce = get_random_string(import_from_settings('OIDC_NONCE_SIZE', 32))
        params.update({
            'nonce': nonce
        })
        request.session['oidc_nonce'] = nonce

    request.session['oidc_state'] = state
    request.session['oidc_login_next'] = request.GET.get(redirect_field_name)

    query = urlencode(params)
    redirect_url = '{url}?{query}'.format(url=self.OIDC_OP_AUTH_ENDPOINT, query=query)
    return HttpResponseRedirect(redirect_url)


class OIDCLogoutView(View):
    """Logout helper view"""

    http_method_names = ['get', 'post']

    @property
    def redirect_url(self):
        """Return the logout url defined in settings."""
        return import_from_settings('LOGOUT_REDIRECT_URL', '/')
```

Figure 21: Detection of vulnerability (Open redirect)

The application we have created differs in some respects from all other previous work. Unlike the approaches of Li et al. [18], Pang et al. [25], Hovsepyan et al. [15], and Dam et al. [7], it uses a broad code base rather than a select number of projects. The predictions are not only applicable within the same file or project, but can be generalized to any other source code. In contrast to these four works, a fine granularity is also chosen. The aforementioned works all classify entire files or, as in the case of Li et al. [18], consider only API and function calls. Our approach is more in line with the work of Russell et al. [26] and Ma et al. [21] in that vulnerabilities are detected at specific locations in the code rather than just at the file level, which is likely to be more useful to developers; different tokens can even be color-coded depending on the confidence level of the classification. Similar to the research of Hovsepyan et al. [15] and in contrast to the work of Ma et al. [21], Yamaguchi et al. [37], and Liu et al. [20], this work does not convert the source code into a structure such as an abstract syntax tree but assumes that it is plain text. It follows the natural hypothesis and aims to use as few assumptions as possible, leaving the extraction of features from the source code entirely to the trained model.

The labels for the dataset are not generated using a static analysis tool, as is the case in the work of Russel et al. [26], Dam et al. [7], and Hovsepyan et al. [15]. The basic idea of our approach is independence from manually designed features,



```
#!/usr/bin/python3
""" script that takes in the name of a state as an argument and lists
    all cities of that state, using the database hbtn_0e_4_usa
"""

if __name__ == '__main__':
    # Standard Library imports
    import sys

    # related third party imports
    import MySQLdb as sql

    user = sys.argv[1]
    passwd = sys.argv[2]
    database = sys.argv[3]

    ui = sys.argv[4] # ui = userinput


    conn = sql.connect(
            host='localhost',
            port=3306,
            user=user,
            passwd=passwd,
            db=database)

    cur = conn.cursor()
        FROM states s\
        JOIN cities c\
        ON s.id = c.state_id\
        WHERE s.name = '{}' ORDER BY c.id".format(ui))


    rows = cur.fetchall()
    print(", ".join([row[0] for row in rows]))

    cur.close()
    conn.close()
```

'source-test-sql103' inserted

Figure 22: Overview of the VulDetective application

which is the major limitation of previous static analysis tools. The goal is not to model an existing static tool, but to learn features without initial assumptions. Therefore, it is based on a similar assumption as Liu et al. [20], namely that code that has been patched or patched was most likely vulnerable before the fix. The flag is based solely on the Github commits, which (at least in theory) allows the discovery of vulnerability patterns that have not yet been manually included in static analysis tools. The dataset used as a basis for training consists of natural code from real software projects, rather than synthetic databases designed to provide clear examples of vulnerabilities.

This makes the whole task more difficult, as real code is much messier and less clean than synthetic code. In this respect, our method differs from the approaches of Russell et al. [26] and Li et al. [18]. However, this also makes our approach independent of specific projects with their characteristics and therefore robust to some degree to the threats to validity that would arise from a narrower approach. The machine learning model used is an LSTM and Bi-LSTM, as also used by

Li et al. [18] and Dam et al. [7]. Compared to the latter, the architecture and preprocessing of the data in our approach are much simpler. Many other approaches use either different deep learning models (CNNs and RNNs in the case of the work by Russell et al. [26]) or completely different machine learning approaches (support vector machines in the case of the work by Pang et al. [25]).

To conclude the list of contributions: The focus is on code written in Python, unlike most other research projects that are primarily concerned with Java, C, C++, or PHP. No other approach has been found that uses even remotely similar techniques and works with Python. Of course, the proposed approach could be applied to other languages as well. The various embedding layer models that have been trained for Python are another contribution to this work.

4.6 Result Comparison with Other Works

To give a framework for the assessment of this study, Table 6 and 7 includes comparisons with related research in the field. Each approach has inherent variances, hence it is difficult to directly compare them. Approaches are compared under the following aspects:

- Language: what language is subject of the classification efforts
- Data: does the data stem from real-life projects or synthetic databases
- Labels: how are the labels for the training data originally generated
- Granularity: is the code evaluated on a rough granularity (whole classes or files) or a fine granularity (lines or tokens)
- Method: what class of neural network or machine learning approach is used (CNN, RNN, LSTM)

5 Conclusion

This paper presents a vulnerability detection method based on deep learning on source code. Its purpose is to relieve human vulnerability detection experts of the time-consuming and subjective effort of manually defining vulnerability detection criteria. Via LSTM models, this research demonstrates the feasibility of learning vulnerability attributes straight from source code using machine learning. It can detect seven different types of errors in Python source code. We were able to identify specific sections of code that are likely to be vulnerable, as well as provide confidence levels for our predictions. We get an accuracy of 93.8%, a recall of 83.2%, a precision of 91.4%, and an F1 score of 87.1% on average. We also demonstrate how the trained model can be applied in practice, therefore opening up the possibility of building a hands-on developer tool for detecting vulnerable code blocks in arbitrary Python programs. Moreover, the presented method is language agnostic, it can be adapted to other languages as well. Higher measurements in precision, recall, and

Table 6: Comparisons with related researches

Name	Lang.	Data	Labels	Scope	Gran.	Method
Russel et al. [26]	C/C++	real and synth.	static tool	general	token level	CNN RNN
Pang et al. [25]	Java	real	pre existing	4 apps	whole classes	SVM
VuRLE [21]	Java	real	manually	general	edits (fine)	10-fold CV
VulDeePecker [18]	C/C++	real and synth.	patches and manual	general	API function calls	BLSTM
Dam et al. [7]	Java	real	static tool	18 apps	whole file	LSTM
Hovsepyan et al. [15]	Java	real	static tool	1 project	whole file	grid search
Bagheri et al.	Python	real	patches	general	token level	LSTM

Table 7: Comparisons with related researches

Name	Accuracy	Precision	Recall	F1
Russel et al.	-	-	-	57%
Pang et al.	63%	67%	63%	65%
VuRLE	-	65%	66%	65%
VulDeePecker	-	-	-	85%-95%
Dam et al.	81%	82%	76%	80%
Hovsepyan et al.	87%	85%	88%	85%
Bagheri et al.	93%	91%	83%	87%

F1 are a lot simpler to accomplish if the methodology centers around forecasts inside a solitary task, as Hovsepyan et al. [15] and Dam et al. [7] do when they train a classifier to predict vulnerabilities inside the same application. Preparing a classifier that is relevant for general recognition of vulnerabilities is a lot harder - yet additionally prompts a substantially more valuable final product. Note that similar two methodologies, as well as the one taken by Pang et al. [25], are likewise simply attempting to predict regardless of whether an entire record is defenseless without having the option to bring up the specific area of the vulnerability. Since it expects to foster an overall vulnerability identifier that can be utilized at the fine granularity of code tokens, it has a significantly more confounded undertaking to satisfy. With basically a similar methodology, Russel et al. [26] accomplished 56% on regular code from Github, yet 84% on the Satisfy test suite because of its spotless and predictable structure and design. our methodology seemingly performs all around given that it works absolutely on normal real-life source code.

References

[1] Allamanis, M. and Sutton, C. Mining source code repositories at massive scale using language modeling. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 207–216. IEEE, 2013. DOI: 10.1109/MSR.2013.6624029.

[2] Amirreza, B. and Hegedűs, P. A Comparison of Different Source Code Representation Methods for Vulnerability Prediction in Python. In *Proceedings of the 14th International Conference on the Quality of Information and Communications Technology (QUATIC 2021)*, 2021. DOI: 10.48550/arXiv.2108.02044.

[3] Bhoopchand, A., Rocktäschel, T., Barr, E., and Riedel, S. Learning Python code suggestion with a sparse pointer network. *arXiv preprint arXiv:1611.08307*, 2016. DOI: 10.48550/arXiv.1611.08307.

[4] Chowdhury, I. and Zulkernine, M. Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities. *Journal of Systems Architecture*, 57(3):294–313, 2011. DOI: 10.1016/j.sysarc.2010.06.003.

[5] Church, K. W. Word2vec. *Natural Language Engineering*, 23(1):155–162, 2017. DOI: 10.1017/S1351324916000334.

[6] Dam, H. K., Tran, T., Grundy, J., and Ghose, A. Deepsoft: A vision for a deep model of software. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 944–947, 2016. DOI: 10.1145/2950290.2983985.

[7] Dam, H. K., Tran, T., Pham, T., Ng, S. W., Grundy, J., and Ghose, A. Automatic feature learning for vulnerability prediction. *arXiv preprint arXiv:1708.02368*, 2017. DOI: 10.48550/arXiv.1708.02368.

[8] Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018. DOI: 10.48550/arXiv.1810.04805.

[9] Duchi, J., Hazan, E., and Singer, Y. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(7), 2011. DOI: 10.1109/TSMC.2021.3097714.

[10] Ghaffarian, S. M. and Shahriari, H. R. Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey. *ACM Computing Surveys (CSUR)*, 50(4):1–36, 2017. DOI: 10.1145/3092566.

[11] Grieco, G., Grinblat, G. L., Uzal, L., Rawat, S., Feist, J., and Mounier, L. Toward large-scale vulnerability discovery using machine learning. In *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, pages 85–96, 2016. DOI: 10.1145/2857705.2857720.

[12] Gupta, R., Pal, S., Kanade, A., and Shevade, S. Deepfix: Fixing common C language errors by deep learning. In *Thirty-First AAAI Conference on Artificial Intelligence*, 2017. DOI: 10.1609/aaai.v31i1.10742.

[13] Hall, T., Beecham, S., Bowes, D., Gray, D., and Counsell, S. A systematic literature review on fault prediction performance in software engineering. *IEEE Transactions on Software Engineering*, 38(6):1276–1304, 2011. DOI: 10.1109/TSE.2011.103.

[14] Harer, J., Ozdemir, O., Lazovich, T., Reale, C., Russell, R., Kim, L., et al. Learning to repair software vulnerabilities with generative adversarial networks. In Bengio, S., Wallach, H., Larochelle, H., Grauman, K., Cesa-Bianchi, N., and Garnett, R., editors, *Advances in Neural Information Processing Systems*, Volume 31, 2018. URL: https://proceedings.neurips.cc/paper_files/paper/2018/file/68abef8ee1ac9b664a90b0bbaff4f770-Paper.pdf.

[15] Hovsepyan, A., Scandariato, R., Joosen, W., and Walden, J. Software vulnerability prediction using text analysis techniques. In *Proceedings of the 4th International Workshop on Security Measurements and Metrics*, pages 7–10, 2012. DOI: 10.1145/2372225.2372230.

[16] Joulin, A., Grave, E., Bojanowski, P., Douze, M., Jégou, H., and Mikolov, T. Fasttext.zip: Compressing text classification models. *arXiv preprint arXiv:1612.03651*, 2016. DOI: 10.48550/arXiv.1612.03651.

[17] Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014. DOI: 10.48550/arXiv.1412.6980.

[18] Li, Z., Zou, D., Xu, S., Ou, X., Jin, H., Wang, S., Deng, Z., and Zhong, Y. Vuldeepecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681*, 2018. DOI: 10.48550/arXiv.1801.01681.

[19] Li, Z. and Zhou, Y. PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software code. *ACM SIGSOFT Software Engineering Notes*, 30(5):306–315, 2005. DOI: 10.1145/1095430.1081755.

[20] Liu, K., Kim, D., Bissyandé, T. F., Yoo, S., and Le Traon, Y. Mining fix patterns for findbugs violations. *IEEE Transactions on Software Engineering*, 47(1):165–188, 2018. DOI: 10.1109/TSE.2018.2884955.

[21] Ma, S., Thung, F., Lo, D., Sun, C., and Deng, R. H. Vurle: Automatic vulnerability detection and repair by learning from examples. In *European Symposium on Research in Computer Security*, pages 229–246. Springer, 2017. DOI: 10.1007/978-3-319-66399-9_13.

[22] Morrison, P., Herzig, K., Murphy, B., and Williams, L. Challenges with applying vulnerability prediction models. In *Proceedings of the 2015 Symposium and Bootcamp on the Science of Security*, pages 1–9, 2015. DOI: 10.1145/2746194.2746198.

[23] Nagappan, N., Murphy, B., and Basili, V. The influence of organizational structure on software quality. In *2008 ACM/IEEE 30th International Conference on Software Engineering*, pages 521–530. IEEE, 2008. DOI: 10.1145/1368088.1368160.

[24] Neuhaus, S., Zimmermann, T., Holler, C., and Zeller, A. Predicting vulnerable software components. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, pages 529–540, 2007. DOI: 10.1145/1315245.1315311.

[25] Pang, Y., Xue, X., and Namin, A. S. Predicting vulnerable software components through n-gram analysis and statistical feature selection. In *2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA)*, pages 543–548. IEEE, 2015. DOI: 10.1109/ICMLA.2015.99.

[26] Russell, R., Kim, L., Hamilton, L., Lazovich, T., Harer, J., Ozdemir, O., Ellingwood, P., and McConley, M. Automated vulnerability detection in source code using deep representation learning. In *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 757–762. IEEE, 2018. DOI: 10.1109/ICMLA.2018.00120.

[27] Scandariato, R., Walden, J., Hovsepyan, A., and Joosen, W. Predicting vulnerable software components via text mining. *IEEE Transactions on Software Engineering*, 40(10):993–1006, 2014. DOI: 10.1109/TSE.2014.2340398.

[28] Shar, L. K. and Tan, H. B. K. Predicting SQL injection and cross site scripting vulnerabilities through mining input sanitization patterns. *Information and Software Technology*, 55(10):1767–1780, 2013. DOI: 10.1016/j.infsof.2013.04.002.

[29] Shin, Y., Meneely, A., Williams, L., and Osborne, J. A. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *IEEE Transactions on Software Engineering*, 37(6):772–787, 2010. DOI: 10.1109/TSE.2010.81.

[30] Shin, Y. and Williams, L. An empirical model to predict security vulnerabilities using code complexity metrics. In *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 315–317, 2008. DOI: 10.1145/1414004.1414065.

[31] Shin, Y. and Williams, L. Can traditional fault prediction models be used for vulnerability prediction? *Empirical Software Engineering*, 18(1):25–59, 2013. DOI: 10.1007/s10664-011-9190-8.

[32] Spadini, D., Aniche, M., and Bacchelli, A. Pydriller: Python framework for mining software repositories. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 908–911, 2018. DOI: 10.5281/zenodo.1327411.

[33] Tieleman, T. and Hinton, G. Lecture 6.5-rmsprop, coursera: Neural networks for machine learning. Technical report, University of Toronto, 2012.

[34] Tu, Z., Su, Z., and Devanbu, P. On the localness of software. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 269–280, 2014. DOI: 10.1145/2635868.2635875.

[35] Wang, S., Liu, T., and Tan, L. Automatically learning semantic features for defect prediction. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 297–308. IEEE, 2016. DOI: 10.1145/2884781.2884804.

[36] Yamaguchi, F., Lottmann, M., and Rieck, K. Generalized vulnerability extrapolation using abstract syntax trees. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 359–368, 2012. DOI: 10.1145/2420950.2421003.

[37] Yamaguchi, F., Rieck, K., et al. Vulnerability extrapolation: Assisted discovery of vulnerabilities using machine learning. In *5th USENIX Workshop on Offensive Technologies (WOOT 11)*, 2011. DOI: 10.5555/2028052.2028065.

[38] Yu, Z., Theisen, C., Williams, L., and Menzies, T. Improving vulnerability inspection efficiency using active learning. *IEEE Transactions on Software Engineering*, 47(11):2401–2420, 2019. DOI: 10.1109/TSE.2019.2949275.

[39] Zhou, Y. and Sharma, A. Automated identification of security issues from commit messages and bug reports. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 914–919, 2017. DOI: 10.1145/3106237.3117771.

[40] Zimmermann, T., Nagappan, N., and Williams, L. Searching for a needle in a haystack: Predicting security vulnerabilities for Windows Vista. In *2010 Third International Conference on Software Testing, Verification and Validation*, pages 421–428. IEEE, 2010. DOI: 10.1109/ICST.2010.32.

Acta Cybernetica **26** (2024) 373–404.

A Formalisation of Core Erlang, a Concurrent Actor Language*

Péter Bereczky[ab], Dániel Horpácsi[ac], and Simon Thompson[ade]

Abstract

In order to reason about the behaviour of programs described in a programming language, a mathematically rigorous definition of that language is needed. In this paper, we present a machine-checked formalisation of concurrent Core Erlang (a subset of Erlang) based on our previous formalisations of its sequential sublanguage. We define a modular, frame stack semantics, show how program evaluation is carried out with it, and prove a number of properties (e.g. determinism, confluence). Finally, we define program equivalence based on bisimulations and prove that side-effect-free evaluation is a bisimulation. This research is part of a wider project that aims to verify refactorings to prove that particular program code transformations preserve program behaviour.

Keywords: formal semantics, formal verification, concurrency, actor model, program equivalence, bisimulation, Erlang, Core Erlang, Coq

1 Introduction

Our work here contributes to a wider project [17] to reason about the correctness of refactorings for functional languages in general, and for Erlang [7] in particular. In our terminology, refactoring is a code transformation that preserves the observable behaviour of programs. Our understanding of the state-of-the-art refactoring tools scene suggests that behaviour preservation (i.e. correctness) is subject to extensive testing, but formal verification is not yet used in practice. We aim to change this, at least in the case of Erlang, and develop higher assurance for refactorings by

*Supported by the ÚNKP-21-3 New National Excellence Program of the Ministry for Innovation and Technology from the source of the National Research, Development and Innovation Fund. Supported by "Application Domain Specific Highly Reliable IT Solutions" financed under the Thematic Excellence Programme TKP2020-NKA-06 (National Challenges Subprogramme) funding scheme by the National Research, Development and Innovation Fund of Hungary.

[a]Eötvös Loránd University, Budapest, Hungary
[b]E-mail: berpeti@inf.elte.hu, ORCID: 0000-0003-3183-0712
[c]E-mail: daniel-h@elte.hu, ORCID: 0000-0003-0261-0091
[d]University of Kent, United Kingdom
[e]E-mail: S.J.Thompson@kent.ac.uk, ORCID: 0000-0002-2350-301X

DOI: 10.14232/actacyb.298977

developing formal, machine-checked theories for program semantics, equivalence and program transformation.

Erlang is a dynamically-typed, impure, functional programming language, which excels at concurrency. Core Erlang [6] is a standard subset of Erlang that contains all the essential elements of Erlang, so that a semantics of Core Erlang can be extended to a semantics for the full language in a straightforward way. In earlier work we defined and implemented formal semantics for the sequential parts of Erlang and Core Erlang, including a reduction semantics for a subset of Erlang using the $\mathbb{K}$ framework [20], and a natural semantics for a subset of Core Erlang, implemented in Coq [2, 3]. We have also implemented a functional big-step [28] semantics for this subset of Core Erlang, and shown [8] that this semantics is equivalent to the natural semantics. In turn, the semantics was validated [4] against the reference implementation of Erlang, namely the Erlang/OTP compiler [11].

Having these semantics defined, we focused on proving the equivalence of programs. On the one hand, we are interested in using the semantics to prove particular pairs of programs equivalent, and on the other, the correctness of many local refactoring steps can be reduced to the equivalence of simple expressions. When developing precise, standard definitions of equivalence, we decided to bring our results to smaller-step semantics and developed a frame stack semantics and equivalence definitions [32] built on that for sequential Core Erlang [19]. The frame stack style for semantics is beneficial for two reasons: it is well-suited to express various standard equivalence definitions [29], and furthermore, the semantics of concurrent expressions can be defined more easily in small-step approaches [25].

Our formalisations of (Core) Erlang are not the first ones. There are a number of other semantics for both sequential and concurrent subsets of (Core) Erlang on which our work has been based. The novelty of our work presented here lies in the fact that it remains more faithful to the language specification [6] and the reference manual [12] than the others; for instance, unlike other works, we formalised exit signals and the signal ordering guarantee closely following the specification. We give a more detailed comparison and an overview of the related research in Section 6. Also worth pointing out is that our formal development is accompanied by a machine-checked implementation [9].

We continue our formalisation efforts and in this paper we add concurrency to our frame stack semantics for Core Erlang. In particular, we create the definition in a modular way: the sequential and process-local parts of the semantics can be replaced by a more complete formalised part of Core Erlang or Erlang (or indeed another programming language) without the need to rewrite the whole semantics. The main contributions of this paper are the following:

- A modular, frame stack semantics for a concurrent subset of Core Erlang;
- Proofs about the properties of the concurrent semantics;
- Results on Core Erlang program equivalence verification using bisimulation.

The rest of the paper is structured as follows. In Section 2 we introduce (Core) Erlang and our previous work informally, and define the syntax of the formalised sublanguage. In Section 3 we describe a modular, dynamic semantics of Core Erlang, focusing on the concurrent sublanguage, then in Section 4 we show the evaluation of simple concurrent programs and prove properties of the semantics. Section 5 defines the concepts of program equivalence and the corresponding results, and then we discuss related work in Section 6. Finally, Section 7 discusses future work and concludes.

2 Background

As mentioned before, Erlang is a dynamically-typed, impure functional programming language. The biggest strength of Erlang is that it really excels at concurrent computation, based on the actor model [1]. For this reason, Erlang was initially used in telecommunication and banking systems, but it now plays a role in high-availability, scalable web-based systems.

2.1 The Erlang Model of Concurrency

Erlang implements and extends the actor model [1]. An Erlang system contains lightweight processes (actors) that can spawn other processes to execute a particular task. Each process executes in its own space, and so they do not share memory. Processes can only communicate by asynchronous message passing. Each process has a message queue (mailbox), where incoming messages are stored in the order of their arrival. A process can select which messages to handle from its mailbox: messages do not need to be handled in the order in which they are received.

Besides messages, processes can also send and receive other signals [13], such as *link*, *unlink* and *exit*. These additional signals can trigger potential changes in the state of the process immediately upon their arrival without being placed into the mailbox. The *link* and *unlink* signals create and remove, respectively, a bi-directional link between two processes, which represents a mutual dependency, and affects the handling of *exit* signals. In general, *exit* signals are used to indicate and initiate termination; they include a reason (describing why they were sent), and a flag indicating whether they were sent through a link (we call this value the *link flag* of the *exit* signal). If one of a pair of linked processes terminates, it will send an *exit* signal to the other process via the link. Processes can terminate for a number of reasons: having finished evaluation, receiving a particular *exit* signal, or terminating abnormally (e.g. with an exception).

Processes have a flag called `'trap_exit'` which, when set, causes *exit* signals to be converted into messages (except in very particular circumstances), i.e. the process *traps exits*. Based on this flag, the reason of the exit signal, and whether the *exit* signal was sent through a link, there are three different outcomes (see [13] and Section 3.3): the receiver process 1) terminates, 2) drops the *exit* signal, or 3) converts the *exit* signal to a message and adds it at the end of its mailbox.

In the next section, we present the syntax of the language under formalisation, which is a sublanguage of Core Erlang. Note that Core Erlang is not merely a standard subset of Erlang, it is also used in the compilation process as an intermediate step, and numerous programming languages based on the BEAM platform can be compiled to Core Erlang [16]. Furthermore, as for concurrency, the two languages implement essentially the same model. For more details, we refer to the Erlang Programming book [7] and the reference manual [12].

2.2 Language Syntax

In this section we discuss and extend the formal syntax of the sequential sublanguage of Core Erlang as presented in our previous work [19]. For better readability, we use a syntax definition that abstracts over the concrete syntax of the language; however, any expressions written in this syntax can be simply transformed to Core Erlang.

Definition 1 (Language syntax)**.**

$$\begin{aligned}
v \in \mathit{Val} &::= i \mid a \mid \iota \mid \texttt{[]} \mid \texttt{[}v_1\texttt{|}v_2\texttt{]} \mid \texttt{fun}\ f/k(x_1,\dots,x_k) \to e \\
p \in \mathit{Pat} &::= i \mid a \mid \iota \mid \texttt{[]} \mid \texttt{[}p_1\texttt{|}p_2\texttt{]} \mid x \\
e \in \mathit{Exp} &::= v \mid x \mid f/k \mid \texttt{apply}\ e(e_1,\dots,e_k) \mid \texttt{case}\ e\ \texttt{of}\ p\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \\
&\quad \mid \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \mid \texttt{[}e_1\texttt{|}e_2\texttt{]} \mid \texttt{letrec}\ f/k(x_1,\dots,x_k) \to e_0\ \texttt{in}\ e_1 \\
&\quad \mid \texttt{call}\ e(e_1,\dots,e_k) \mid \texttt{receive}\ p_1 \to e_1;\dots p_k \to e_k\ \texttt{end}
\end{aligned}$$

We use i, k, n to range over integers, a, f over atoms, x over variables, and ι over process identifiers. f/k denotes a function identifier, where f is the function name, and k is its arity. The primitive values of the language are integers (denoted by numbers), atoms (strings of characters, enclosed in single quotation marks), and process identifiers (for simplicity, also denoted by numbers). Besides these, lists and functions are also values, and patterns are built from variables, integers, atoms and process identifiers, and formed into composite patterns as lists[1].

Note that process identifiers are not patterns in Core Erlang, but with process identifiers as patterns, we can distinguish them from other values of the language; in Erlang, the `is_pid` function can be used instead. This distinction is needed to maintain the proof of coincidence of sequential equivalence definitions described in [19] (namely, the coincidence of behavioural and contextual equivalence).

For simplicity of formalisation, functions are always named to enable explicit recursive calls, but in this paper we omit function names for readability when there are no recursive calls in the body expression. For lists, we use the standard notations of Erlang, that is a list $\texttt{[}e_1\texttt{|[}e_2\texttt{|[}\dots\texttt{|[}e_n\texttt{|[]]]}\dots\texttt{]]}$ will be denoted by $\texttt{[}e_1\texttt{,}e_2\texttt{,}\dots\texttt{,}e_n\texttt{]}$. Note that we also include Erlang's improper lists (such as `[1|2]`), but these do not require specific care in the semantics rules.

Expressions of the sequential sublanguage are values, variables, function identifiers, binding expressions (both `let` and `letrec`), function applications (`apply`), pattern matching (`case`) expressions[2].

[1] Tuples are not included in this language, but would be handled similarly to parameter lists.
[2] This expression is a simplified version of Core Erlang's `case`, restricting it to only two branches.

We extend the syntax (as in [19]) with two language elements in this work, the first one is BIF (built-in function) call (denoted by `call` $e(e_1, \dots, e_k)$), the second is the `receive` expression. BIFs are used to implement both sequential (e.g. addition of integers) and concurrent features of the language.

In particular, the concurrency model introduced in the previous subsection is implemented as follows:

- the `'!'` BIF is used to send messages;
- a `receive` expression is used to select a message from the process mailbox by means of pattern matching;
- processes are created with the `'spawn'` BIF (taking a function and its parameters as arguments for the new process to evaluate);
- *link*, *unlink* and *exit* signals can be sent with the identically named BIFs;
- the `'process_flag'` BIF is used to set the `'trap_exit'` flag.

The syntax we presented here is implemented in Coq using the nameless variable representation [10]. This way, we reuse existing approaches to define capture-avoiding, parallel substitutions [30]. Nonetheless, we use named variables in this paper for readability. Substitutions are denoted by $e[x_1 \mapsto v_1, \dots, x_k \mapsto v_k]$, which results in replacing $x_1, \dots, x_k$ variables simultaneously with $v_1, \dots, v_k$ values in the expression e. We omit further details about substitutions and static semantics since they are not in the scope of this paper. For further details we refer to our previous work [19] and to the formalisation [9]. Next, we show an example expression in the syntax presented above:

Example 1 (A simple map function in Core Erlang)**.** The following snippet shows a simple sequential Core Erlang function that transforms the elements of a list by applying the function `F` to each member. Since it is a rather simple definition, we present it in concrete syntax for better readability. To evaluate the function, it suffices to substitute the body of the `letrec` (denoted by `...`) with an application of `'mm'/2`.

```
letrec 'mm'/2 =
  fun(F, E) ->
    case E of [H|T]
      then [ apply F(H) | apply 'mm'/2(F, T) ]
      else []
    end
  in ...
```

△

The syntax of the sequential sublanguage is minimal, but representative.

3 Dynamic Semantics

In this section we explain the dynamic semantics of the formalised Core Erlang subset. We present a three-layered, modular semantics for the language such that the sequential parts of the semantics can be replaced by a more complete formalised part of Core Erlang, Erlang, or another programming language entirely.

Table 1: Layers of the semantics

Layer name	Notation	Description
Inter-process semantics (Section 3.4)	$\xrightarrow{\iota:a}$	System-level reductions
Process-local semantics (Section 3.3)	$\xrightarrow{a}$	Process-level reductions
Sequential semantics (Section 3.1)	$\longrightarrow$	Computational reductions

3.1 Sequential Semantics

First, we briefly present the sequential semantics [19] on which we base the concurrent formalisation. We highlight that the specification of Core Erlang [6] does not define the evaluation order of subexpressions, but the compiler employs a leftmost-innermost strategy [26]: during the standard translation of Erlang, the evaluation order is enforced by nested `let` expressions in Core Erlang. Furthermore, lists in Core Erlang are evaluated from the right[3]. The compiler's evaluation strategy *is* reflected in our definition.

The semantics has been formally defined as a frame stack semantics [29]. This definition style resembles reduction semantics [14], but the reduction context is deconstructed into a stack of evaluation frames with holes denoted by $\square$. The frame stack can be regarded as the continuation of the computation.

Definition 2 (Syntax of frames, frame stacks)**.**

$$\begin{aligned} F \in \mathit{Frame} ::= {} & \texttt{call}\ \square(e_1,\ldots,e_k) \mid \texttt{call}\ v(\square,\ldots,e_k) \mid \cdots \mid \texttt{call}\ v(v_1,\ldots,\square) \\ & \mid \texttt{apply}\ \square(e_1,\ldots,e_k) \mid \texttt{apply}\ v(\square,\ldots,e_k) \mid \cdots \mid \texttt{apply}\ v(v_1,\ldots,\square) \\ & \mid \texttt{let}\ x = \square\ \texttt{in}\ e_2 \mid \texttt{case}\ \square\ \texttt{of}\ p\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \\ & \mid \texttt{[}e_1\,\texttt{|}\,\square\texttt{]} \mid \texttt{[}\square\,\texttt{|}\,v_2\texttt{]} \\ K \in \mathit{FrameStack} ::= {} & \mathcal{I}d \mid F :: K \end{aligned}$$

For the stacks, we use the following notations: $\mathcal{I}d$ denotes the empty stack and $F :: K$ denotes adding frame F to the top of stack K. Next, we introduce two metatheoretical functions for pattern matching:

[3]The reference implementation generates a bytecode sequence that evaluates the list tail before evaluating the list head.

$\langle K, \texttt{let}\ x = e_1\ \texttt{in}\ e_2\rangle \longrightarrow \langle \texttt{let}\ x = \Box\ \texttt{in}\ e_2 :: K, e_1\rangle$
$\langle K, \texttt{[}e_1\texttt{|}e_2\texttt{]}\rangle \longrightarrow \langle \texttt{[}e_1\texttt{|}\Box\texttt{]} :: K, e_2\rangle$
$\langle K, \texttt{apply}\ e(e_1, \ldots, e_k)\rangle \longrightarrow \langle \texttt{apply}\ \Box(e_1, \ldots, e_k) :: K, e\rangle$
$\langle K, \texttt{call}\ e(e_1, \ldots, e_k)\rangle \longrightarrow \langle \texttt{call}\ \Box(e_1, \ldots, e_k) :: K, e\rangle$
$\langle K, \texttt{letrec}\ f/k(x_1, \ldots, x_k) \rightarrow e_0\ \texttt{in}\ e\rangle \longrightarrow$
$\quad\langle K, e[f/k \mapsto \texttt{fun}\ f/k(x_1, \ldots, x_k) \rightarrow e_0]\rangle$
$\langle K, \texttt{case}\ e_1\ \texttt{of}\ p\ \texttt{then}\ e_2\ \texttt{else}\ e_3\rangle \longrightarrow \langle \texttt{case}\ \Box\ \texttt{of}\ p\ \texttt{then}\ e_2\ \texttt{else}\ e_3 :: K, e_1\rangle$

$\langle \texttt{apply}\ \Box(e_1, \ldots, e_k) :: K, v\rangle \longrightarrow \langle \texttt{apply}\ v(\Box, \ldots, e_k) :: K, e_1\rangle$
$\langle \texttt{call}\ \Box(e_1, \ldots, e_k) :: K, v\rangle \longrightarrow \langle \texttt{call}\ v(\Box, \ldots, e_k) :: K, e_1\rangle$
$\langle \texttt{apply}\ v(v_1, \ldots, v_{i-1}, \Box, e_{i+1}, \ldots, e_k) :: K, v_i\rangle \longrightarrow$
$\quad\langle \texttt{apply}\ v(v_1, \ldots, v_{i-1}, v_i, \Box, e_{i+2}, \ldots, e_k) :: K, e_{i+1}\rangle \qquad (\text{if } i < k)$
$\langle \texttt{call}\ v(v_1, \ldots, v_{i-1}, \Box, e_{i+1}, \ldots, e_k) :: K, v_i\rangle \longrightarrow$
$\quad\langle v(v_1, \ldots, v_{i-1}, v_i, \Box, e_{i+2}, \ldots, e_k) :: K, e_{i+1}\rangle \qquad (\text{if } i < k)$
$\langle \texttt{[}e_1\texttt{|}\Box\texttt{]} :: K, v_2\rangle \longrightarrow \langle \texttt{[}\Box\texttt{|}v_2\texttt{]} :: K, e_1\rangle$

$\langle \texttt{apply}\ \Box() :: K, \texttt{fun}\ f/0() \rightarrow e\rangle \longrightarrow \langle K, e[f/0 \mapsto \texttt{fun}\ f/0() \rightarrow e]\rangle$
$\langle \texttt{apply}\ (\texttt{fun}\ f/k(x_1, \ldots, x_k) \rightarrow e)(v_1, \ldots, \Box) :: K, v_k\rangle \longrightarrow$
$\quad\langle K, e[f/k \mapsto \texttt{fun}\ f/k(x_1, \ldots, x_k) \rightarrow e, x_1 \mapsto v_1, \ldots, x_k \mapsto v_k]\rangle$
$\langle \texttt{call}\ \texttt{'+'}(i_1, \Box) :: K, i_2\rangle \longrightarrow \langle K, i_1 + i_2\rangle$
$\langle \texttt{let}\ x = \Box\ \texttt{in}\ e_2 :: K, v\rangle \longrightarrow \langle K, e_2[x \mapsto v]\rangle$
$\langle \texttt{[}\Box\texttt{|}v_2\texttt{]} :: K, v_1\rangle \longrightarrow \langle K, \texttt{[}v_1\texttt{|}v_2\texttt{]}\rangle$
$\langle \texttt{case}\ \Box\ \texttt{of}\ p\ \texttt{then}\ e_2\ \texttt{else}\ e_3 :: K, v\rangle \longrightarrow \langle K, e_2[\mathit{match}(p, v)]\rangle \quad (\text{if } \mathit{is\_match}(p, v))$
$\langle \texttt{case}\ \Box\ \texttt{of}\ p\ \texttt{then}\ e_2\ \texttt{else}\ e_3 :: K, v\rangle \longrightarrow \langle K, e_3\rangle \quad (\text{if } \neg\mathit{is\_match}(p, v))$

Figure 1: Sequential semantics of Core Erlang

- $\mathit{is\_match}(p, v)$: determines whether the value v matches the pattern p: that is they have been built up with the same constructs of Core Erlang up to pattern variables.

- $\mathit{match}(p, v)$: if the value v matches the pattern p, this function returns a substitution which contains the result of the pattern matching in form of a mapping from pattern variables to values.

We present the sequential semantics rules in Section 1. We use $\langle K, e\rangle \longrightarrow \langle K', e'\rangle$ to denote one reduction step between configurations consisting of a frame stack and an expression to be evaluated. Recall that v, v_i are used for values, i, i_j for integer values, and e, e_k for (unevaluated) expressions.

The biggest advantage of this semantics definition is that there are no premises in the reduction rules about the reduction of subexpressions since they have been put into the frame stack. Therefore the propagation of concurrent actions to this level is not necessary (i.e. there are no labels on the reduction rules). On the other hand, its disadvantage is that the complex syntax of frames is needed to be defined separately from the syntax of the language.

The reduction rules can be categorised into three groups:

- Rules that extract the first redex from language constructs, and put the remainder with a hole into the frame stack.
- Rules that modify the top frame of the stack by putting the calculated value into the hole, and obtaining the next reducible expression from the same frame.
- Rules that remove the top element of the frame stack, which also marks that the subexpression has been completely reduced.

The evaluation of any language element (except `letrec`) includes using exactly one rule once from the first and third categories. We note that this would change if exceptions and exception handler expressions were present in the sequential language. The connection between exceptions and signals is that when an exception terminates a process, it will emit an exit signal with the details of the exception. The presence of exceptions does not affect the modularity of the definition, but it would require consistent modifications in multiple layers.

Example 2 (Sequential evaluation of Section 1)**.** We use $\longrightarrow^*$ to denote the reflexive, transitive closure of the relation $\longrightarrow$. For simplicity, we denote the successor function `fun(X)` $\rightarrow$ `call '+'/2(X, 1)` with f in the following example. We also use mm to denote the function bound inside the `letrec` expression in Section 1.

The first step is to evaluate the head of the application mm to itself (since it is a function). Next, the parameter function f is reduced to itself. Thereafter, the parameter list is reduced (`[0,1,2]`) by deconstructing it starting from the back, pushing the head elements of the sublists into the frame stack. Actually, the semantics just checks in this case that all of these elements are values, and then the list is reconstructed. These actions transform the type of the parameter list from `[`e_1`|`e_2`]` to `[`v_1`|`v_2`]`, this is the reason why they are necessary, although, there are two seemingly identical configurations in the reduction sequence.

$$\begin{aligned}
&\langle \mathcal{I}d, \texttt{letrec 'mm'/2} = \mathit{mm}\ \texttt{in apply 'mm'/2}(f, \texttt{[0,1,2]})\rangle \longrightarrow\\
&\langle \mathcal{I}d, \texttt{apply}\ \mathit{mm}(f, \texttt{[0,1,2]})\rangle \longrightarrow\\
&\langle \texttt{apply}\ \mathit{mm}(\square, \texttt{[0,1,2]}) :: \mathcal{I}d, f\rangle \longrightarrow\\
&\langle \texttt{apply}\ \mathit{mm}(f, \square) :: \mathcal{I}d, \texttt{[0,1,2]}\rangle \longrightarrow^*\\
&\langle \texttt{[2|}\square\texttt{]} :: \texttt{[1|}\square\texttt{]} :: \texttt{[0|}\square\texttt{]} :: \texttt{apply}\ \mathit{mm}(f, \square) :: \mathcal{I}d, \texttt{[]}\rangle \longrightarrow^*\\
&\langle \texttt{apply}\ \mathit{mm}(f, \square) :: \mathcal{I}d, \texttt{[0,1,2]}\rangle
\end{aligned}$$

Thereafter, the function *mm* is applied by substituting the previous list into its body expression. The pattern in the `case` expression matches the parameter list, thus the first clause will be evaluated.

$$\begin{aligned}
&\langle \texttt{apply}\ \mathit{mm}(f, \square) :: \mathcal{I}d, \texttt{[0,1,2]}\rangle \longrightarrow\\
&\langle \mathcal{I}d, \texttt{case [0,1,2] of}\\
&\qquad\quad \texttt{[H|T] then [apply}\ f(\texttt{H})\texttt{|apply}\ \mathit{mm}(f, \texttt{T})\texttt{] else []}\rangle \longrightarrow\\
&\langle \mathcal{I}d, \texttt{[apply}\ f(\texttt{0})\texttt{|apply}\ \mathit{mm}(f, \texttt{[1,2]})\texttt{]}\rangle
\end{aligned}$$

Next, we continue the evaluation with the tail of the list (since lists are evaluated backwards). Again we evaluate the application of *mm* and reduce the `case` expression, etc. The recursion stops when the list has been consumed. The last sublist, `[]` will not match the pattern of the `case` expression, thus the application of *mm* will leave `[]` unchanged while being removed from the stack. These reduction steps built up a sequence of applications inside the stack.

$$\begin{aligned}
&\langle \mathcal{I}d, \texttt{[apply}\ f(\texttt{0})\texttt{|apply}\ \mathit{mm}(f, \texttt{[1,2]})\texttt{]}\rangle \longrightarrow^*\\
&\langle \texttt{apply}\ \mathit{mm}(f, \square) :: \texttt{[apply}\ f(\texttt{0})\texttt{|}\square\texttt{]} :: \mathcal{I}d, \texttt{[1,2]}\rangle \longrightarrow^*\\
&\langle \texttt{apply}\ \mathit{mm}(f, \square) :: \texttt{[apply}\ f(\texttt{1})\texttt{|}\square\texttt{]} :: \texttt{[apply}\ f(\texttt{0})\texttt{|}\square\texttt{]} :: \mathcal{I}d, \texttt{[2]}\rangle \longrightarrow^*\\
&\langle \texttt{apply}\ \mathit{mm}(f, \square) :: \texttt{[apply}\ f(\texttt{2})\texttt{|}\square\texttt{]} :: \texttt{[apply}\ f(\texttt{1})\texttt{|}\square\texttt{]} ::\\
&\qquad \texttt{[apply}\ f(\texttt{0})\texttt{|}\square\texttt{]} :: \mathcal{I}d, \texttt{[]}\rangle \longrightarrow^*\\
&\langle \texttt{[apply}\ f(\texttt{2})\texttt{|}\square\texttt{]} :: \texttt{[apply}\ f(\texttt{1})\texttt{|}\square\texttt{]} :: \texttt{[apply}\ f(\texttt{0})\texttt{|}\square\texttt{]} :: \mathcal{I}d, \texttt{[]}\rangle
\end{aligned}$$

Thereafter, the function applications can be evaluated for the elements of the list. First, the top element of the frame is extracted while `[]` is placed back. The application of f increases `2` to `3`. Combining the top element of the frame (`[`$\square$`|[]]`) and `3`, we obtain the list value `[3]`.

$$\begin{aligned}
&\langle \texttt{[apply}\ f(\texttt{2})\texttt{|}\square\texttt{]} :: \texttt{[apply}\ f(\texttt{1})\texttt{|}\square\texttt{]} :: \texttt{[apply}\ f(\texttt{0})\texttt{|}\square\texttt{]} :: \mathcal{I}d, \texttt{[]}\rangle \longrightarrow\\
&\langle \texttt{[}\square\texttt{|[]]} :: \texttt{[apply}\ f(\texttt{1})\texttt{|}\square\texttt{]} :: \texttt{[apply}\ f(\texttt{0})\texttt{|}\square\texttt{]} :: \mathcal{I}d, \texttt{apply}\ f(\texttt{2})\rangle \longrightarrow^*\\
&\langle \texttt{[}\square\texttt{|[]]} :: \texttt{[apply}\ f(\texttt{1})\texttt{|}\square\texttt{]} :: \texttt{[apply}\ f(\texttt{0})\texttt{|}\square\texttt{]} :: \mathcal{I}d, \texttt{3}\rangle \longrightarrow\\
&\langle \texttt{[apply}\ f(\texttt{1})\texttt{|}\square\texttt{]} :: \texttt{[apply}\ f(\texttt{0})\texttt{|}\square\texttt{]} :: \mathcal{I}d, \texttt{[3]}\rangle
\end{aligned}$$

For the other two elements, we omit the previous steps and just show how the list inside the frame stack is reconstructed.

$$\langle \texttt{[apply } f\texttt{(1)|}\square\texttt{]} :: \texttt{[apply } f\texttt{(0)|}\square\texttt{]} :: \mathcal{Id}, \texttt{[3]} \rangle \longrightarrow^{*}$$
$$\langle \texttt{[apply } f\texttt{(0)|}\square\texttt{]} :: \mathcal{Id}, \texttt{[2,3]} \rangle \longrightarrow^{*} \langle \mathcal{Id}, \texttt{[1,2,3]} \rangle$$

△

In the next section, we show how we built the concurrent semantics on top of the frame stack relation.

3.2 Processes, Signals and Actions

In this section we formalise the notions of *processes*, *signals* and *actions*, on which we build in the next two sections where we describe the concurrent semantics of Core Erlang, first the process-local semantics and then the inter-process semantics. In the remainder of this section we also establish some metatheoretical notation that we use in presenting the semantics.

Definition 3 (Core Erlang processes)**.** *A process ($p \in Process$) is either dead or alive.*

- *A live process is a quintuple $(K, e, q, pl, flag)$, where K denotes a frame stack, e is an expression, q is the mailbox (represented as a list of values). pl is the set of linked processes (a list of process identifiers), and $flag$ is the status of the* ***'trap_exit'*** *flag.*

- *A terminated (or dead) process is a list of linked process identifiers.*

As described earlier, Erlang and Core Erlang implement the actor model [1] for asynchronous communication between processes by message passing. Besides messages, there are other signals that can be sent between the processes (we formalise *exit*, *link*, and *unlink* signals beside messages) which potentially change the state of the process upon arrival without being put into the mailbox.

Actions represent the effects that characterise concurrency: message send and arrival in a mailbox, processing a mailbox with *receive*, process creation, and so on. An action will have an effect on individual processes (in the process-local semantics) and also *between* processes in the system level, inter-process semantics.

We define the following signals and actions of the semantics.

Definition 4 (Signals and Actions)**.**

$$\begin{aligned} s \in Signal &::= msg(v) \mid exit(v, b) \mid link \mid unlink \\ a \in Action &::= send(\iota_1, \iota_2, s) \mid rec(v) \mid self(\iota) \mid arr(\iota_1, \iota_2, s) \mid spawn(\iota, e_1, e_2) \\ &\quad\;\; \mid \tau \mid \Downarrow \mid flag \end{aligned}$$

Signals can be messages (parametrised by a value), exits (parametrised by a reason value and a flag whether the exit was sent through a link), links, and unlinks (which do not have parameters). The source and destination process identifiers are handled by actions, thus they are not included in the signals. We explain the syntax of actions as follows:

- Signal sending (*send*) and signal arrival (*arr*) actions carry a signal as a parameter, as well as the source and target process identifiers which are propagated from the inter-process semantics.
- *rec* actions have as parameter the message that is to be removed from the mailbox. There is no need to include process identifiers since these actions denote a process-local step and the removable message is already present in the mailbox of the process.
- *self* actions contain the identifier of the executing process as a parameter, which was obtained from the inter-process semantics.
- *spawn* actions include the new process identifier (propagated from the inter-process semantics), a function expression, and its actual parameters (as a Core Erlang list). The spawned process will execute this function with the given parameters.
- A sequential (τ) action denotes one reduction step with the sequential semantics.
- Termination ($\Downarrow$) actions denote either normal termination or the execution of the single-parameter `'exit'` BIF.
- *flag* actions denote the execution of the `'process_flag'` BIF (which does not necessarily change the state of the `'trap_exit'` flag).

Actions are used as the labels of the one-step evaluation relation. Next, we define the following *metatheoretical* functions and notations for the next sections:

- *tt* denotes the metatheoretical true, while *ff* denotes false.
- $x :: xs$ denotes a list with x as the first element and xs as the tail.
- $[]$ denotes the empty list.
- $[x_1, \ldots, x_n] = x_1 :: (x_2 :: (\ldots x_n :: []) \ldots)$.
- $\mathit{rem}_1(x, l)$: creates a list by removing the first occurrence of x from l.
- $\mathit{rem}(x, l)$: creates a list by removing all occurrences of x from l.
- $\mathit{map}(\mathit{fn}, l)$: constructs a list by applying the metatheoretical function fn to the elements of l.
- $l_1 +\!\!+\, l_2$: constructs a list to represent the concatenation of l_1 and l_2.

- $convert(b)$: maps tt to `'true'` and ff to `'false'`.
- $convert(v)$: maps `'true'` to $Some\ tt$ and `'false'` to $Some\ \mathit{ff}$, for other inputs, it returns $None$.

3.3 Process-Local Semantics

Next, we show the process-local semantics (see Section 2, Section 3, and Section 4), denoted by $p \xrightarrow{a} p'$, which describes the one-step evaluation of actions by individual processes. We primarily built this semantics by following the techniques of Fredlund's formalisation [15], since it has the widest coverage of language features among previous semantics. We note that the evaluation of the parameters of BIF calls `call` $e(e_1, \dots, e_k)$ is handled by the sequential semantics (see Section 3.1), while the final reductions are formalised in the process-local level of BIF calls, with concrete BIF names, as shown in Section 3 below.

In the following, we make a brief description of the process-local reduction rules. The process identifiers in the reduction rules are propagated from the inter-process semantics via actions. First we detail the rule for sequential steps, and the rules for signal arrival (Figure 2):

- Seq lifts the computational layer to the process-local level. This is the sequential (τ) reduction rule of the semantics. In this rule, the computational layer could be replaced by any other frame stack semantics, such as a semantics for Erlang.
- Msg describes message arrival. Whenever a message arrives, it is appended to the mailbox of the process.
- ExitDrop describes when should an exit signal be dropped without modifying the state of the process [13, Receiving Exit Signals].
- ExitTerm describes when an exit signal terminates the process [13, Receiving Exit Signals]. The process becomes a terminated process by pairing the exit reason with the linked process identifiers. When an exit signal was sent explicitly, and the reason was `'kill'`[4], it also has to be converted to `'killed'` for the links (to prevent unnecessary termination of additional processes that are trapping exits).
- ExitConv describes when an exit signal should be converted to a message and appended at the end of the mailbox [13, Receiving Exit Signals]. This action can only occur when the `'trap_exit'` flag of the process is set.
- LinkArr, UnlinkArr rules describe arrival of link and unlink signals. In the first case, a process identifier is added to the links of the process, while in the second case, all occurrences of the process identifier are removed from the links.

[4]The `'kill'` reason causes unconditional termination almost always. We explain the only exception in Section 4.1 with Example 4.

$$\frac{\langle K, e\rangle \rightarrow \langle K', e'\rangle}{(K, e, q, pl, b) \xrightarrow{\tau} (K', e', q, pl, b)} \qquad (\textsc{Seq})$$

$$(K, e, q, pl, b) \xrightarrow{arr(\iota_1,\iota_2,msg(v))} (K, e, q +\!\!+ [v], pl, b) \qquad (\textsc{Msg})$$

$$\frac{(\iota_1 \neq \iota_2 \land b = \mathit{ff} \land v = \texttt{'normal'}) \lor (\iota_1 \notin pl \land b_e = \mathit{tt} \land \iota_1 \neq \iota_2)}{(K, e, q, pl, b) \xrightarrow{arr(\iota_1,\iota_2,exit(v,b_e))} (K, e, q, pl, b)} \qquad (\textsc{ExitDrop})$$

$$\frac{\begin{array}{c}(v = \texttt{'kill'} \land b_e = \mathit{ff} \land v' = \texttt{'killed'})\lor \\ (b = \mathit{ff} \land v \neq \texttt{'normal'} \land v' = v \land (b_e = \mathit{tt} \rightarrow \iota_1 \in pl))\lor \\ (b = \mathit{ff} \land v = \texttt{'normal'} = v' \land \iota_1 = \iota_2)\end{array}}{(K, e, q, pl, b) \xrightarrow{arr(\iota_1,\iota_2,exit(v,b_e))} map\ (\lambda\iota \Rightarrow (\iota, v'))\ pl} \qquad (\textsc{ExitTerm})$$

$$\frac{b = \mathit{tt} \land ((b_e = \mathit{ff} \land v \neq \texttt{'kill'}) \lor (b_e = \mathit{tt} \land \iota_1 \in pl))}{(K, e, q, pl, b) \xrightarrow{arr(\iota_1,\iota_2,exit(v,b_e))} (K, e, q +\!\!+ [\texttt{['EXIT',}\iota_1\texttt{,}v\texttt{]}], pl, b)} \qquad (\textsc{ExitConv})$$

$$(K, e, q, pl, b) \xrightarrow{arr(\iota_1,\iota_2,link)} (K, e, q, \iota_1 :: pl, b) \qquad (\textsc{LinkArr})$$

$$(K, e, q, pl, b) \xrightarrow{arr(\iota_1,\iota_2,unlink)} (K, e, q, rem(\iota_1, pl), b) \qquad (\textsc{UnlinkArr})$$

Figure 2: Process local semantics (part 1)

Next, we describe the formal rules of signal sending (Figure 3):

- Send describes message sending. If the BIF `'!'` is on the top of the frame stack with the target process identifier, and the message is evaluated to a value, a send action is emitted containing the source (which is propagated from the inter-process semantics in the NSend rule) and target identifiers and the message value, while the send expression itself is reduced to the message value.

- Exit describes explicitly sending an exit signal to a process. If the two-parameter `'exit'` BIF is on the top of the frame stack with the target process identifier, and the reason is evaluated to a value, an exit action is emitted with the source (which is propagated from the inter-process semantics in NSend), target identifiers, and the exit reason value, while the expression is reduced to `'true'`. Note that when sending an explicit exit signal, the link flag of the signal is false.

- Link, Unlink rules both reduce the evaluable expression to `'ok'`. In the first case, a link signal is emitted with the source and target identifier, and the target is appended to the links of the process. In the second case, an unlink signal is emitted with the source and target identifier, and the target is removed from the links of the process.
- Dead describes the communication of a terminated process. In this rule, the first item of the links of the dead process is removed while an exit signal is emitted to the target with the reason that is specified in this first item. Note that the link flag of this exit signal is *true*, because this exit is sent through a link.

$$(\texttt{call '!'}(\iota_2, \square) :: K, v, q, pl, b) \xrightarrow{send(\iota_1,\iota_2,msg(v))} (K, v, q, pl, b) \qquad (\textsc{Send})$$

$$(\texttt{call 'exit'}(\iota_2, \square) :: K, v, q, pl, b) \xrightarrow{send(\iota_1,\iota_2,exit(v,\mathit{ff}))} (K, \texttt{'true'}, q, pl, b) \qquad (\textsc{Exit})$$

$$(\texttt{call 'link'}(\square) :: K, \iota_2, q, pl, b) \xrightarrow{send(\iota_1,\iota_2,link)} (K, \texttt{'ok'}, q, \iota_2 :: pl, b) \qquad (\textsc{Link})$$

$$(\texttt{call 'unlink'}(\square) :: K, \iota_2, q, pl, b) \xrightarrow{send(\iota_1,\iota_2,unlink)} (K, \texttt{'ok'}, q, rem(\iota_2, pl), b) \qquad (\textsc{Unlink})$$

$$(\iota_2, v) :: pl \xrightarrow{send(\iota_1,\iota_2,exit(v,tt))} pl \qquad (\textsc{Dead})$$

Figure 3: Process-local semantics (part 2)

Finally, we detail the rest of the process-local rules (Figure 4):

- Self receives the identifier of the process from the inter-process semantics, and evaluates the `'self'` BIF call to this identifier.
- Spawn describes process creation. The spawned process receives its identifier from the inter-process semantics, and this identifier will be the result of this

rule. Note that it is necessary that the first parameter of the `'spawn'` is a function value, while the second is a correct, object-level parameter list (which is checked in the inter-process semantics).

- Receive describes message processing. With pattern matching, the first (oldest) message is selected from the mailbox of the process that matches any clause of the `receive` expression (if more patterns are matching to the same message, the first matching clause is selected). The evaluation continues with the body expression of the selected clause, substituted by the result (pattern variable - value) bindings.
- Flag describes when the process flag `'trap_exit'` changes. The result of this rule is the original value of the flag.
- Term describes normal termination, i.e. there are no more continuations in the frame stack, and the evaluable expression has already been reduced to a value. The result is a dead process, which will send exit signals to its links with the reason `'normal'`.
- ExitSelf describes the call of the single-parameter `'exit'` BIF. It imme-

$$(\texttt{call } \square() :: K, \texttt{'self'}, q, pl, b) \xrightarrow{self(\iota)} (K, \iota, q, pl, b) \qquad (\textsc{Self})$$

$$\frac{f = \texttt{fun } f/k(x_1, \ldots, x_k) \to e}{(\texttt{call 'spawn'}(f, \square) :: K, vs, q, pl, b) \xrightarrow{spawn(\iota, f, vs)} (K, \iota, q, pl, b)} \qquad (\textsc{Spawn})$$

$$\frac{\begin{array}{ll} l = match(p_i, v) & \\ is\_match(p_i, v) & \forall j < i : \neg is\_match(p_j, v) \\ q = [v_1, \ldots, v_n, v, \ldots] & (\forall m, j : 1 \le m \le k \wedge 1 \le j \le n \implies \neg is\_match(p_m, v_j)) \end{array}}{(K, \texttt{receive } p_1 \to e_1; \ldots; p_k \to e_k \texttt{ end}, q, pl, b) \xrightarrow{rec(v)} (K, e_i[l], rem_1(v, q), pl, b)} \qquad (\textsc{Receive})$$

$$\frac{convert(v) = Some\ v' \qquad v'' = convert(b)}{(\texttt{call 'process\_flag'}(\texttt{'trap\_exit'}, \square) :: K, v, q, pl, b) \xrightarrow{flag} (K, v'', q, pl, v')} \qquad (\textsc{Flag})$$

$$(\mathfrak{I}d, v, q, pl, b) \xrightarrow{\Downarrow} map\ (\lambda\iota \Rightarrow (\iota, \texttt{'normal'}))\ pl \qquad (\textsc{Term})$$

$$(\texttt{call 'exit'}() :: K, v, q, pl, b) \xrightarrow{\Downarrow} map\ (\lambda\iota \Rightarrow (\iota, v))\ pl \qquad (\textsc{ExitSelf})$$

Figure 4: Process-local semantics (part 3)

diately terminates the process, and exit signals will be sent to the linked processes with the parameter reason value. We note that when introducing exceptions in the future, this version of exit signals will be capable of being caught by exception handlers.

3.4 Inter-Process Semantics

In this section we discuss the inter-process reduction rules for the semantics. The advantage of this formalisation is that the dynamic semantics of the system is described by only 5 rules (by combining the rules from related work [15, 18, 23] with the same premises but different actions), which resulted in shorter proofs. First, we introduce the necessary concepts.

Definition 5 (Ether)**.** *An ether (denoted by Δ) is a mapping of source and target identifier pairs to lists of signals. We use $\emptyset$ to denote the empty ether, which maps everything to the empty list*[5]*.*

We use an ether to express non-atomic signal passing (unlike a number of related works [15, 18]); that is, the signals sent from one process do not arrive immediately, but they are transferred via the ether. This is also described in the reference manual [13]: "The amount of time that passes between the time a signal is sent and the arrival of the signal at the destination is unspecified but positive".

In addition, the ether is used to implement the signal ordering guarantee [13], that is "if an entity sends multiple signals to the same destination entity, the order is preserved". If a source sends multiple signals to the same target, these signals will be appended to the end of the list associated with the source and target in the ether. However, if multiple processes send signals to the same destination, the arrival order of these signals is not specified, thus they are included in separate lists in the ether based on their source.

Definition 6 (Node)**.** *A node is a pair ($(\Delta, \Pi) \in Node$) of an ether and a process pool. The process pool (denoted by Π) is a mapping that associates process identifiers with processes. We denote nodes with Σ and the empty process pool with $\emptyset$.*

On top of these concepts, we introduce notations and metatheoretical functions:

- $\iota : p \parallel \Pi$: Appends process p associated with the identifier ι to the process pool Π. In formalising this we used function update, so that the order of identifiers is irrelevant. Because of this, we are justified in abusing the notation somewhat when we write the rules using pattern matching: without loss of generality, we assume that the item of interest appears in the head position of the collection of processes given.

- *remFirst*(Δ, ι, ι'): Removes the first element in the ether Δ from the list associated with ι source and ι' destination, and returns a pair of this removed

[5]In the implementation, we formalised the ether as a function which maps (source) process identifiers to a function mapping (target) process identifiers to a list of signals.

signal and the result ether inside *Some*. If the associated list was empty, it returns *None*.

- $\Delta[(\iota, \iota') \overset{+}{\mapsto} s]$: Creates an ether by appending the signal s to the end of the list associated with ι source and ι' destination in the ether Δ (while keeping other parts of Δ unchanged).
- $\Pi \setminus \iota$: Creates a process pool by removing the process associated with ι from process pool Π. This operation was also formalised by function updates.
- $\iota \notin \Pi$: Checks whether there is no process associated with ι in Π.
- $convert\_list(vs)$: Creates a metatheoretical list of expressions based on an object-level Core Erlang list (constructed with `[_|_]` and `[]`); if successful the result is wrapped with a *Some* constructor; if not, *None* is returned.

Next, we define the semantics in Figure 5. This one-step reduction is denoted by $\Sigma \xrightarrow{\iota:a} \Sigma'$ that means the node Σ is reduced to Σ' by taking a reduction step determined by the action a with the process identified by ι. The rules always include a "first" process $(\iota : p \parallel \Pi)$, nevertheless, any process from the pool can take this place, since any process in a $\parallel$ chain can be the outermost one, as mentioned before. We give a brief, informal description of the inter-process rules now:

- NSend describes signal sending. While a process with the identifier ι is reduced by emitting a send action, the contents of this action (target, source, and signal) are placed into the ether.
- NArrive describes how an element is (nondeterministically) removed from the ether. Any signal can be removed from the lists in the ether, if the signal is the first element of that list, and there is a live process with the destination identifier in the process pool.
- NTerm describes how a process identifier is freed. When a dead process has notified all of its links, its identifier is removed from the process pool.
- NSpawn describes the creation of a new process. The new process is assigned a non-used identifier, and it starts evaluating the function application described in the spawn action of the rule (note that conversion from object-level to meta-level lists is needed). The initial configuration of the new process is the empty frame stack (continuation), the given function application as the evaluable expression, empty mailbox, it has no links, and it does not trap exit signals.
- NOther describes the reduction in case of any other action, that is, this rule propagates these actions to the process-local level.

We note that in every rule of this semantics, exactly one process is reduced. Furthermore, all reduction rules (except NTerm) actually propagate the action to the process-local semantics, while modifying the ether or the process pool. We also introduce the following notations on top of the inter-process semantics:

$$\frac{p \xrightarrow{send(\iota_1,\iota_2,s)} p'}{(\Delta, \iota_1 : p \parallel \Pi) \xrightarrow{\iota_1 : send(\iota_1,\iota_2,s)} (\Delta[(\iota_1,\iota_2) \overset{+}{\mapsto} s], \iota_1 : p' \parallel \Pi)} \quad \text{(NSend)}$$

$$\frac{p \xrightarrow{arr(\iota_1,\iota_2,s)} p' \qquad remFirst(\Delta, \iota_1, \iota_2) = Some\ (s, \Delta')}{(\Delta, \iota_1 : p \parallel \Pi) \xrightarrow{\iota_1 : arr(\iota_1,\iota_2,s)} (\Delta', \iota_1 : p' \parallel \Pi)} \quad \text{(NArrive)}$$

$$(\Delta, \iota : [] \parallel \Pi) \xrightarrow{\iota : \Downarrow} (\Delta, \Pi \setminus \iota) \quad \text{(NTerm)}$$

$$\frac{\begin{array}{cc} \iota_2 \notin (\iota_1 : p \parallel \Pi) & v = \texttt{fun}\ f/k(x_1, \dots, x_k) \to e \\ p \xrightarrow{spawn(\iota_2,v,vs)} p' & convert\_list(vs) = Some\ [v_1, \dots, v_k] \end{array}}{(\Delta, \iota_1 : p \parallel \Pi) \xrightarrow{\iota_1 : spawn(\iota_2,v,vs)} (\Delta, \iota_2 : ([], \texttt{apply}\ v(v_1, \dots, v_k), [], [], ff) \parallel \iota_1 : p' \parallel \Pi)} \quad \text{(NSpawn)}$$

$$\frac{p \xrightarrow{a} p' \qquad a \in \{self(\iota), \Downarrow, \tau, flag\} \cup \{rec(v) \mid v \in Value\}}{(\Delta, \iota : p \parallel \Pi) \xrightarrow{\iota : a} (\Delta, \iota : p' \parallel \Pi)} \quad \text{(NOther)}$$

Figure 5: Formal semantics of communication between processes

- $\Sigma \xrightarrow{l} {}^*\Sigma'$ denotes a special reflexive, transitive closure of the relation $\xrightarrow{\iota:a}$, which traces the actions in the list l in forms of (ι, a) pairs. We use $\Sigma \xrightarrow{\dots} {}^*\Sigma'$ when the trace is not relevant. For example, if a node Σ can be reduced to Σ' in the three following steps: 1) the process identified by ι sends a message v to the process identified by ι', 2) this message arrives to the target, 3) the message is received by the target, we use

$$\Sigma \xrightarrow{[(\iota, send(\iota,\iota',msg(v))), (\iota', arr(\iota,\iota',msg(v))), (\iota', rec(v))]} {}^*\Sigma'.$$

- $\Sigma \longrightarrow^* \Sigma$ denotes a reduction sequence from node Σ to node Σ' that contains only sequential (τ) reduction steps.

Discussion. There are other approaches (e.g. [15]) which define fewer actions for the semantics by defining input, output, spawn, and silent actions. In these approaches, $rec(v), flag, \Downarrow, \tau$ could all be handled as silent actions, since they affect the state of a single process and do not communicate. Still, we formalised the more fine-grained version, because this allows us to group the rules of the semantics into more categories. By coupling the aforementioned actions, the less detailed approach can also be simulated. Moreover, we proved theorems (specifically, Theorem 9) which would not be provable if other actions were also considered to be silent.

4 Semantics Validation

After defining a formal semantics, the next step is to validate it [5]. We use two approaches: 1) we evaluate simple parallel programs and compare the results to the results of the Erlang/OTP compiler, and 2) we prove properties of the semantics. We are also investigating ways in which the concurrent semantics can be executed efficiently, which is a necessary step to enable extensive validation against the reference implementation.

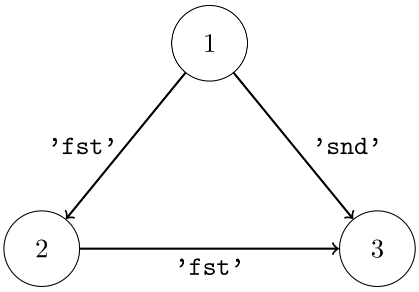


Figure 6: Actor diagram for Example 3

4.1 Example Program Evaluation

In this section we present some simple program evaluation case studies that demonstrate how the semantics operates.

Example 3 (Signal ordering)**.** The first example illustrates when the signal ordering guarantee cannot be applied. Let us consider three processes (with the identifiers 1, 2, 3), which evaluate the following expressions.

1. `let X = call '!'(2, 'fst') in call '!'(3, 'snd')`
2. `receive X -> call '!'(3, X) end`
3. `receive X -> X end`

Next, we construct a node with the empty ether from these processes, and start evaluating it. We use Π, to denote the process pool constructed from 2 and 3. For simplicity, we omit the list of linked processes and the trap flag, since they are not used during this evaluation. First, we reduce process 1, since the others are all blocked by `receive` expressions. This evaluation puts the two messages into the ether.

$$(\emptyset, 1 : (\mathcal{I}d, \texttt{let X = call '!'(2, 'fst') in call '!'(3, 'snd')}, []) \parallel \Pi) \xrightarrow{\ldots}{}^{*}$$
$$(\emptyset[(1,2) \overset{+}{\mapsto} \mathit{msg}(\texttt{'fst'})][(1,3) \overset{+}{\mapsto} \mathit{msg}(\texttt{'snd'})], 1 : (\mathcal{I}d, \texttt{'snd'}, []) \parallel \Pi) \qquad (\mathit{Init})$$

We denote the result process pool with Π_1 without process 3. Next, we can evaluate process 3, to which the message `'snd'` arrives. Then the `receive` expression removes it from the mailbox and processes it. Thus the final value upon termination in process 3 is `'snd'`.

$$\begin{aligned}
&(\emptyset[(1,2) \overset{+}{\mapsto} \mathit{msg}(\texttt{'fst'})][(1,3) \overset{+}{\mapsto} \mathit{msg}(\texttt{'snd'})],\\
&\quad 3 : (\mathcal{I}d, \texttt{receive X -> X end}, []) \parallel \Pi_1) \xrightarrow{3: \mathit{arr}(1,3,\mathit{msg}(\texttt{'snd'}))}\\
&(\emptyset[(1,2) \overset{+}{\mapsto} \mathit{msg}(\texttt{'fst'})], 3 : (\mathcal{I}d, \texttt{receive X -> X end}, [\texttt{'snd'}]) \parallel \Pi_1) \xrightarrow{\ldots}{}^{*}\\
&(\emptyset[(1,2) \overset{+}{\mapsto} \mathit{msg}(\texttt{'fst'})], 3 : (\mathcal{I}d, \texttt{'snd'}, []) \parallel \Pi_1)
\end{aligned}$$

Note that the last configuration we presented above could still progress, because process 2 can receive and forward the message `'fst'`.

However, this was not the only option to evaluate this simple program. Instead of evaluating process 3 in the previous reductions, we can progress with process 2 (from the state reached in *Init*). We denote the process pool containing the terminated process 1 and process 3 with Π_2. First, the message `'fst'` arrives to process 2 which removes it from the mailbox, and forwards it to process 3.

$$\begin{aligned}
&(\emptyset[(1,2) \overset{+}{\mapsto} \mathit{msg}(\texttt{'fst'})][(1,3) \overset{+}{\mapsto} \mathit{msg}(\texttt{'snd'})],\\
&\quad 2 : (\mathcal{I}d, \texttt{receive X -> call '!'(3, X) end}, []) \parallel \Pi_2) \xrightarrow{2: \mathit{arr}(1,2,\mathit{msg}(\texttt{'fst'}))}\\
&(\emptyset[(1,3) \overset{+}{\mapsto} \mathit{msg}(\texttt{'snd'})],\\
&\quad 2 : (\mathcal{I}d, \texttt{receive X -> call '!'(3, X) end}, [\texttt{'fst'}]) \parallel \Pi_2) \xrightarrow{\ldots}{}^{*}\\
&(\emptyset[(1,3) \overset{+}{\mapsto} \mathit{msg}(\texttt{'snd'})][(2,3) \overset{+}{\mapsto} \mathit{msg}(\texttt{'fst'})], 2 : (\mathcal{I}d, \texttt{'fst'}, []) \parallel \Pi_2)
\end{aligned}$$

After processes 1 and 2 are terminated (we denote the pool containing these with Π_3), we evaluate process 3. At this point, either of the messages in the ether could arrive first at process 3, which will be processed then by the `receive` expression, since their source is different. We present the case when `'fst'` arrives first.

$$(\emptyset[(1,3) \overset{+}{\mapsto} \mathit{msg}(\texttt{'snd'})][(2,3) \overset{+}{\mapsto} \mathit{msg}(\texttt{'fst'})],$$
$$3 : (\mathcal{I}d, \texttt{receive X -> X end}, []) \parallel \Pi_3) \xrightarrow{3:\mathit{arr}(2,3,\mathit{msg}(\texttt{'fst'}))}$$
$$(\emptyset[(1,3) \overset{+}{\mapsto} \mathit{msg}(\texttt{'snd'})],$$
$$3 : (\mathcal{I}d, \texttt{receive X -> X end}, [\texttt{'fst'}]) \parallel \Pi_3) \xrightarrow{3:\mathit{arr}(1,3,\mathit{msg}(\texttt{'snd'}))}$$
$$(\emptyset, 3 : (\mathcal{I}d, \texttt{receive X -> X end}, [\texttt{'fst'}, \texttt{'snd'}]) \parallel \Pi_3) \overset{\cdots}{\longrightarrow}{}^{*}$$
$$(\emptyset[(1,2) \overset{+}{\mapsto} \mathit{msg}(\texttt{'snd'})], 3 : (\mathcal{I}d, \texttt{'fst'}, [\texttt{'snd'}]) \parallel \Pi_3)$$

However, with this reduction sequence, process 3 terminates with `'fst'`. The signal ordering guarantee was not applicable in this scenario, because the messages that process 3 received are from different sources. △

Example 4 (Exit signals)**.** Next, we present an example about sending exit signals, specifically we show the difference between the one- and two-parameter `'exit'` BIFs. Consider two processes:

1. `let X = call 'link'(2) in call 'exit'(1, 'kill')`
2. `receive X -> X end`

The second process is set to trap exit signals. Once again, we start the evaluation with the first process. In the first steps, process 1 creates the link between the two processes:

$$(\emptyset, 1 : (\mathcal{I}d, \texttt{let X = call 'link'(2) in call 'exit'(1, 'kill')}, [], [], \mathit{ff}) \parallel$$
$$2 : (\mathcal{I}d, \texttt{receive X -> X end}, [], [], \mathit{tt}) \parallel \emptyset) \overset{\cdots}{\longrightarrow}{}^{*}$$
$$(\emptyset[(1,2) \overset{+}{\mapsto} \mathit{link}], 1 : (\mathcal{I}d, \texttt{call 'exit'(1, 'kill')}, [], [2], \mathit{ff}) \parallel$$
$$2 : (\mathcal{I}d, \texttt{receive X -> X end}, [], [], \mathit{tt}) \parallel \emptyset) \xrightarrow{2:\mathit{arr}(1,2,\mathit{link})}$$
$$(\emptyset, 1 : (\mathcal{I}d, \texttt{call 'exit'(1, 'kill')}, [], [2], \mathit{ff}) \parallel$$
$$2 : (\mathcal{I}d, \texttt{receive X -> X end}, [], [1], \mathit{tt}) \parallel \emptyset)$$
(*Link*)

Next, the first process terminates itself with the two-parameter `'exit'`. This involves multiple reduction steps, because the signal needs to be put into the ether, and then retrieved from it. Then the reason will be converted to `'killed'` because the two-parameter `'exit'` always sets the link flag of the exit signal to *ff*.

$$(\emptyset, 1 : (\mathcal{I}d, \texttt{call 'exit'(1, 'kill')}, [], [2], \mathit{ff}) \parallel$$
$$2 : (\mathcal{I}d, \texttt{receive X -> X end}, [], [1], \mathit{tt}) \parallel \emptyset) \overset{\cdots}{\longrightarrow}{}^{*}$$
$$(\emptyset[(1,1) \overset{+}{\mapsto} \mathit{exit}(\texttt{'kill'}, \mathit{ff})], 1 : (\mathcal{I}d, \texttt{'true'}, [], [2], \mathit{ff}) \parallel$$
$$2 : (\mathcal{I}d, \texttt{receive X -> X end}, [], [1], \mathit{tt}) \parallel \emptyset) \xrightarrow{1:\mathit{arr}(1,1,\mathit{exit}(\texttt{'kill'},\mathit{ff}))}$$
$$(\emptyset, 1 : [(2, \texttt{'killed'})] \parallel 2 : (\mathcal{I}d, \texttt{receive X -> X end}, [], [1], \mathit{tt}) \parallel \emptyset)$$

Next, we propagate the exit signal through the link, and it will be converted to a message because of the trap flag in the execution of process 2.

$$(\emptyset, 1 : [(2, \texttt{'killed'})] \parallel 2 : (\mathcal{I}d, \texttt{receive X -> X end}, [], [1], tt) \parallel \emptyset) \xrightarrow{\cdots}{}^{*}$$
$$(\emptyset[(1,2) \overset{+}{\mapsto} \mathit{exit}(\texttt{'killed'}, tt)],$$
$$2 : (\mathcal{I}d, \texttt{receive X -> X end}, [], [1], tt) \parallel \emptyset) \xrightarrow{2:\mathit{arr}(1,2,\mathit{exit}(\texttt{'killed'},tt))}$$
$$(\emptyset, 2 : (\mathcal{I}d, \texttt{receive X -> X end}, [\texttt{['EXIT', 1, 'killed']}], [1], tt) \parallel \emptyset) \xrightarrow{\cdots}{}^{*}$$
$$(\emptyset, 2 : (\mathcal{I}d, \texttt{['EXIT', 1, 'killed']}, [], [1], tt) \parallel \emptyset$$

However, if we use the single parameter `'exit'` BIF, the reduction would be carried out otherwise. We start the evaluation from the analogous state to the point *Link* above. It immediately terminates the process without sending signals into the ether. This also causes the reason `'kill'` not to be converted to `'killed'`. Next, this exit signal will be sent through a link (the link flag of the signal is tt), which enables the use of ExitConv in process 2.

$$(\emptyset, 1 : (\mathcal{I}d, \texttt{call 'exit'('kill')}, [], [2], \mathit{ff}) \parallel$$
$$2 : (\mathcal{I}d, \texttt{receive X -> X end}, [], [1], tt) \parallel \emptyset) \xrightarrow{\cdots}{}^{*}$$
$$(\emptyset, 1 : [(2, \texttt{'kill'})] \parallel 2 : (\mathcal{I}d, \texttt{receive X -> X end}, [], [1], tt) \parallel \emptyset) \xrightarrow{\cdots}{}^{*}$$
$$(\emptyset[(1,2) \overset{+}{\mapsto} \mathit{exit}(\texttt{'kill'}, tt)],$$
$$2 : (\mathcal{I}d, \texttt{receive X -> X end}, [], [1], tt) \parallel \emptyset) \xrightarrow{2:\mathit{arr}(1,2,\mathit{exit}(\texttt{'kill'},tt))}$$
$$(\emptyset, 2 : (\mathcal{I}d, \texttt{receive X -> X end}, [\texttt{['EXIT', 1, 'kill']}], [1], tt) \parallel \emptyset) \xrightarrow{\cdots}{}^{*}$$
$$(\emptyset, 2 : (\mathcal{I}d, \texttt{['EXIT', 1, 'kill']}, [], [1], tt) \parallel \emptyset$$

We should note that the `'kill'` reason is normally used to terminate a process regardless of its current state. Although, in this case (when the signal is sent through a link, i.e. its link flag is set) `'kill'` does *not* terminate the process in question. $\triangle$

4.2 Properties of the Semantics

After formally evaluating simple programs, we proved some fundamental properties of the layers of the semantics, and formalised the proofs in the Coq theorem prover [9]. In this section we highlight the most important properties, and provide sketches of the proofs; for more insights, we refer to the formalisation. First, we show the determinism of the sequential and process-local levels.

Theorem 1 (Sequential and process-local evaluation is deterministic)**.** *For all frame stacks K, K', K'' and expressions e, e', e'', if $\langle K, e\rangle \longrightarrow \langle K', e'\rangle$ and $\langle K, e\rangle \longrightarrow \langle K'', e''\rangle$, then $K' = K''$ and $e' = e''$.*

Similarly, for all processes p, p', p'', and actions a, if $p \xrightarrow{a} p'$ and $p \xrightarrow{a} p''$, then $p' = p''$.

Proof. To prove determinism (in both semantics), we carried out case distinction based on the two reduction premises. If both use the same reduction rule, their result is equal, otherwise a contradiction is found between the premises of the different rules. □

The determinism of these layers of the semantics is a natural property; one process should handle an incoming action in the same way in the same inner state. However, we found that the conditions in the reference manual [13, Receiving Exit Signals] are ambiguous in the way that they describe how to handle exit signals. We checked with the reference implementation what the correct conditions are, and encoded them in the premises for reduction rules about exit signals: ExitConv, ExitDrop, and ExitTerm.

In the previous sections, we emphasised why the concept of the ether is necessary to ensure the signal ordering guarantee. We formally verified this property.

Theorem 2 (Signal ordering guarantee). *For all nodes $\Sigma_1, \Sigma_2, \Sigma_3$, process identifiers ι, ι', and unique signals*[6] $s_1 \neq s_2$*, if* $\Sigma_1 \xrightarrow{\iota : send(\iota, \iota', s_1)} \Sigma_2$ *and* $\Sigma_2 \xrightarrow{\iota : send(\iota, \iota', s_2)} \Sigma_3$*, then for all nodes Σ_4 and action traces l which satisfy* $\Sigma_3 \xrightarrow{l}{}^{*} \Sigma_4$ *and also* $(\iota', arr(\iota, \iota', s_1)) \notin l$ *there is no node Σ_5 at which s_2 can arrive:* $\Sigma_4 \xrightarrow{\iota' : arr(\iota, \iota', s_2)} \Sigma_5$.

Proof. We proved this theorem by induction on the length of the reduction chain $\Sigma_3 \xrightarrow{l}{}^{*} \Sigma_4$. In the base case, the first removable element in the ether is either s_1 or another signal which is different from s_2. In the inductive case, we suppose that there is a reduction chain of length k which does not remove s_1 from the ether. Then there is the $(k + 1)$th reduction step, which also cannot remove s_1 from the ether, based on the hypotheses. Thus once again, the first removable element from the ether is either s_1 or another signal which is different from s_2. □

This theorem informally states the following: if two signals have been sent from the same sender to the same target, after taking any number of reduction steps, which do not contain the arrival of the first signal, it is not possible that the second signal will arrive to the target.

We also proved a number of confluence properties, which are the basis of proving bisimulation-based program equivalence. Our goal is to prove that sequential evaluation ($\Sigma \longrightarrow^{*} \Sigma'$) produces equivalent nodes. The first theorem expresses that a sequential reduction can be carried out after another reduction step if this step does not terminate the process. Otherwise, the sequential reduction cannot be executed after the other action. This property holds for both process-local and inter-process semantics.

Theorem 3 (Confluence of sequential reductions in the same process). *For all processes p_1, p_2, p_2', and action a, supposing that $p_1 \xrightarrow{\tau} p_2$ and $p_1 \xrightarrow{a} p_2'$, then there exists a process p_3 that satisfies $p_2 \xrightarrow{a} p_3$ and $(p_2' \xrightarrow{\tau} p_3 \vee p_2' = p_3)$.*

[6] They are different from any other signal in the starting configuration.

Similarly, for all nodes $\Sigma_1, \Sigma_2, \Sigma_2'$, process identifiers ι, and actions a, supposing that $\Sigma_1 \xrightarrow{\iota:\tau} \Sigma_2$ and $\Sigma_1 \xrightarrow{\iota:a} \Sigma_2'$, then there exists a node Σ_3 that satisfies $\Sigma_2 \xrightarrow{\iota:a} \Sigma_3$ and $(\Sigma_2' \xrightarrow{\iota:\tau} \Sigma_3 \lor \Sigma_2' = \Sigma_3)$.

Proof. The proof for both semantics relies on case distinction in the derivation of $\Sigma_1 \xrightarrow{\iota:a} \Sigma_2'$. There are actually two separate cases:

- If the action a does not terminate the process (still, it potentially modifies either the mailbox, or the list of linked processes, or the `'trap_exit'` flag), then the sequential reduction step can be taken after this action too, since these steps are not influenced by the mentioned attributes of the process.
- If the action a terminates the process, then p_2' and p_3 denote the same terminated process, since sequential steps do not modify the list of linked processes, which is the only attribute of a live process that is kept when it terminates.

□

This theorem is used when two reductions for the same process need to be chained after each other. Actually, this theorem is a stepping stone towards proving Theorem 5.

The next theorem concerns different processes: possible reduction steps can be carried out after each other if they are not both *spawn* actions.

Theorem 4 (Action ordering)**.** *For all nodes $\Sigma_1, \Sigma_2, \Sigma_2'$, process identifiers $\iota \neq \iota'$, actions a, a', which are not both spawn actions, if $\Sigma_1 \xrightarrow{\iota:a} \Sigma_2$ and $\Sigma_1 \xrightarrow{\iota':a'} \Sigma_2'$ then there exists a node Σ_3, which can be reached from Σ_2 with action a': $\Sigma_2 \xrightarrow{\iota':a'} \Sigma_3$.*

Proof. This theorem is proved by case separation on the two reduction premises. There is no scheduling algorithm formalised in the semantics, thus any process can be reduced if it is not in a stuck configuration (i.e. if it is waiting for a message to evaluate a `receive` expression). We can define any order for the reductions of different processes (except if both reductions are labelled by *spawn* actions), because both of the reductions in the premise can always be carried out. The only action a in the first reduction that could prevent making the second reduction (with action a') is the arrival of an exit signal that terminates the process identified by ι', but the premise $\iota \neq \iota'$ rules this case out. □

The premise that restricts *spawn* actions is necessary because we cannot assure that these spawned processes obtain the same process identifiers if their spawn order is reversed (currently, the semantics assigns fresh process identifiers to spawned processes based on the list of process identifiers already in use). This theorem is also a stepping stone towards Theorem 6.

The following theorems contain any-step reduction chains. The first of these theorem expresses that if there are τ actions and an additional action that can be executed in a configuration, then either this additional action can be executed at the final node after executing the chain, or it was τ-reduction inside the chain.

Theorem 5 (Chaining a reduction to the end of an sequential sequence)**.** *For all nodes $\Sigma_1, \Sigma_4, \Sigma_4'$, process identifier ι, action a, and action traces l, which only include internal actions paired with any process identifiers, if $\Sigma_1 \xrightarrow{l}{}^{*} \Sigma_4$ and $\Sigma_1 \xrightarrow{\iota:a} \Sigma_4'$, then there are two potential scenarios:*

- *Either there is a node Σ_5 which can be reached by a reduction from Σ_4: $\Sigma_4 \xrightarrow{\iota:a} \Sigma_5$.*
- *Or $a = \tau$ and there are nodes Σ_2, Σ_3 and action traces l_1, l_2, which can be used to split the sequential reduction steps: $\Sigma_1 \xrightarrow{l_1}{}^{*} \Sigma_2$, $\Sigma_2 \xrightarrow{\iota:a} \Sigma_3$ and $\Sigma_3 \xrightarrow{l_2}{}^{*} \Sigma_4$, moreover $l = l_1 \mathbin{++} [(\iota, a)] \mathbin{++} l_2$.*

Proof. We proved this theorem by induction on the reduction chain $\Sigma_1 \xrightarrow{l}{}^{*} \Sigma_4$. The base case is solved by the premise $\Sigma_1 \xrightarrow{\iota:a} \Sigma_4'$ (by choosing $\Sigma_5 = \Sigma_4'$), since $\Sigma_1 \xrightarrow{l}{}^{*} \Sigma_4$ was a 0-step reduction, thus $\Sigma_1 = \Sigma_4$. In the inductive case, we did case distinction whether $a = \tau$ and (ι, a) is included in l. If this is not true, we can make the reduction determined by (ι, a) from the configuration Σ_4 based on the induction hypothesis and Theorem 3. Otherwise $(\iota, a) = (\iota, \tau)$ is included in the action trace l. $\square$

This theorem expresses one of the fundamental properties needed to prove that $\longrightarrow^*$ is a weak bisimulation, (Theorem 9 below). The next theorem is the other fundamental property required. If there is an action that is executed at the end of the execution of a sequential reduction chain, and it can be executed in the starting configuration too, then from the result of the second derivation the result of the first one can be reached by only sequential steps.

Theorem 6 (Confluence of sequential reductions)**.** *For all nodes $\Sigma_1, \Sigma_2, \Sigma_2', \Sigma_3$, process identifier ι, and action a, if $\Sigma_1 \longrightarrow^* \Sigma_2$, and a reduction can be done in the starting and in the final configuration too: $\Sigma_1 \xrightarrow{\iota:a} \Sigma_2'$, and $\Sigma_2 \xrightarrow{\iota:a} \Sigma_3$, then $\Sigma_2' \longrightarrow^* \Sigma_3$.*

Proof. The proof of this property is also carried out by induction on the reduction chain $\Sigma_1 \longrightarrow^* \Sigma_2$. In the base case $\Sigma_1 = \Sigma_2$ and by Theorem 1, $\Sigma_2' = \Sigma_3$, while the proof of the inductive case is based on Theorem 4 and the induction hypothesis. $\square$

What if this potentially non-sequential action was the arrival of an exit signal? That will potentially terminate a process, which could have taken some internal steps. We note that with the $\longrightarrow^*$ reduction in the conclusion we do not say that the steps are preserved, thus the result node after the arrival of the exit signal can take fewer internal steps by leaving the steps for the terminated process out.

5 Program Equivalence

In this section, we investigate program equivalence using bisimulation. Bisimulations are relations between nodes that are preserved by the reduction steps.

Definition 7 (Bisimulation). *A relation R is a bisimulation if it satisfies the following two properties:*

- *For all nodes $\Sigma_1, \Sigma_2, \Sigma_1'$, process identifiers ι, and actions a, if $(\Sigma_1, \Sigma_2) \in R$ and $\Sigma_1 \xrightarrow{\iota:a} \Sigma_1'$, then there is a node Σ_2', which is reducible from Σ_2 with the action a: $\Sigma_2 \xrightarrow{\iota:a} \Sigma_2'$, and $(\Sigma_1', \Sigma_2') \in R$.*
- *For all nodes $\Sigma_1, \Sigma_2, \Sigma_2'$, process identifiers ι, and actions a, if $(\Sigma_1, \Sigma_2) \in R$ and $\Sigma_2 \xrightarrow{\iota:a} \Sigma_2'$, then there is a node Σ_1', which is reducible from Σ_1 with the action a: $\Sigma_1 \xrightarrow{\iota:a} \Sigma_1'$, and $(\Sigma_1', \Sigma_2') \in R$.*

We can show that equality satisfies the above conditions of being a bisimulation.

Theorem 7. *The equality of nodes is a bisimulation.*

Proof. This property is just a simple consequence of the definition of bisimulation. □

We also defined a relaxed variant: weak bisimulations omit τ actions taken in the semantics, so only communication actions should preserve the relation.

Definition 8 (Weak bisimulation). *A relation R is a weak bisimulation if it satisfies the following two properties:*

- *For all nodes $\Sigma_1, \Sigma_2, \Sigma_1'$, process identifiers ι, and actions $a \neq \tau$, if $(\Sigma_1, \Sigma_2) \in R$ and $\Sigma_1 \xrightarrow{\iota:a} \Sigma_1'$, then there are nodes $\Sigma_2^1, \Sigma_2^2, \Sigma_2'$, which are reducible from Σ_2 in the following way: $\Sigma_2 \longrightarrow^* \Sigma_2^1$, $\Sigma_2^1 \xrightarrow{\iota:a} \Sigma_2^2$, and $\Sigma_2^2 \longrightarrow^* \Sigma_2'$, and $(\Sigma_1', \Sigma_2') \in R$.*
- *For all nodes $\Sigma_1, \Sigma_2, \Sigma_1'$, process identifiers ι, and actions $a \neq \tau$, if $(\Sigma_1, \Sigma_2) \in R$ and $\Sigma_2 \xrightarrow{\iota:a} \Sigma_2'$, then there are nodes $\Sigma_1^1, \Sigma_1^2, \Sigma_1'$, which are reducible from Σ_1 in the following way: $\Sigma_1 \longrightarrow^* \Sigma_1^1$, $\Sigma_1^1 \xrightarrow{\iota:a} \Sigma_1^2$, and $\Sigma_1^2 \longrightarrow^* \Sigma_1'$, and $(\Sigma_1', \Sigma_2') \in R$.*

Bisimulations satisfy the natural property of being weak bisimulations.

Theorem 8. *Bisimulations are weak bisimulations.*

Proof. This property is also a simple consequence of the definitions, since we can choose 0-step reductions for $\Sigma_2 \longrightarrow^* \Sigma_2^1$ and $\Sigma_2^2 \longrightarrow^* \Sigma_2'$ in Definition 8, while the middle step $\Sigma_2^1 \xrightarrow{\iota:a} \Sigma_2^2$ is obtained from Definition 7. □

We consider two programs (Σ, Σ') equivalent if there is a relation R that is a weak bisimulation and $(\Sigma, \Sigma') \in R$. Next, we prove that sequential evaluation is a weak bisimulation. For this proof we used the chaining properties (Theorem 5 and Theorem 6) of the semantics.

Theorem 9. $\longrightarrow^*$ *(between nodes) is a weak bisimulation.*

Proof. To avoid ambiguity, we use Λ to denote the available nodes in the proof, while we keep Σ for the definitions. To prove that a relation is a weak bisimulation, two properties need to be proved:

- For the first part of Definition 8 we have $\Lambda_1 \longrightarrow^* \Lambda_2$ and $\Lambda_1 \xrightarrow{\iota:a} \Lambda_1'$ as assumptions. We can chain the reduction determined by a to the end of the sequential reduction sequence by Theorem 5 ($\Lambda_2 \xrightarrow{\iota:a} \Lambda_3$ for some Λ_3). Actually, the second possible conclusion (i.e. the $a = \tau$) of this theorem can not occur here, because of the restriction $a \neq \tau$ in Definition 8. We need to prove that $\Lambda_2 \longrightarrow^* \Sigma_2^1$, $\Sigma_2^1 \xrightarrow{\iota:a} \Sigma_2^2$, $\Sigma_2^2 \longrightarrow^* \Sigma_2'$, and $\Lambda_1' \longrightarrow^* \Sigma_2'$ for suitable Σ nodes. We can choose $\Sigma_2^1 = \Lambda_2$, $\Sigma_2^2 = \Lambda_3$, and $\Sigma_2' = \Lambda_3$, thus the first and second $\longrightarrow^*$ reductions are 0-step reductions, while the single-step reduction is among the assumptions. The reduction $\Sigma_1' \longrightarrow^* \Sigma_2'$ remains, which can be proved by Theorem 6.
- To satisfy the second part of Definition 8 we have $\Lambda_1 \longrightarrow^* \Lambda_2$ and $\Lambda_2 \xrightarrow{\iota:a} \Lambda_2'$ as assumptions. We need to prove $\Lambda_1 \longrightarrow^* \Sigma_1^1$, $\Sigma_1^1 \xrightarrow{\iota:a} \Sigma_1^2$, and $\Sigma_1^2 \longrightarrow^* \Sigma_1'$, and $\Sigma_1' \longrightarrow^* \Lambda_2'$ for suitable nodes. We choose $\Sigma_1^1 = \Lambda_2$, $\Sigma_1^2 = \Lambda_2'$, and $\Sigma_1' = \Lambda_2'$, thus the first two reductions are among the assumptions, while the third and fourth ones are 0-step reductions.

□

With this proof, we can state that a node is equivalent to the nodes to which it reduces by using sequential steps only. For example, we can derive the following property:

Example 5. For all nodes Π, ethers Δ, frame stacks K, process identifiers ι, mailboxes q, list of process identifiers *pl*, and process flags *flag*, the following nodes are equivalent (where *mm* denotes the function expression inside `letrec` in Example 1, and f denotes the successor function from Example 2).

$$(\Delta, \iota : (K, \texttt{letrec 'mm'/2} = mm \texttt{ in apply 'mm'/2}(f, \texttt{[0,1,2]}), q, pl, \mathit{flag}) \parallel \Pi)$$

$$(\Delta, \iota : (K, \texttt{[1,2,3]}, q, pl, \mathit{flag}) \parallel \Pi)$$

Proof. We have already shown in Example 2 how the complex `letrec` expression can be reduced to a list of values. Using this fact together with Theorem 9 we can prove this equivalence (note that the sequential steps of the evaluation can be lifted to the inter-process level with rules Seq and NOther). □

There is a natural question, whether any evaluation sequence could be proved to be a bisimulation. Unfortunately, that is not the case.

Theorem 10. *For all l action traces, $\xrightarrow{l}$ is not a weak bisimulation.*

Proof. We can prove this theorem by providing a counterexample. Here, we just give the idea of it: consider the process (with identifier 0) that evaluates `let X =`

`0 in X`. This process terminates in two sequential steps according to the semantics. By the definition of the weak bisimulation, taking a reduction step determined by any action (specifically for $\mathit{arr}(0, 1, \mathit{exit}(\texttt{'kill'}, \texttt{'false'})))$, the result configurations should be reducible to each other (by two sequential steps). `let X = 0 in X` evaluates to a dead process with the action above, which naturally could not be reduced to anything by sequential steps. □

In this section we defined program equivalence based on bisimulations and proved that sequential evaluation is a bisimulation. With the help of these definitions and theorems, we can establish the equivalence of simple programs. As we noted at the start of the section, all of the definitions and theorems presented here are formalised in the Coq proof assistant [9]. We plan to further investigate bisimulations to enable reasoning about more complex programs.

6 Related Work

The results presented in this paper are extensions to our work on sequential Core Erlang [2–4, 19]. We mainly based the concurrent semantics on the work of Fredlund [15], Harrison [18], and Lanese et al. [23]. The general idea of an interaction semantics of actor languages is described in the work of Mason and Talcott [24].

The formalisation of Fredlund [15] is the most detailed regarding both the sequential and concurrent parts of Erlang, which also faithfully follows the documentation of Erlang [12]. However, it considers signal transfer as an atomic operation (i.e. when a signal is sent, it immediately arrives), while according to signal ordering guarantee [13], the order of the signals sent from an entity to the same entity is preserved, which means that two signals that are targeting different entities can arrive in arbitrary order. The semantics of Fredlund [15] differentiates active and passive termination signals, while we denote these by the link flag of the exit signal.

Moreover, the work of Fredlund [15] differentiates only three actions on the inter-process level semantics: input, output, and silent. This approach closely follows the general idea of the interaction semantics [24]. With our approach, we can simulate input and output actions: *send* actions can be considered as output actions, *arr* actions are the input actions, while every other action can be regarded as silent. The advantage of our semantics is that we can distinguish more classes of reduction sequences, moreover, we also exploit this property: the theorems discussed in Section 4.2 and Section 5 involving sequential reductions would not hold, if other actions (e.g. *flag*) were considered as silent.

Lanese et al. [23] describe their results on bisimulations, and prove a number of system equivalences (e.g. renaming, normalisation). They also use ethers to store messages, however, their approach (deliberately) ignores the guarantee for signal ordering [13]. Moreover, they do not formalise signals except messages, and used only a small subset of Core Erlang. Still, we incorporated some of their ideas in the formalisation of program equivalence, and plan to pursue this topic more in detail.

The work of Vidal et al. [21, 22, 27, 31] is also related to ours, they define multiple semantics (reduction semantics and small-step semantics) for Core Erlang to express reversible computation. The language they formalised has a similar coverage to our formalisation, they also formalised concurrent semantics with an ether and action traces, moreover, they also proved similar theorems about the properties. However, their formalisations do not include signals except messages.

Harrison [18] presented a formalisation of a minimal subset of Core Erlang in his paper, which has also been formalised in Isabelle. His formalisation techniques aided us while creating a usable Coq definition of the concurrent semantics, however, his formalisation includes only a few of the language elements, and he too treated signal transfer as an atomic operation.

An important advantage of our formalisation compared to most of the existing ones is that it is also implemented in Coq in an open-source project. Most of the existing works are paper-based formalisations or the machine-checked version is no longer available to the public. Furthermore, our semantics implements the signal ordering guarantee [13] more faithfully than the other discussed approaches.

7 Conclusion and Future Work

In this paper, we described our three-level, modular formal semantics for concurrent Core Erlang. We discussed a number of theorems about the determinism and confluence properties of the semantics, defined bisimulations to be able to reason about program equivalence, and proved that side-effect-free evaluations of a program provide equivalent programs. Finally, we compared our approach with the results of other authors. The formalisation has also been implemented as an open-source project in the Coq proof assistant [9].

In the future we are planning to further extend this formalisation. Our future goals include the following points:

- Investigating bisimulations in more depth, potentially by following a similar path to Lanese *et al* [23] who defined barbed congruence, that enabled them to develop a proof technique to effectively reason about program equivalence.
- Proving the equivalence between more complex examples of concurrent programs equivalent, as well as investigating equivalence between sequential and concurrent algorithms.
- Extending the semantics to cover exceptions and other side effects (e.g. input-output) based on our previous results [3].
- Implementing a formalisation of the module system within this semantics.
- In the longer term, an extensive, usable formalisation of Erlang is our ultimate goal.

References

[1] Agha, G. and Hewitt, C. Concurrent programming using actors: exploiting large-scale parallelism. In Bond, A. and Gasser, L., editors, *Readings in Distributed Artificial Intelligence*, pages 398–407. Morgan Kaufmann, 1988. DOI: 10.1016/B978-0-934613-63-7.50042-5.

[2] Bereczky, P., Horpácsi, D., and Thompson, S. A proof assistant based formalisation of a subset of sequential Core Erlang. In Byrski, A. and Hughes, J., editors, *Trends in Functional Programming*, pages 139–158, Cham, 2020. Springer International Publishing. DOI: 10.1007/978-3-030-57761-2_7.

[3] Bereczky, P., Horpácsi, D., and Thompson, S. Machine-checked natural semantics for Core Erlang: exceptions and side effects. In *Proceedings of Erlang 2020*, page 1–13. ACM, 2020. DOI: 10.1145/3406085.3409008.

[4] Bereczky, P., Horpácsi, D., Kőszegi, J., Szeier, S., and Thompson, S. Validating formal semantics by property-based cross-testing. In *Proceedings of the 32nd Symposium on Implementation and Application of Functional Languages (IFL '20)*, pages 150–161. ACM, New York, NY, USA, 2021. DOI: 10.1145/3462172.3462200.

[5] Blazy, S. and Leroy, X. Mechanized semantics for the Clight subset of the C language. *Journal of Automated Reasoning*, 43(3):263–288, 2009. DOI: 10.1007/s10817-009-9148-3.

[6] Carlsson, R., Gustavsson, B., Johansson, E., Lindgren, T., Nyström, S.-O., Pettersson, M., and Virding, R. Core Erlang 1.0.3 language specification. Technical report, 2004. URL: https://www.it.uu.se/research/group/hipe/cerl/doc/core_erlang-1.0.3.pdf.

[7] Cesarini, F. and Thompson, S. *Erlang programming*. O'Reilly Media, Inc., 1st edition, 2009. URL: https://www.oreilly.com/library/view/erlang-programming/9780596803940/.

[8] Core Erlang formalization. URL: https://github.com/harp-project/Core-Erlang-Formalization, 2022. Accessed on 20th of September, 2022.

[9] Core Erlang mini. URL: https://github.com/harp-project/Core-Erlang-mini/releases/tag/v1.6, 2022. Accessed on 20th of September, 2022.

[10] de Bruijn, N. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75(5):381–392, 1972. DOI: 10.1016/1385-7258(72)90034-0.

[11] Erlang/OTP compiler, version 24.0. URL: https://www.erlang.org/patches/otp-24.0. Accessed on 30th of September 2022.

[12] Erlang documentation. URL: https://www.erlang.org/docs, 2022. Accessed on 20th of September, 2022.

[13] Erlang documentation, Processes. URL: https://www.erlang.org/doc/reference_manual/processes.html, 2022. Accessed on 20th of September, 2022.

[14] Felleisen, M. and Friedman, D. Control operators, the SECD-machine, and the λ-calculus. In *Formal Description of Programming Concepts - III: Proceedings of the IFIP TC 2/WG 2.2 Working Conference on Formal Description of Programming Concepts - III*, pages 193–222, 1987.

[15] Fredlund, L.-Å. *A framework for reasoning about Erlang code.* PhD thesis, Mikroelektronik och informationsteknik, 2001. URL: https://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-3210.

[16] Gumbs, K. The core of Erlang. URL: https://8thlight.com/blog/kofi-gumbs/2017/05/02/core-erlang.html, 2017. Accessed on 17th of March, 2022.

[17] High-Assurance Refactoring Project. URL: https://github.com/harp-project, 2023. Accessed on 27th of March 27th, 2023.

[18] Harrison, J. Towards an Isabelle/HOL formalisation of Core Erlang. In *Proceedings of the 16th ACM SIGPLAN International Workshop on Erlang*, Erlang 2017, page 55–63, New York, NY, USA, 2017. Association for Computing Machinery. DOI: 10.1145/3123569.3123576.

[19] Horpácsi, D., Bereczky, P., and Thompson, S. Program equivalence in an untyped, call-by-value functional language with uncurried functions. *Journal of Logical and Algebraic Methods in Programming*, 132:100857, 2023. DOI: 10.1016/j.jlamp.2023.100857.

[20] Kőszegi, J. KErl: Executable semantics for Erlang. *CEUR Workshop Proceedings*, 2046:144–160, 2018. URL: http://ceur-ws.org/Vol-2046/koszegi.pdf.

[21] Lanese, I., Nishida, N., Palacios, A., and Vidal, G. A theory of reversibility for Erlang. *Journal of Logical and Algebraic Methods in Programming*, 100:71–97, 2018. DOI: 10.1016/j.jlamp.2018.06.004.

[22] Lanese, I., Palacios, A., and Vidal, G. Causal-consistent replay reversible semantics for message passing concurrent programs. *Fundamenta Informaticae*, 178(3):229–266, 2021. DOI: 10.3233/FI-2021-2005.

[23] Lanese, I., Sangiorgi, D., and Zavattaro, G. Playing with bisimulation in Erlang. In Boreale, M., Corradini, F., Loreti, M., and Pugliese, R., editors, *Models, Languages, and Tools for Concurrent and Distributed Programming*, pages 71–91. Springer, Cham, 2019. DOI: 10.1007/978-3-030-21485-2_6.

[24] Mason, I. and Talcott, C. Equivalence in functional languages with effects. *Journal of Functional Programming*, 1(3):287–327, 1991. DOI: 10.1017/S0956796800000125.

[25] Mosses, P. Formal semantics of programming languages: — An overview —. *Electronic Notes in Theoretical Computer Science*, 148(1):41–73, 2006. DOI: 10.1016/j.entcs.2005.12.012, Proceedings of the School of SegraVis Research Training Network on Foundations of Visual Modelling Techniques (FoVMT 2004).

[26] Neuhäußer, M. and Noll, T. Abstraction and model checking of Core Erlang programs in Maude. *Electronic Notes in Theoretical Computer Science*, 176(4):147–163, 2007. DOI: 10.1016/j.entcs.2007.06.013, Proceedings of the 6th International Workshop on Rewriting Logic and its Applications (WRLA 2006).

[27] Nishida, N., Palacios, A., and Vidal, G. A reversible semantics for Erlang. In Hermenegildo, M. and Lopez-Garcia, P., editors, *International Symposium on Logic-Based Program Synthesis and Transformation*, pages 259–274, Cham, 2017. Springer, Springer International Publishing. DOI: 10.1007/978-3-319-63139-4_15.

[28] Owens, S., Myreen, M., Kumar, R., and Tan, Y. Functional big-step semantics. In Thiemann, P., editor, *Programming Languages and Systems*, pages 589–615. Springer Berlin Heidelberg, 2016. DOI: 10.1007/978-3-662-49498-1_23.

[29] Pitts, A. and Stark, I. Operational reasoning for functions with local state. *Higher order operational techniques in semantics*, pages 227–273, 1998. DOI: 10.5555/309656.309671.

[30] Schäfer, S., Tebbi, T., and Smolka, G. Autosubst: Reasoning with de Bruijn terms and parallel substitutions. In Urban, C. and Zhang, X., editors, *Interactive Theorem Proving*, pages 359–374, Cham, 2015. Springer International Publishing. DOI: 10.1007/978-3-319-22102-1_24.

[31] Vidal, G. Towards symbolic execution in Erlang. In Voronkov, A. and Virbitskaite, I., editors, *International Andrei Ershov Memorial Conference on Perspectives of System Informatics*, pages 351–360, Berlin, Heidelberg, 2015. Springer, Springer Berlin Heidelberg. DOI: 10.1007/978-3-662-46823-4_28.

[32] Wand, M., Culpepper, R., Giannakopoulos, T., and Cobb, A. Contextual equivalence for a probabilistic language with continuous random variables and recursion. *Proceedings of the ACM on Programming Languages*, 2(ICFP), 2018. DOI: 10.1145/3236782.

Acta Cybernetica **26** (2024) 405–429.

Identifying Client-Server Behaviours in Legacy Erlang Systems*

Zsófia Erdei[ab], Melinda Tóth[ac], and István Bozó[ad]

Abstract

In Erlang, behaviours are special forms of design patterns. There are many benefits to using behaviours. For example, behaviours can help abstract away the most common parts when solving similar problems. Design pattern recognition may help understand the source code of the software. It can provide structured information about the purpose of specific parts and the design decisions behind the implementation. For object-oriented languages, several tools exist that use different approaches and methods to identify design patterns. We present a method for identifying source code fragments in legacy Erlang systems amenable to transforming into client-server Erlang design patterns. In our analysis, we identify the base set of server candidates using concurrent process analysis and narrow down the result using further static analysis knowledge using the RefactorErl framework.

Keywords: Erlang, design patterns, client-server behaviour, concurrent behaviours, static analysis

1 Introduction

Design patterns are developed best practices that provide general, reusable solutions to common problems. There are several benefits to using design patterns. Their use promotes transparent and easy-to-maintain code, reduces the possibility of errors, and speeds up the development process. Design patterns are not specific to any programming language, they are general solutions that can be implemented in many programming languages. However, most design patterns are designed for

*Supported by EFOP-3.6.3-VEKOP-16-2017-00001: Talent Management in Autonomous Vehicle Control Technologies – The Project is supported by the Hungarian Government and co-financed by the European Social Fund. Application Domain Specific Highly Reliable IT Solutions project has been implemented with the support provided from the National Research, Development and Innovation Fund of Hungary, financed under the Thematic Excellence Programme TKP2020-NKA-06 (National Challenges Subprogramme) funding scheme.

[a]Eötvös Loránd University, Budapest, Hungary
[b]E-mail: zsanart@inf.elte.hu, ORCID: 0000-0002-5089-4984
[c]E-mail: toth_m@inf.elte.hu, ORCID: 0000-0001-6300-7945
[d]E-mail: bozo_i@inf.elte.hu, ORCID: 0000-0001-5145-9688

DOI: 10.14232/actacyb.299529

object-oriented environments. In object-oriented programming, a program design pattern typically depicts the relationships between objects, and documents the inheritance, association, and aggregation relationships in design.

Design patterns can be broadly divided into three categories [5]. Structural patterns are used to define relationships between classes, creation patterns are used to represent the instantiation process, and behavioural patterns are used to describe communication and interaction between objects. Recognising design patterns cannot only help to understand the source code of software but also provide information about the purpose of specific parts of the system and the design decisions behind the implementation. Manually searching for design patterns in larger software is an extremely cumbersome and time-consuming task. For this reason, a large number of methodologies, approaches and tools have been proposed for detecting design patterns and accordingly transforming the source code [21, 9, 22].

In the case of distributed software, supporting the understanding and transformation of source code with software tools is an even more critical task. Erlang [6] is a programming language specifically designed to build fault-tolerant, distributed systems that can contain a large number of concurrent processes. In software written in Erlang, many processes can have similar structures and behaviours. The formalization of these patterns is called behaviour. Using behaviours makes it easier to read, understand and maintain legacy codes. Improvised programming structures, while possibly more efficient, are always more difficult to understand. In the Erlang/OTP libraries, there are several common concurrent design patterns implemented. The programmer only needs to implement a callback module to define the specific behaviour. Using behaviours also enables/makes it possible to use verification tools and techniques developed for Erlang [4]. A simple example of the pre-implemented design patterns included in the Erlang/OTP is the implementation of a client-server behaviour, the so-called *gen_server* behaviour. The client-server model consists of a central server process and an arbitrary number of clients. Its most common application is resource management, where multiple clients share a common resource. The server is then responsible for managing this resource.

Identifying design patterns manually is not efficient. Therefore, various approaches have been developed to automatise this process. Static analysis-based methods are widely used in this domain. The goal of our work is to identify source code fragments in legacy Erlang systems that are amenable to transforming into concurrent Erlang behaviours. This paper presents our first result in order to achieve this, namely to analyse and to identify the client-server behaviour candidates. We base our work on the static source code analysis and transformation tool RefactorErl.

RefactorErl [19] is a static analyser and transformer tool for Erlang. The tool uses static code analysis techniques and provides a wide range of features, like dataflow analysis, dynamic function call detection, side-effect analysis, a user-level query language to gather semantic information or structural complexity metrics about Erlang programs, dependency examination among functions or modules, function call graph with information about dynamic calls, etc.

We propose a two-staged behaviour recognition analysis. At first, we use the communication graph of RefactorErl [20] to find process candidates based on the communication pattern. In the second stage, we filter those elements by a predefined rule set to check the internal structure of the process.

The paper is structured as follows. In Section 2 we discuss related works, where we present multiple methods designed to recognise design patterns in object-oriented programming languages. In Section 3 we first introduce Erlang, its advantages and some of the features of the language that can be used to develop highly scalable soft real-time systems. We describe the characteristics of server processes and present the basic architecture of the client-server behaviour. In Section 4 with the use of an example, we present the structure and operation of an Erlang server process and how it has similar properties to the *gen_server* behaviour included in the Erlang/OTP library. We also show a few examples of Erlang processes that have similar characteristics to server processes but could not be implemented with *gen_server* behaviour. In Section 5 we introduce the RefactorErl tool, and in Section 6 we present a method to identify the client-server behaviour in Erlang programs based on static analysis. Finally, Section 7 presents the first results of the prototype implementation and Section 8 summarises our results.

2 Related work

Different approaches can be used to identify design patterns, both in terms of the method of identification (searching for the components of samples or recognising the full structure of a design pattern) and the type of analysis (static or dynamic analysis). Methods based on static analysis are based on the analysis of the source code, while dynamic analysis collects information while the software is running. Information obtained only from the static or dynamic analysis is seldom sufficient to effectively recognize most design patterns, so many methods work with a hybrid solution. Recognition methods can be broadly classified into several categories: those that rely on database queries, those that use metrics, those that utilize graph or matrix representations, those that are based on the Unified Modeling Language (UML), and those that combine multiple of these techniques [1].

The paper [10] proposes a solution using metrics and a machine learning algorithm for recognising micro-architectures similar to design patterns in the architecture. This can be used to better understand the design problems solved by software developers when designing the program architecture. Fingerprints are sets of metric values characterising classes playing a given role. These fingerprints are based on a set of external attributes that help categorise classes and can be used to reduce the search space of micro-architectures similar to design motifs. The fingerprints were created based on a rule-learner algorithm that inferred rules characterising design motifs' roles with the metric values of the classes playing these roles. The identification process described in the paper consisted of two steps: identifying candidate classes for design patterns by eliminating classes that did not match the expected fingerprint and identifying candidate classes for the remaining roles starting from key-role candidates and using structural matching.

A method based on both static and dynamic analysis was presented in [12] for automatic design pattern recognition in Java. They use static analysis to compute the potential program parts playing a certain role in a design pattern and dynamic analysis to further examine those candidates. The static analysis reads the source code and constructs an attributed AST, then computes the pattern relation on the AST nodes and provides a result as a set of candidates consisting of tuples of AST nodes. The dynamic analysis takes this set as an input and monitors the execution of the nodes. Depending on the node's unique role, dynamic test actions are executed on the object sets of the candidates. In the paper, different approaches for detecting the Observer, Composite, Mediator, Chain of Responsibility and Visitor Patterns are discussed.

Another hybrid method for recognising design patterns is presented in [2]. They developed the software prototype JADEPT (JAva DEsign Pattern deTector) for design pattern recognition based on a predefined set of rules describing properties that may be either structural or behavioural and may define relationships between classes or families of classes. Weights have been associated with rules indicating how much a rule is able to describe a specific property of a given design pattern. JADEPT collects structural and behavioural information through dynamic analysis of Java software by exploiting JPDA (Java Platform Debugger Architecture) and stores the extracted information in a database. A rule is implemented by one or more queries and the existence of a design pattern can be verified through the validation of its associated rules.

Li and Thompson's paper [15] presents a technique for detecting and eliminating similar code in Erlang programs. The technique involves analysing the abstract syntax trees (ASTs) of the source code, computing a similarity measure between the ASTs, and then merging the similar code into a single function.

The authors argue that similar code is a common problem in Erlang programs, and that it can lead to maintenance and readability issues. They propose a solution that involves using a combination of structural and lexical analysis to identify similar code, and then using a technique called "code merging" to eliminate the duplication.

Duplicated code detection and design pattern recognition are two related but distinct techniques used in software development. Duplicate code is code that is identical or very similar to other code in the system mostly caused by copy-pasting and reusing already existing code. While two code snippets that implement the same design pattern can be very different even in the AST, in the case of duplicated code we usually expect them to closely match. While the examination of AST alone is not sufficient to detect design patterns, the recognition of certain similarities could help to filter out results.

The structure of parallel computations in a program can be defined conveniently, and at a high level of abstraction, using parallel design patterns. Algorithmic skeletons [7] implement common patterns of parallelism, allowing the programmer to instantiate parallel skeletons with application-specific code fragments. PaRTE [18] integrates capabilities of the RefactorErl and Wrangler [16] refactoring/program analysis tools into a parallelisation framework that can be used to identify parallel patterns and determine the best implementations of those patterns.

Some well-known patterns are pipe (parallel pipeline) and task farm (applies a given function to a sequence of independent inputs in parallel), the map-reduce and the divide-and-conquer patterns. Skel [17] is a library of algorithmic skeletons for Erlang, providing a small number of useful, classical skeletons. Since both pipe and farm can be defined to operate on lists of inputs, the analysis described in the paper focuses on identifying certain operations on lists, and also on identifying those data structures that can be transformed to lists.

The tool developed by our research group targets three constructs: list comprehensions, library calls and recursive functions. List comprehensions are categorised based on the output expression as a possible farm or pipe candidate. Calls to functions that exhibit a map-like or pipeline-like behaviour over a sequence of data and certain map-like and pipeline-like recursive functions can also be transformed into farms or pipes. These patterns can be identified based on the syntax of the code but not all code fragments that match a pattern can be safely executed in parallel. In order to guarantee that the transformations preserve the semantics of the program further semantic analysis is required.

The transformation process itself comprises two distinct phases, an initial program shaping phase and the actual transformation into instances of skeletons. The role of program shaping is to prepare the source code for the introduction of parallel skeletons since it is necessary to shape the code into an appropriate canonical form before the transformations can be applied.

The use of PaRTE is demonstrated on a simple worked example, showing how the tool can be used to transform a sequentially implemented image merge to a parallel version and automatically obtain significant and scalable speedups over the original version.

3 Modelling client-server behaviour in Erlang

Our research focuses on legacy Erlang systems. In this section, we start with introducing Erlang. Then we specify the properties of a general server process structure implemented in Erlang, and extend these properties to express the client-server behaviour.

3.1 Erlang

Erlang [6] is a general-purpose, dynamically typed, concurrent functional programming language, which enables developers to write highly scalable, soft real-time systems. Erlang was originally designed for developing telecommunication software, since then, it is also widely used in the world of banking, chat services, and database management systems. Due to its robustness and fault tolerance, it is suitable for the development of large-scale distributed systems.

One of the main advantages of using Erlang instead of other functional languages is Erlang's ability to handle concurrency and distributed programming. In Erlang, the unit of concurrency is the process. A process is a lightweight task that runs concurrently and is independent of the other processes. Processes do not share memory, the only way for processes to interact with each other is through message passing, where the message can be any Erlang term. Message passing is asynchronous, so the process can continue processing once a message is sent. Each process has its own input queue for messages it receives. New messages received are put at the end of the queue. Received messages can be processed selectively, it is not necessary to handle messages in the order of arrival. When a process executes a receive, the first message in the queue is matched against the first pattern in the receive. If it does not match, the next branch of the receive is matched. It is repeated until the first match is found. If none of the patterns matches, the next message from the message queue is examined. Once a match is found, the message is removed from the queue and the actions corresponding to the pattern are executed. If none of the messages match, the process is blocked until a matching message arrives.

Every process in Erlang is identified by a unique process identifier (*PID*). The Erlang *BIF* (Built-In Function) *spawn* is used to create a new process:

spawn(Module, Exported_Function, List_of_Arguments).

This function creates a new process executing the function *Exported_Function* of the module *Module* with the given list of arguments.

Processes can be registered with a given name using the built-in function call *register(Name, Pid).* After registration, the process can be addressed, either with its PID or with its registered name. The process can be unregistered with the built-in function *unregister(Name).* The registered process is automatically unregistered when the process terminates.

3.2 Server processes

When implementing client-server behaviour, clients and servers are represented as Erlang processes. Processes – including server processes – have common characteristics and follow similar patterns. As a result of this, they share a similar code base. Regardless of their function, processes must first be spawned and possibly initialised. A process can be addressed by using its PID, but a server process is also usually registered. After that, we have to initialise the state of the process. The state is specific to the function of our process. This step handles all initialisation of data required for the main loop function to work well. Once the process has been initialised, it is ready to communicate with other processes. The main loop is a tail-recursive function that handles the events, it usually has a receive block and is used to process messages and send replies. A special message can be used to terminate the process when needed. In the case of a normal termination when necessary a cleanup procedure is completed.

Figures 1 and 2 show a general server process skeleton – a general, reusable structure for implementing a process in a concurrent or distributed system. It

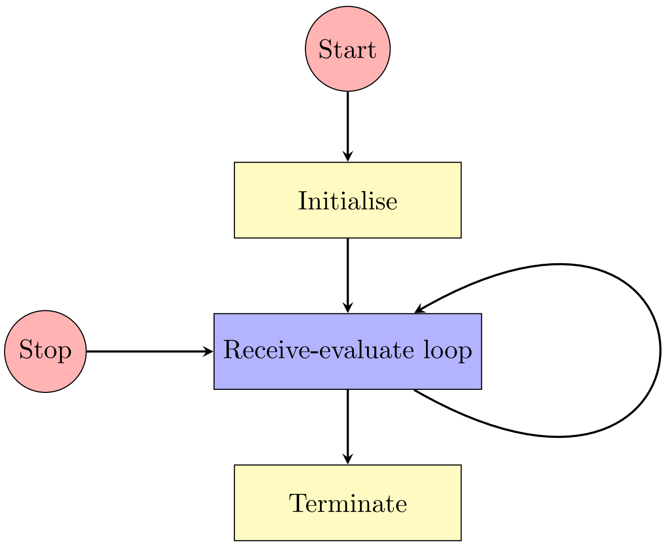


Figure 1: A process skeleton

```
start(Args) ->
    register(server, spawn_link(?MODULE, init, [Args])).

stop() -> server ! stop.

init(Args)->
    InitState = initialise_state(Args),
    loop(InitState).

loop(State)->
    receive
        stop -> terminate(State);
        {handle, Msg} ->
            NewState = handle_req(Msg, State),
            loop(NewState)
    end.
```

Figure 2: A server process skeleton source

typically includes the basic components and structure that are common to many types of processes, such as initialisation, message handling, and termination. A general process skeleton typically includes the following components:

- **Initialisation**: This is the first step in the process, where the process is set up and initialised. This may include allocating resources, setting up data structures, and starting any other necessary sub-processes.
- **Message handling**: This is the core component of the process, where the process waits for and handles incoming messages. This may include performing calculations, updating data structures, and sending messages to other processes.
- **Termination**: This is the final step in the process, where the process is cleaned up and resources are deallocated. This may include stopping sub-processes and closing any open files or connections.

While the general process skeleton fits server processes of the client-server behaviour very well, the communication pattern of this behaviour is unique (see Figure 3). In the next section, we describe the main characteristics of the client-server process architecture and its implementation in Erlang.

3.3 Client-server behaviour

The client-server model describes a way of distributing tasks and services within a network. It is characterised by a central server and an arbitrary number of clients. The client-server model is often used for resource management operations, where several different clients share a resource. Clients implemented as Erlang processes use these resources by sending the server requests.

Figure 3 shows the typical process architecture and communication of a client-server model. Most often there are multiple instances of the client and a single server. The server process receives requests, handles them, and can respond with an acknowledgement and a return value if the request was successful, or with an error if the request did not succeed. Interaction between them takes place through message sending and receiving. If a client using the service or resource handled by the server expects a reply to the request, the call to the server has to be synchronous. If the client does not need a reply, the call to the server can be asynchronous.

The *gen_server* behaviour module provides the server of a client-server relation. A generic server process (*gen_server*) implemented with this behaviour has a standard set of interface functions and includes functionality for tracing and error reporting. The process can be divided into a generic part (a behaviour module) and a specific part (a callback module). The behaviour module contains all the generic functionality reused from one implementation to another. The specific parts of the process implemented by the user are located in the callback module exporting a predefined set of functions.

4 Motivating example

In Figure 4 an example server implementation is shown. In Figure 5 an equivalent server implementation is presented using the generic server library of Erlang/OTP.

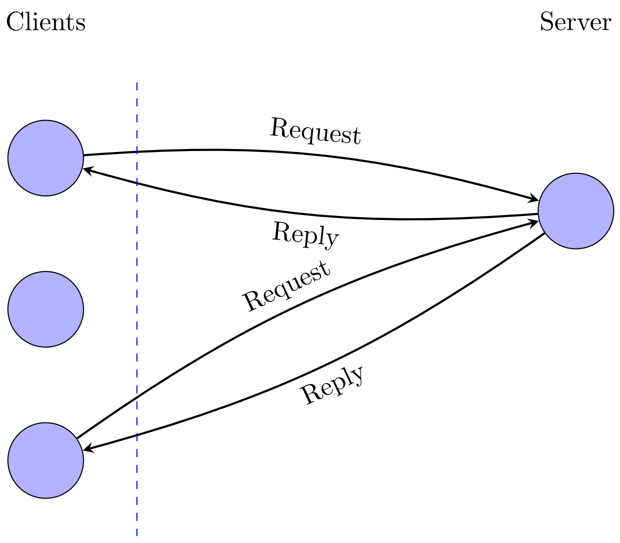


Figure 3: The client-server process architecture

The goal of our work is to identify codes similar to the code fragment shown in Figure 4 as a candidate and later transform them into an application of a client-server design pattern, as presented in the example in Figure 5.

In Figure 4 a simple job server is demonstrated that waits for {*job, ReqId,* {*M, F, A*}} messages where the third element of the tuple represent a function by the name of the implementing module, the name of the function and the list of arguments of the function call to be evaluated. The server spawns a new worker process to evaluate the function. The worker sends the calculated result to the server as a message tagged with the *result* atom. The server adds the result to its state when a *result* message arrives. Once a *reply* message arrives from a client, the server searches the result in its state and sends the value to the client. If the result is not ready yet, the server sends a *pending* answer. The *stop* message terminates the server.

As the simple server implementation example shows, the server module provides interface functions for starting (*start/0*) and stopping (*stop/0*) the server process. The *start* function spawns and registers the process with the name *jobsrv*. The server implements the function *init/0* to initialise the server process with a state and the iterating function (*loop/1*) that receives messages and performs the requested tasks. Creating a process that calls the *init/1* function can be generic. The arguments passed to the call and the implementation of the function that initialises the main loop are specific to the task. The function *loop(State)* stores the connected clients in the server state variable (*State*) and handles incoming messages

```
-module(server).
-export([start/0, stop/0]).

start() -> register(jobsrv, spawn(fun init/0)).

stop() ->
  jobsrv ! stop.

init() ->
  loop(#{}).

loop(State) ->
  receive
    stop ->
      io:format("Server stopped in state: ~p~n", State);
    {job, ReqId, {M, F, A}} ->
      spawn(fun() ->
                jobsrv ! {result, ReqId, apply(M, F, A)}
            end),
      loop(State);
    {reply, ReqId, To} ->
      case State of
        #{ReqId:=Data} -> To ! {final, ReqId, Data};
                     _ -> To ! {pending, ReqId}
      end,
      loop(State);
    {result, ReqId, Result} ->
      loop(State#{ReqId => Result})
  end.
```

Figure 4: Non gen-server implementation

in the receive block. All incoming messages are matched against the patterns in the receive and the corresponding branch is executed. Storing the loop data in between calls is the same from one process to another, but the loop data itself will be different depending on the function of the process. Sending requests to the server process will also be generic, but the types and contents of the messages and how they are handled will differ depending on the task. While each response is specific, the method of sending it back to the client process is handled in a generic way. When a *stop* message is received, the process calls the terminate function, which is responsible for a clean termination. While sending a stop messageis generic, the steps to clean up the state prior to termination will be specific.

The previously described generic parts of the server could be separated and reused for many different applications, and only the specific parts of the code would have to be reimplemented. The Erlang/OTP module solves this with the *gen_server* library that formalizes the general server behaviour of a client-server model. The *gen_server* module contains the generic part of server implementation and the user only has to implement a callback module to define the specific behaviour.

```
-module(gserver).
-export([start/0, stop/0, init/1,
         handle_call/3, handle_cast/2, handle_info/2]).
-behaviour(gen_server).

start() ->
  gen_server:start({local, jobsrv}, gserver, [], []).

stop() ->
  gen_server:cast({local, jobsrv}, stop).

init(_) ->
  {ok, #{}}.

handle_cast({job, ReqId, {M, F, A}}, State) ->
  spawn(fun() ->
           jobsrv ! {result, ReqId, apply(M, F, A)}
         end),
  {noreply, State};
handle_cast(stop, State) ->
  io:format("Server stopped in state: ~p~n", State),
  {noreply, stop, State}.

handle_info({result, ReqId, Result}, State) ->
  {noreply, State#{ReqId => Result}}.

handle_call({reply, ReqId}, _, State) ->
  case State of
    #{ReqId:=Data} -> {reply, {final, ReqId, Data}, State};
    _ -> {reply, {pending, ReqId}, State}
  end.
```

Figure 5: Gen-server implementation

With the *gen_server* behaviour (Figure 5), instead of using the *spawn* and *spawn_link* BIFs, the *gen_server:start/4* and *gen_server:start_link/4* functions are used. In the example shown here, the server can be started by calling the *gserver:start()* function call, which then calls the *gen_server:start_link/4* function. This function spawns and links to the created server process. The first argument of the function specifies the name in the form of a tuple, in this case, {*local,jobsrv*}.

The server is then locally registered as jobsrv[1]. The second argument is the name of the callback module, which is the module where the callback functions are located. The third argument is a list of arguments that are passed to the *init/1* callback function when the process is started. Usually, lists are used to pass multiple arguments. These arguments can be used to initialise the process's state or to configure its behavior. Here, *init* does not need any data and ignores the argument. The fourth argument is a list of options, for example, it enables the user to set memory management flags as well as tracing and debugging flags. Most behaviour implementations, like in the example, just pass the empty list as an argument.

The *gen_server* start functions will spawn a new process that calls the *init/1* callback function from the callback module, with the arguments supplied. The task of the *init* function in the *gserver* module (Figure 5) is same as it was in the *server* module (Figure 4): to initialise the state of the server. Synchronous communication can be initialised by calling the *gen_server:call/2* function that sends a message to the server, while *gen_server:cast/2* calls are responsible for asynchronous communication. When a client sends a message to the server process by calling these functions, the *handle_call/3* or *handle_cast/2* callback function is called. The *handle_call/3* function is used to handle synchronous requests, where the client expects a reply. The *handle_cast/2* function is used to handle asynchronous requests, where the client does not expect a reply. Stopping the server can be handled synchronously or asynchronously by calling *gen_server:call/2* or *gen_server:cast/2*. Messages that are not sent to the server through *gen_server:call/2* or *gen_server:cast/2* function calls can be handled in the *handle_info/2* function definitions.

In our example, *job* messages and the stopping are asynchronous request, *reply* messages are synchronous, and the *result* messages sent from the worker processes are handled by the *handle_info* definition.

4.1 Other server-like processes

In Erlang, a process can be identified with its evaluating function. The process is created when we spawn the function and the process is alive until the evaluation of its function is finished. Long-living processes usually evaluate recursive functions. Thus when we are identifying server processes we need to identify recursive function definitions. However, we do not want to consider all recursive definitions which were spawned in the program as server processes. In this subsection, we will introduce a few counterexamples.

4.1.1 Taskfarm

Let us consider the code skeleton on Figure 6 that implements a parallel taskfarm in Erlang: where we want to evaluate a function on the elements of a list. We start a dispatcher, a collector process and some worker processes to evaluate the function depending on the number of available resources. The dispatcher and the

[1]The first argument can be omitted, so the server might not be registered. In this case, the process id of the newly created process could be used to refer to the server.

collector are registered processes. The dispatcher waits for *free* messages from the workers and sends an element of the list back. The collector waits for results from the workers and stores those in its state. It also notifies about the collected data on request. Workers are notifying the dispatcher if they are free to work and send the result of the computation to the collector process.

```
run(F, L) ->
    register(disp, spawn(fun() -> dispatcher(L) end)),
    register(coll, spawn(fun() -> collector([]) end)),
    N = erlang:system_info(logical_processors_available),
    [spawn(fun() -> worker(F) end) || _ <- lists:seq(1, N)].

dispatcher([H|T]) ->
    receive
        {free, Worker} -> Worker ! {data, H}, dispatcher(T)
    end;
dispatcher([]) ->
    receive
        stop -> terminate
    end.

collector(Acc) ->
    receive
        {result, Result} -> collector([Result | Acc]);
        {give_me, From} -> From ! Acc, collector(Acc)
    end.

worker(F) ->
    disp ! {free, self()},
    receive
        {data, Data} -> coll ! {result, F(Data)}, worker(F)
    end.
```

Figure 6: Parallel taskfarm implementation

The most important characteristic of a server process is a containing receive expression to handle messages from client processes and answering to them. If we consider only this condition, we might say that all the processes in this example could be server processes. However, we do not want to consider worker processes as server processes in client-server behaviour. We might have the expectation that a server process is unique in the system, it has some special role. Thus when we create multiple instances of an actor we do not want to consider those as servers. We can delete some processes from our server candidate list based on the context of the initialisation. The worker processes are spawned in a list comprehension, therefore we can assume that multiple occurrences exist, and thus we will not consider them.

The dispatcher process could be considered as a server where the clients are the worker processes. Later we might prioritise our candidate list and put functions like dispatcher at the end if we want our servers to do more work.

4.1.2 Timeout looping

Server processes have to be tail-recursive. However, we do not want to consider all of them. The code snippet on Figure 7 shows a process definition which waits for cancelling messages and terminates. Otherwise, it waits for T seconds and recursively calls itself if there is no more event to handle. We do not want to consider the recursion in the after branch of the receive expression as a proper server behaviour. It would not fit the client-server behaviour, thus we cannot transform it.

```
loop(S = #state{server=Server, to_go=[T|Next]}) ->
    receive
        {Server, Ref, cancel} -> Server ! {Ref, ok}
    after T*1000 ->
        if Next =:= [] -> Server ! {done, S#state.name};
           Next =/= [] -> loop(S#state{to_go=Next})
        end
    end.
```

Figure 7: An event handler [14]

4.1.3 Multiple recursive calls

Figure 8 contains a parallel implementation of the Fibonacci number calculation based on a caching optimisation to store the already calculated values. The cache process has no termination branch, it is an infinite recursive definition. However, we might consider it a server process. On the other hand, the process evaluating the fib function could not be considered a server process. It has multiple recursive calls in its body, thus it would not be possible to transform it into a *gen_server* behaviour-based process.

5 RefactorErl

The RefactorErl tool uses a directed, rooted graph, with typed nodes and edges as an internal representation to store the source code. The graph is called Semantic Program Graph (SPG) [13]. The SPG stores lexical, syntactic and semantic information about the source code, calculated by various static semantic analysers. Every module, function, and expression is a node with a unique identifier and a set of properties. Figure 9 shows a small part of a generated SPG. Besides the

```
fib(0) -> 1;
fib(1) -> 1;
fib(N) when is_integer(N), N > 1 ->
    cache ! {fib, N, self()},
    receive
        {value, N, FibN} -> FibN;
        no_value ->
            Fib1 = fib(N-1),
            Fib2 = fib(N-2),
            Fib = Fib1+Fib2,
            cache ! {store, N, Fib},
            Fib
    end.

start_cache() ->
    register(cache, spawn(fun() -> cache(#{}) end)).

cache(State) ->
    receive
        {fib, N, From} ->
            case State of
                #{N := Fib} -> From ! {value, N, Fib}, cache(State);
                _ -> From ! no_value, cache(State)
            end;
        {store, N, Fib} -> cache(State#{N=>Fib})
    end.
```

Figure 8: Fibonacci calculation

syntactic (black) nodes and edges, various semantic (coloured) and lexical (blue) information are presented there.

Based on the initial static analyser framework provided by RefactorErl, various complex static analysers have been implemented. For example, the tool provides control flow, control dependence, data-flow, data-dependence analysis, dynamic function reference analysis, concurrent message flow analysis, etc [19].

RefactorErl supports the analysis of concurrent programs as well. It is able to identify the spawned processes and the communication between them based on data-flow analysis and expression value calculation. RefactorErl builds a communication graph as a result of the analysis [20]. The nodes of the graph represent the processes in the system. The edges represent various forms of communication between the processes, for example, process creation, process name registration, message passing, ETS table creation and reading from or writing into an ETS table.

ETS (Erlang Term Storage) [8] tables are a built-in feature of the Erlang/OTP

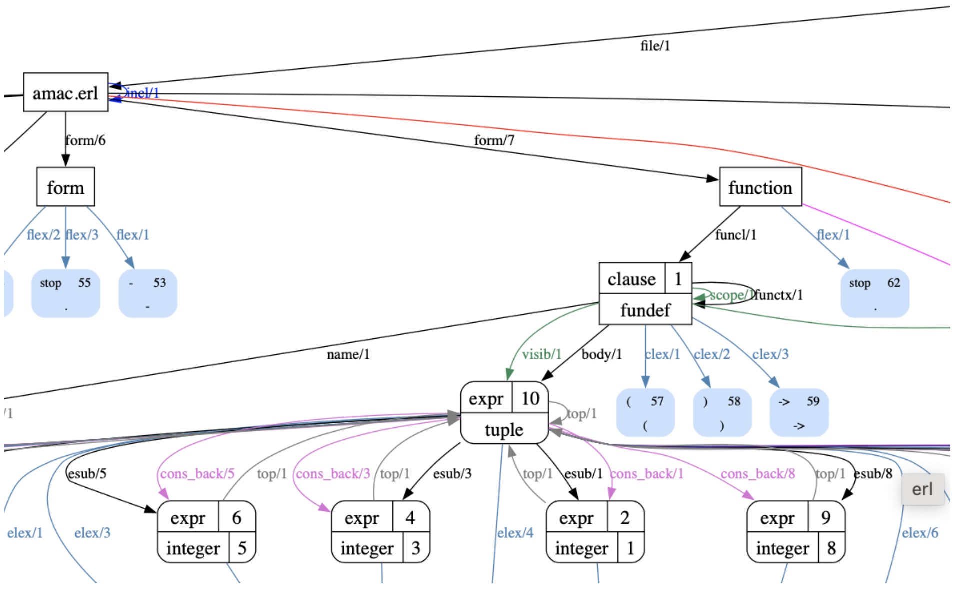


Figure 9: Part of a Semantic Program Graph

system that allows for fast and efficient storage and retrieval of data among multiple processes. They are similar to hash tables or key-value stores, but they are implemented in the Erlang virtual machine and are designed to work well with the Erlang concurrency model. Since one process can put some data into the table that others can read, thus it can be considered as a special form of process communication.

The root of the communication graph is a 'super process' (SP) node which represents the runtime environment. It represents the fact that the communicating functions can be called from the currently running process, for example from the Erlang shell.

Figure 10 shows an example communication graph [20]. These graphs can be useful when we want to find client/server communication in the code as the first step in our analysis. This helps us to find potential candidates and narrow the scope of the analysis. Figure 11 contains an even more detailed communication graph with hidden communication through ETS tables [20].

6 Identifying server processes

In this section, we present our approach to detecting one of the Erlang design patterns, the generic server process. The proposed method is built on the capabilities of the RefactorErl tool introduced in the last section. The method can be divided

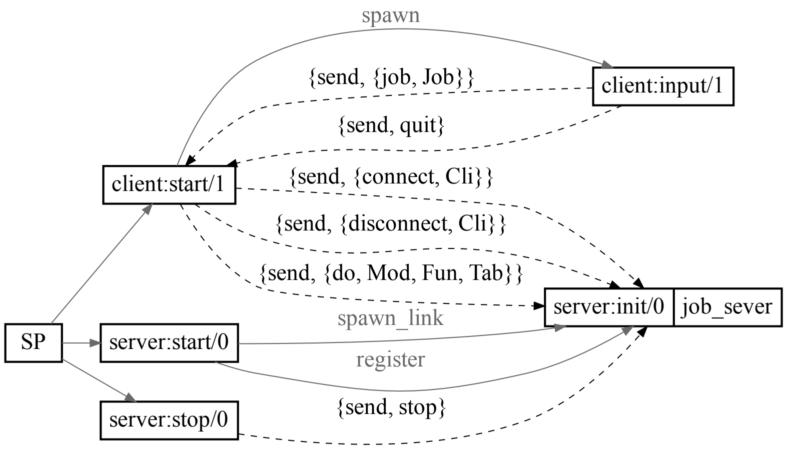


Figure 10: Communication graph [20]

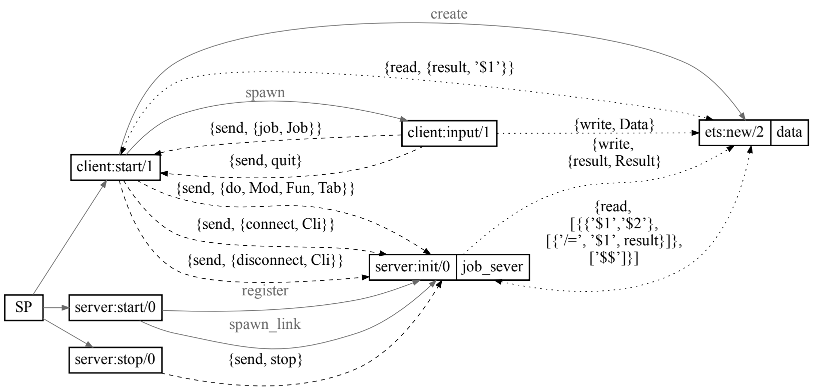


Figure 11: Communication graph with hidden data sharing [20]

into two major steps. We use initial filtering based on the communication graph to reduce the search space and eliminate processes not matching the client-server behaviour. To identify possible candidates, we use data-flow analysis to examine relations between processes. After the initial filtering, we use the Semantic Program Graph to identify functions that match the structure of the generic server design pattern. To do this we have developed a set of rules that the previously determined candidates must comply with.

6.1 Detecting candidates

For the first part of the method, the communication graph is used to find the possible candidate processes. First, we are looking for processes that start and possibly register a process. This can be achieved by examining the communication graph and filtering nodes that are connected with an edge that signifies process creation. The process nodes of the communication graph and the data associated with them are stored in the "processes" ETS table.

We can use the *match_object(Table, Pattern)* function from the *ets* module to find edges that represent process creation, from this we can determine our set of candidate functions.

After the initial filtering of the candidates, we use a set of rules to identify the ones that match the patterns we are looking for. Since the communication graph does not provide enough information for this we use the SPG built by the RefactorErl tool. The information we need to check if a candidate satisfies the rules can be gathered efficiently from the SPG using the query language RefactorErl provides.

6.2 Filtering the initial candidates

To effectively recognize server processes, we need to examine their structure. The Figure 1 shows an example skeleton for a general server process [6]. After the process has been spawned and possibly registered, it initialises the process loop data. The loop data is often the result of arguments passed to the spawn function. The receive-evaluate function receives messages and handles them, updates the state, and passes it back as an argument to a tail-recursive call. If one of the messages it handles is a stop message, the receiving process will clean up after itself and terminate.

Based on this general behaviour we determined a set of rules that candidates must comply with to be considered to be equivalent to the *gen_server* behaviour. These criteria mostly apply to the structure of the processes and can be checked based on the syntactic and semantic information contained in the SPG.

We consider to server processes those functions that satisfy the following criteria:

- the spawned function must be tail-recursive;
- it must contain a receive block;
- one of the branches of the function must be non-recursive to ensure the process can terminate;
- the receive block might contain a reply (synchronous), or no reply (asynchronous) branch;
- the recursive call cannot be in an after block;
- the spawned process is registered; and

- the process is unique.

The first rule comes from a common behaviour of processes, which is typical not only for server processes but also for many other types. Since processes can handle thousands of messages per second over sustained periods of time, using tail-recursion, where the very last thing the function does is to call itself, we can ensure that it executes in constant memory space without increasing the recursive call stack every time a message is handled. To determine whether the candidate complies with this rule, we have to examine the spawned function. It either has to be a tail-recursive function, or the last expression has to be a function call such that the called function itself complies with this rule. Using the SPG and following the function calls we must ultimately check if the last function on the chain is a tail-recursive function. If it is not, we can rule it out from our set of candidates.

If we have established that our function is tail-recursive, the second rule can also be checked in the same step. For this, we only need to examine whether the last expression of the path we followed is a receive block or not. This rule differs from the first one in that it is not a strict rule. We can implement a process complying with the generic server pattern such that it does not have receive block, but it would be incredibly uncommon in practice.

Often one of the messages handled by the server process is a 'stop' message that when received the process will clean up after itself and terminate. For this behaviour, the spawned function must have a non-recursive branch in the receive block. This is also not a strict rule, but using this, we can rank the results according to how well they match the general behaviour. Usually, the function of a server is to receive requests, handle them, and respond with some appropriate message. In order to do this, the receive block must contain replies. Similar to the second rule a process complying with the *gen_server* behaviour can be implemented without this rule being met, but it would not be general use of a server. This rule can be checked by examining the receive block found during the checking of the first rule.

When a process is spawned, it can be registered with a name. After registration, the process can be addressed with its registered name or if registration was omitted with its PID. Not every process needs to be registered, as we have shown previously in the task farm example 4.1.1 where only the dispatcher and collector were registered processes but the workers were not. In contrast to this server processes are almost always registered, so checking if a given process is registered can help us filter the candidates.

Server processes have a special role in the system, they communicate with multiple clients receiving, processing and handling multiple requests. Some server-like processes exist that comply with most of our structural requirements but still cannot (or should not) be considered server processes. A good example of such processes is the worker processes of the task farm. When we create multiple instances of an actor we do not want to consider those as servers. For this reason, it is worth examining if there are multiple instances spawned from a given candidate process.

A process being tail-recursive in itself is not enough for it to be a valid server process. There can be a special case where the tail-recursive call of the spawned function is in an after block. We have to filter out such cases because they would not fit the client-server behaviour. An example of such a process is shown in Section 4.1.2.

It can be the case that a process has multiple recursive calls in its body. Since types of processes would be not transformable to a *gen_server* behaviour-based process, so they cannot be considered server processes despite being structurally similar. Such an example can be found in Section 4.1.3 where a parallel implementation of the Fibonacci number calculation is shown. These types of processes have to be also filtered out from our results.

To refine the set of candidates, it might also be worth examining if the receive block has branches for certain special messages that a server usually handles, for example an *'EXIT'* or *'DOWN'* messages sent to the server process when linked/monitored processes exit[2].

7 Evaluation

Our method for identifying source code fragments in legacy Erlang systems that can be transformed into client-server Erlang behaviors is based on the static source code analysis and transformation tool RefactorErl. We present examples of server-like processes we found in some analysed projects that could have been implemented using the *gen_server* behavior.

We analysed example source codes and solutions[3] to the exercises to the books *Erlang Programming* [6], *Programming Erlang: Software for a Concurrent World* [3] and *Learn You Some Erlang for Great Good* [11]. We found snippets that match the pattern of the client-server behaviour.

The first server-like process we identified is the basic server implementation example from the book Learn You Some Erlang for Great Good. The *kitty_server* is a simple Erlang application that demonstrates the use of the client-server pattern and message-passing between processes. The example is a simulation of a server that manages a collection of 'kitty' objects, which are represented as Erlang processes. Clients can interact with the *kitty_server* process by sending messages to it, such as requesting to create a new kitty, or asking for the status of a particular kitty. The *kitty_server* process then communicates with the appropriate process to fulfill the request, and sends a response back to the client. The prototype algorithm identifies the loop function shown in Figure 12 as a server-like process as it satisfies all the established criteria.

From the example codes provided to *Erlang Programming* the prototype algorithm found multiple server-like processes. In Chapter 4 of [6] there are two small examples (Figure 13) demonstrating message passing between processes (the loop function is the same in both). While it might be unnecessary because of the simplicity of the example, it would be possible to convert both to a *gen_server*

[2] https://learnyousomeerlang.com/errors-and-processes

[3] https://github.com/francescoc/erlangprogramming, https://github.com/Stratus3D/programming_erlang_exercises, https://learnyousomeerlang.com/

```
loop(Cats) ->
    receive
        {Pid, Ref, {order, Name, Color, Description}} ->
            if Cats =:= [] ->
                Pid ! {Ref, make_cat(Name, Color, Description)},
                loop(Cats);
               Cats =/= [] -> % got to empty the stock
                Pid ! {Ref, hd(Cats)},
                loop(tl(Cats))
            end;
        {return, Cat = #cat{}} ->
            loop([Cat|Cats]);
        {Pid, Ref, terminate} ->
            Pid ! {Ref, ok},
            terminate(Cats);
        Unknown ->
            %% do some logging here too
            io:format("Unknown message: ~p~n", [Unknown]),
            loop(Cats)
    end.
```

Figure 12: Main loop function of the kitty server [11]

implementation[4]. In the modules provided to Chapter 5 the algorithm identified several examples where a process could be implemented with the *gen_server* behaviour. For example in the module *frequency* a server process is responsible for managing radio frequencies on behalf of its clients, the mobile phones connected to the network. The phone requests a frequency whenever a call needs to be connected, and releases it once the call has terminated. This is an example that demonstrates the client-server design pattern and the *loop/1* function fits the criteria perfectly.

Although in the analysed simpler examples we successfully found the possible server-like processes with the help of the implemented prototype algorithm, not all rule-checks have yet been fully implemented, so we also received a few false positives. Such an example was the *loop/1* function shown in Figure 14 (Exercise 3 from Chapter 12 of [3]). The example code implements a process ring, where a number of Erlang processes are connected in a ring-like structure, with each process communicating with one neighbor in the ring. The process with ID 1 then sends a message to the process with ID 2, which in turn sends the message to the process with ID 3, and so on, until the message has been passed around the entire ring. At first this process seems like a server based on its structure and communication but it does not comply with the rule of uniqueness. Such candidates have to be eliminated in the future.

[4]In the future, we might implement a prioritisation to present the results in the order of relevance. Simple candidates might be listed at the end of the candidate list.

```
loop() ->
   receive
     {From, Msg} ->
        From ! {self(), Msg},
        loop();
     stop ->
       true
   end.
```

Figure 13: Loop function from the simple example in Chapter 4 of [6]

```
loop(NextPid) ->
    receive
        stop ->
            NextPid ! stop,
            ok;
        Value ->
            NextPid ! Value,
            loop(NextPid)
    end.
```

Figure 14: Loop function from a process ring in Chapter 12 of [3]

In Exercises 5 and 6 from Chapter 13 of [6] (Figure 15) the prototype algorithm identified the spawned *handle_crashes/1* function as a server-like process. The example code implements a supervisor process which is responsible for starting, stopping, and monitoring the other processes. The worker processes are responsible for performing specific tasks, and the supervisor process is responsible for monitoring and managing the worker processes. When a worker process crashes or exits, the supervisor process is notified and restarts the worker process. This could be potentially implemented with the *gen_server* behaviour, but there exists a separate behaviour for exactly this type of process, the *supervisor* behaviour. In this case, it would be preferable to use the latter behaviour. However, we would like to note here that the *supervisor* behaviour is implemented as a server using a *gen_server* behaviour. Thus our result is correct.

We examined the edge cases presented in Section 4.1. The prototype implementation had a false positive hit, the *worker* function (Figure 6). This is a known limitation, since the uniqueness check is not yet implemented.

8 Conclusions

Design patterns provide solutions to recurring issues in software development. For object-oriented languages, various tools exist that use different approaches and

```
handle_crashes(WorkerData) ->
  receive
      {get_workers, Pid} ->
          Pid ! {self(), workers, WorkerData},

          handle_crashes(WorkerData);
      {'DOWN', Ref, process, Pid, Why} ->

          ...

          % Recursively call this function to handle later crashes
          handle_crashes(NewWorkerData)
  end.
```

Figure 15: Function `handle_crashes` from the example in Chapter 13 of [6]

methods to identify these patterns. In Erlang, behaviours are the formalised versions of these design patterns. In this paper, we proposed a method for identifying a specific design pattern, the client-server behavior, in legacy Erlang systems.

In this paper, we proposed a method based on static analysis of Erlang programs to identify processes complying with the client-server behaviour. The method is based on the analyses provided by the RefactorErl tool and can be divided into two major steps. Initial filtering based on the communication graph is used to reduce the search space and eliminate processes not matching the server behaviour. After the initial filtering, the Semantic Program Graph, an intermediate representation of the source code built by the RefactorErl tool is used to identify functions that match the structure of the generic client-server design pattern. To achieve this, we have developed a set of rules that the previously determined candidates must comply with.

We implemented the prototype algorithm and tested it on small open-source examples. Using this prototype implementation, we identified basic server processes that could be turned to equivalent *gen_server* process. We also examined a few edge cases where the described rules might fail. In the future, we would like to analyse further open-source projects and refine the rules based on the findings.

References

[1] Al-Obeidallah, M., Petridis, M., and Kapetanakis, S. A survey on design pattern detection approaches. *International Journal of Software Engineering*, 7:41–59, 2016. URL: https://www.cscjournals.org/manuscript/Journals/IJSE/Volume7/Issue3/IJSE-163.pdf.

[2] Arcelli Fontana, F., Perin, F., Raibulet, C., and Ravani, S. Design pattern detection in Java systems: A dynamic analysis based approach. *Communications in Computer and Information Science*, 69:163–179, 2010. DOI: 10.1007/978-3-642-14819-4_12.

[3] Armstrong, J. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007. DOI: 10.1017/S0956796809007163.

[4] Arts, T., Benac Earle, C., and Derrick, J. Development of a verified Erlang program for resource locking. *International Journal on Software Tools for Technology Transfer*, 5(2):205–220, 2004. DOI: 10.1007/s10009-003-0114-9.

[5] Brown, K. Design reverse-engineering and automated design-pattern detection in smalltalk. Technical report, North Carolina State University at Raleigh, USA, 1996. URL: https://repository.lib.ncsu.edu/items/ec9a80d5-c9c6-47c5-afd5-03f21a36bb63.

[6] Cesarini, F. and Thompson, S. *Erlang Programming: A Concurrent Approach to Software Development*. O'Reilly Media, Inc., 2009. URL: https://www.oreilly.com/library/view/erlang-programming/9780596803940/.

[7] Cole, M. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, Cambridge, MA, USA, 1991. URL: https://dl.acm.org/doi/10.5555/128874.

[8] Ericsson AB. Erlang Reference Manual: `ets` module. URL: https://www.erlang.org/doc/man/ets.html.

[9] Gaitani, M., Zafeiris, V., Diamantidis, N., and Giakoumakis, E. Automated refactoring to the Null Object design pattern. *Information and Software Technology*, 59:33–52, 2015. DOI: 10.1016/j.infsof.2014.10.010.

[10] Guéhéneuc, Y.-G., Sahraoui, H., and Zaidi, F. Fingerprinting design patterns. In *11th Working Conference on Reverse Engineering*, pages 172–181. IEEE, 2004. DOI: 10.1109/WCRE.2004.21.

[11] Hebert, F. *Learn You Some Erlang for Great Good! A Beginner's Guide*. No Starch Press, USA, 2013. URL: https://learnyousomeerlang.com/.

[12] Heuzeroth, D., Holl, T., Högström, G., and Löwe, W. Automatic design pattern detection. In *11th IEEE International Workshop on Program Comprehension*, pages 94– 103, 2003. DOI: 10.1109/WPC.2003.1199193.

[13] Horváth, Z., Lövei, L., Kozsik, T., Kitlei, R., Nagyné Víg, A., Nagy, T., Tóth, M., and Király, R. Modeling semantic knowledge in Erlang for refactoring. In *Knowledge Engineering: Principles and Techniques, Proceedings of the International Conference on Knowledge Engineering, Principles and Techniques*, Volume 54 Special Issue of *Studia Universitatis Babeş-Bolyai, Series Informatica*, pages 7–16, Cluj-Napoca, Romania, 2009.

[14] Learn you some Erlang. An event module. URL: https://learnyousomeerlang.com/designing-a-concurrent-application#an-event-module.

[15] Li, H. and Thompson, S. Similar code detection and elimination for Erlang programs. In *International Symposium on Practical Aspects of Declarative Languages: Practical Aspects of Declarative Languages*, Volume 5937 of *Lecture Notes in Computer Science*, pages 104–118. Springer Berlin Heidelberg, 2010. DOI: 10.1007/978-3-642-11503-5_10.

[16] Li, H., Thompson, S., Orosz, G., and Tóth, M. Refactoring with Wrangler, updated: Data and process refactorings, and integration with Eclipse. In *Proceedings of the 7th ACM SIGPLAN Workshop on ERLANG*, ERLANG '08, page 61–72, New York, NY, USA, 2008. Association for Computing Machinery. DOI: 10.1145/1411273.1411283.

[17] skel: A streaming process-based skeleton library for Erlang, 2012. URL: https://github.com/ParaPhrase/skel.

[18] Tóth, M., Bozó, I., and Kozsik, T. Pattern candidate discovery and parallelization techniques. In *Proceedings of the 29th Symposium on the Implementation and Application of Functional Programming Languages*, IFL 2017, New York, NY, USA, 2017. Association for Computing Machinery. DOI: 10.1145/3205368.3205369.

[19] Tóth, M. and Bozó, I. Static analysis of complex software systems implemented in Erlang. In *Central European Functional Programming School*, Volume 7241 of *Lecture Notes in Computer Science*, pages 451–514. Springer Berlin Heidelberg, 2012. DOI: 10.1007/978-3-642-32096-5_9.

[20] Tóth, M. and Bozó, I. Detecting and visualising process relationships in Erlang. *Procedia Computer Science*, 29:1525–1534, 2014. DOI: 10.1016/j.procs.2014.05.138.

[21] Yu, D., Zhang, P., Yang, J., Chen, Z., Liu, C., and Chen, J. Efficiently detecting structural design pattern instances based on ordered sequences. *Journal of Systems and Software*, 142:35–56, 2018. DOI: https://doi.org/10.1016/j.jss.2018.04.015.

[22] Zafeiris, V., Poulias, S., Diamantidis, N., and Giakoumakis, E. Automated refactoring of super-class method invocations to the Template Method design pattern. *Information and Software Technology*, 82:19–35, 2017. DOI: 10.1016/j.infsof.2016.09.008.

Acta Cybernetica **26** (2024) 431–454.

Using Version Control Information to Visualize Developers' Knowledge*

Anett Fekete[ab] and Zoltán Porkoláb[ac]

Abstract

It is not always clear in case of a software project who has the right amount of knowledge concerning a certain module or file. Programmers frequently ask questions like "Who knows the most about this code?" or "Who can I ask for help when I work on this module?". In a large, long-term software product, knowledge is distributed in an uneven way among developers. Developer fluctuation during the product lifetime might cause some parts of the code to be known very well by a multitude of developers, while other parts might sink to the "gray zone", where developer competence is dangerously scarce. It is important for the project management to identify such critical points, to avoid the complete loss of competence. Version control repositories contain loads of useful information about the evolution of a software project. This paper presents a novel developer-centered method implemented as a plugin in the open-source code comprehension tool, CodeCompass. The method is intended to detect individual, team-bound and company-bound knowledge of large legacy projects. The competence information is computed from the extracted version control information from Git repositories. The calculated competence value is based on the number of commits per developer and their significance. The method weighs all changes according to their added value computed by a plagiarism detection software. Aggregated views for teams and companies are available based on various heuristics. The results are visualized as graph-based diagrams. Project managers and individual developers may both profit from the tool, whether it concerns software evolution, human-resource management, architecture, knowledge catch-up, or blame.

1 Introduction

In case of long-running software projects, the fluctuation of developers is inevitable. This is true for smaller projects with only a few developers at once, such as a uni-

*Prepared with the professional support of the Doctoral Student Scholarship Program of the Co-operative Doctoral Program of Ministry of Innovation and Technology financed by the National Research, Development and Innovation Fund.

[a]Faculty of Informatics, Eötvös Loránd University, Budapest, Hungary

[b]E-mail: afekete@inf.elte.hu, ORCID: 0000-0001-8466-7096

[c]E-mail: gsd@inf.elte.hu, ORCID: 0000-0001-6819-0224

DOI: 10.14232/actacyb.299322

versity lab project, or large, industrial projects that count tens or hundreds of developers. When fluctuation is so great, there is always risk that certain components of the software suffer neglect. If all or even most developers that have knowledge about a component leave the project, maintenance problems might emerge, and it will cause much more problems to debug or develop the component than it would be in possession of decent expertise [3].

During development, lots of questions emerge that concern other developers, like ”Who wrote this code?”, ”Why did they decide to write this code like that?”, ”Who understands this code the best?” and so on. These questions might be hard to answer, especially in large companies, where dozens of programmers develop a software. Nowadays, using some type of version control system for our projects like Git or SVN is fundamental. Repositories are able to store every piece of information that is bound to development since the start of the project. Naturally, the information we need is not stored explicitly but in the form of commits that can be processed and analyzed.

In this paper, we present the *competence plugin*, a tool that is capable of analyzing the commit history of a Git repository and tell various information about the project files and developers as graph-based visualizations. The plugin serves multiple different developer-related purposes with its visualizations. We would like to facilitate code comprehension for programmers by providing them a visualization that methodizes the obtained knowledge about project files. Observing the already familiar part of the code and the unknown territories, the programmer can consciously select the next comprehension target. It is also our goal to provide useful team- and company-level information, thus we present the team view and the affiliation views. They tell about the most competent developer on a certain file and their affiliation. They also provide information about the teams or companies who have the most code-related knowledge. The calculations and visualizations are all handled on file level.

For ethical reasons, all data in this paper has been anonymized. The original research has been done in a real industrial software development environment with actual development teams and programmers. We replaced the email addresses, usernames, and team or company names. We use the form *Dev email x* and *Team y* in place of the original names.

The rest of the paper is structured as follows: Section 2 describes the various methods and approaches for code comprehension supporting visualization, paying particular attention to version control related software. Section 3 presents our methodology of version control data analysis. Section 4 describes the visualization of the version control data. Section 5 provides a report of the details of implementation. In Section 6 we present the results of our work on some well-known open source projects. Section 7 discusses the possible threats to the method’s validity. Finally, in Section 8 we give a heads-up about future work and conclude our paper.

2 Related work

There has been much research done before about code comprehension supporting visualization possibilities [6,32,41], and using version control data for visualizations in particular. Code comprehension supporting tools usually focus on a group of important and coherent aspects of a software, which means they do not cover every need of a developer that seeks deeper knowledge of the project. Visualization tools generally stick to 2D or 3D imaging exclusively, thus they can be divided in these two groups.

In this section, we give an overview of recent visualization tools that support code comprehension, and then we discuss tools that use version control information in their visualizations in detail.

2.1 Visualization for code comprehension support

Ever since large software projects started spreading, the need for reliable, transparent, informative code comprehension tools and visualizations has been growing [4, 30]. There are several software to facilitate source code comprehension, in the form of plugins (e.g. in IDEs) and standalone tools [8, 46]. These software always have some focus points that they put the most emphasis on, usually code metrics based on structure or content [44].

A returning approach is displaying source code metrics in the form of some type of map view, created with complex geoinformatical methods [20]. Tools with this approach usually represent source code as continents, states, cities, and buildings on a map. Metrics can be diversely shown by the size, color, borders and surroundings of the representing shapes. This technique guarantees that abstract modularity levels are depicted correctly, and transparency is provided for the user. Code comprehension software with map views can be divided into 2D and 3D tools. A couple 2D software include *CodeSurveyor* [20], *Software Cartography* [26], 3D software include *CodeCity* [47, 48] and *CityVR* [33] which is also a modern step towards code comprehension supporting visualization with an experimental usage of VR technology and gamification. Even more recent approaches combine traditional visualization techniques with virtual and augmented reality [25].

Another commonly used visualization method includes using simple shapes, such as rectangles or circles [2], and diagrammatic figures (e.g. UML diagrams, statistic diagrams) to represent code metrics. We can also apply grouping the tools by their dimensions; 2D software include *CodeCrawler* [28], *ExplorViz* [15], and *CommunityExplorer* [35], 3D software include *sv3D* [31] and *TraceCrawler* [19]. er erom

2.2 Usage of version control information

Version control information is used for various software research areas. Most frequently, in the center of these researches is the connection between commit actions and software quality, and the cost of maintenance. Naturally, this is used as the prediction of distribution of software bugs. In [37], the authors developed a regression model that accurately predicts the likelihood of post-release defects for new entities. Similarly, in his PhD thesis [12] the author describes the connection between code maintenance activities as it is reflected by version control information and the deterioration of the code quality. In a related paper [13] the authors show that a connection between version control operations and maintainability really exists, in spite of the fact that the data is coming from different sources.

Apart from software quality, other researches target the developer's team. The authors of [22] utilize version control information for mining and visualizing networks of software developers. They detect similarities among developers based on common file changes, and construct the network of collaborating developers. The authors show that this approach performs well in revealing the structure of development teams and improving the modularity in visualizations of developer networks.

Unfortunately, information retrieved from version control systems has its limitations. As an example, attempts to predict code quality or developer efficiency cannot be achieved according to a research described in [36].

A frequent problem with information mining from version control systems is that they store only atomic information. In [23], the authors suggest a set of heuristics for grouping change-sets files that frequently change together. The results show that the approach is able to find sequences of changed-files.

Version control information can be used not only for extracting data from the source code for code comprehension purposes, but it is also a frequent research target for analyzing comments – either structured, or natural language text – in order to get additional information about the system [42].

Version control information is used for code comprehension purposes to connect related code modifications submitted in the same commit as described in [5].

2.3 Version control data visualization

Utilizing version control data forms a subset of code visualization techniques. While repositories contain lots of valuable information, repository mining and data analysis is an additional challenge in visualization pursuits [49].

Alcocer et al. [2] created a circular visualization called *Spark Circle* to visualize the changes in various source code metrics between commits. The visualizations vary based on the number of different metrics calculated. A spark circle consists of one annulus if only one metric is applied or multiple annulus sectors in case of multiple metrics. They also boost the visualization by using different filler and border colors and different shape sizes according to the concrete metric data. Spark Circle is a useful tool for the analysis of software evolution based on commit difference.

Not only commits, but entire Git repositories and the branches they contain can be effectively analyzed and visualized. Elsen [11] has proposed *VisGi*, a version control visualization software that is capable of displaying detailed graphs of Git branches and version structures. The software highlights structural and code-related differences. It also shows that time-bound visualization provides valuable information about software and repository evolution.

Greene et al. [17, 18] developed a browser tool that intends to give answers to collaborator- and repository-related questions among others, named *ConceptCloud*. They put focus on identifying abstract relations between objects and attributes. Their tag cloud visualization is mainly text-based, and capable of handling multiple selected tags for more accurate search results. Naturally, this method provides the user with an opportunity to find answers for code-related questions as well.

There are also attempts for the integration of version control data visualization into the development environment. The Eclipse IDE has a plugin which calculates the number of changes of methods during a given amount of time [45]. Another, less recent Eclipse plugin supports data visualization from the CVS version control system [7].

There can be correlation detected between the changes made to source code and performance. In order to facilitate discovering changes and their causes, Alcocer et al. [1] developed *Performance Evolution Matrix* which is capable of comparing multiple versions of the same software. The tool helps the user notice the modifications that might have caused changes in performance. The visualizations mostly aim changes in software metrics, such as additions and deletions to source code, call graph changes, and execution time differences.

As mentioned above, besides traditional 2D visualization techniques, modern approaches are becoming more widespread in version control visualization methods, such as virtual reality [38].

Apart from version control repositories, data from the supporting project hosting systems (e.g. GitHub, GitLab) is also useful for understanding software. Kumar et al. use Elastic search and Kibana for data visualization through mining GitHub for further information that cannot be found in plain repositories [27].

3 Methodology

3.1 Background

Repositories, especially those of long-running projects tell lots about development history and workflow. It creates an image of the gradual changes in the architectural design of the software. When it comes to maintenance and debugging, the first questions that usually emerge are "Who can I turn to for explanation of the logic, structure and objective behind this code? Who is the expert in it?". Version control systems contain all the answers to these questions in their commit history in an implicit way that requires meticulous analysis to be brought to surface. Commits provide all the information about who is responsible for each line of code ever written for that repository in their blame data. This information can be easily obtained and used in various different visual depictions. Our visualization intends to present the developer data to facilitate source code comprehension by providing developer statistics for the software.

In the competence computing, we consider one commit as a unit, this is why the plugin parses a given part of the commit history commit-by-commit, which is then divided into *deltas* that are equivalent to the files that were modified by the commit author. The most important factor of developer competence in our calculation is the significance of modifications which is calculated using *JPlag*[1] [40], a code similarity checker. The idea is that we want to measure how important is a change and want to ignore irrelevant formal modifications. We compare the modified version of a file to its previous version in the commit history. JPlag returns a percentage value applying token-based comparison. When comparing different versions of a file, JPlag breaks down the file to tokens, and checks the changes between versions. To minor changes that do not affect the actual file content, such as adding or removing comments, or renaming identifiers, JPlag returns a 100% match. We use this number in our calculation as a threshold to detect relevant changes in the source code. If the compared versions are not 100% equal, then relevant modifications were committed to the code. This way minor changes are filtered, and the focus is put on actual content change.

3.2 Data parsing

The competence plugin consists of a parser and a service component. The parser performs code analysis, repository mining, and information extraction. Figure 1 shows the parser workflow.

The first and most important precondition of parsing is for the project to have a Git repository. Without one, the parser will finish parsing without processing any information. This is why the very first step during parsing is to find the *.git* directory. Since this directory is usually placed in the root directory of the source code, the parser looks for it in the user-provided source path. The parser can be provided with an optional input variable, n, the number of commits to parse. If n is not provided, the entire commit history is parsed.

Once the repository is found, we need to collect information about the modifications done in the commits. We traverse through each commit on the current branch, starting from *HEAD*, look through the list of modified files and calculate actual developer competence data. For this, we need to traverse all relevant commits to obtain the blame data. In order to traverse the commits, we need a revision walker that is sorted in a backward sequence in time and in topological order. The walker provides the next commit. Commit history processing is done as follows:

1. Let C be the commit history, and let c_j be the current commit that is retrieved from the walker ($C = c_1, c_2, ..., c_j, ..., c_n$). If n is provided, we check if c_j meets the conditions. If not, execution is terminated.

2. Let p_{c_j} be the direct parent of c_j which is retrieved from the repository. If p_{c_j} does not exist, we are at the beginning of the commit history ($c_j = c_n$), and execution is terminated.

3. Let $d_{c_j,p_{c_j}}$ be the difference between the two Git trees built from c_j and p_{c_j}.

[1]GitHub repo: https://github.com/jplag/JPlag

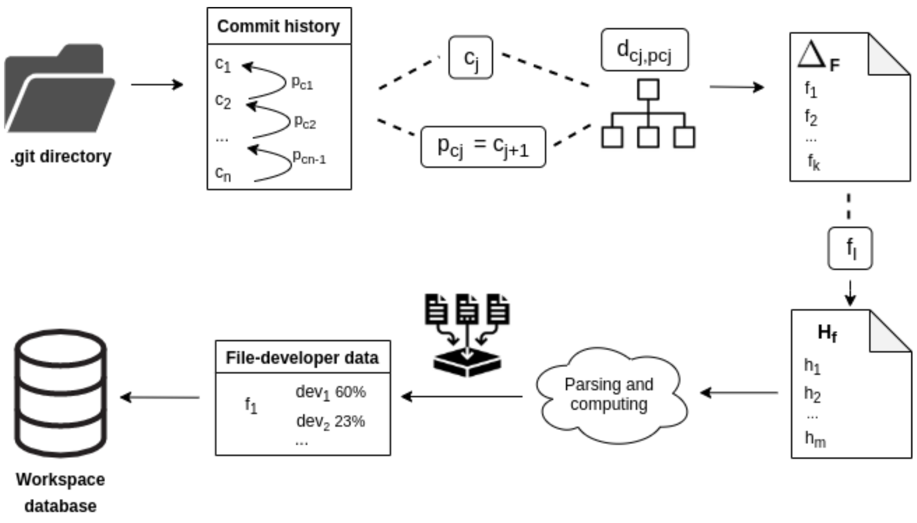


Figure 1: The methodology of commit history parsing. The commit history of a Git repository is parsed commit-by-commit. We take the diff of each consecutive pair of commits, and analyze every file that was modified in the latter one. The file versions are compared with a software similarity checker to determine the structural significance of modifications as a percentage value. The summarized data is persisted into the database.

4. As mentioned before, $d_{c_j,p_{c_j}}$ consists of *deltas*, each of which contains the modifications in one file. Let f_i be an individual file that has been modified in a commit ($F = \{f_1, f_2, ...f_k\}$).

5. $\forall f_i \in d_{c_j,p_{c_j}}$ we compare f_i with its previous version in p_{c_j} using JPlag. The returned percentage value shall be the competence data of the author for f_i.

6. Go to 4): move on to the next delta if it exists, otherwise, go to 7).

7. Go to 1), and start over with c_{j+1}, if all the conditions are met (i.e. there are further commits to parse), otherwise, go to 8).

8. Persist the calculated data in the database for every file and every commit author.

Despite not being able to extract usernames, we can, however, use author email addresses to extract affiliation, i.e. company or team names. Large companies usually have their own domain name and give a workplace email address within this domain to their employees (e.g. *microsoft.com*, *ericsson.com*). Based on this convention, we can assume, that if a developer authors their commits with a certain email address, we can conclude their affiliation.

The competence plugin automatically extracts well-known company names from email addresses during parsing time. It works with a list of possible domain names mapped up with company names prepared in advance. When the plugin is done with competence data calculation, all email addresses are checked for company name extraction. Private email addresses and others not in the company list can be completed with affiliation manually.

4 Visualization

The competence plugin is implemented as a part of CodeCompass[2] [39], which is an LLVM/Clang based open-source code comprehension framework developed by Eötvös Loránd University and Ericsson. The CodeCompass parser applies static analysis on the given source code and the corresponding build commands that are logged during compilation. Various information is stored about the project including structural data, code metrics, version control information, etc. This information is stored in the workspace database which is then accessed by the CodeCompass webserver. The webserver provides several various textual and graphic services through a web browser, such as detailed searching, structural and code-level visualizations, and Git blame data.

CodeCompass has a pluginable framework. Plugins work independently, thus a certain plugin can be easily skipped from the parsing process if it is not needed. Plugins consist of a *parser* and a *service* component. The parser component takes care of the analysis, while the service component is responsible for constructing the visualizations based on the stored information in the database and displaying it through the CodeCompass webserver. Currently, there are 4 different available diagram types.

4.1 Personal view

When browsing software components or learning the source code, it is useful for the user to keep track of which parts in the code base they have reasonable knowledge about, and what else is there to investigate and learn. The personal view intends to show this information. CodeCompass is capable of text-based authentication, thus it is possible to show the user their personal information stored in the database.

The personal view diagram shows the maximum competence percentage that belongs to the authenticated user for every file in the project. The percentage is converted to a color code that appears in the corresponding node in the diagram. Generated colors are on a scale from red, assigned to 0%, to green, assigned to 100%. Figure 2 shows an example of the personal view about the structure of the competence plugin.

This view is also useful for project managers to decide who to assign a certain task, or for other teammates to see who they can ask if a question emerges or a

[2]GitHub repo: `https://github.com/Ericsson/CodeCompass`

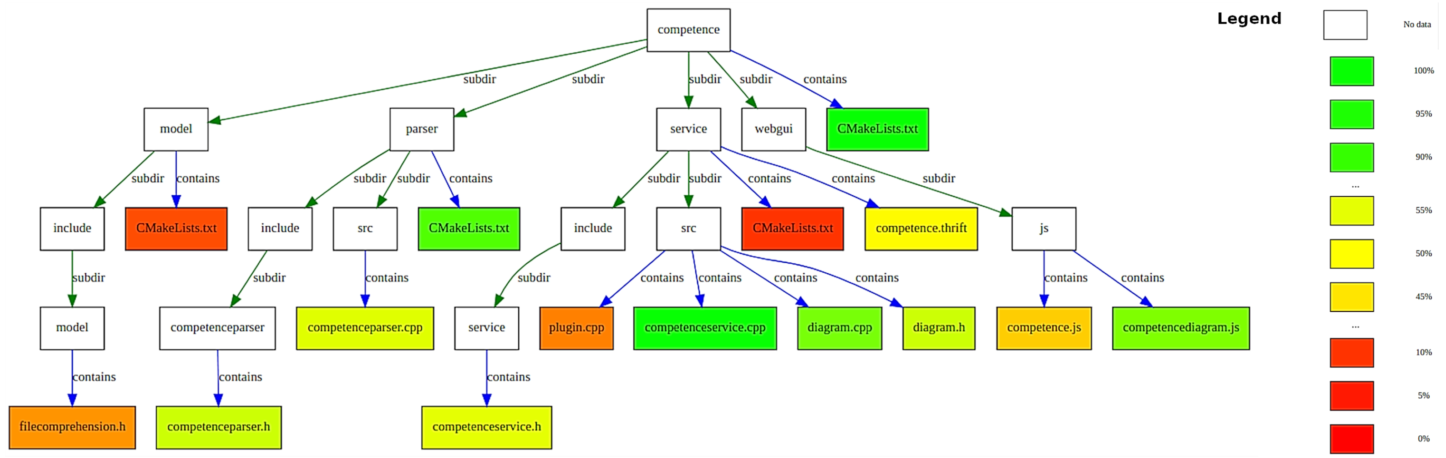


Figure 2: The personal view of a developer concerning the source code of the competence plugin. The non-white nodes represent files in the plugin. They are colored on a scale from red to green, according to the competence percentage of the developer. The containing directories are left white as they have no assigned data.

bug is found etc. However, the plugin is now only capable of showing the user their own information.

4.2 Team view

If we are looking to see who the most competent developer is in the current state of a file, it is likely the one who recently committed a larger significant modification to the file in question. The investigated number of commits can be set by the user before parsing. This way, the last n commits will be parsed, starting from the very last one. Team view displays the most competent developer of every file, based on the parsed information. It selects the maximum percentage stored for the file. The diagram nodes are colored according to the color code map that was previously calculated from the developers' email addresses. An example of the team view is shown in Figure 3.

4.3 Affiliation views

In case of a collaboration project or an open-source software, it is useful if we are aware which unit is the most competent in a software module. This is why it is advantageous to display affiliation-focused diagrams. The competence plugin provides 2 different affiliation views:

4.3.1 Individual affiliation view

In its logic, this type of diagram is similar to *team view*, but it is focused on affiliations. We calculate who the most competent developer is in a file, but instead of coloring the node with the personal color of the developer, we color it with the assigned color of their company or team.

4.3.2 Accumulated affiliation view

In order to learn which team is the most competent in a file, we need to consider all data that belongs to the file in question. In this diagram, we group the records of a file by developer affiliations, and take the average competence of every team. A diagram node gets the color of the team with the maximum average value.

Although they derive from the same data, the affiliation views might display different results for the same file: the first diagram focuses on the individual competence rate and the corresponding affiliation, while the second one calculates the average competence rate of the members of each team that worked on the examined file, and shows which team has the highest competence in that file. An example of the two views and their comparison is shown in Figure 4.

The latter 3 visualizations can be displayed without authentication.

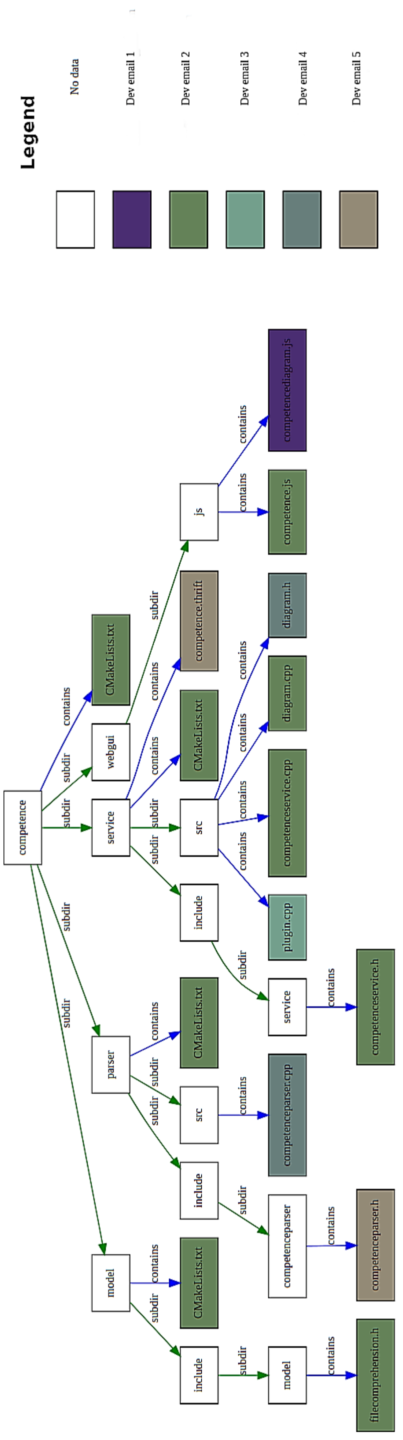


Figure 3: The team view of the competence plugin. Like in Figure 2, the white nodes represent directories, and the colored nodes represent the contained files. The file nodes are given the color of the most competent developer in that file. The colors are automatically generated from hashing email addresses.

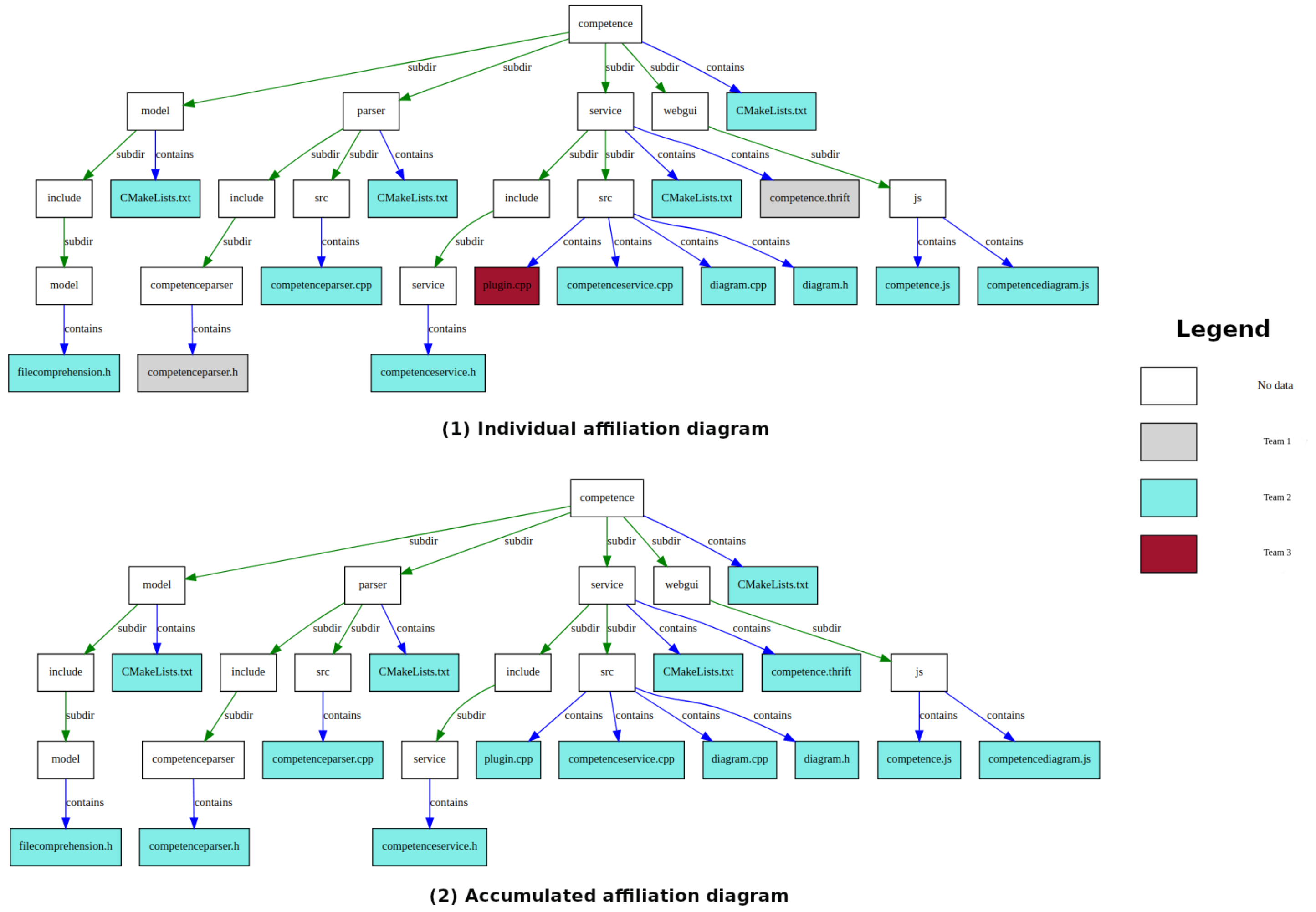


Figure 4: Comparison of the individual (1) and the accumulated (2) affiliation diagram for the source code of the competence plugin. (1) shows the affiliation of the most competent developer in a file, while (2) shows the most competent team in a file on average.

4.4 Coloring

In the personal view, node coloring is trivial, since a node is assigned a color on the scale from red to green depending on the corresponding competence percentage from 0% to 100% respectively. In the affiliation diagrams, each team is assigned a random color. However, node color generation in the team view is based on the corresponding email address. Email addresses are hashed, and the hash code is converted to a unique color.

5 Implementation

CodeCompass provides a stable core for a pluginable framework. The backend (parser and web server) is written in C/C++, while the frontend is written in JavaScript, using the *dojo.js* library. The competence plugin is implemented as an extension of CodeCompass as a new plugin. It relies entirely on version control data and supports only Git. Repository mining and data analysis is implemented using the *libgit2* API library. The graph-based visualizations are generated using *GraphViz* [10]. Since JPlag is capable of parsing several popular programming languages, we consider the plugin fairly language-independent, meaning that the competence calculation is applicable to all languages supported by JPlag.

6 Case studies

The plugin was tested on multiple long-running open-source projects: *Google Test*[3], *libgit2*[4], and *LLVM-Clang*[5] [29]. All of these projects are continuously developed with hundreds or even thousands of commits a month, and their developer teams consist of large numbers of programmers from several different companies. We also took CodeCompass itself[6] into the test projects, since we thoroughly know its history and structure, and it facilitated the evaluation of the results. The tests were run on an average personal computer with 16 GB RAM and 3 CPU cores.

Table 1 shows the results of parsing the test projects. All of their repositories contain hundreds, thousands, or even hundreds of thousands of commits. In order to provide easily comparable information, we parsed the latest few hundred commits of every project. Average execution time was calculated considering that 3 cores were used.

We can see significant difference in execution time, the number of modified files, and the number of committer developers between the projects. The most spectacular difference is the execution time of LLVM which can be measured in hours, compared to the other projects where hundreds of commits have been parsed

[3]GitHub repo: `https://github.com/google/googletest`

[4]GitHub repo: `https://github.com/libgit2/libgit2`

[5]GitHub repo: `https://github.com/llvm/llvm-project`

[6]The repository of CodeCompass was migrated from SVN to Git in 2016. The earlier version control data is not accessible.

Table 1: Competence parser results

Project	**All commits**	**Parsed commits**	**Exec. time**	**Modified files**	**Devs**
CodeCompass	955	500	8m 31s	553	15
Google Test	3,913	500	26m 21s	151	63
libgit2	14,550	500	92m 41s	353	33
LLVM-Clang	425,138	500	15h 34m	1340	159

in less than an hour. The cause of this phenomenon is that, although other test projects are also open-source and developed by numerous programmers, LLVM is still an edge-case compared to them; if we take a look at the LLVM's GitHub statistics, we can see that over 3000 commits are pushed to master during a single month, while other continuously developed projects are expanded by only a few hundreds of commits at most. What's more, the commits pushed to LLVM are large ones, affecting many files, and they frequently reach the size of a full value pull request[7], which means parsing one commit in this project means actually parsing several smaller commits. This also explains why the average execution time of one commit is significantly higher in LLVM.

Aside from the visualizations, the workspace database also provides a great deal of information about developers. Table 2 contains the answers to the developer-related questions asked above, "Who knows the most about the code?". In all test projects, less than a dozen developers can be named who are competent in larger parts of the code. This circumstance means increased risk from the aspect of the project's future. If any of the highly competent programmers leave the developer team for any reason, the software might suffer serious damage from a code and software quality perspective among others.

Table 2: Developer data of the test projects

Project	**>25 files known**	**>50 files known**	**Most competent developer**	**Files known**	**Most competent team**
CodeCompass	6	4	user1	399	Company1
Google Test	3	3	user2	87	N/A
libgit2	4	2	user3	200	N/A
LLVM-Clang	15	2	user4	107	Company4

[7]Commits on LLVM are mostly actual size pull requests where lots of commits are compressed into a patch file.

The "most competent developer[8]" at each software, while they possess a large amount of information and they are very important in the project, still have limited knowledge about the code. This can be considered a normal situation, since we cannot expect, even from a lead developer or an architect, to know everything about every component. The programmers with the highest amount of knowledge supposedly know the entire project thoroughly, and they are aware of all the important developer decisions.

We are also able to evaluate the results from the affiliations' aspect. If a project is developed by multiple teams or companies – which is usually the case in open-source projects – it might be an interesting and useful information to know which team is the most competent in certain parts of a software. As mentioned before, it is not easy to determine a programmer's affiliation by their commits only, but we can rely on their committer email addresses for some information. The competence parser is currently equipped with a list of international companies that give distinctive workplace email addresses to their employees, e.g. Apple (*apple.com*), Ericsson (*ericsson.com*), and Intel (*intel.com*). By running simple queries on the parsed data, we can identify the most competent company or team in a software project. In this case, "most competent" means the highest number of developers from a team that committed modifications to the project in the investigated time period. Of course, this value can be scaled by considering the amount and significance of the modifications.

Although the affiliation is obvious in some cases, some companies make their distinctive email domains accessible for non-employees as well, such as Google (*google.com*) and Yahoo (*yahoo.com*). This makes accurate analysis more difficult, since we cannot decide by a mere commit if the committer is an actual employee at one of these companies or not. This is why, even though the natural assumption is that e.g. Google is the most competent in Google Test, they are not clearly identifiable.

Another important information concerns the aforementioned risk factor of files that have not been modified in a long time, thus they are in danger of neglect. The risk of completely forgetting the purpose and content of these files gets significantly lower if their associated files are regularly maintained. However, this requires further deep analysis of the version control history.

6.1 Validation

The results of the plugin have been verified by applying the calculations on *CodeChecker*[9], an open-source source code analyzer which also applies static analysis to detect errors and malfunctions in programs. We conducted an experiment where we asked the developers of CodeChecker to evaluate their knowledge of the various modules of the project. The development team of CodeChecker consists of a small group of full-time developers and students and interns. The project is de-

[8]Personal names, email addresses and company names are anonymized due to privacy reasons.
[9]GitHub repo: `https://github.com/Ericsson/codechecker`

veloped in a multi-language environment with Python, C++ and JavaScript being the primary languages, which makes CodeChecker an excellent test project.

First, we determined the main modules of CodeChecker, then the entire commit history of more than 4800 commits was analyzed by the our method. Afterwards, the participating developers had to answer the question, *To what extent do you know the source code of this module: (module name)?*. The participants graded their knowledge in each module on a five-point scale, from 1 if they were unfamiliar with the module to 5 if they had detailed knowledge of the source code of the module. Each point on the scale corresponds to an interval of 20% understanding: 1 corresponds to 0-20% of source code knowledge, 2 to 20-40%, and so on.

After the analysis and the data collection from developers, we accumulated the results of competence calculation to match modules instead of files, and compared the results to the data given by developers. The experiment showed that the tool gave correct results in 48% of cases, with an 18% average deviation from the answers to our questions. The data indicated that the participants overestimated their knowledge in 50% of the time, and underestimated in 1.8%. The high value of overestimation suggests that developers on average have more knowledge of the source code than the commit history data shows. We also concluded that applying the results to modules instead of files could provide more useful results in everyday usage.

The average deviation of 18% indicates that the scale of proportions should be recalibrated to show more accurate and more detailed representation of knowledge in a software. We also concluded that the "unseen" knowledge which is inevitable for contribution should also be included in the method. More information of developer knowledge can be extracted from additional input, such as contributions in the project hosting system (e.g. GitHub, GitLab).

More details of the experiment and its results can be found in our previous paper [14].

6.2 Evaluation

Considering earlier studies about the possible aspects of data visualization tool evaluation [21, 24, 34, 43], we evaluated the usefulness of our tool can be evaluated based on the following criteria system:

Relevance: Most version control visualization tools focus on the actual source code (its architecture, evolution, change-proneness, etc.), and omits analyzing developer-related data (see Section 2 for examples). Our tool puts focus on collecting and visualizing information about the project-related knowledge of developers and teams, which can be used for developer-centered support within the development team.

Usability: The user interface of the plugin is integrated with the core frontend of CodeCompass. The functionalities (diagrams) are available through right-click menus on the source files, which is intuitive and easy to learn. The diagrams are generated by the graphic tools of GraphViz, which provides a versatile and clean tool set for graph-based diagrams. However, the entire current frontend of CodeCompass is obsolete. User-friendliness will be improved by a new, modern web frontend in the near future.

Functionality: The plugin is capable of providing a broad overview of the most knowledgeable developers and teams in a software project, as well as mapping the overall familiarity of individuals with every source file. This information can be used by developers, team leaders, scrum masters, project owners, etc. to improve the efficiency of task distribution, and provide better support for less experienced team members. The extracted data leaves more space for improvement, as the plugin could offer support for further developer- and knowledge-related questions. For example, based on the frequency of modifications in a source file, and the number of active developers who have contributed to the file, the tool could calculate which files are in danger of forgetting in case some developers leave the team.

Scalability: In Table 1 we can see that 500 commits were parsed from each test repository. However, the tool was tested on the reopsitory of CodeChecker which contained more than 4800 commits at the time of testing. Furthermore, during the development of the plugin, we continuously executed testing on a larger (10000) set of commits from the repository of LLVM repository. LLVM receives hundreds of contributions every day which makes it an excellent test project because the commits include several edge cases and exceptions that had to be handled to guarantee secure operation. Thus, the plugin is capable of handling large repositories.

Performance: Table 1 shows the runtime of the tool on the four test projects. We can see that the larger and the more complex commits get in a project, the more time the plugin takes to analyze them. The performance of the plugin can be improved by using an extension or wrapper library instead of the raw libgit2 API. The API includes many memory issues that wrappers tackle which may improve performance and reduce runtime significantly.

Flexibility: The plugin itself can be easily switched on and off during the usage of CodeCompass. On development level, the skeleton of diagram generation is readily provided in case of implementing new features. However, in order to improve or extend parsing, the algorithm needs to be understood first. Currently, commits from one branch can be analyzed at one parse. If a user, for example, wants to compare branches, the parser component of the plugin must be extended for which the user has to understand the appropriate libgit2 API elements.

Integration: The plugin is integrated with CodeCompass. The user may provide the number of commits they wish to analyze, and include the repository with the source code. This way, the plugin can be easily integrated with continuous integration systems. Future work includes analyzing data from project hosting systems.

7 Threats to validity

Although token-based comparison of edited files guarantees high-level accuracy in our change analysis, there can be some distorting factors. Just because a programmer has committed to a file some time ago in the past, does not mean that they are still competent in the file in its current state. For example, what if the programmer wrote a big chunk of a file, but someone came a couple days later and entirely refactored it? The two programmers would get similar results for this file, but the deleted content or overlap of modifications can cause distortions in the results. Refining the calculations from file level to smaller units, like classes or functions may help eliminate such anomalies in the future.

Another question that might come up is also content-related: what if someone modifies every file in a project by adding some minor change, such as license or copyright information to the comments? This way, this person is granted to have some percent of comprehension for every file for a while, even though they have not contributed to the project in its function and might not know anything about the code at all. Fortunately, software similarity checkers take care of this problem with token-based comparison. Renaming a variable, reordering a file, or adding comments do not count as significant modifications, there are no semantic differences are detected between file versions.

A more positively distorting factor lies in the nature of programming, which is particularly present when a programmer makes changes to the previously written work of someone else, let that be debugging, maintenance or further development: programmers can hardly (correctly) contribute to code without understanding at least some of it. That means actual competence in a file is most likely higher than our calculations say it to be.

In our calculations, we focus on exclusively the objectively measurable data, the structural significance of modifications. However, expertise and knowledge calculations include more subjective human factors, such as the capability of memory retention of a developer. According to the work of Ebbinghaus [9], people tend to forget detailed information quite quickly after hearing or reading said information. The forgetting curve has been tested by making people read and remember unrelated information like random words. Program code consists of more tightly connected units to which the forgetting curve may not apply. Future research includes human factors in the competence calculation.

Another threat to validity is that certain code modifications which seem like serious modifications to a code similarity checker might not really require deep knowledge of the code in question. Trivial refactoring patterns such as the extract function [16] can be applied with minimal understanding of the purpose of the code. However, we might usually assume that such tasks are assigned to a person who can take responsibility of the target module.

8 Conclusion and future work

In this research, we have developed the competence plugin, a visualization tool which uses information obtained from version control repositories to make developer-related information accessible for the user. The plugin answers some frequently asked questions during development, such as "Who knows the most about this part of the code?" and "Who can I ask for help in this task?". Also, the tool is essential to better plan the use of human resources, e.g. detecting when the knowledge in some part of the code dropped below a certain threshold, or when the knowledge is unevenly distributed between developers. In such cases, the project management can apply preventive actions to avoid complete loss of information about certain parts of the code. Aggregated views can be constructed from individual developers' knowledge information, using various heuristics. We tested the plugin as part of the CodeCompass open-source code comprehension framework, on several long-term open-source software projects that are continuously developed with frequent commits in their repositories. Our study has shown that the plugin is an effective code comprehension supporting tool that is useful for individual developers and project teams as well.

The competence plugin will be further developed by implementing new visualizations focusing on other programmer-related questions and utilizing more information from the version control system. Mapping the comprehension visualizations to project dependency graphs could more effectively help the developer in the process of learning about the software in a more conscious way. In the future, we will develop the calculation to not only apply to files, but modules as well. We plan to evolve the available visualizations with more subtle coloring, mouse hover functionality, and some logical diversity in node shapes. We also plan to implement an interactive interface where the users can map parsed email addresses and affiliations to their user accounts, and fill in the missing user data.

References

[1] Alcocer, J. P. S., Beck, F., and Bergel, A. Performance evolution matrix: Visualizing performance variations along software versions. In *Proceedings of the 2019 Working Conference on Software Visualization (VISSOFT)*, pages 1–11. IEEE, 2019. DOI: `10.1109/VISSOFT.2019.00009`.

[2] Alcocer, J. P. S., Jaimes, H. C., Costa, D., Bergel, A., and Beck, F. Enhancing commit graphs with visual runtime clues. In *Proceedings of the Working Conference on Software Visualization (VISSOFT)*, pages 28–32. IEEE, 2019. DOI: `10.1109/VISSOFT.2019.00012`.

[3] Bao, L., Xing, Z., Xia, X., Lo, D., and Li, S. Who will leave the company?: A large-scale industry study of developer turnover by mining monthly work report. In *Proceedings of the 2017 IEEE/ACM 14th International Conference*

on Mining Software Repositories (MSR), pages 170–181. IEEE, 2017. DOI: 10.1109/MSR.2017.58.

[4] Bassil, S. and Keller, R. K. Software visualization tools: Survey and analysis. In *Proceedings of the 9th International Workshop on Program Comprehension*, pages 7–17. IEEE, 2001. DOI: 10.1109/WPC.2001.921708.

[5] Brunner, T. and Porkoláb, Z. Advanced code comprehension using version control information. *IPSI Transactions on Internet Research*, 16(2):47–54, 2020. URL: http://ipsitransactions.org/journals/papers/tir/2020jul/p7.pdf.

[6] Chotisarn, N., Merino, L., Zheng, X., Lonapalawong, S., Zhang, T., Xu, M., and Chen, W. A systematic literature review of modern software visualization. *arXiv preprint arXiv:2003.00643*, 2020. DOI: 10.1007/s12650-020-00647-w.

[7] da Silva, I. A., Mangan, M. A., and Werner, C. M. CVS Watch: A group awareness tool applied to collaborative software development, 2004. URL: https://www.researchgate.net/profile/Marco-Mangan/publication/266471809_CVS_Watch_a_Group_Awareness_Tool_Applied_to_Collaborative_Software_Development/links/55a79ec308aea2222c747c4f/CVS-Watch-a-Group-Awareness-Tool-Applied-to-Collaborative-Software-Development.pdf.

[8] de F Carneiro, G., Magnavita, R., and Mendonça, M. Combining software visualization paradigms to support software comprehension activities. In *Proceedings of the 4th ACM Symposium on Software Visualization*, pages 201–202. ACM, 2008. DOI: 10.1145/1409720.1409755.

[9] Ebbinghaus, H. *Über Das Gedachtnis.* 1885. URL: https://home.uni-leipzig.de/wundtbriefe/wwcd/opera/ebbing/memory/GdaechtI.htm.

[10] Ellson, J., Gansner, E., Koutsofios, L., North, S. C., and Woodhull, G. Graphviz—open source graph drawing tools. In *Proceedings of the International Symposium on Graph Drawing*, pages 483–484. Springer, 2001. DOI: 10.1007/3-540-45848-4_57.

[11] Elsen, S. Visgi: Visualizing GIT branches. In *Proceedings of the 2013 First IEEE Working Conference on Software Visualization (VISSOFT)*, pages 1–4. IEEE, 2013. DOI: 10.1109/VISSOFT.2013.6650522.

[12] Faragó, C. *Maintainability of Source Code and its Connection to Version Control History Metrics.* PhD thesis, Department of Software Engineering, University of Szeged, Hungary, 2016.

[13] Faragó, C., Hegedűs, P., Végh, A. Z., and Ferenc, R. Connection between version control operations and quality change of the source code. *Acta Cybernetica*, 21(4):585–607, 2014. DOI: 10.14232/actacyb.21.4.2014.4.

[14] Fekete, A., Cserép, M., and Porkoláb, Z. Measuring developers' expertise based on version control data. In *Proceedings of the 2021 44th International Convention on Information, Communication and Electronic Technology (MIPRO)*, pages 1607–1612. IEEE, 2021. DOI: 10.23919/MIPRO52101.2021.9597103.

[15] Fittkau, F., Krause, A., and Hasselbring, W. Software landscape and application visualization for system comprehension with ExplorViz. *Information and Software Technology*, 87:259–277, 2017. DOI: 10.1016/j.infsof.2016.07.004.

[16] Fowler, M. *Refactoring*. Addison-Wesley Professional, 2018. URL: https://martinfowler.com/books/refactoring.html.

[17] Greene, G. J., Esterhuizen, M., and Fischer, B. Visualizing and exploring software version control repositories using interactive tag clouds over formal concept lattices. *Information and Software Technology*, 87:223–241, 2017. DOI: 10.1016/j.infsof.2016.12.001.

[18] Greene, G. J. and Fischer, B. Interactive tag cloud visualization of software version control repositories. In *Proceedings of the IEEE 3rd Working Conference on Software Visualization (VISSOFT)*, pages 56–65. IEEE, 2015. DOI: 10.1109/VISSOFT.2015.7332415.

[19] Greevy, O., Lanza, M., and Wysseier, C. Visualizing feature interaction in 3-D. In *Proceedings of the 3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 1–6. IEEE, 2005. DOI: 10.1109/VISSOF.2005.1684317.

[20] Hawes, N., Marshall, S., and Anslow, C. Codesurveyor: Mapping large-scale software to aid in code comprehension. In *Proceedings of the IEEE 3rd Working Conference on Software Visualization (VISSOFT)*, pages 96–105. IEEE, 2015. DOI: 10.1109/VISSOFT.2015.7332419.

[21] Isenberg, T., Isenberg, P., Chen, J., Sedlmair, M., and Möller, T. A systematic review on the practice of evaluating visualization. *IEEE Transactions on Visualization and Computer Graphics*, 19(12):2818–2827, 2013. DOI: 10.1109/TVCG.2013.126.

[22] Jermakovics, A., Sillitti, A., and Succi, G. Mining and visualizing developer networks from version control systems. In *Proceedings of the 4th International Workshop on Cooperative and Human Aspects of Software Engineering*, CHASE '11, pages 24—31, New York, NY, USA, 2011. Association for Computing Machinery. DOI: 10.1145/1984642.1984647.

[23] Kagdi, H., Yusuf, S., and Maletic, J. I. Mining sequences of changed-files from version histories. In *Proceedings of the 2006 International Workshop on Mining Software Repositories*, MSR '06, page 47–53, New York, NY, USA, 2006. Association for Computing Machinery. DOI: 10.1145/1137983.1137996.

[24] Kienle, H. M. and Muller, H. A. Requirements of software visualization tools: A literature survey. In *Proceedings of the 2007 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 2–9. IEEE, 2007.

[25] Krause-Glau, A., Bader, M., and Hasselbring, W. Collaborative software visualization for program comprehension. In *Proceedings of the 2022 Working Conference on Software Visualization (VISSOFT)*, pages 75–86. IEEE, 2022. DOI: 10.1109/VISSOFT55257.2022.00016.

[26] Kuhn, A., Erni, D., Loretan, P., and Nierstrasz, O. Software cartography: Thematic software visualization with consistent layout. *Journal of Software Maintenance and Evolution: Research and Practice*, 22(3):191–210, 2010. DOI: 10.1002/smr.414.

[27] Kumar J., M., Dubey, S., Balaji, B., Rao, D., and Rao, D. Data visualization on github repository parameters using elastic search and kibana. In *Proceedings of the 2018 2nd International Conference on Trends in Electronics and Informatics (ICOEI)*, pages 554–558, 2018. DOI: 10.1109/ICOEI.2018.8553755.

[28] Lanza, M., Ducasse, S., Gall, H., and Pinzger, M. Codecrawler: An information visualization tool for program comprehension. In *Proceedings of the 27th International Conference on Software Engineering*, pages 672–673, 2005. DOI: 10.1145/1062455.1062602.

[29] Lattner, C. and Adve, V. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 75–86. IEEE, 2004. DOI: 10.1109/CGO.2004.1281665.

[30] Löwe, W., Ericsson, M., Lundberg, J., and Panas, T. Software comprehension-integrating program analysis and software visualization. *Software Engineering Research and Practice*, 2002. URL: http://arisa.se/files/LELP-02.pdf.

[31] Marcus, A., Feng, L., and Maletic, J. I. Comprehension of software analysis data using 3D visualization. In *Proceedings of the 11th IEEE International Workshop on Program Comprehension*, pages 105–114. IEEE, 2003. DOI: 10.1109/WPC.2003.1199194.

[32] Mattila, A.-L., Ihantola, P., Kilamo, T., Luoto, A., Nurminen, M., and Väätäjä, H. Software visualization today: Systematic literature review. In *Proceedings of the 20th International Academic Mindtrek Conference*, pages 262–271, 2016. DOI: 10.1145/2994310.2994327.

[33] Merino, L., Ghafari, M., Anslow, C., and Nierstrasz, O. CityVR: Gameful software visualization. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 633–637. IEEE, 2017. DOI: 10.1109/ICSME.2017.70.

[34] Merino, L., Ghafari, M., Anslow, C., and Nierstrasz, O. A systematic literature review of software visualization evaluation. *Journal of Systems and Software*, 144:165–180, 2018. DOI: 10.1016/j.jss.2018.06.027.

[35] Merino, L., Seliner, D., Ghafari, M., and Nierstrasz, O. Communityexplorer: A framework for visualizing collaboration networks. In *Proceedings of the 11th edition of the International Workshop on Smalltalk Technologies*, pages 1–9, 2016. DOI: 10.1145/2991041.2991043.

[36] Mierle, K., Laven, K., Roweis, S., and Wilson, G. Mining student CVS repositories for performance indicators. In *Proceedings of the 2005 International Workshop on Mining Software Repositories*, MSR '05, page 1–5, New York, NY, USA, 2005. Association for Computing Machinery. DOI: 10.1145/1083142.1083150.

[37] Nagappan, N., Ball, T., and Zeller, A. Mining metrics to predict component failures. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, page 452–461, New York, NY, USA, 2006. Association for Computing Machinery. DOI: 10.1145/1134285.1134349.

[38] Oberhauser, R. VR-Git: Git repository visualization and immersion in virtual reality. In *Proceedings of the the Seventeenth International Conference on Software Engineering Advances*, pages 9–14, 2022. URL: https://www.thinkmind.org/index.php?view=article&articleid=icsea_2022_1_20_10032.

[39] Porkoláb, Z., Brunner, T., Krupp, D., and Csordás, M. Codecompass: An open software comprehension framework for industrial usage. In *Proceedings of the 26th Conference on Program Comprehension*, pages 361–369, 2018. DOI: 10.1145/3196321.3197546.

[40] Prechelt, L., Malpohl, G., Philippsen, M., et al. Finding plagiarisms among a set of programs with JPlag. *Journal of Universal Computer Science*, 8(11):1016–1038, 2002. URL: https://pdfs.semanticscholar.org/6281/93dbaa4b88101b8d7dd0a7c2eee86af5e32c.pdf.

[41] Shahin, M., Liang, P., and Babar, M. A. A systematic review of software architecture visualization techniques. *Journal of Systems and Software*, 94:161–185, 2014. DOI: 10.1016/j.jss.2014.03.071.

[42] Shinyama, Y., Arahori, Y., and Gondow, K. Analyzing code comments to boost program comprehension. In *Proceedings of the 2018 25th Asia-Pacific Software Engineering Conference (APSEC)*, pages 325–334, 2018. DOI: 10.1109/APSEC.2018.00047.

[43] Shneiderman, B. and Plaisant, C. Strategies for evaluating information visualization tools: Multi-dimensional in-depth long-term case studies. In *Proceedings of the 2006 AVI workshop on Beyond time and errors: Novel evaluation methods for information visualization*, pages 1–7, 2006.

[44] Slater, J., Anslow, C., Dietrich, J., and Merino, L. CorpusVis–Visualizing software metrics at scale. In *Proceedings of the 2019 Working Conference on Software Visualization (VISSOFT)*, pages 99–109. IEEE, 2019. DOI: 10.1109/VISSOFT.2019.00020.

[45] Svitkov, S. and Bryksin, T. Visualization of methods changeability based on VCS data. In *Proceedings of the 17th International Conference on Mining Software Repositories*, pages 477–480, 2020. DOI: 10.1145/3379597.3387451.

[46] Teyseyre, A. R. and Campo, M. R. An overview of 3D software visualization. *IEEE Transactions on Visualization and Computer Graphics*, 15(1):87–105, 2008. DOI: 10.1109/TVCG.2008.86.

[47] Wettel, R. and Lanza, M. Visualizing software systems as cities. In *Proceedings of the 2007 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 92–99. IEEE, 2007. DOI: 10.1109/VISSOF.2007.4290706.

[48] Wettel, R. and Lanza, M. Codecity: 3D visualization of large-scale software. In *Companion of the 30th International Conference on Software Engineering*, pages 921–922, 2008. DOI: 10.1145/1370175.1370188.

[49] Williams, C. C. and Hollingsworth, J. K. Automatic mining of source code repositories to improve bug finding techniques. *IEEE Transactions on Software Engineering*, 31(6):466–480, 2005. DOI: 10.1109/TSE.2005.63.

Acta Cybernetica **26** (2024) 455–474.

Ontology Supported Domain Knowledge Module for E-Tutoring System[*]

Ghanim Hussein Ali Ahmed[ab] and László Kovács[ac]

Abstract

E-tutoring Systems are computer applications that provide direct customized education to learners. This paper introduces a domain knowledge module for an E-tutoring system that allows knowledge stored in a well defined form to support reusability, shareability, flexibility, and standardability and to assists the storage of transfer and prerequisite knowledge relationships. The introduced knowledge domain module is designed in two ways the general concepts domain knowledge module and a specific domain knowledge module ontology. This innovative technique is helpful for students in enhancing their learning progress. Combining the proposed ontology domain knowledge module with an E-tutoring system can enhance the quality of intelligent problem solving. Also, it will be possible to reuse the knowledge domains. As a result, the proposal of the domain knowledge module for the E-tutoring system can enhance the teaching and learning process, support recommendations, generate hints. In the future the suggested module can be improved by adding some functionalities and automatically support the generation of problems and their solutions.

Keywords: E-tutoring system, domain knowledge module, ontology, SPARQL

1 Introduction

E-learning environments are increasingly getting popular in different contexts in academies, universities, and vocational training [1]. Therefore, suitable support of learners is also getting great significance. Besides, collaborative learning is also growing, which puts greater demands on learners, especially when the collaboration is implemented and tailored to enhance learning and teaching process [1].

[*]This work was carried out as part of the EFOP-3.6.1-16-00011 "Younger and Renewing University, Innovative Knowledge City, institutional development of the University of Miskolc aiming at intelligent specialization" project implemented in the framework of the Szechenyi 2020 program. The realization of this project is supported by the European Union, co-financed by the European Social Fund.

[a]Institute of Information Science, University of Miskolc, Hungary

[b]E-mail: ghanim@iit.uni-miskolc.hu, ORCID: 0000-0001-9582-259X

[c]E-mail: kovacs@iit.uni-miskolc.hu, ORCID: 0000-0003-2703-7228

DOI: 10.14232/actacyb.297804

In the earlier decades, learning platforms such as e-learning have been controlled by a technology called Learning Management Systems (LMS) [11]. such as Moodle, ATutor, or Blackboard, these LMS present integrated systems that allow and support a wide range of academic activities. Thus, instructors can use LMS to create courses and test suites, communicate with monitor learners, and evaluate their work. In addition, learners can learn, share, and collaborate through LMS. The problem is that LMS cannot offer only limited personalized learning services [12]. All learners are given access to the same learning resources and implements without considering the differences in knowledge level, interests, and goals.

Regarding this problem, a new learning platform is coming to provide learning facilities for customizing the education as one-to-one learning. This technology is an E-tutoring system. An E-tutoring system is defined by [7], as computerbased software that provides immediate personalized learning or feedback to learners without the involvement of humans while conducting a task. According to [14]. E-tutoring systems commonly involve four modules: the Knowledge Module, which incorporates content related to the rules and facts for a specific domain of interest to be given to the learner; the Tutoring Module, which creates and controls instructional interactions with the learners; the Learner Module, which is a dynamic representation of the current state of student knowledge; and the Learner Interface, which governs the interaction between the learner and the system [20].

The creation of an E-tutoring system can concentrate on different issues, including the tutoring decisions which will take place in the tutoring module as well as the rules and facts represented in the knowledge module. The goal of E-tutoring systems is to allow students to gain knowledge and develop skills in a particular field of study. However, delivering such tutoring services effectively, these systems must provide an explicit representation of the domain knowledge module that the topic of the education activity [21]. It must also be prepared with the tools by which the representation can be employed on the E-tutoring system for reasoning to solve problems in given domain of interest. E-tutoring systems must also include a domain specific knowledge module capable of generating and resolving domain problems as well as providing access to such knowledge to promote the dissemination and acquisition of this knowledge by students [2].

Developing and describing a domain knowledge module is a challenging problem that has been the issue of many investigations in the disciplines of both artificial intelligence and artificial intelligence in the educational domain (AIED) [4]. Scholars offered many approaches to expressing the knowledge base explicitly for domain knowledge modules [17]. The approaches presented have been drawn from several fields such as artificial intelligence, education science, knowledge engineering, knowledge management, knowledge representation, and software engineering. These techniques are semantic networks, frames, knowledge graphs, ontologies, rulebased, casebased, logicbased and belief networks. In this work, the authors focus on developing the domain knowledge module using ontology as acknowledge representation techniques.

Investigation into an ontology is evolving and increasingly becoming widespread in the computer science community. Its importance is recognized in many research and application areas, including knowledge engineering, database design and integration, information retrieval and extraction, and educational systems [21]. Ontology is a standard structure that offers a shared understanding of a specific area [2]. It represents the domain semantically explicitly, allowing intelligent access to the knowledge module. Ontology is a building block of semantic technologies. It is a formal description of the relevant knowledge concepts and their relations. The formal definition of ontology given by Gruber [10] ”ontology is an explicit specification of a conceptualization”. Ontologies determine or model the domain using concepts, attributes, and relationships, and this explicit formal representation provides meaning for the vocabulary. In computer and information science, ontology is a formal concept describing an artifact designed for a purpose, enabling the modeling of the domain knowledge module [21].

Most of the current E-tutoring systems solutions are developed for a particular domain, meaning the provided solution of a given knowledge domain will not be suitable for other knowledge domain [17]. Therefore, these systems developed for isolated knowledge bases have some limitations and drawbacks to using local knowledge bases. These limitations are limited knowledge base shareability and lack of standardability, flexibility, reusability, and manual control. Due to this problem, the solution in this work of the ontology domain knowledge module proposed to avoid the limitations and drawbacks of using local knowledge bases.

This paper constructs an ontology supported domain knowledge module for an E-tutoring system that provides help for enhancing the teaching and learning process. The applications of this modules are implemented in Python, which can be used to support the problem solving process. Accordingly, to improve and increase the learning quality and their processes, a novel ontology domain knowledge module develops by defining a set of relationships that would be adequate and clear to represent all possible relationships for developing and building the ontology domain knowledge model. In the proposed ontology domain knowledge module, two domain ontologies were introduced: a) general concepts for domain knowledge module ontology and b) specific domain knowledge module ontology. The general concepts for the domain knowledge module deal with the domain knowledge model concepts and define the relationship related to these concepts. The ontology of a selected domain knowledge module deals with the selected subject area that can relate the selected subject domain to the general concepts for the domain knowledge module. It seems like individuals or instances for the general concepts of the domain knowledge module.

This article is organized as follows: the introduction in the first section. The second section displays the related work, the third section illustrates the proposed the domain knowledge module and explains the proposal module in detail, the fourth section presents the implementation of the proposed module using Python and Owlready2 module, the fifth section demonstrates the result and discussion, and the conclusion in the sixth section.

2 Related Work

Researchers have followed a modern technology known as an E-tutoring system in different disciplines such as teaching, psychology, and artificial intelligence. The aim is to support the benefits of one-to-one education and allow learners to train their skills by conducting exercises on many interactive learning platforms. E-tutoring System is a software system developed to support students with immediate and personalized instruction or feedback, usually without human teacher intrusion. Several researchers, designers, and developers define E-tutor systems in different ways according to their interests. According to authors in [14], E-tutoring systems are intelligent instruction techniques using computer software and communication technologies their capabilities, practices to improve a human tutor who is an expert in the subject matter to enhance personalized learning in the form of one-to-one learning. Scholars have struggled since the early invention of computer applications to create intelligent learning systems that are more successful than human instructors [20], The fundamental role of using an E-tutoring system is to facilitate and customize student learning and achieve their activities effectively [20].

Nowadays, ontologies have become a proper representation scheme, and several application domains are considering adopting it. Recently, scholars found that the benefits of using ontologies in the learning field have greater support for designing and developing the coming E-learning platform [18]. However, concentrating on this technique is reusing domain knowledge resources for developing domain ontologies. Ontologies have been discussed in the context of E-Learning since early 2004 [21]. Ontologies are employed in different ways in E-Learning systems, depending on the E-Learning tasks they perform. Every part of E-learning activities can represent by a collection of welldefined associated entities that can have the same semantic representation, especially when dealing with concepts in particular domain knowledge module.

Ontology is the mostly used approach in the evolution of AI applications to model concepts in a particular domain of knowledge. In different terms, ontology is used to represent classes, concepts, and properties that usually exist in a specific domain and their relations. Ontology is a building block of semantic technologies. It is a formal description of the relevant knowledge concepts and their relations [2]. Ideally, ontologies should describe these welldefined meanings that can be formalized in languages such as Ontology Web Language (OWL) [4], Resource Description Framework (RDF) [17], or Resource Description Framework Scheme (RDFS) [17]. The formal nature of ontologies allows intelligent machines to interpret the meaning of concepts. In computer technology and information science areas, Ontology is the formal description of a specific domain by representing the concepts of a specific domain their properties and relationships among these concepts [2]. Concepts are usually classified regarding a hierarchical relationship of specialization, generalization, and containment among these concepts.

Data Model (DM) is a commonly used notion in many disciplines to deliver an abstract representation of its structure, function, behavior, or others [9]. DM expresses as a conceptual model that organizes knowledge structure, relationship, semantics, and consistency constraints [9]. According to authors [12]. Domain Model deals with a collection of knowledge concerning a specific topic, concept, or domain. Artificial intelligence is a tool that can extract and work with this data. The data represented in this case relates to the form of knowledge in a given domain. As an additional point, this model is not designed to replace the role of the instructor.

Domain knowledge module (DKM) introduces as a knowledge base for courses, topics, fundamental concepts, teaching units, or knowledge units in the E-tutoring system. In other terms, the DKM represents the knowledge base structure of a particular domain in a specific discipline which used as a component in E-tutoring systems. DKM is essential and valuable to educational communities because it is usually a targeted skill for developers to build a knowledge base [5]. DKM within the E-tutoring system describes the course, topics, knowledge domains, key concepts, teaching units, including the knowledge contents and competencies. However, the primary role of developing a DKM is to promote and reuse this knowledge on different E-tutoring frameworks. Domain module is the most significant part of E-tutoring systems providing a base for operational components such as learning material, content recommenders, or collaboration tools.

Scholars have investigated practices of knowledge representation such as semantic-based, rule-based, case-based, frame-based, Bayesian network, logic-based, and ontology-based. Rule-based models are also called Cognitive tutors. The rulebased models are built from cognitive task analysis, producing problem spaces or task models. These problem spaces or task models are constructed by observing the expert and novice users. Task models represent a set of production rules in which each rule represents an action corresponding to a task [16]. When a user tries to solve a given task, the user's reasoning ability is analyzed based on the rules applied by the user, i.e., the user's solution is compared step-by-step to the solution given by the expert.

Case-based is an artificial intelligence problem solving method that records experience into cases and associates the current problem with an experience [15]. The Case-based approach is operated in several domains, including pattern recognition, diagnosis, troubleshooting and planning, and intelligent e-learning.

A logical representation language has some definite rules for dealing with propositions and reasoning for knowledge and representation. Logical representation entails deducing a conclusion from many circumstances. This representation establishes many fundamental communication principles. It is composed of well-defined syntax and semantics that facilitate sound inference. Each phrase can transform into a logical form through syntax and semantics.

Frame-Based is one of the artificial intelligence techniques to structure the data, and it is used to separate information into substructures via the representation of stereotyped scenarios [3]. Frame-Based seems like a record form build-up of many characteristics and values used to describe an object in the real world. In addition, this object contains a collection of slots and their associated values. However, these slots come in a combination of shapes and sizes. Facets are the names and values assigned to slots.

A Bayesian network is also referred to as a belief network or Bayes net. Bayesian network is probabilistic and graphical in a form of graph with directed acyclic devoid of loops and self-connections used for knowledge representation for an uncertain domain, with each node indicating a random variable [22]. Each edge denotes a conditional probability associated with the associated random variables.

Semantic-based is a knowledge representation method that enables visualization of the knowledge via graphical networks [19]. This network incorporates nodes representing entities and arcs that reflect their relationships. Moreover, semantic-based classify concepts in different ways and can also connect them in a form of a graph. Ontologies, as semantic-based representation, have gained vital significance as one of the most commonly used techniques to describe and share knowledge in several disciplines such as E-learning systems, business modeling, software engineering, knowledge engineering [6].

Regarding the continuous development of new technology, it can change the way of teaching and learning. According to Fensel [23], the primary reason for the popularity of ontologies is due to providing "a shared and common understanding of a domain that can be communicated between people and application systems. Ontology can be constructed as a representation required for scale and variety in the design of educational frameworks. In the e-learning field, ontologies are employed in various applications extending from domain knowledge modules representation to automate generation and assessment of personalized learning materials. The concept of ontology is a useful technology that incorporates related resources, shares knowledge, and eliminates unnecessary data. Ontology is a fundamental description of the information in the world [13]. The ontology in computing refers to knowledge representation applying a collection of concepts and connections among them [8]. In the context of a targeted discipline, ontology is used to rationally reason and validate concepts in the semantic knowledge model. In theory, ontology is a "formal, explicit specification of a shared conceptualization [10]. It offers a shared vocabulary that can be employed to construct the domain knowledge model, involving objects, concepts, properties, and relationships.

A comparison was made using selected criteria according to the representation schemas used in the current works for representing the domain knowledge module as a part of E-tutoring systems. The criteria used to compare the proposed model with others in the literature covered the terms: standardbility, shareability, reusability, flexibility, simplicity, and reasoning engine, which are indicated in Table 1. In the columns, the following properties are represented: C1: standardability, C2: shareability, C3: reusability, C4: flexibility,C5: simplicity and C6: reasoning engine.

3 Proposed Domain Knowledge Module

Based on the properties of the learning content, two kinds of ontologies were introduced: a) general concepts for domain knowledge module ontology and b) specific domain knowledge module ontology. The general concepts of the proposed ontology

Table 1: Comparison of the models

Representation schema	C1	C2	C3	C4	C5	C6
knowledge graph	+	+	+	-	-	+
semantic network	+	+	+	-	-	+
rule-based	-	-	-	-	-	+
case-based	-	-	-	-	-	+
belief network	-	-	-	-	-	-
DITA architecture	+	+	+	-	-	-
ontology-based	+	+	+	+	+	+

domain knowledge module deal with the domain knowledge module's concepts and define the relationship related to these concepts. The ontology of a specific domain knowledge module deals with the selected subject area that can relate the selected subject domain to the general concepts for the domain knowledge module. It seems like individuals or instances for the general concepts of the domain knowledge module. These modules describe the topic to be learned, provide input to the domain module, provide specific feedback, select problem topic, generate suggestions, and support the learner module. The key structure of our proposed domain knowledge module is shown in Figure 1.

The proposed model is based on topics, attributes, task assessments, material forms, learning levels, learning rules and relations. To share and reuse the knowledge module in E-tutoring systems, ontology is utilized to manage and represent the domain knowledge module. The benefit of this model is to personalize the material forms, make suggestions, and automatic assessments for students.

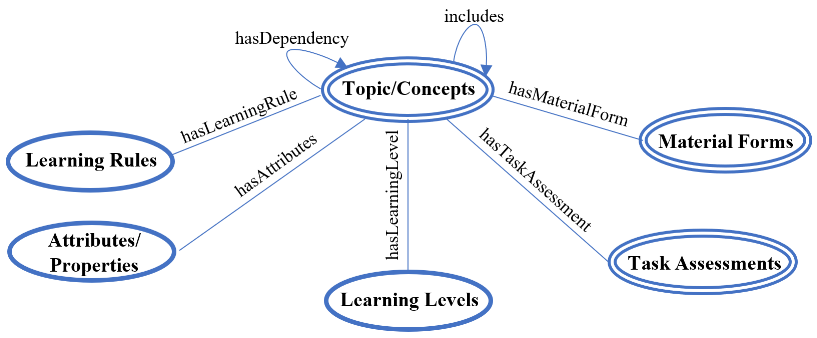


Figure 1: The proposed domain knowledge module

Based on the general concepts of the proposed domain knowledge module ontology displayed in Figure 1, topics, attributes, task assessments, learning levels, learning riles, and material forms terms refer to the following: Topic refers to knowledge modules as a unit of instruction representing domains, key concepts, or education units of the learning materials. Attributes refer to slots as an atomic property of the topics. Every slot has a value domain and concept type. Learning rule refers to rules or constraints defined on the topics and the attributes. A learning rule is a group of explicit or implicit ontology constraints or principles managing behavior or procedure in a particular activity area. Task assessment refers to the task as an activity related to the topics and attributes/properties. Task assessment describes an activity to be performed by learners. Material forms refers to teaching material for the topic. Material forms are teaching materials used to learn the topic. Material forms contain any parts of the academic institution or education material. Regarding the primary relationships, the ontology model contains the following elements: Topics taxonomy relationship: it defines the specialization among the topics. Topics component relationship: one topic consists of other topics. Topics and competency relationship among the topics. Figure 2. Indicates the case study structure of a specific domain knowledge module ontology for the world history domain in the E-tutoring system. We use various kinds of relationships in the case study, such as specialization or generalization, association, and containment. Containment denotes that a specific topic within a domain includes different concepts (has-a). The specialization or generalization indicates that topic has specific topics (is-a). Finally, association means a specific topic associated with attributes/properties, material forms, and task assessments. Based on Figure 1 and Figure 2, the following shows a brief description of a subject:

- *Topic Concepts*: Loop, Condition, Iterative Loop.
- *Dependency*: Logical Operator, Relational Operator.
- *Task Assessments*: program output, code review.
- *Attributes*: syntax, operators.
- *Material Forms*: Web, Textbook, Media.

4 Implementation of the Proposed Module

The proposed module is implemented as a prototype system includes the back-end, which is the ontology domain knowledge of the selected module, and the front-end, which is the interface design of the prototype system that allows the learner to use the functionality of the proposed module that can integrate the ontology domain knowledge module with the E-tutoring framework using Python. Python is the most commonly used language for implementing the ontology domain knowledge module, which can apply to the E-tutoring system. It is an object-oriented and extensible programming language [10]. It offers different modules, frameworks, and packages for handling and implementing ontology. Python can be integrated with

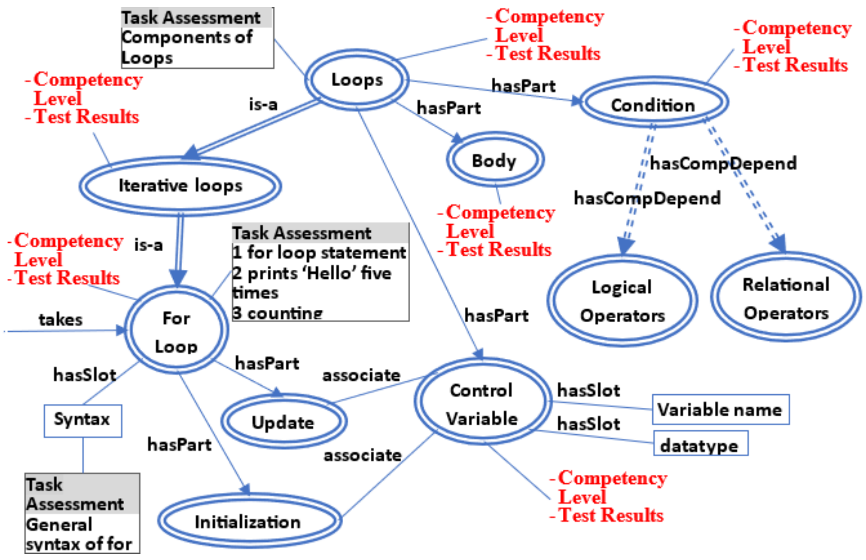
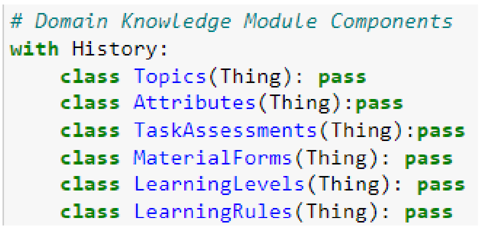


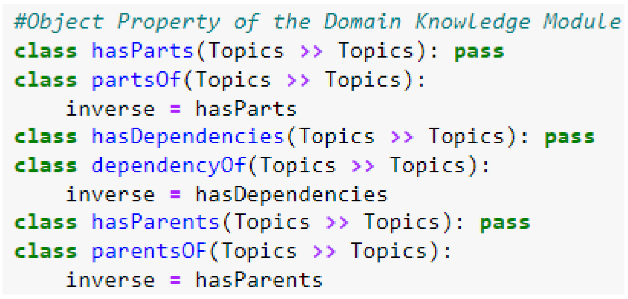
Figure 2: Domain knowledge module sample

an OWL ontology using Owlready2 and Flask. Owlready2 was employed to get transparent access to ontologies, manipulating the classes, object and data properties, individuals, property domains, ranges, annotations, constrained datatypes, disjoints, and class expressions. Flask is a Python Web framework that allows the rapid design of web applications [10]. The domain knowledge module considered here is "History," The ontology created consisted of the "World History." Figures 3, 4, 5, 6, 7, and 8 show the implementation of the proposed module. In Figure 3 a snippet shows topic components construction while. Figure 4 displays a snippet of the object property of the topic module. Figure 7 presents a topic instance. Checking the consistency of the ontology is shown in Figure 5. Rule construction and adding new knowledge are indicated in Figure 6. Figure 8 represents the topics and their task assessments.



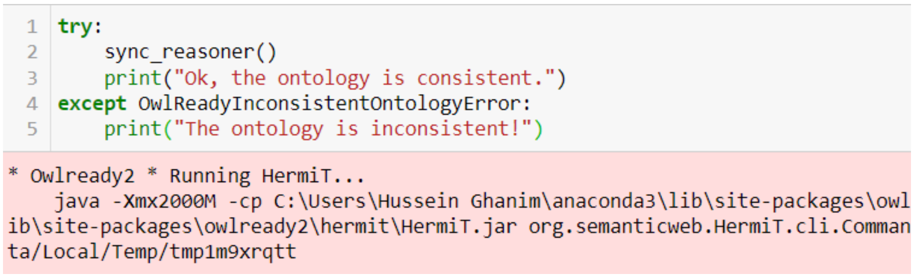
```
# Domain Knowledge Module Components
with History:
    class Topics(Thing): pass
    class Attributes(Thing):pass
    class TaskAssessments(Thing):pass
    class MaterialForms(Thing): pass
    class LearningLevels(Thing): pass
    class LearningRules(Thing): pass
```

Figure 3: Domain knowledge model components



```
#Object Property of the Domain Knowledge Module
class hasParts(Topics >> Topics): pass
class partsOf(Topics >> Topics):
    inverse = hasParts
class hasDependencies(Topics >> Topics): pass
class dependencyOf(Topics >> Topics):
    inverse = hasDependencies
class hasParents(Topics >> Topics): pass
class parentsOF(Topics >> Topics):
    inverse = hasParents
```

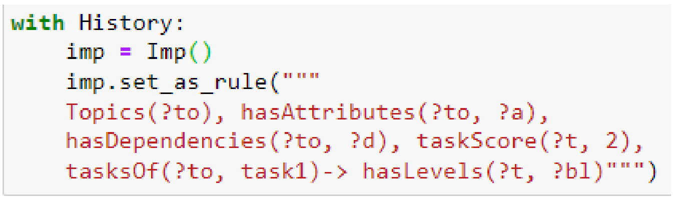
Figure 4: Object property of the domain knowledge



```
try:
    sync_reasoner()
    print("Ok, the ontology is consistent.")
except OwlReadyInconsistentOntologyError:
    print("The ontology is inconsistent!")
```

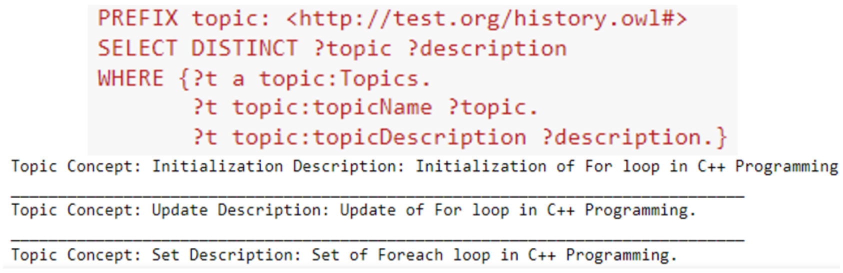
```
* Owlready2 * Running HermiT...
    java -Xmx2000M -cp C:\Users\Hussein Ghanim\anaconda3\lib\site-packages\owl
ib\site-packages\owlready2\hermit\HermiT.jar org.semanticweb.HermiT.cli.Comman
ta/Local/Temp/tmp1m9xrqtt
```

Figure 5: Checking the consistency of the ontology



```
with History:
    imp = Imp()
    imp.set_as_rule("""
    Topics(?to), hasAttributes(?to, ?a),
    hasDependencies(?to, ?d), taskScore(?t, 2),
    tasksOf(?to, task1)-> hasLevels(?t, ?bl)""")
```

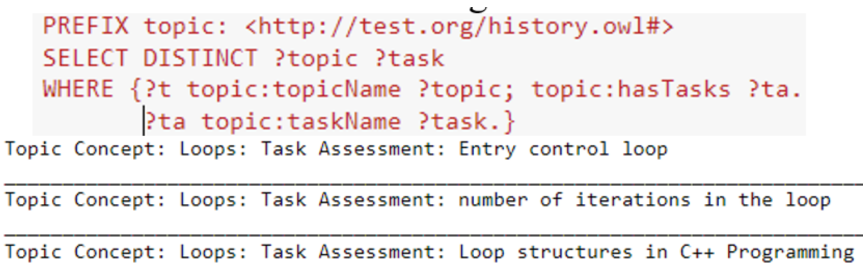
Figure 6: Rule construction for adding new knowledge



```
PREFIX topic: <http://test.org/history.owl#>
SELECT DISTINCT ?topic ?description
WHERE {?t a topic:Topics.
       ?t topic:topicName ?topic.
       ?t topic:topicDescription ?description.}
```

```
Topic Concept: Initialization Description: Initialization of For loop in C++ Programming
______________________________________________________________________________
Topic Concept: Update Description: Update of For loop in C++ Programming.
______________________________________________________________________________
Topic Concept: Set Description: Set of Foreach loop in C++ Programming.
```

Figure 7: A SPARQL query to retrieve the topics



```
PREFIX topic: <http://test.org/history.owl#>
SELECT DISTINCT ?topic ?task
WHERE {?t topic:topicName ?topic; topic:hasTasks ?ta.
       ?ta topic:taskName ?task.}
Topic Concept: Loops: Task Assessment: Entry control loop
_______________________________________________________________________
Topic Concept: Loops: Task Assessment: number of iterations in the loop
_______________________________________________________________________
Topic Concept: Loops: Task Assessment: Loop structures in C++ Programming
```

Figure 8: A SPARQL query to retrieve the task assessments

5 Functionality tests of the e-tutor module

In the functionality test of the proposed module, a specific field in IT, namely the loop structure in programming was selected as target domain. The main Topic Concepts in the corresponding knowledge model are the following elements: iterative loops, conditional loops, while loops, do-while loops, for loops, foreach loops, conditions, body, loop variable, logical operators (see Figure 9).

The domain knowledge module was extended with a set of task elements, too. Task assessments (T-assessments) refer to activities related to the T-concepts and attributes. T-assessments describes an activity to be performed by a student. It contains a task type followed by a list of arguments. It also can be given as a question-answer pair. In addition, the T-assessments can be represented in the form of activities performed by the learners. Syntactically, a T-assessment contains a task type followed by an argument list. The T-assessments may be either primitive or compound. A primitive T-assessments was considered to be performed by a planning operator: the task type is the planning operator's name to apply. The task arguments are the parameters for the operator. A compound task requires separating into smaller tasks using a method; any method whose title unifies the task type, and its arguments may probably be suitable for satisfying the task unit. A Task is aimed at a procedure that defines how to accomplish it. A Task is a combination of steps users follow to produce an expected outcome.

The task can be represented as a pair of questions Q and answer A. The set of all T-assessments is denoted by T=t where t is a T-assessment, the T-assessment is given as a question-answer applying as a function form as shown below: Q(S, Student): list the S of ST ¡S is a field, ST is a table¿ A(S, Student): select S from Student Another key element in the domain knowledge module is the competency relationship which is used to link the Topic concepts. Topic T1 is linked to topic T2 if T1 is a foundation to understand the concept T2. Based on this relationship, if a user fails a test on T2, the engine will suggest studying T1 to the user before retaking the test. The E-tutoring framework also involves a student model database, beside the domain models. It will register in this database the progress and the current knowledge level of the students. The knowledge level indicator is

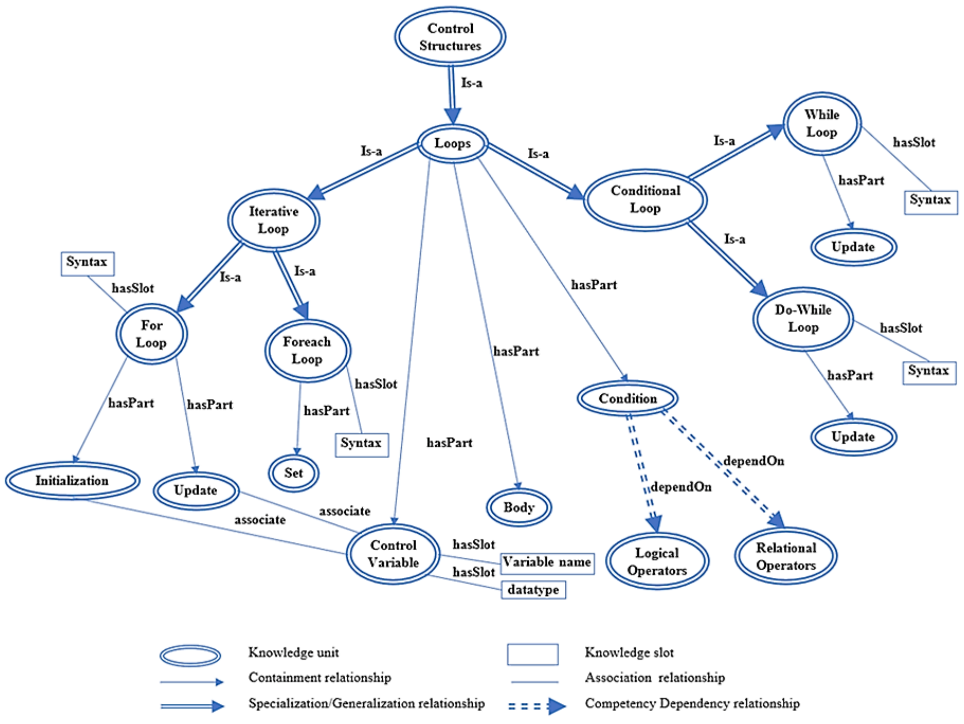


Figure 9: Topic Concepts in Loop Structures domain

given with a pair describing separately the grade of correct and incorrect answers similar to the intuitionistic logic approach (see Figure 10).

A functional test was done to evaluate the student knowledge level in the prototype system, which is integrated with the ontology domain knowledge module. Several task assessments with MCQ were performed. According to that, the result gives feedback according to the student's answer showing if the answer is correct or incorrect. The student status will update and offer suggestions. The suggestion related to task questions, student correct and wrong answers, the student result, knowledge level progress, and suggested reading materials. Figures 11, 12 display a test evaluation of the task assessments for different practices that allow the student to answer the task questions related to the topic concept, and then the student chooses the correct answer and moves to the next task question. After that, the prototype system checks whether the answer is correct or not and provides feedback according to the student's answer.

Figures 13, 14, 15 shows the generated result in detail for the task assessments, the selected task question related to the topic concept, and the learner's answer, and suggests related material if the learner wants to learn more about the chosen topic concept.

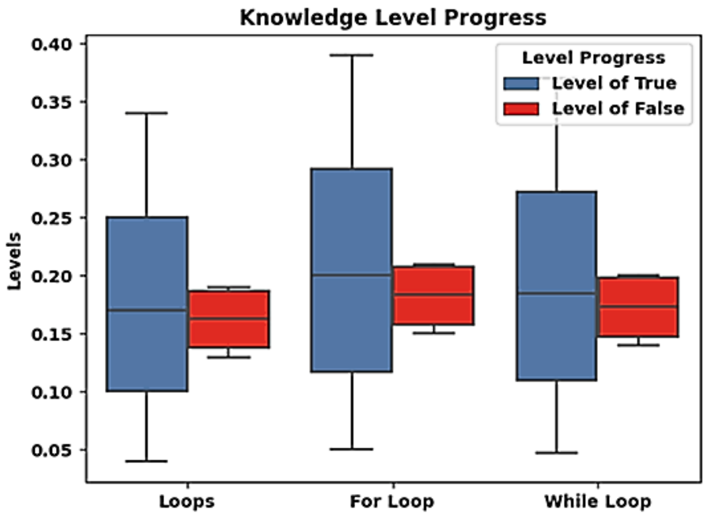


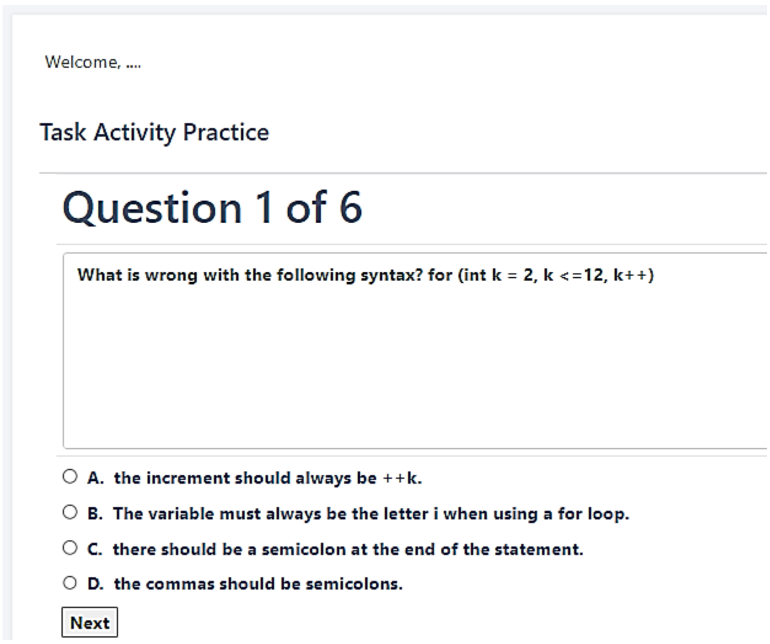
Figure 10: Knowledge level progress

6 Results and Discussions

A proposed model for an ontology domain knowledge module is given in this work. In Section 3, a theoretical module was described based on two kinds of ontologies. First, a general domain knowledge module ontology based on topics, attributes, task assessments, learning rules, learning levels, material forms, and their relations is indicated in Figure 1. Second, a specific domain knowledge module is designed as a case study for the History domain using different relationships such as specialization, generalizations, association, and containment, as shown in Figure 2. In Section 4, an implementation of the domain ontology using Python and Owlready2.

The current work deals with a knowledge module that helps learners understand all the key concepts in a specific topic. The future work recommended is coping with problem solving to support learners in understanding how to use knowledge modules in solving a practical problem. The majority of the domain knowledge module uses a separate knowledge base, which can satisfy the features of reusability, standardability, open knowledge, and flexibility. By employing ontology in constructing the domain knowledge module, we can avoid the problem of isolating knowledge bases, which is the problem of the most current solutions. The domain ontology can be involved in managing adaptive intelligent E-learning frameworks, supporting personalized learning, generating tasks, suggesting materials, and giving hints automatically.

The domain ontology considered in this work has diverse pedagogical goals. These goals include understanding specific domain facts and solving standard problems, obtaining a conceptual and intuitive understanding of the material in the



Welcome, ...

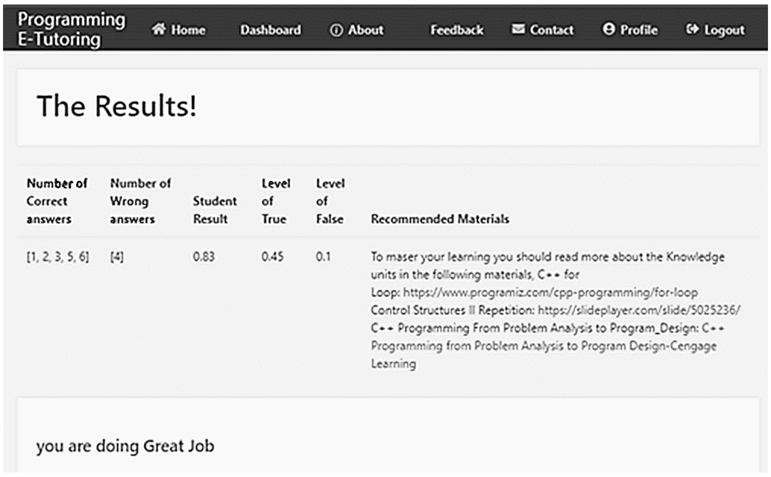
Task Activity Practice

Question 1 of 6

What is wrong with the following syntax? for (int k = 2, k <=12, k++)

- A. the increment should always be ++k.
- B. The variable must always be the letter i when using a for loop.
- C. there should be a semicolon at the end of the statement.
- D. the commas should be semicolons.

Figure 11: Task assessment interface

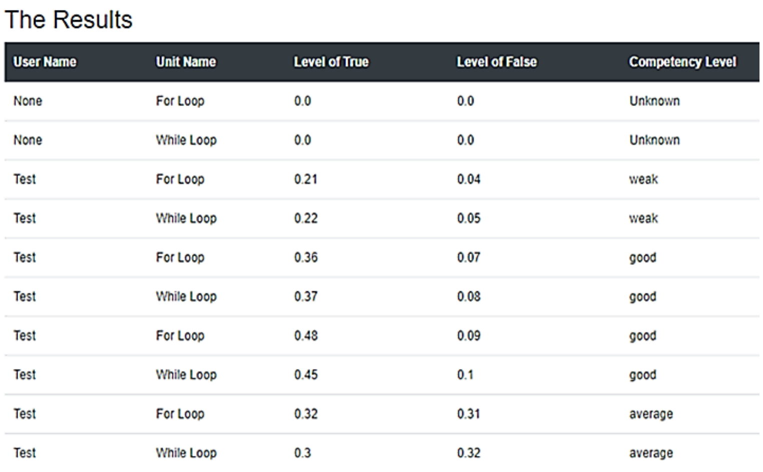


The Results!

Number of Correct answers	Number of Wrong answers	Student Result	Level of True	Level of False	Recommended Materials
[1, 2, 3, 5, 6]	[4]	0.83	0.45	0.1	To maser your learning you should read more about the Knowledge units in the following materials, C++ for Loop: https://www.programiz.com/cpp-programming/for-loop Control Structures II Repetition: https://slideplayer.com/slide/5025236/ C++ Programming From Problem Analysis to Program_Design: C++ Programming from Problem Analysis to Program Design-Cengage Learning

you are doing Great Job

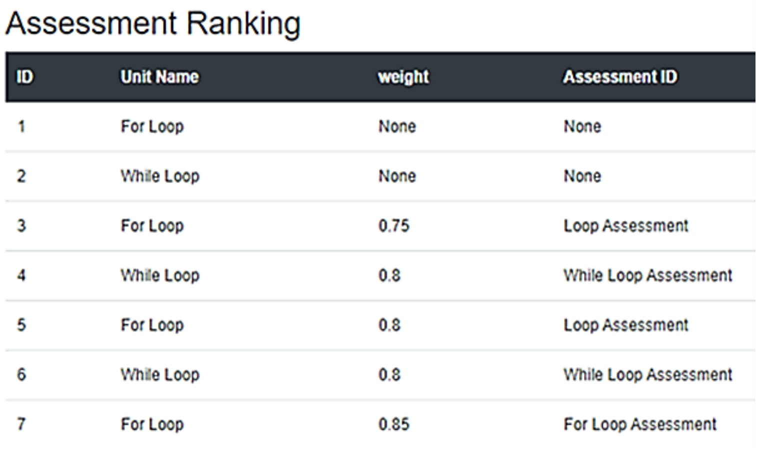
Figure 12: Task assessment result interface



The Results

User Name	Unit Name	Level of True	Level of False	Competency Level
None	For Loop	0.0	0.0	Unknown
None	While Loop	0.0	0.0	Unknown
Test	For Loop	0.21	0.04	weak
Test	While Loop	0.22	0.05	weak
Test	For Loop	0.36	0.07	good
Test	While Loop	0.37	0.08	good
Test	For Loop	0.48	0.09	good
Test	While Loop	0.45	0.1	good
Test	For Loop	0.32	0.31	average
Test	While Loop	0.3	0.32	average

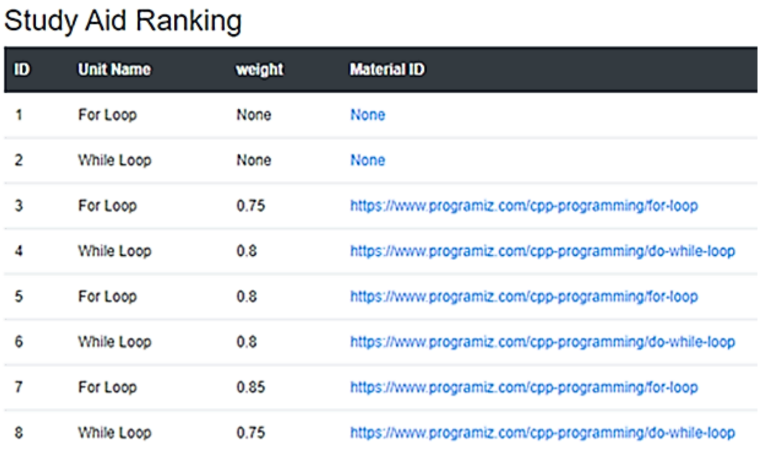
Figure 13: Task results interface



Assessment Ranking

ID	Unit Name	weight	Assessment ID
1	For Loop	None	None
2	While Loop	None	None
3	For Loop	0.75	Loop Assessment
4	While Loop	0.8	While Loop Assessment
5	For Loop	0.8	Loop Assessment
6	While Loop	0.8	While Loop Assessment
7	For Loop	0.85	For Loop Assessment

Figure 14: Assessment Ranking


Study Aid Ranking

ID	Unit Name	weight	Material ID
1	For Loop	None	None
2	While Loop	None	None
3	For Loop	0.75	https://www.programiz.com/cpp-programming/for-loop
4	While Loop	0.8	https://www.programiz.com/cpp-programming/do-while-loop
5	For Loop	0.8	https://www.programiz.com/cpp-programming/for-loop
6	While Loop	0.8	https://www.programiz.com/cpp-programming/do-while-loop
7	For Loop	0.85	https://www.programiz.com/cpp-programming/for-loop
8	While Loop	0.75	https://www.programiz.com/cpp-programming/do-while-loop

Figure 15: Study Ranking

selected domain, and learning general problem solving and metacognitive skills. A major feature of our model is that the knowledge representation techniques used have a standard structure. This standard structure allows general representational inference tools and control mechanisms, facilitating the pedagogical analysis of knowledge.

Several task assessments were tested with correct and incorrect answers using list of questions for evaluating the prototype system, which is integrated with the proposed ontology domain knowledge model. The prototype system generates feedback for the result according to the student answers, it suggests materials for learning more about the knowledge units if the answer is correct and it provide links to related materials if the answer is incorrect. The developed ontology domain knowledge module integrated with the prototype system can be used in managing adaptive intelligent e-learning frameworks in the future. Furthermore, the domain ontology knowledge model can meet various pedagogical goals. These goals include understanding specific domain facts and solving standard problems, obtaining a conceptual and intuitive understanding of the material in the selected domain, and learning general problem solving and metacognitive skills. A significant feature of the selected module is that the knowledge representation techniques have a standard structure. However, the standard structure of the proposed model can allow for general representational inference tools, control mechanisms, and facilitating pedagogical analysis of knowledge. In addition, combining the proposed ontology domain knowledge module with an E-tutoring system can enhance the quality of intelligent problem solving. Also, it will be possible to reuse the knowledge domains. Finally, a proposal of the domain knowledge module for the E-tutoring system can enhance the teaching and learning process, support recommendations, generate hints, and automatically support the generation of problems and solutions.

7 Conclusion

An E-tutoring requires content-specific knowledge and pedagogical, social, and technical factors to manage the complicated procedure affected in an E-learning platform. We first developed general concepts for domain knowledge module ontology which deals with the general concepts of the domain knowledge module and defines the relationship related to these concepts. Secondly, we design a specific domain knowledge module ontology that deals with the selected subject area that can link the selected subject domain to the general concepts for the domain knowledge module. It seems like an individual or instances for the general concepts of the domain knowledge module. Therefore, we created domain ontology for the Knowledge Module, especially in History Domain, to integrate with E-tutoring System. Using ontologies is concerned as a knowledge representation for describing the domain knowledge module, which can provide solutions to fundamental problems in this subject—also, an approach for organizing the ontology domain knowledge module presented and discussed. Furthermore, a proposed method explains how the ontology domain knowledge module can be combined with an E-tutoring system to enhance the quality of intelligent problem solving. Also, it will be possible to reuse the knowledge domains and design E-tutoring frameworks. Finally, a proposal of the domain knowledge module for the E-tutoring system can enhance teaching and learning, support recommendations, generate hints, and support the generation of problems and solutions automatically.

The developed ontology domain knowledge module can be used in managing adaptive intelligent e-learning frameworks in the future. Furthermore, the domain ontology knowledge module can meet various pedagogical goals. These goals include understanding specific domain facts and solving standard problems, obtaining a conceptual and intuitive understanding of the material in the selected domain, and learning general problem solving and metacognitive skills. A significant feature of the selected module is that the knowledge representation techniques have a standard structure. However, the standard form of the proposed module can allow for general representational inference tools, control mechanisms, and facilitating pedagogical analysis of knowledge. In addition, combining the proposed ontology domain knowledge module with an E-tutoring system can enhance the quality of intelligent problem solving. Also, it will be possible to reuse the knowledge domains. Finally, a proposal of the domain knowledge module for the E-tutoring system can enhance the teaching and learning process, support recommendations, generate hints. In the future the suggested module can be improved by adding some functionalities and automatically support the generation of problems and solutions.

References

[1] Çağatay Baz, F. New trends in e-learning. In Sinecen, M., editor, *Trends in E-learning*, chapter 1. IntechOpen, Rijeka, 2018. DOI: 10.5772/intechopen.75623.

[2] Abdoune, R., Lazib, L., and Dahmani-Bouarab, F. Disciplinary e-tutoring based on the domain ontology onto-tdm. In *2022 4th International Conference on Computer Science and Technologies in Education (CSTE)*, pages 143–147. IEEE, 2022. DOI: 10.1109/CSTE55932.2022.00033.

[3] Abu-Dawwas, W. and Abu-Dawas, M. Proposed frame-based expert system to construct student's knowledge model in intelligent tutoring systems. *Journal of Mathematical and Computational Science*, 10(5):1529–1537, 2020. DOI: 10.28919/jmcs/4567.

[4] Akinwalere, S. N. and Ivanov, V. Artificial intelligence in higher education: Challenges and opportunities. *Border Crossing*, 12(1):1–15, 2022. DOI: 10.33182/bc.v12i1.2015.

[5] Al-Yahya, M., George, R., and Alfaries, A. Ontologies in e-learning: review of the literature. *International Journal of software engineering and its applications*, 9(2):67–84, 2015. DOI: 10.14257/ijseia.2015.9.2.07.

[6] Alshboul, J., Ghanim, H. A. A., and Baksa-Varga, E. Semantic modeling for learning materials in e-tutor systems. *Journal Of Software Engineering & Intelligent Systems*, 6(2):17–24, 2021.

[7] Barnová, S., Krásna, S., and Gabrhelová, G. E-mentoring, e-tutoring, and e-coaching in learning organizations. In *11th International Conference on Education and New Learning Technologies*, Volume 1, page 6488–6493, 2019. DOI: 10.21125/edulearn.2019.1548.

[8] Chimalakonda, S. and Nori, K. V. An ontology based modeling framework for design of educational technologies. *Smart Learning Environments*, 7(1), 2020. DOI: 10.1186/s40561-020-00135-6.

[9] Gandon, F. L. Ontologies in computer science. *DIDACTICA MATHEMATICA*, 31(1):4346, 2013. DOI: 10.4018/978-1-61520-859-3.ch001.

[10] Gruber, T. R. A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5(2):199–220, 1993. DOI: 10.1006/knac.1993.1008.

[11] Kasim, N. N. M. and Khalid, F. Choosing the right learning management system (lms) for the higher education institution context: A systematic review. *International Journal of Emerging Technologies in Learning*, 11(6):55–61, 2016. DOI: 10.3991/ijet.v11i06.5644.

[12] Kraleva, R., Sabani, M., and Kralev, V. S. An analysis of some learning management systems. *International Journal on Advanced Science, Engineering and Information Technology*, 9(4):1190–1198, 2019. DOI: 10.18517/IJASEIT.9.4.9437.

[13] Ma, C. and Molnár, B. Use of ontology learning in information system integration: A literature survey. In *Asian Conference on Intelligent Information and Database Systems*, Volume 1178, page 342–353, 2020. DOI: 10.1007/978-981-15-3380-8_30.

[14] Marouf, A., Yousef, M., Mukhaimer, M. N., and Abu-Naser, S. S. An intelligent tutoring system for learning introduction to computer science. *International Journal of Academic Multidisciplinary Research (IJAMR)*, page 1–8, 2018. URL: https://philarchive.org/archive/MARATS-3.

[15] Masood, M. and Mokmin, N. A. M. Case-based reasoning intelligent tutoring system: An application of big data and IoT. *Proceedings of the 1st International Conference on Big Data Research*, Part F1325:28–32, 2017. DOI: 10.1145/3152723.3152735.

[16] Mendjoge, N., Joshi, A. R., and Narvekar, M. Review of knowledge representation techniques for intelligent tutoring system. In *2016 3rd International Conference on Computing for Sustainable Global Development (INDIACom)*, pages 2508–2512, 2016. URL: https://ieeexplore.ieee.org/document/7724713.

[17] Nkambou, R. Modeling the domain: An introduction to the expert module. In *Advances in Intelligent Tutoring Systems*, Volume 308, page 15–32, 2010. DOI: 10.1007/978-3-642-14363-2_2.

[18] Noy, N., McGuinness, D., Amir, E., Baral, C., et al. Research challenges and opportunities in knowledge representation. *Computer Science and Engineering Faculty Publications*, 2013. URL: http://corescholar.libraries.wright.edu/cse/218.

[19] Ram, A. Knowledge representation in intelligent tutoring system. In *Proceedings of the International Conference on Advanced Intelligent Systems and Informatics*, Volume 533, 2017. DOI: 10.1007/978-3-319-48308-5.

[20] Ramírez-Noriega, A., Martínez-Ramírez, Y., García, J. E. S., Bojórquez, E. M., Francisco Figueroa Pérez, J., Mendivil-Torres, J., and Miranda, S. Towards the automatic construction of an intelligent tutoring system: Domain module. In *New Knowledge in Information Systems and Technologies: Volume 1*, pages 293–302. Springer, 2019. DOI: 10.1007/978-3-030-16181-1_28.

[21] Sani, S. M., Aris, T. N. M., Mustapha, N., and Sulaiman, N. M. " ontology": A tool for managing domain module in an intelligent tutoring system. *International Journal of Advances in Computer Science and Its Applications*, 5(2):117–124, 2015. DOI: 10.15224/978-1-63248-056-9-38.

[22] Tato, A., Nkambou, R., Brisson, J., Kenfack, C., Robert, S., and Kissok, P. A Bayesian network for the cognitive diagnosis of deductive reasoning. In Verbert, K., Sharples, M., and Klobučar, T., editors, *Adaptive and Adaptable Learning*, Volume 9891 of *Lecture Notes in Computer Science*, pages 627–631. Springer International Publishing, 2016. DOI: 10.1007/978-3-319-45153-4_78.

[23] Verma, A. An ontology of ontological engineering. *International Journal of Engineering Sciences*, 24(63019):97–107, 2017. URL: https://ijoes.vidyapublications.com/paper/Vol24/11-Vol24.pdf.

Comparing Structural Constraints for Accelerated Branch and Bound Solver of Process Network Synthesis Problems[*]

Emília Heinc[ab] and Balázs Bánhelyi[ac]

Abstract

The P-Graph methodology can be used to find the optimal solution for large processing system. This methodology solves the combinatorial part of the problem more efficiently than the traditional branch and bound method due to the utilized relationships inherent in the structure. However, reducing the number of possibilities developed in the constraint functions also plays a major role in this algorithm. In this publication, we present a new constraint function that also takes into account the minimum cost structure and compares it with earlier versions.

Keywords: P-Graph, Accelerated Branch and Bound, structural constraint

1 Introduction

The task of process network synthesis is to determine the optimal structure of a process system, the optimal configurations, and operating sizes of the functional units that make up the system and perform various operations [12]. Process synthesis plays a critical role in reducing material, energy consumption, and negative environmental impacts, thereby increasing profitability. Several examples in the literature demonstrate that efficient process synthesis can reduce energy consumption by up to 50% and costs by 35% [13]. Ideally, the structure of a process and the operational configurations that make up the process could be designed and synthesized simultaneously because their performance interacts. In practice, however, it is extremely difficult due to the simultaneous continuous and discrete nature of the task. The discrete nature is caused by the structure of the process, which leads to

[*]The research was supported by the Government of Hungary and funded by the European Union Recovery and Resilience Plan within the framework of the Artificial Intelligence National Laboratory Program (RRF-2.3.1-21-2022-00004).

[a]Institute of Informatics, University of Szeged, Hungary

[b]E-mail: heincze@inf.u-szeged.hu, ORCID: 0009-0000-2172-505X

[c]E-mail: banhelyi@inf.u-szeged.hu, ORCID: 0000-0003-4408-9054

DOI: 10.14232/actacyb.297735

the combinatorial complexity of the problem that makes it complex to find an optimal solution to the problem. The process network synthesis problems formulate a MILP problem with many binary variables. Finding the optimal subnetwork is an NP-hard problem. Combinatorial analysis can be applied to this type of problem. The method is used to reduce the number of possible solutions by exploiting the unique properties of the so-called PNS (Process Network Synthesis) problems is the ABB (Accelerated Branch and Bound) method [10]. It is based on the branch and bound method, i.e. the method uses a lower bound submethod to exclude solutions that cannot provide a better solution than the currently known best solution. It is critical for the computation time of solving the problem with the B&B method to find a tighter lower bounding submethod. The currently available implementations and the previous studies do not exploit all the information, considering only the continuous part of the problem by calculating the LP relaxation of the MILP problem. In this article, we introduce a better lower bounding sub-method taking into consideration not just the continuous but also the structural nature of the PNS problem.

2 The Process Network Synthesis

2.1 P-Graph and basic notations methodology

The P-Graph (Process Graph) methodology was developed in the early 1990s for the complex chemical production system to model and optimize. Its name derives from a directed graph obtained by P-Graph, which provides the ability to use combinatorially feasible solution structures to determine the optimum for large tasks [8]. The P-Graph methodology based on graph theory and combinatorial techniques provides a solution to facilitate finding the optimal Process Network Synthesis (PNS) subproblem. The P-Graph can be described with (M, O) structures, where the M is the finite set of materials, and the finite set of operating units, $O \in \wp(M) \times \wp(M)$. The two sets are disjoint, i.e. $M \cap O = \emptyset$.

Definition 1. *The* P-Graph(M', O') *is the subgraph of the* P-Graph(M, O)*, i.e.* P-Graph$(M', O') \subseteq$ P-Graph(M, O) *if* $M' \subseteq M$ *and* $O' \subseteq O$.

The Process Network Synthesis problems, or PNS problems, in short, are defined as (P, R, O) triplets, where P stands for the set of products, R stands for the set of resources or raw materials and O is the set of operating units, where $P \cap R = \emptyset$, $P \subseteq M$, $R \subseteq M$, and $M \cap O = \emptyset$. If $(\alpha, \beta) \in O$, then α is the *input-set*, and β is the *output-set* of (α, β). The sets of input and output materials of set o of operating units are denoted by $mat^{in}(o)$ and $mat^{out}(o)$ separately, which are defined as:

$$mat^{in}(o) = \bigcup_{(\alpha,\beta)\in o} \alpha \text{ and } mat^{out}(o) = \bigcup_{(\alpha,\beta)\in o} \beta.$$

Let the either consumed or produced materials by the operating unit o be:

$$mat(o) = mat^{in}(o) \cup mat^{out}(o).$$

M is the set of materials in the PNS problem that are used (consumed or produced) by at least one operating unit from the set O, i.e. $M = \bigcup_{o \in O} mat(o)$.

In the methodology, a directed bipartite graph was used to represent the structure of a process system. We distinguish two kinds of nodes, the material (set of M) and operating units (set of O) in the graph. The directed edges represent the connection between the operating units and materials. The edges from the materials to the operating units mark the relation of the operating units that consume the materials. The edges from the operating units to the materials represent the relation of producing the given materials. In the PNS problems, costs can be assigned to the operating units and raw materials. In the following sections, the fix, installation cost is denoted by $fix\_cost(O') : \wp(O) \rightarrow \mathbb{R}_{\geq 0}$ and the operating cost is marked by $op\_cost(O') : \wp(O) \rightarrow \mathbb{R}_{\geq 0}$.

Definition 2. *The* P-Graph(m, o) *is a combinatorially feasible structure or solution structure, in short of the PNS problem (P, R, O) if it satisfies the following five listed axioms:*

(S1) $P \subset m$

(S2) $\forall X \in m, X \notin mat^{out}(o)$ *if and only if* $X \in R$

(S3) $o \subseteq O$

(S4) $\forall y_0 \in o, \exists \ path[y_0, y_n]$, *where* $y_n \in P$

(S5) $\forall X \in m, \exists(\alpha, \beta) \in o$ *such that* $X \in (\alpha \cup \beta)$

The aim of the problem is that the solution structure with the optimal summed cost is found, i.e. to produce all of the products from the raw materials at minimum cost. The algorithm that finds all the possible solution structures, will be the SSG algorithm, and the method that finds the solution structures with the optimal summed cost will be the ABB algorithm.

To solve this problem, we have to first find the maximum solution structure which contains all combinatorially feasible process structures. This method is called the MSG method (Maximal Structure Generation) [7].

Definition 3. *Let* $\Delta : M \rightarrow \wp(O)$, *where* $\Delta(X) = \{(\alpha, \beta) : (\alpha, \beta) \in O$ *and* $X \in \beta\}$ *i.e. determines the set of operating units producing all materials* $X \in M$.

Definition 4. *Let the decision mapping* $\delta(X)$ *be the subset of* $\Delta(X)$, *i.e.* $\delta(X) \in \Delta(X)$ $X \in M$.

Definition 5. *Expanding the decision mapping definition for the set of materials let the* $\delta[m] = (X, \delta(X) \,|\, X \in m)$.

Let the set of operating units of decision-mapping $\delta[m]$ be marked as $op(\delta[m])$, where

$$op(\delta[m]) = \bigcup_{X \in m} \delta(X).$$

Let the complement of decision-mapping $\delta[m]$ defined by

$$\overline{\delta}[m] = \{(X, Y) \mid X \in m \text{ and } Y = \Delta(X) \setminus \delta(X)\}.$$

Let $\overline{\delta}(X)$ be a set of operating units not included in $\delta(X)$:

$$\overline{\delta}(X) = \Delta(X) \setminus \delta(X).$$

Definition 6. *Decision mapping $\delta[m]$ is consistent if $|m| \leq 1$, or $(\delta(X) \cap \delta(Y)) \cup (\overline{\delta}(X) \cap \overline{\delta}(Y)) = \Delta(X) \cap \Delta(Y) \,\forall\, X, Y \in m$.*

Definition 7. *Let $\delta_1[m_1]$ and $\delta_2[m_2]$ be consistent decision-mappings. Then $\delta_1[m_1]$ is an extension of $\delta_2[m_2]$, i.e. $\delta_1[m_1] \geq \delta_2[m_2]$ if $m_2 \subseteq m_1$ and $\delta_1(X) = \delta_2(X)$ for $X \in m_2$.*

It can easily be proved that there is a bijective transition between the consistent decision mapping $\delta[m]$ and the P-Graph (m, o), where $m = \bigcup_{(\alpha,\beta)\in o}(\alpha \cup \beta)$ and $o = \bigcup_{X\in m} \delta(X)$ [6].

Definition 8. *Let the set of included operating units in $\delta[m]$ decision mapping be noted as O_I, where then*

$$O_I = op(\delta[m]).$$

Let the set of excluded operating units in the $\delta[m]$ decision mapping be O_E, where

$$O_E = op(\overline{\delta}[m]).$$

2.2 Solution Structure Generation algorithm

Further investigation is aided by the SSG (Solution Structure Generation) algorithm, which generates each combinatorially feasible structure exactly once. The algorithm is based on decision mappings. Decision mappings involve deciding which operating units produce the materials, i.e. which operating units are involved in a given solution structure [9]. Consequently, during decision mapping, we also decide which operating units will be excluded from the given structure. We must be consistent in our decisions because even if it has already been decided that an operating unit for one material should not be included in the structure, we cannot choose again when deciding on another material. All output materials for an operating unit, if included in the structure, must be specified, an inconsistent decision would result in certain substances being produced and certain substances not. The SSG implementation of the decision mapping-based algorithm calls itself recursively [6, 5].

As we see in the Algorithm 1, the procedure returns with all possible decision mappings over the PNS(P, R, O) problem. In the further section, the ABB algorithm will be introduced which is based on the SSG algorithm as working over all the possible decision mappings in the input PNS(P, R, O) problem but it returns with the optimal cost decision mapping.

Algorithm 1 Main and recursive part of SSG algorithm

```
Input $M, PNS(P, R, O)$
  procedure SSG($M, PNS(P, R, O)$)
  if $P = \emptyset$ then                                   If there is nothing to produce
    Stop
  end if
  SSGD($p, \emptyset, \emptyset$)
  return
  procedure SSGD(p, m, $\delta[m]$)
  if $p = \emptyset$ then
    write $\delta[m]$                                $\delta[m]$ defines a solution-structure
    return
  end if
  let $x \in p$
  $C := \wp(\Delta(x)) \setminus \emptyset$
  for $\forall c \in C$ do
    if $\forall y \in m$, $(c \cap \overline{\delta}(y) = \emptyset$ and $(\Delta(x) \setminus c) \cap \delta(y) = \emptyset)$ then
      $\delta[m \cup \{x\}] := \delta[m] \cup (x, c)$
      $SSGD\left(\left(p \cup mat^{in}(c)\right) \setminus (R \cup m \cup \{x\}), m \cup \{x\}, \delta[m \cup \{x\}]\right)$
    end if
  end for
  return
```

2.3 Mathematical model for P-Graph

The continuous variables of the model are denoted by x and the binary variables by y.

These variables are assigned to operating units. The continuous variable x_i indicates the operational size of the operating unit $O_i (\in O)$, and the binary variable y_i indicates whether the unit is in the structure or not: if the value of the binary variable $y_i \in \{0, 1\}$ is 0, then the operating unit O_i is not in the structure, and if it is 1, then it is involved. If the operation unit O_i is part of the structure, i.e. $y_i = 1$, then the operation size of the operation unit, which is a continuous variable x_i, can take any value from 0, and the operation unit between its upper capacity limit, U_i. Formally: $x_i \leq y_i U_i$, where U_i is the upper bound of the capacity of the operating unit O_i. If nothing of the sort is defined in the task boundary, an arbitrarily large number M can be used instead of U_i.

The objective function is to minimize cost. The cost is composed of the investment cost, the operating cost of the operating units, and the price of the raw material. These components cover the full cost of the network, i.e. the process to be synthesized considers the full cost. In the model, the costs of the operating units are simply entered with the relationship $a + bx$, where x is the size or capacity of each operating unit, a is the fixed cost, and b is the proportional cost which contains the price of the raw material.

In addition to the capacity constraints listed above, additional constraints are imposed on material balances, products, and raw materials. For products, we usually set lower limits to determine how much we need to produce at least of a given product, while for raw materials we may set upper limits if these types of raw material quantities are not available in unlimited amounts. Material balance conditions should be defined for intermediate products. These conditions state that at least as much of each intermediate product must be produced as is necessary for the operation of the operating units that use it, otherwise the manufacturing process would come to a standstill.

2.4 Accelerated Branch and Bound method

Since this is a mixed-integer programming problem, a general branch-and-bound-type method can be used to solve this model. Although the optimal solution to the problem can also be determined by using these methods, their efficiency can be further improved since the special properties of the synthesis tasks are not taken into account in the search for a solution. Accordingly, the P-Graph method for determining the optimal solution is a special algorithm of the Constraint and Separation type, ABB is used.

This algorithm uses the previously described decision mappings of the SSG algorithm for binary variables in the branch-and-bound tree. Earlier on, the branch-and-bound method used continuous relaxation of the mathematical model in addition to the structural constraints of the constraint SSG. In this relaxed model, the binary variables (y_i) were not considered, and the model was limited to determining the optimal values of the continuous variables (x_i). This optimization task provided a lower bound on the operating costs. In the following section, when the lower bounds are defined precisely for our branch-and-bound method, the relaxation type of the bound is nominated as $Lower\_Bound_{relaxed}$ (see Algorithm 2).

Algorithm 2 Relaxed lower bound algorithm

```
Input $PNS(P, R, O)$ problem, $O_I$, $O_E$
  procedure $Lower\_Bound_{relaxed}(PNS(P, R, O), O_I, O_E)$
  return $LP_{Solver}(PNS(P, R, O), O_I, O_E)$
```

The ABB algorithm is given as Algorithm 3 below. The method's inputs are the $PNS(P, R, O)$ problem in which the algorithm is running over, and the $Lower\_Bound$ function that will be used to prune the branch-and-bound tree. The M and $neutralExtension$ variables are used implicit by ABB algorithm. The M denotes the set of materials that will be considered, and the $neutralExtension$ is a Boolean variable that decides whether the neutral extension acceleration will be used or not.

The ABBD sub-method is called recursively in the ABB algorithm. It can be defined as a node from the branch-and-bound tree when it is called. The decision-

Algorithm 3 Main and recursive part of ABB algorithm

```
Input $PNS(P, R, O), Lower\_Bound$
Global variables $R, \Delta(x), (x \in M), U, currentbest$
  procedure ABB($PNS(P, R, O), Lower\_Bound$)
  $U := \infty; currentbest := \infty$
  $O := MSG(PNS(P, R, O))$
  ABBD(P, $\emptyset$, $\delta[\emptyset]$)
  return
  procedure ABBD(p, m, $\delta[m]$)
  if neutralExtension then
    let $\hat{\delta}[\hat{m}]$ be the maximal neutral extension of $\delta[m]$
    $p := (mat^{in}(op(\hat{\delta}[\hat{m}])) \cup P) \setminus (\hat{m} \cup R)$
    $O_I := op(\hat{\delta}[\hat{m}])$
    $O_E = op(\bar{\hat{\delta}}[\hat{m}])$
  end if
  $bound = Lower\_Bound(PNS(P, R, 0), O_I, O_E)$
  if $p = \emptyset$ then                                    Halting condition.
    if $U \geq bound$ then
      $U = bound$;
      update currentbest;
    end if
    return
  end if
  if $bound \geq U$ then                                     Cutting the branch.
    return
  end if
  $x \in p$;
  $C := \wp(\Delta(x)) \setminus \{\emptyset\}$;
  for $\forall c \in C$ do
    if $\forall y \in m, c \cap \bar{\delta}(y) = \emptyset \& (\Delta(x) \setminus c) \cap \delta(y) = \emptyset$ then
      $m' := m \cup \{x\}$;
      if $S(\delta[m']) = \emptyset$ then
        Continue;
      end if
      $\delta[m'] := \delta[m] \cup \{(x, c)\}$;
      $p := (mat^{in}(op(\delta[m'])) \cup P) \setminus (m' \cup R)$;
      $O_I := op(\delta[m'])$;
      $O_E := op(\bar{\delta}[m'])$;
      $ABBD(p, m', \delta[m'])$
    end if
  end for
  return
```

mapping is the fundamental tool for finding the optimal solution structure, as it defines obviously the node, or partial problem, S of the branch-and-bound tree. First, the partial problem has to be defined precisely for the algorithm. The definition of the S will be:

Definition 9. *Let $S(\delta[m_i])$ be the partial problem of ABB in solving PNS problem (P, R, O):*

$$S(\delta[m_i]) := \{\delta[m_k] : \delta[m_k] \geq \delta[m_i] \textit{ and } graph(\delta[m_k]) \in PNS(P, R, O)\},$$

where $graph(\delta[m_k]) \in PNS(P, R, O)$ means that $\delta[m_k]$ is the combinatorially feasible structure of $PNS(P, R, O)$.

The ABBD(P, $\emptyset$, $\delta[\emptyset]$) or $S(\emptyset)$ is called for the root node in the tree. The first parameter is the set of materials that are obligatory to produce. In the root node, it is the products from the PNS problem. The second parameter is the set of materials that have been produced already. The third parameter is the decision mapping which is valid in the current branch. As there is no material produced in the root node, the last two parameters are zero in the root node.

As an acceleration of the algorithm, the neutral extension of the current decision mapping, $\delta[m]$ can be used. The method extends the decision mapping with the materials that have to be produced. If we have one possible way to produce it, then the decision of which operating unit will produce these materials is straightforward [10]. It could be either that (1) all of the operating units are included and excluded, i.e. $\Delta(m) \cap (O_I \cup O_E) = \Delta(m)$, or (2) it is not decided which operating units produce it, but there is one possible operating unit that could produce it, the other operating units have already been excluded, i.e.$(O_I \cap \Delta(m) = \emptyset)$ & $(|\Delta(m) \setminus O_E| = 1)$.

The halting condition has been defined in the ABBD algorithm. If there aren't any obligatory products then the algorithm returns with the currently best cost. In this case from the validity of the lower bound sub-method, the return value equals the optimal cost of the current examined branch.

In the ABBD sub-method, as we calculate the minimal cost of the sub-tree in the branch-and-bound tree, the lower bound has to be calculated for the summed cost. The *Lower_Bound* must be a valid lower bound sub-method. The validity of the newly introduced lower bound will be proved in Theorem 1.

If the lower bound is greater than the actual best solution then the branch is going to be cut because the optimal solution can't be better than the current best solution.

After the examination of the halting conditions, the branching part is executed: as it is introduced in the SSG algorithm, we select a material x from the set of mandatory products, p that hasn't been decided which operating units produce yet. The son of the current partial problem will be the recursive sub-problem. The recursively called method's parameter is the decision mapping that consists of the material x and it is consistent with the current decision mapping in the main problem. Formally defined as:

$$\begin{aligned} son(S(\delta[m]), x) := \quad & \{S(\delta'[m']) : S(\delta'[m']) \neq \emptyset \ \&\ \delta'[m'] = \delta[m] \cup \{\, (x, c) \,\} \\ & \textit{for } c \in (\wp(\Delta(x))) \setminus \emptyset \ \&\ \delta'[m'] \textit{ is consistent}\}\,. \end{aligned}$$

3 Lower bound submethods in ABB algorithm

In each of our relaxed models in our ABB algorithm with a new lower bound, the minimum cost of the structurally feasible part of the residual was determined by the modified SSG algorithm for the free part of the P-Graph. The modified SSG algorithm is similar to the ABB method that has been called recursively with the sum of the fixed costs as the lower bound. The combination of these two optimization models gives a better lower bound for the sub-problems of B&B. Of course, operational and structural lower bounds need not necessarily come from the same feasible structure. The general $Lower\_Bound$ function inputs are the $PNS(P, R, O)$ problem the algorithm is running over, and the currently included and excluded operating units in the branch by the decision mapping $\delta[m]$.

3.1 Relaxed lower bound

Let us first consider the well-known and commonly used relaxed model, which is typically used for B&B algorithms and basic P-Graph solutions.

Definition 10. *$LP_{Solver}(PNS(P, R, O), O_I, O_E)$ is the optimal value of PNS problem with $y_i = 0$, where $o_i \in O_E$ and $y_i = 1$, where $o_i \in O_I$.*

The most obvious lower bound will be the optimum of the relaxation of the current MILP model derived from the decision mapping. The relaxed optimum of the current MILP problem is denoted as $LP_{Solver}(PNS(P, R, O), O_I, O_E)$, where O_I is the included operating unit, which means that also in the relaxed problem the $y_i = 1 \,|\, \forall o_i \in O_I$ is set. As the $x_i \leq y_i U_i$ constraints have been set for the operation number of operating unit i.

The same is true for O_E which denotes the excluded operating units also in the relaxed problem, i.e. $y_i = 0 \,|\, \forall o_i \in O_E$. In this case, the operating unit i has not been installed so from the constraints also listed above the operating unit cannot do any operation. (Also it has been excluded.)

The $x_i \leq y_i U_i$ relation is excluded from the constraints for all the free operating units. It means that the y_i for the free operating units will be set to 0 in the optimal solution as all the y variables have non-negative coefficients in the objective function.

To obtain the optimum of the relaxed LP problem that is get from the transformation of the current decision mapping, $\delta[m]$ to an LP problem is essential to choose a reliable LP solver. These LP solvers could be CPLEX [4], XPress [3], or Gurobi [11]. The Gurobi was used in our implementation.

3.2 Defining the new lower bound of ABB algorithm

Our aim is to introduce a new lower bound which gives a tighter bound to the optimum value than the current relaxed lower bound, and the LP solver calling number is not greater than the total LP callings in the case with the relaxed bound.

The new lower bound has been improved to take into consideration not only the currently summed included operating units' cost, but the remained operating units' fix costs as well.

The main idea was that the same ABB algorithm can be used for the calculation of the free or remained operating units with modified parameters. Both parameters of ABB, the PNS problem, and the lower bound have been modified to make the algorithm calculate the optimal structural value. A modified PNS problem is used to calculate the lower bound of the MILP model's integer part. Let $PNS_{IP}^{\delta[m]}(P', R', O')$ be the new PNS problem derived from the original $PNS(P, R, O)$ problem, where

- $R' = mat^{out}(O_I) \cup R$
- $P' = \big(P \cup mat^{in}(O_I)\big) \setminus R'$
- $O' = \{(\alpha', \beta') : \alpha' = \alpha, \beta' = \beta \setminus mat^{out}(O_I), (\alpha, \beta) \in O \setminus O_E\}$.

In the previously introduced ABB algorithm, the $PNS_{IP}^{\delta[m]}$ problem was formulated by the $get\_IP\_problem(PNS(P, R, O), O_I, O_E)$ method, where $O_I = op(\delta[m])$ and $O_E = op(\overline{\delta}[m])$ and $\delta[m]$ is the current decision mapping in the original problem. The second parameter is changed to the $Summed\_Weight$ lower bound (listed in Algorithm 4), which returns the summed cost of the included units. This lower bound cost gives a lower bound to the actual optimal structural cost of the free operating units. It will be proved in the Lemma 4.

Algorithm 4 The current summed structural cost

```
Input $PNS(P', R', O')$ problem, $O_I$, $O_E$
  procedure $Summed\_Weight(PNS(P, R, O), O_I, O_E)$
  return $\sum_{o \in O_I} fix\_cost(o)$
```

The optimal free operating units' structural cost will be added to the previously defined remained graph's optimal operating cost.

The detailed method is listed in Algorithm 5. In the rest of the section, the validity of our new lower bound will be proved.

Definition 11. *Let the A be all the possible extensions of decision mapping $\delta[m]$ over arbitrary $PNS(P, R, O)$ problem.*

$$A(\delta[m]) = \{\delta^*[m^*] \mid \delta^*[m^*] \geq \delta[m] \text{ and } \exists \delta^+[m^+], graph(\delta^+[m^+]) \in PNS(P, R, O) \text{ where } \delta^*[m^*] \leq \delta^+[m^+]\}.$$

Definition 12. *Let the δ' be all the possible decision of materials, and A' be all δ' decision mapping over the problem $PNS_{IP}^{\delta[m]}(P', R', O')$, i.e. $A' = A(\delta'[\emptyset])$.*

Definition 13. *Let the bijective transition F from $O \setminus O_E$ to $O' \setminus O_E$ be $F((\alpha, \beta)) = (\alpha, \beta \setminus mat^{out}(O_I))$, where O' is the operating unit set of $PNS_{IP}^{\delta[m]}$ problem, and O is the operating unit set of the PNS problem.*

Algorithm 5 New lower bound algorithm

```
Input $PNS(P,R,O)$ problem, $O_I$, $O_E$
  procedure $Lower\_Bound_{new}(PNS(P,R,O), O_I, O_E)$
  $PNS_{IP}^{\delta[m]}(P',R',O')$ :=get_IP_problem($PNS(P,R,O), O_I, O_E$)
  $O'$ :=MSG($PNS_{IP}^{\delta[m]}(P',R',O')$)
  IPCurrentBest:=0
  if $P' \neq \emptyset \,||\, O' \neq \emptyset$ then
    IPCurrentBest:=
            ABB($PNS_{IP}^{\delta[m]}(P',R',O')$, $Summed\_Weight$).currentbest
  end if
  return $LP_{Solver}(PNS(P,R,O), O_I, O_E) + IPCurrentBest$
```

Definition 14. *Let the $\mathcal{F}$ transition be the following: $\mathcal{F} : \wp(O \setminus O_E) \to \wp(O' \setminus O_E)$ $\mathcal{F}(Op) = \bigcup_{o \in Op} F(o)$.*

Definition 15. *Let $PNS_{IP}^{\delta[m]}(P',R',O')$ derived from $PNS(P,R,O)$ problem, where the current decision mapping is $\delta[m]$, i.e.*

$$PNS_{IP}^{\delta[m]}(P',R',O') = get\_IP\_problem(PNS(P,R,O), op(\delta[m]), op(\overline{\delta}[m])).$$

Definition 16. *Let the f be a transition between the $PNS(P,R,O)$ problem's set of decision mappings and the $PNS_{IP}^{\delta[m]}(P',R',O')$ problem's set of decision mappings. f is derived from the F in the following way:*

$$f_{\delta[m]} : A(\delta[m]) \to A', f_{\delta[m]}(\delta^*[m^*]) := \{(X, \mathcal{F}(\delta^*(X))) \,|\, \exists (X, Op) \in \delta^+[m^+] \text{ and } X \in m^*, \text{ where } graph(\delta^+[m^+]) \in PNS_{IP}^{\delta[m]}(P',R',O')\} \text{ and } Op \subseteq O'.$$

where $\delta^*[m^*] \geq \delta[m]$ and $PNS_{IP}^{\delta[m]}(P',R',O')$ is calculated over $\delta[m]$ and $\delta[m] \in A(\delta[\emptyset])$ and also the (S4) axiom is satisfied in $\delta[m]$.

Example 1. In the example shown in Figure 1, produce D are $\{o_1\}$, $\{o_2\}$, $\{o_1, o_2\}$.

If the D is produced by o_1, then $\delta[m] = \{(D, \{o_1\})\}$. If the o_1 operating unit is included, then o_2 gets in the excluded unit set. $\delta^*[m^*]$ can be $\{(D, \{o_1\})\}$ or $\{(D, o_1), (E, o_3)\}$. $f_{\delta[m]}(\{(D, \{o_1\})\}) := \emptyset$, $f_{\delta[m]}(\{(D, o_1), (E, o_3)\}) := \{(E, \{\mathcal{F}(o_3)\})\}$.

If the D is produced by o_2, then $\delta[m] = \{(D, \{o_2\})\}$. The O_E is $\{o_1\}$. The $\delta^*[m^*]$ can be $\{(D, \{o_2\})\}$ or $\{(D, \{o_2\}), (E, \{o_2\})\}$ or $\{(D, \{o_2\}), (E, \{o_2, o_3\})\}$. In the last two cases, f returns with $\emptyset$ since in the original $\delta[m]$, all products have already been produced by o_2, and R' includes the O_I outputs. In the first case, since all materials have also been produced from m^*, return with $\emptyset$.

If the D is produced by o_1 and o_2, then $\delta[m] := \{(D, \{o_1, o_2\})\}$. The $\delta^*[m^*]$ can be $\{(D, \{o_1, o_2\})\}$, $\{(D, \{o_1, o_2\}), (E, \{o_2\})\}$, $\{(D, \{o_1, o_2\}), (E, \{o_2, o_3\})\}$. In all cases, f returns with $\emptyset$ for the same reasons as in the previous case.

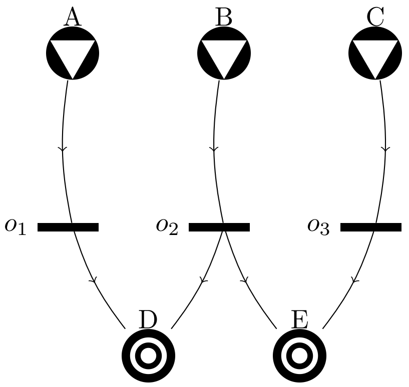


Figure 1: The illustration of an example for representing all possible cases of $f_{\delta[m]}(\delta^*[m^*])$.

Lemma 1. *Let's suppose that $\delta^*[m^*]$ and $\delta^{**}[m^{**}]$ are $\in A(\delta[m])$ and $\delta^{**}[m^{**}] \geq \delta^*[m^*](\geq \delta[m])$. Then $f_{\delta[m]}(\delta^{**}[m^{**}]) \geq f_{\delta[m]}(\delta^*[m^*])$.*

Proof. If $\delta^{**}[m^{**}] \geq \delta^*[m^*]$, then $m^* \subseteq m^{**}$ and $\forall X \in m^* : \delta^*(X) = \delta^{**}(X)$. Let's suppose that $X \in m^*$ and $(X, \mathcal{F}(\delta^*(X))) \in f_{\delta[m]}(\delta^*[m^*])$. Because of the definition of $f_{\delta[m]}$, exists at least one $\delta^+[m^+]$, where $\delta^+[m^+] \in graph(PNS_{IP}^{\delta[m]}(P', R', O'))$ and $X \in m^+$. As X also $\in m^{**}$, then it is also will be produced in $f_{\delta[m]}(\delta^{**}[m^{**}])$. Because $\delta^*(X) = \delta^{**}(X)$, as $(X, \mathcal{F}(\delta^{**}(X))) \in f_{\delta[m]}(\delta^{**}[m^{**}])$, then $(X, \mathcal{F}(\delta^*(X)))$ $\in f_{\delta[m]}(\delta^{**}[m^{**}])$.

$\square$

Lemma 2. *Let suppose that $\delta^*[m^*] \in A(\delta[m])$ and $graph(\delta^*[m^*]) \in PNS(P, R, O)$. Then $graph(f_{\delta[m]}(\delta^*[m^*])) \in PNS_{IP}^{\delta[m]}(P', R', O')$.*

Proof. It is wanted to be proven that $graph(f_{\delta[m]}(\delta^*[m^*]))$ is a solution structure of the $PNS_{IP}^{\delta[m]}(P', R', O')$ problem if $graph(\delta^*[m^*])$ is a solution structure in the $PNS(P, R, O)$ problem. A P-Graphs is considered a solution structure if and only if it satisfies the five axioms listed previously. In the proof, it is shown that $graph(f_{\delta[m]}(\delta^*[m^*]))$ satisfies all the five axioms in the $PNS_{IP}^{\delta[m]}(P', R', O')$ problem.

The first axiom is satisfied if $P' \subset mat^{out}(f_{\delta[m]}(\delta^*[m^*]))$. The P' is equal to $(P \cup mat^{in}(O_I)) \setminus (mat^{out}(O_I) \cup R)$. The P' set is equivalent to $(P \setminus mat^{out}(O_I)) \cup (mat^{in}(O_I) \setminus (mat^{out}(O_I) \cup R))$, as $P \cap R = \emptyset$. It can be divided into two disjoint sets, $P \setminus mat^{out}(O_I)$ and $(I \cap mat^{in}(O_I)) \setminus (mat^{out}(O_I))$, where I is the set of intermediate materials from the original problem $(I = M \setminus (R \cup P))$. Since all connections to $mat^{out}(O_I)$ were eliminated in the $PNS_{IP}^{\delta[m]}$ problem, none of the previously produced materials (i.e. $mat^{out}(O_I)$) are chosen in $f_{\delta[m]}(\delta^*[m^*])$. All materials that are in both I and $mat^{in}(O_I)$ had to be produced. In addition, all elements of P have already been produced. Therefore, all elements from $(P \setminus$

$mat^{out}(O_I)) \cup (mat^{in}(O_I) \setminus (mat^{out}(O_I) \cup R))$ are produced in the resulting P-Graph.

The second axiom states that $\forall X \in m^f, X \notin mat^{out}(o^f)$ if and only if $X \in R'$. From the proof of the first axiom, it is known that the elements of $mat^{out}(O_I)$ are not in $mat^{out}(f_{\delta[m]}(\delta^*[m^*]))$. Based on the second axiom, it can be concluded that R is also not in the set of produced materials.

The third axiom is satisfied if $op(f_{\delta[m]}(\delta^*[m^*])) \subseteq O'$. It is trivially satisfied since the operating units in the P-Graph are determined as $(F(o) \,|\, \forall o \in f_{\delta[m]}(\delta^*[m^*])$.

The fourth axiom states that $\forall y_0 \in op(f_{\delta[m]}(\delta^*[m^*])), \exists\; path[y_0, y_n],\; where\; y_n \in P'$, i.e. for all operating units there is a path to at least one product. All connections to any operating unit to the previously produced materials $(mat^{out}(O_I))$ are eliminated in $f_{\delta[m]}(\delta^*[m^*])$. It induced that, if an operating unit only produces these materials, then it will be left out in $f_{\delta[m]}$, as none of the solution structures in the $PNS_{IP}^{\delta[m]}$ problem contains it. If the operating unit is not eliminated in the resulting decision mapping, then whether it has a path to a product from P, because $\delta^*[m^*]$ is a solution structure in the original problem, or if it does not have a path to any original product, then it has at least one path to material from $mat^{in}(O_I)$. From this material from $mat^{in}(O_I)$ as the fourth axiom also applied for $\delta[m]$, originally, in the $\delta^*[m^*]$ has a path to a product from P.

The fifth axiom states that for all materials, there is at least one operating unit that either produces or consumes them. This axiom is trivially satisfied because the materials in the P-Graph correspond to the consumed and produced materials of the decision mapping. □

Lemma 3. *Images of all extension of $\delta[m]$ in the PNS problem are possible decision mapping in $PNS_{IP}^{\delta[m]}$ problem, i.e. $f_{\delta[m]}(\delta^*[m^*]) \in A' \; \forall \delta^*[m^*] \geq \delta[m]$.*

Proof. The lemma claims that $\exists \delta^+[m^+]$ where $\delta^+[m^+] \in PNS_{IP}^{\delta[m]}(P', R', O')$ and $f_{\delta[m]}(\delta^*[m^*]) \leq \delta^+[m^+]$. As $\delta^*[m^*] \in A(\delta[m])$, this implies that $\exists \delta^+_{PNS}[m^+_{PNS}]$ where $graph(\delta^+_{PNS}[m^+_{PNS}]) \in PNS(P, R, O)$ and $\delta^*[m^*] \leq \delta^+_{PNS}[m^+_{PNS}]$. Because of Lemma 2, $graph(f_{\delta[m]}(\delta^+_{PNS}[m^+_{PNS}]) \in PNS_{IP}^{\delta[m]}(P', R', O')$.

From Lemma 1 and $\delta^*[m^*] \leq \delta^+_{PNS}[m^+_{PNS}]$, $f_{\delta[m]}(\delta^*[m^*]) \leq f_{\delta[m]}(\delta^+_{PNS}[m^+_{PNS}])$. As $graph(f_{\delta[m]}(\delta^+_{PNS}[m^+_{PNS}])) \in PNS_{IP}^{\delta[m]}(P', R, O')$, the $\delta^+[m^+]$ is looking for will be $f_{\delta[m]}(\delta^+_{PNS}[m^+_{PNS}])$. □

Lemma 4. *The summed weight of included units of $f_{\delta[m]}(\delta^*[m^*])$ always is a value less than the summed weight of $\delta^*[m^*]$, i.e. $Summed\_Weight(O_I(f_{\delta[m]}(\delta^*[m^*]))) \leq Summed\_Weight(O_I(\delta^*[m^*]))$.*

Proof. Derived from the definition of $f_{\delta[m]}(\delta^*[m^*])$, if $(X, Op') \in f_{\delta[m]}(\delta^*[m^*])$, then $\exists Op \subseteq O$ where $\mathcal{F}(Op) = Op'$ and $(X, Op) \in \delta^*[m^*]$. It is previously known from the definition of F that the costs of the operating units have not been changed by the transition. It induces that the $Summed\_Weight(Op) = Summed\_Weight(Op')$. □

Lemma 5. *If there is an optimal solution to the $PNS(P, R, O)$ problem, and suppose it is $\delta^*[m^*]$, then $f_{\delta[m]}(\delta^*[m^*])$ is also the optimal solution among all possible $f_{\delta[m]}(\delta^{**}[m^{**}])$ where $\delta^{**}[m^{**}] \in A(\delta[m])$, if the aim of the optimization problem is to find the minimal fix costed solution structure, i.e. P-Graphs that satisfy only the axioms.*

Proof. The statement that $f_{\delta[m]}(\delta^*[m^*])$ is the optimal solution among all possible $f_{\delta[m]}(\delta^{**}[m^{**}])$ where $\delta^{**}[m^{**}] \in A(\delta[m])$ can be proved indirectly. Suppose that, $\exists\delta^{opt}[m^{opt}](\in A)$ where $f_{\delta[m]}(\delta^{opt}[m^{opt}])$ is the optimal solution among the $f_{\delta[m]}(\delta^{**}[m^{**}])$ in the sense of summed fix cost. In this case, the materials from M can be divided into three disjoint categories. The three parts are (1) m previously produced materials, (2) $bp = mat^{out}(O_I) \cap \overline{m}$ set of byproducts, (3) $np = (M \backslash mat^{out}(O_I))$ non-produced materials. It is obvious that $np \cup bp \cup m = m^*$, and the three sets are disjoint.

(1) If $mat \in m$. It is known that $\delta[m] \leq f_{\delta[m]}(\delta^*[m^*])$ and also $\delta[m] \leq f_{\delta[m]}(\delta^{opt}[m^{opt}])$. This part of the material production will be the same in the two possible solution structures.

(2) If $mat \in bp$. In this case, the mat can be produced neither in $f_{\delta[m]}(\delta^*[m^*])$, nor in $f_{\delta[m]}(\delta^{opt}[m^{opt}])$, as $mat \in mat^{out}(O_I)$ and it implies that none of any operating units from the possible arbitrary $f_{\delta[m]}(\delta^{**}[m^{**}])$ can produce mat. If $\delta^*[m^*]$ is the optimal solution in the PNS problem, then none of any mat from bp will be produced, as $mat \in mat^{out}(O_I)$, and only the installation costs count when the objective function was calculated.

(3) If $mat \in np$. In this case, the input and the output materials are remained the same compared to the F mapping of the operating units from $\Delta(mat)$. In other words, if $op \in \Delta(mat)$, then $mat^{in}(op) = mat^{in}(F(op))$ and $mat^{out}(op) = mat^{out}(F(op))$. This means that the operating units, that can produce the mat have remained the same in the sense that they can be defined by the same pair of material sets. It is known that only the materials from np can be produced in $f_{\delta[m]}(\delta^{opt}[m^{opt}])$ and $f_{\delta[m]}(\delta^*[m^*])$ because only these materials are produced in the arbitrary $f_{\delta[m]}$ mapping from the possible three categories.

The $\delta^*[m^*]$ is the optimal solution of the PNS problem, and $op(f_{\delta[m]}(\delta^*[m^*])) \subseteq \mathcal{F}(op(\delta^*[m^*]))$. If the objective function is also divided into three part according to the three previously defined disjoint finite sets of materials (m, bp, and np), then the $Summed\_Weight(op(\delta^*[m^*])) = Summed\_Weight(op(\delta[m])) + Summed\_Weight(op(\delta^*[np \cap m^*]))$, as none of any materials from bp has been produced. $Summed\_Weight(op(\delta^{opt}[m^{out}])) = Summed\_Weight(op(\delta[m])) + Summed\_Weight(op(\delta^{opt}[bp \cap m^{opt}])) + Summed\_Weight(op(\delta^{opt}[np \cap m^{opt}]))$. The $Summed\_Weight(op(\delta^*[np \cap m^*])) > Summed\_Weight(op(\delta^{opt}[np \cap m^{opt}]))$ is known from the indirect statement. Only the materials from np are produced in the $f_{\delta[m]}$ mappings, and the summed weight of $\delta^{**}[m^{**} \cap np]$ is equal to the summed weight of $f_{\delta[m]}(\delta^{**}[m^{**} \cap np])$ for arbitrary $\delta^{**}[m^{**}] \in A(\delta[m])$. It is a contradiction because $\exists\delta'[m'] \in A(\delta[m])$ where $Summed\_Weight(op(\delta'[m'])) < Summed\_Weight(op(\delta^*[m^*]))$ and $\delta'[m'] = \delta[m] \cup \delta^{opt}[np \cap m^{opt}]$ is also a possible solution structure in the PNS problem because $bp \subseteq mat^{out}(O_I)$. It follows that

$bp \subseteq m'$. □

Lemma 6. *$ABB(PNS_{IP}^{\delta[m]}(P', R', O'), Summed\_Weight)$ always returns with a valid lower bound for the $PNS_{IP}^{\delta[m]}(P', R', O')$ problem.*

Proof. Let's suppose that, running the ABB algorithm over $PNS(P, R, O)$ problem, $\delta[m]$ is the decision mapping in the current branch, i.e. the $PNS_{IP}^{\delta[m]}(P', R', O')$ problem is derived from $\delta[m]$. From Lemma 3 is known that, for all possible subnodes of the current branch-and-bound node $\delta^*[m^*]$ (i.e. $\delta^*[m^*] \geq \delta[m]$), the $f_{\delta[m]}(\delta^*[m^*])$ is a possible decision mapping in the $PNS_{IP}^{\delta[m]}$ problem.

It is also known from Lemma 5 that, the optimal solution structure among the possible $f_{\delta[m]}(\delta^*[m^*])$ decision mappings is at least the optimal solution structure of the $PNS_{IP}^{\delta[m]}$ problem.

From the definition of $f_{\delta[m]}$, it is known that, when a $(X, \mathcal{F}(op(X)))$ is added to decision mapping of the current branch, $\delta^*[m^*]$ where $X \in M$ and $op(X) \in \Delta(X)$, then in $f_{\delta[m]}(\delta^*[m^*] \cup (X, op(X)))$ is equal to either $f_{\delta[m]}(\delta^*[m^*])$ or $f_{\delta[m]}(\delta^*[m^*]) \cup (X, \mathcal{F}(op(X)))$.

From Lemma 4 is known that for each step, the summed weight of $f_{\delta[m]}(\delta^*[m^*])$ is less than the summed weight of $\delta^*[m^*]$.

Summing up the previous claims, it implies that, the decision mappings that the $f_{\delta[m]}$ function returns with are possible decision mapping over the $PNS_{IP}^{\delta[m]}$ problem. It is also known that among the possible solution structure, there is an optimal solution structure according to the minimal summed cost. The ABB algorithm always managed to find it over the $PNS_{IP}^{\delta[m]}$ problem only if the ABB algorithm over the PNS problem can find the optimal solution, i.e. the optimal solution exists among the possible $f_{\delta[m]}(\delta^*[m^*])$. And it is also known, that solution structure always gives a lower bound for the optimal solution structure in the original PNS problem, i.e. it provides a lower bound for all possible solution structures in the original problem. □

Theorem 1. *The new lower bound is correct, so*

$$\begin{aligned} Lower\_Bound_{new}(PNS(P, R, O), O_I, O_E) :=& \\ & ABB(PNS_{IP}^{\delta[m]}(P', R', O'), Summed\_Weight) + \\ & LP_{Solver}(PNS(P, R, O), O_I, O_E), \end{aligned}$$

if P' is not empty, otherwise $LP_{Solver}(PNS(P, R, O), O_I, O_E)$.

Proof. Let $X^* \in \mathbb{R}^n_{\geq 0}$ and $Y^* \in \{0, 1\}^n$, where (X^*, Y^*) is the optimal solution of the $PNS(P, R, O)$ problem and n stands for the number of operating units when the $\delta[m]$ is the current decision mapping in the branch, i.e. $O_I = op(\delta[m])$ and $O_E = op(\overline{\delta}[m])$. Here Y^* stands for the selected operating units, and X^* marks the operational size in the solution. From Lemma 3 we have that $ABB(PNS_{IP}^{\delta[m]}(P', R', O'), Summed\_Weight) \leq fix\_cost(O)^T Y^*$. It is known that $LP_{Solver}(PNS(P, R, O), O_I, O_E) \leq op\_cost(O)^T X^*$. A valid lower bound for the proportional part of the cost is given by $LP_{Solver}(PNS(P, R, O), O_I, O_E)$ because (1) the excluded units' proportional costs are not included due to the constraints $x \leq yM$ and $y = 0$, (2) the included units' proportional costs are included because of the above-listed constraint with $y = 1$, (3) the free units' variables will be set to 0 during the optimization as all the coefficients of the x-s and y-s are nonnegative numbers in the objective function. Because of the statements above, $ABB(PNS_{IP}^{\delta[m]}(P', R', O')) + LP_{Solver}(PNS(P, R, O), O_I, O_E)$ is a valid lower bound for the optimal cost in the $PNS(P, R, O)$ problem with O_I included and O_E excluded sets. □

Corollary 1. *As the fix costs are always non-negative values, our new lower bound always gives a tighter bound than the relaxed version, i.e.*

$$\begin{aligned}&Lower\_Bound_{new}(PNS(P, R, O), O_I, O_E) \geq \\ &Lower\_Bound_{relaxed}(PNS(P, R, O), O_I, O_E)\end{aligned}$$

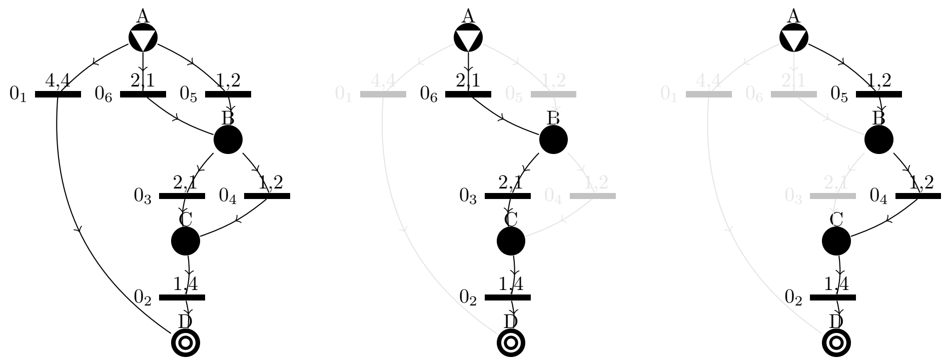
3.3 Illustration of new constraint and relaxed constraint

The following simple example illustrates the efficiency of our algorithm. In our example, we want to produce one product (D) from the raw material (A), using the operating units $O_1, O_2, \ldots, O_5$.

The final product (D) can be produced by either O_1 or O_2 or both operating units. If the machine O_1 chooses to produce the D final product, the O_1 unit consumes only the A raw material. The total production cost will be the sum of the fixed and proportional costs of the O_1 operating unit. The optimal solution is $y_1 = 1, x_1 = 1$, and the other variables are 0.

Consider another branch that chooses O_2. In this case, our previous production cost is $1 + 4$, because $y_2 = 1$ and $x_2 = 1$. In the original version, the operating cost of producing C is added to this cost. The optimal solution for $x_3 = 1$ and $x_6 = 1$ is 2 (see Figure 2b). For the lower bound, we obtain a value of 7, which is smaller than the previous value of 8. That is, this branch is explained by the previous constraints. However, structurally, the minimum cost of the operating units needed to produce C is 2, which is the minimum in the $y_4 = 1$ and $y_5 = 1$ case (see Figure 2c). Then the installation cost of $1 + 1$ is added to $5 + 2$. So, in total, the lower bound is 9, which is already worse than 8. With this new lower bound, the ABB algorithm is not explained this case.

The third branch includes both the O_1 and O_2 operating units. In this case, the considered inputs of the included units, A and C will be the new materials to be produced. The recursively called ABB method with modified fix costs for the C material, as it has been calculated in the second branch, returns 2. The optimal relaxed operational configuration for the current whole branch is $x_1 = 1$. The sum with the fixed cost of the included units will be 11 so it is fathomed.



(a) Original P-Graph (b) Optimal proportional cost (c) Optimal fix cost

Figure 2: A simple P-Graph in which the fixed and proportional costs are given above the operating units.

4 Results

To test our new lower bound calculating algorithm running time, we have to generate some test cases. There are many aspects to examine the performance of our new development. It is obvious that the CPU time could be taken into consideration but in this case teh solution of the LP relaxation problem should be added to the computational time. In a further development, it is not fixed whether we use an efficient LP solver method or use LP at all. The relationship between the produced and consumed materials does not have to be linear. It could be nonlinear as well. Because of these reasons, we preferably examine the total number of LP_solver executions and not the actual CPU time.

4.1 Result for P-Graphs without circle cases

Firstly, test cases have to be generated according to some predefined metrics to compare the number of LP_solvers calls in the case of ABB called with the $Lower\_Bound_{relaxed}$ and $Lower\_Bound_{new}$. The metrics in the first case will be the height and the width of the P-Graph. The whole structure of the graph will be a ($height \times width$) matrix of the operating units. Before the first level of operating units, there is a level of width the number of raw materials. Between the operating unit levels, there are also material levels which consist only of the width number of intermediate materials, and after the last level of operating units, there is also the width number of products.

In each n^{th} operating unit level ($n \in \{1, \ldots, weight\}$) operating unit consumes at least one material from the n^{th} material level. In the n^{th} material level ($n \in \{1, \ldots, weight\}$) all the materials consumed by at least one operating unit from the n^{th} operating unit level.

The same is true for the produced materials by the operating units in the n^{th} level: each n^{th} operating unit produces at least one material from the $(n+1)^{th}$ material level and each material from the $(n+1)^{th}$ level is produced at least by one operating unit from the n^{th} level. Alongside these connections listed above all the following layers' nodes (the n^{th} operating unit layer with the n^{th} and $(n+1)^{th}$ material layers) are connected with p probability.

In Figure 4's first plot this kind of P-Graph is illustrated. It is obvious that by running the ABB algorithm over these classes of P-Graphs the MSG algorithm won't exclude any operating unit from the original graph. It is also evident that the graph doesn't contain any operating unit circle because it doesn't hold edges that point to previous layers' elements.

The experiment of running the ABB algorithm and calculating the average number of LP solver calls according to the lower bound methods was completed, and the results are summarized and plotted in Figure 3.

The results are grouped by height. If we examine the branching method from the ABB algorithm, then it is straightforward that the algorithm has to examine all the possible operating unit combinations which can produce all the $x \in P$ material. The cutting can't be done on the first level of the tree, all the sub-branches have to

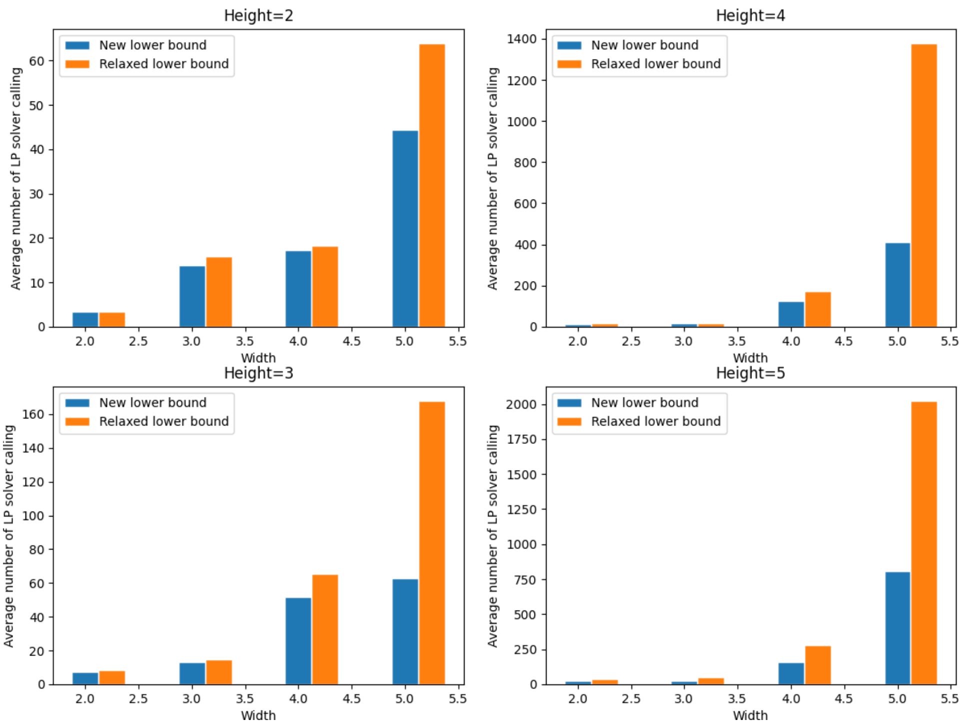
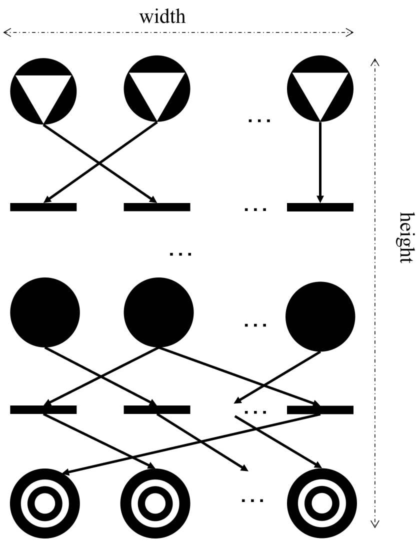
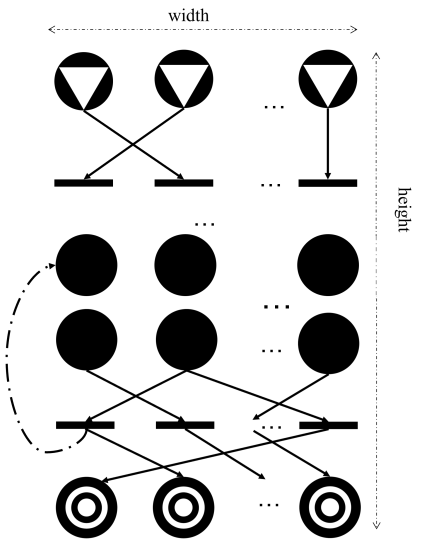


Figure 3: Comparing the ABB algorithm with different lower bounds by the average number of executed LP-solver calls over 30 generated P-Graphs without circles.



(a) A general example for the P-Graph without circles with the parameter setting $p = 0.5$



(b) A general example for the P-Graph with circles with the parameter setting $p = 0.5$ and $p_{circ} = 0.1$

Figure 4: Examples for the randomly generated P-Graphs

be calculated for the cutting, which is in this case one of the first chosen product production with all the possible cases. The possibility of an operating unit in the n^{th} level producing the $x \in P$ material is constant p beside one operating unit which surely can produce it. It implies that $\mathbb{E}(|\Delta(x)|) = p * width + 1$. It means that beside the same heights, the number of the branch grows as $\Omega(2^{width})$ with any lower bound.

As the $Lower\_Bound_{new}$ gives a tighter lower bound than the relaxed type of lower bound with the same height and width, $Lower\_Bound_{new}$ always outperforms the $Lower\_Bound_{relaxed}$. It implies that in all cases the number of LP_{solver} function calling in the ABB algorithm is always smaller with the $Lower\_Bound_{new}$ lower bound, than with the $Lower\_Bound_{relaxed}$. The reason for it is that the calculation of the structural optimization of the free operating units in our new lower bound does not call the LP_{solver} method, and the solver is called exactly once in the lower bound method. It is also true for the relaxed version of the lower bound.

If the submethod gives a tighter lower bound for the actual optimal value then the method would be called fewer times. It also means that the LP_{solver} is called fewer times.

4.2 Results for the P-Graphs with circle cases

Considering the previous generated test cases, the P-Graph with circle cases hasn't been examined yet. The cases with circles are when the previous P-Graph includes some directed edges that point to the previous levels. For example, there is an operating unit in the n^{th} level which produces at least one material in the $m^{th}(m \leq n)$ level with the probability p_{circ}. It is easy to prove that a dependency chain can be formulated when an infinite number of materials should be produced. An example of the generation logic is shown in Figure 4's second plot. These chains evolved when there is no raw materials being part of the chain, just products, and intermediate materials. As a consequence, the chain will be excluded from the graph when the MSG algorithm is running over the $PNS(P, R, O)$ problem. The executions' results with the parameter settings are $p_{circ} = 0.1$ and $p = 0.5$. This means that a portion of ≈ 0.1 is erased from the P-Graph while the MSG algorithm has been running over. The randomly generated summed average LP solver calling results grouped by the height and width is demonstrated in Figure 5. Similar to the previous non-circle running comparison, the average number of calls grows exponentially by increasing the width of the graph. The growth is distorted by the fact that some operating units are erased randomly because of the circle cases.

4.3 Result for a specific P-Graph

To test the new lower bound performance on a specific graph, the graph depicted in Figure 6 is used. The concrete P-Graph is also the maximal structure, as the MSG algorithm running over it doesn't exclude any operating unit. The graph has 65 materials, and 35 operating units. The ABB algorithm has been run over the graph both with our new lower bound, and the relaxed lower bound. The comparison of the performances is listed in Table 1. To make an extended comparison, the fix and operating costs are multiplied with constant values. The constant values are different for the cost types. The first column contains these values. The second column contains the summed LP_Solver calling number during the running of the ABB algorithm. If all the constants are 0 then it equals to the case when there is no costs. In this case the lower bound sub method can't cut on any branch at all, because the branch-and-bound algorithms, also like the ABB, always take advantage of the costs when a cutting is performed. As it is listed in the table's first row, all the possible structures are examined. If the fixed cost is multiplied by 0 and the operational cost by 1, then the ABB algorithm could prune just with the optimal summed operating cost which is calculated by the LP_Solver in both lower bounds.

The two constants which are used to multiply the costs control that algorithm ABB$(PNS^{\delta[m]}_{IP}(P', R', O'), Summed_Weight)$ or $LP_{Solver}(PNS(P, R, O), O_I, O_E)$ is more dominant than the fix cost or op cost multiplier was increased in the new lower bound. As the table shows, the basic case is when the costs are not multiplied then with the relaxed bound the number of LP calls is 48, and with the new lower

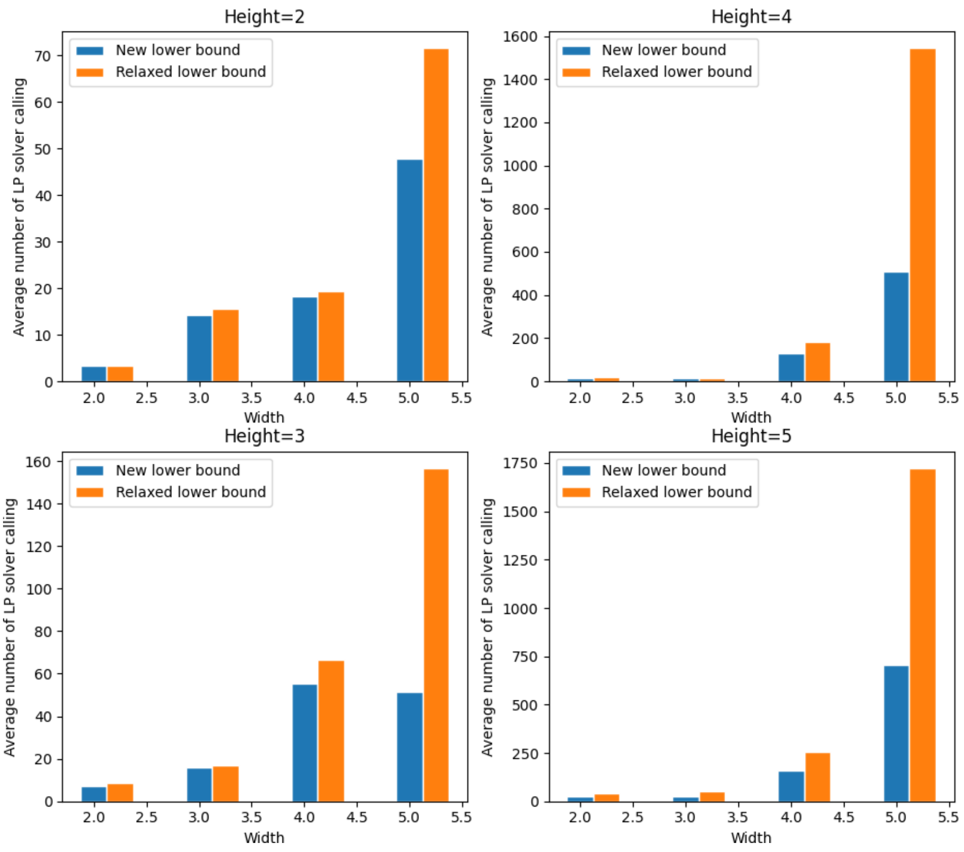


Figure 5: Comparing the ABB algorithm with different lower bounds by the average executed LP-solver numbers over 30 generated P-Graphs with circles, where $p_{circ} = 0.1$ and $p = 0.5$.

bound the number of calls is 31. That means the number of calls is reduced by $\approx 35\%$ with our new lower bound. With increasing the operational multiplier constant compared to the fix cost multiplier as the fix cost remains the same, but the operational cost is increased by the multiplication of $10^1, 10^2, \dots, 10^5$ the number of LP calls remains the same. It has happened because the P-Graph costs aren't modified to the level when the differences between the structural cost and operational cost are so large that the algorithm could cut the branches according to the operational cost. If the operational cost would be significantly larger compared to the structural cost, the LP calls with the relaxed and the new lower bound converge to the same value, 1023.

If the fix cost multiplier constant is 1, and the operational cost multiplier is 0 (plotted in the last row in the table) then the free operating units' summed cost is 0. As a consequence of that, the branch can be cut down by the summed cost of

Table 1: Comparison of the running of ABB algorithm with different lower bound functions: in the first columns are the results of running with relaxed LP model bound, in the second columns are the results with our new lower bound.

Ratio	LP		Time		Comparing the LP ratio
(fix cost/ op. cost)	Relaxed bound	New lower bound	Relaxed bound	New lower bound	(New lower /Relaxed)
0/0	2099	2099	13478	24532	1
0/1	1023	1023	7271	19857	1
$1/10^{1...5}$	48	31	361.6	464.8	0.65
1/1	48	31	352	460	0.65
$10^1/1$	48	31	378	475	0.65
$10^2/1$	44	27	420	453	0.61
$10^3/1$	91	60	837	948	0.66
$10^4/1$	95	70	763	1037	0.74
$10^5/1$	83	57	669	835	0.69
1/0	80	57	577	740	0.71

already included operating units' total cost added to the remained graph optimal structural cost. Then due to this, the relaxed bound functions as $Summed\_Weight$ (Algorithm 4) bound and the newly introduced lower bound uses also the remained part's optimal structural cost for the cut branch. This case is interesting when the structure was to be optimized, and the network wasn't used, or only rarely [1, 2].

In the previous rows in the table the cases are examined when the fix cost is multiplied by $10, 100, \dots 10^5$. In these cases the results converge to the case, when the fix cost has remained the same, and the operating cost was erased.

From the examined cases in the table, the $10^2/1$ case gives the optimal number of LP solver calls with both kinds of lower bounds, and also the LP ratio is the smallest in this case. The 1/0 case doesn't give the optimal ratio. It is caused by the acceleration of the algorithm with the natural extension.

The next column in the table lists the running time of the algorithm in CPU seconds multiplied by 1000. Examining this column shows, that the algorithm with a relaxed lower bound gives almost always a better result than the running with the new lower bound. It is the case since it is not worth to call the LP solver fewer times as the calculation of free operating units' optimal cost is more time-consuming. It isn't the case if the calculation of the derived model is more time-consuming. An example is when the P-Graph isn't linear.

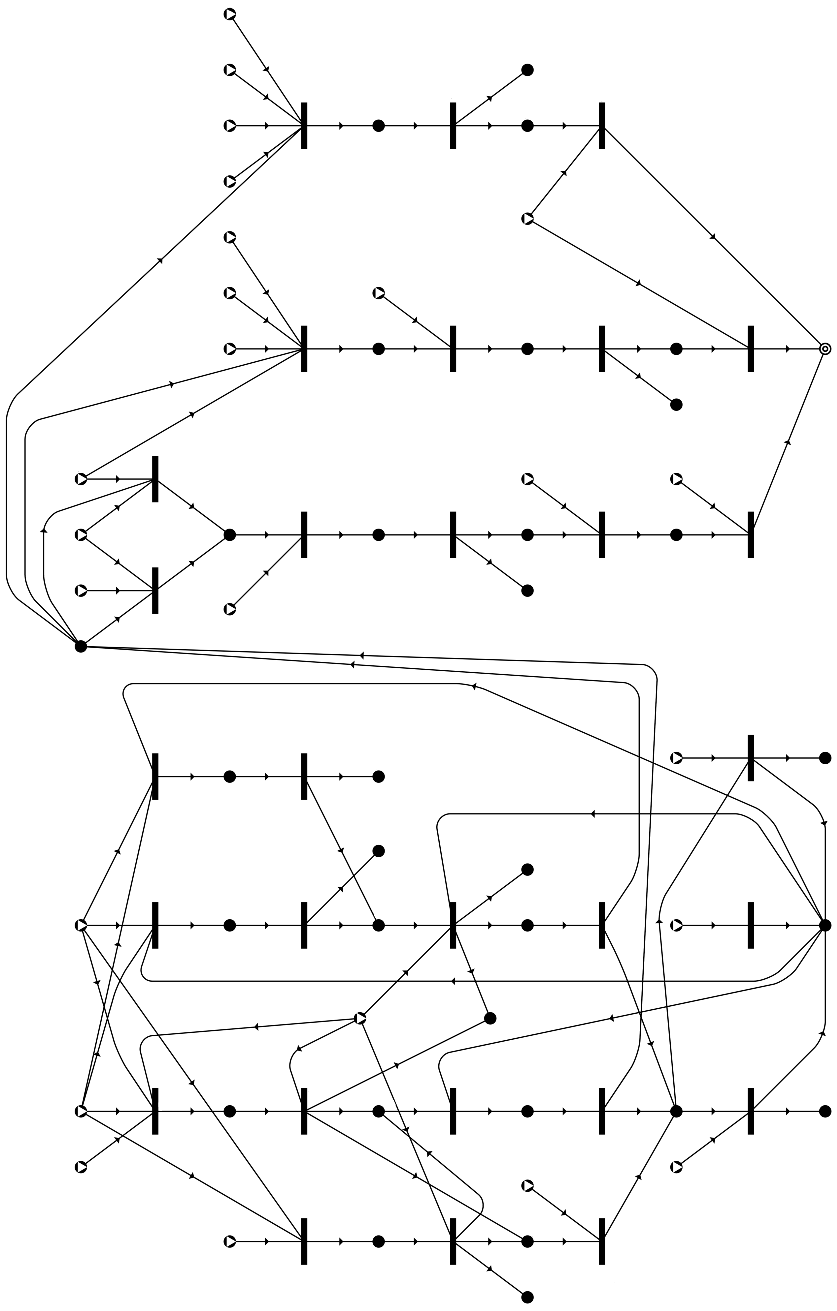


Figure 6: A specific graph with 65 materials and 35 operating units. The graph has been generated with the MSG algorithm.

5 Conclusion

We have created a new constraint calculation procedure for the ABB algorithm that gives a better one than the previously applied constraints. The computational cost of the new constraint is higher than the previous one, but we show that the overall number of linear programming problem solver calls will be reduced. It is advantageous to run the derived problem solver fewer times if in the operating unit model, the connection between the consumed and produced materials is nonlinear, or the variables are stochastic.

References

[1] Aviso, K., Lee, J.-Y., Dulatre, J., Madria, V., Okusa, J., and Tan, R. A p-graph model for multi-period optimization of sustainable energy systems. *Journal of Cleaner Production*, 161:1338–1351, 2017. DOI: 10.1016/j.jclepro.2017.06.044.

[2] Aviso, K., Yu, K., Lee, J.-Y., and Tan, R. P-graph optimization of energy crisis response in Leontief systems with partial substitution. *Cleaner Engineering and Technology*, 9:100510, 2022. DOI: 10.1016/j.clet.2022.100510.

[3] Berthold, T., Farmer, J., Heinz, S., and Perregaard, M. Parallelization of the FICO Xpress-Optimizer. *Optimization Methods and Software*, 33(3):518–529, 2018. DOI: 10.1080/10556788.2017.1333612.

[4] CPLEX, I. I. V12. 1: User's manual for CPLEX. *International Business Machines Corporation*, 46(53):157, 2009. URL: https://public.dhe.ibm.com/software/websphere/ilog/docs/optimization/cplex/ps_usrmancplex.pdf.

[5] Friedler, F., Orosz, ., and Losada, J. *P-graphs for Process Systems Engineering*. Springer Cham, London, 2022. URL: https://link.springer.com/book/10.1007/978-3-030-92216-0.

[6] Friedler, F., Tarjan, K., Huang, Y., and Fan, L. Combinatorial algorithms for process synthesis. *Computers and Chemical Engineering*, 16:S313–S320, 1992. DOI: 10.1016/S0098-1354(09)80037-9.

[7] Friedler, F., Tarjan, K., Huang, Y., and Fan, L. Graph-theoretic approach to process synthesis: Polynomial algorithm for maximal structure generation. *Computers and Chemical Engineering*, 17(9):929–942, 1993. DOI: 10.1016/0098-1354(93)80074-W.

[8] Friedler, F., Tarján, K., Huang, Y., and Fan, L. Graph-theoretic approach to process synthesis: axioms and theorems. *Chemical Engineering Science*, 47(8):1973–1988, 1992. DOI: 10.1016/0009-2509(92)80315-4.

[9] Friedler, F., Varga, J., and Fan, L. Decision-mapping: A tool for consistent and complete decisions in process synthesis. *Chemical Engineering Science*, 50(11):1755–1768, 1995. DOI: 10.1016/0009-2509(95)00034-3.

[10] Friedler, F., Varga, J., Fehér, E., and Fan, L. Combinatorially accelerated Branch-and-Bound method for solving the MIP model of Process Network Synthesis. In Floudas, C. and Pardalos, P., editors, *State of the Art in Global Optimization: Computational Methods and Applications*, pages 609–626, Boston, MA, 1996. Springer US. DOI: 10.1007/978-1-4613-3437-8_35.

[11] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2022. URL: https://www.gurobi.com/documentation/current/refman/index.html.

[12] Nishida, N., Stephanopoulos, G., and Westerberg, A. A review of process synthesis. *AIChE Journal*, 27(3):321–351, 1981. DOI: 10.1002/aic.690270302.

[13] Siirola, J. Industrial applications of chemical process synthesis. *Advances in Chemical Engineering*, 23:1–62, 1996. DOI: 10.1016/S0065-2377(08)60201-X.

Standardized Telemedicine Software Development Kit with Hybrid Cloud Support*

Zoltán Richárd Jánki[ab] and Vilmos Bilicki[ac]

Abstract

In modern Web development, it is expected that systems operating in the same area can be easily integrated. For common integration points, it is recommended to use a standardized data model and a common interface during the development as this will facilitate further integrations. The use of the cloud infrastructure is increasingly popular in telemedicine, but taking into account the goals, the productivity of the development, the availability of the system and the various regulations, choosing the right solution is not trivial. Inclouded platform consists of numerous currently active telemedical microservices that are working with a common software development kit. This tool provides a standardized data model for document-oriented database systems, has support for public and private clouds by using the classic Data Access Object (DAO) analogy and contains a lot of convenient functions as well. Furthermore, it is found that our solutions can significantly increase development productivity and is confirmed by measurements taken which involved software developers.

Keywords: telemedicine, hybrid cloud, software development kit, productivity

1 Introduction

Telemedicine applications are getting more and more attention. Google Trends shows that after the appearance of coronavirus disease the number of available

*This research was supported by the EU-funded Hungarian grant GINOP-2.2.1-15-2017-00073, project no. TKP2021-NVA-09 has been implemented with the support provided by the Ministry of Innovation and Technology of Hungary from the National Research, Development and Innovation Fund, financed under the TKP2021-NVA funding scheme; project no. II-NKFIH-1528-1/2021 has been implemented with the support provided by the Ministry of Innovation and Technology of Hungary from the National Research, Development and Innovation Fund, financed under the II-NKFIH-1528-1 funding scheme. This study was also supported by the Ministry of Innovation and Technology NRDI Office within the framework of the Artificial Intelligence National Laboratory Program (RRF-2.3.1-21-2022-00004).

[a]Department of Software Engineering, University of Szeged, Hungary
[b]E-mail: jankiz@inf.u-szeged.hu, ORCID: 0000-0003-1829-5663
[c]E-mail: bilickiv@inf.u-szeged.hu, ORCID: 0000-0002-7793-2661

DOI: 10.14232/actacyb.299460

telehealth applications increased, not only in the mobile stores but on the World Wide Web (WWW), too. In telemedicine, electronic healthcare records (EHRs) are considered as sensitive data, so it is really important to take into account the regulations.

Since 2018, access to patient healthcare records are governed by the General Data Protection Regulation (GDPR) [13] and individuals have a right to access their own healthcare data, but in limited circumstances they can get information about other people, too. However, the data handlers have to ensure that sensitive data cannot be transferred outside the country. Sometimes it is required to host everything within the boundaries of an organization. In some cases, it is allowed to use the public cloud but data must be stored in an encrypted form. These are limitations which can affect not only the economic solutions but can affect the development processes as well. Today, there is no publicly available software development kit (SDK) that conforms to such requirements and supports both public and private cloud solutions.

The degree of maturity of a research field can be measured by the number of available standards and protocols that belong to the given field. In telemedicine, many standards are adapted from other fields, and only a handful of them are telemedicine specific [4]. The most well-known standard is called Fast Healthcare Interoperability Resources (FHIR)[1] that provides a data model for real telemedicine use-cases. Thanks to its practical design and loose structure, it easily fits into any application.

FHIR defines only the resources that can be present in a medical environment and lists the attributes that can be used to describe these resources. FHIR itself is not appropriate to standardize every component of a telemedicine application but it has recommendations on what and how to use, so it gives vent to other standards, too.

Since FHIR is not a security protocol, it does not provide ready-to-use solutions for authorization, synchronization and digital signatures, but has recommendations. Using OAuth[2] for user authentication is suitable for web-centric applications, but the SMART-On-FHIR[3] specification can be used as an alternative solution.

Clinical terms are also managed by different standards. The most widely spread one is SNOMED CT published by SNOMED International[4]. Diagnoses, clinical documents, vital signs and other data that can be measured are systemized in the so-called LOINC[5] standard. Both standards categorize different terms with coding systems that help group data of the same type. In addition to the recommended standards, FHIR provides the opportunity to extend the predefined data model of resources with custom elements that are not originally part of the standard. However, all extensions must be well defined so that the data stored in an extension

[1]HL7. Fhir overview. https://www.hl7.org/fhir/overview.html

[2]OAuth. Oauth 2.0 - oauth. https://oauth.net/2/

[3]SMART on FHIR. Smart on fhir: Introduction - smile cdr documentation. https://smilecdr.com/docs/smart/smart_on_fhir_introduction.html

[4]International, SNOMED. Snomed - home — snomed international. https://www.snomed.org/

[5]Institute, Regenstrief. Home - loinc. https://loinc.org/

field can be easily identified. Data from special systems such as Enterprise Resource Planning (ERP) and Customer Relationship Management (CRM) systems do not have an appropriate place within FHIR, even though these concepts are critical for performing logistics and health management tasks. FHIR provides a number of extension profiles but it can be time consuming and difficult to find the one that describes the data. If there is no suitable extension profile in the standard, developers have to create a new one.

As the number of EHRs is rapidly increasing, the benefits of cloud solutions can be utilized. However, as we mentioned earlier, there can be project specific regulations that determine which type of cloud infrastructure can be used. To begin with using a public cloud, both the development and the maintenance can be convenient and the performance can be significantly high, but if there are restrictions on the location of the servers in the project, shared infrastructure cannot be an option. Private cloud is also a well-scalable solution, but the additional tasks associated with configuration and maintenance should also be considered. A hybrid solution in which a combination of private and public cloud services are available, can perform well because most of the load is on the public cloud and the critical tasks related to data and security can be handled by the private cloud.

In telemedicine, various data types can be present. An EHR can be a simple JavaScript Object Notation (JSON) object or it can be a high resolution image. In some cases, a Relational Database Management System (RDBMS) is sufficient but if the performance is critical then a non-relational database can be a better choice. Hence, it is recommended to introduce the so-called polyglot persistence concept so that we can use different data storage techniques and vary them to meet the needs. However, it is not trivial how different storages communicate with each other.

In telemedicine, offline capability can be critical. Web applications often go to offline status for seconds but occasionally they cannot come back online for hours. Besides offline status, the performance can be increased by adding caches to the data path. In our recent study, we elaborated a taxonomy for telemedicine applications and provided an easily tunable solution that helps in the design of telemedicine systems taking into account their offline capability. There should be various caching techniques that are available and finely tunable depending on the use-case.

The rest of the paper is organized as follows. Section 2 provides an overview of well-known telemedical platforms and the current status of the areas covered by our SDK. In Section 3, statistical information about FHIR and its prevalence in software development is presented. Section 4 offers a comprehensive list of challenges that developers may encounter. Section 5 introduces our SDK as an all-in-one solution to these challenges. The measurement results that demonstrate the effectiveness of our SDK in aiding telemedicine system development are discussed in Section 6. Finally, in Sections 7 and 8, we summarize the key aspects of this paper and highlight potential opportunities for further development.

2 State of the art

In this section, we present the current status of telemedicine platforms and their applied solutions focusing on the main issues that we may face.

2.1 Telemedicine platforms

- Intelehealth is an open-source telemedicine platform[6] that consists of 4 main components: a web application, an Android mobile application, a middleware layer and a medical record server. The mobile app uses complex data-gathering flowcharts and combines them to form an assisted history-taking system, called Ayu, and the web app allows remote doctors to review uploaded data and make referrals, offer advice or prescribe. OpenMRS[7] is the EHR server in this architecture that stores patient data, but it has not gained popularity in Europe and does not offer as many options as FHIR. OpenMRS can receive and convert data from FHIR-compatible systems, but it is not the main domain.
- AdvancedMD[8] is a more than 20 years old telemedicine platform. Unfortunately, it is not free and open-source but it is known that they store their data in Amazon Web Services (AWS). Due to storing data in a public cloud, this platform does not meet European regulations.
- OpenEMR[9] is a lightweight project that presents a video conferencing and chat platform for consultation purposes. Technologically its basics belong to Danphe Health that provides telemedicine softwares embedded in cloud services.
- The telemedicine network called Unimed Floripa was established in Brazil that was first introduced by R. S. Maia, et. al. [15]. Their platform serves radiological demands by maintaining a Web portal, a medical imaging toolset, teleconference tools and multiple Digital Imaging and COmmunications in Medicine (DICOM) and non-DICOM servers. It is obvious that they use standardized solutions for storing and managing data. Health Level Seven (HL7) and its standards (e.g. FHIR) play important roles in their telemedicine network. However, the capabilities of the platform are limited and too use-case specific.
- Inclouded[10] is an open-source telemedicine and smart-city platform that conforms to a number of standards. Development is based on ISO 13485, domain models follow HL7's FHIR and TMForum standards. It provides both public and private cloud solutions that are using our SDKs as connectors and

[6] Intelehealth — Confluence. URL: https://intelehealthwiki.atlassian.net/wiki/spaces/INTELEHEAL/overview

[7] URL: https://openmrs.org/

[8] Cloud-based patient relationship management software —– AdvancedMD. URL: https://www.advancedmd.com/medical-office-software/cloud/

[9] URL: https://www.open-emr.org/

[10] URL: http://inclouded.hu/

helping functions with high-level FHIR support. Inclouded SDK has already performed well in many currently active telemedicine projects over the years. It is also proved that the SDK is not only a convenient tool for designing and managing telemedicine systems that can be easily integrated, but also significantly speeds up the development processes. This article provides a detailed introduction to the key components of Inclouded SDK.

Table 1 presents a comparative analysis of the five platforms that have been introduced. It is seen that none of the platforms are GDPR-compliant except for Inclouded. Private and public cloud supports vary based on their focus, but hybrid cloud support is not common. Notably, a majority of the platforms provide support for FHIR.

Table 1: Comparison of telemedicine platforms

Platform	**Open-source**	**Private cloud support**	**Public cloud support**	**FHIR support**	**GDPR compliance**
Intelehealth	✓			✓	
AdvancedMD			✓		
OpenEMR	✓		✓		
Unimed Floripa		✓		✓	
Inclouded	✓	✓	✓	✓	✓

2.2 FHIR

Before diving deep into the details of the SDK, it is important to see how popular the standard used is.

Firstly, we have analyzed the available open-source telemedicine projects. We used GitHub as a datasource because it has the biggest public repository store on the Web. It consists of millions of public projects and provides an API for filtering and gathering information regarding repositories. We wrote a GitHub Crawler for filtering metadata of the repositories and finding specific repositories based on their content. We were focusing on telemedicine projects and we were searching for projects with various keywords. By using the term "health", we found more than 100,000 repositories, but unfortunately many of them were useless due to lack of commits or completely different interests. However, we still found quite a good number of projects that really deal with telemedicine. After analyzing these repositories, we can assume that FHIR is the most popular standard used in telemedicine projects.

FHIR was first released in 2013, but its first presentation was held in 2012 [5]. First three releases were just called Draft Standard for Trial Use (DSTU), but after DSTU3, it reached a maturity level that could have been considered as a final version. Since it is an open standard, open-source projects can describe its popularity well. For our statistics, we used GitHub as the source. Figure 1 presents that from 2014 the number of projects using FHIR started to grow exponentially and this growth is still continuing. Unfortunately, there are hundreds of repositories that contain a single readme file or just text files referring to the standard. Based on this experience, we have filtered out the GitHub repositories that contain evaluable projects using FHIR. In Figure 2, it is shown that currently R4 is the most supported version, but the number of new projects are increasing as the maturity of the standard levels up.

Today, FHIR is the most popular healthcare standard but except for some interface libraries there is no available toolkit that implements a Representational State Transfer (REST) endpoint in a typed form with FHIR support.

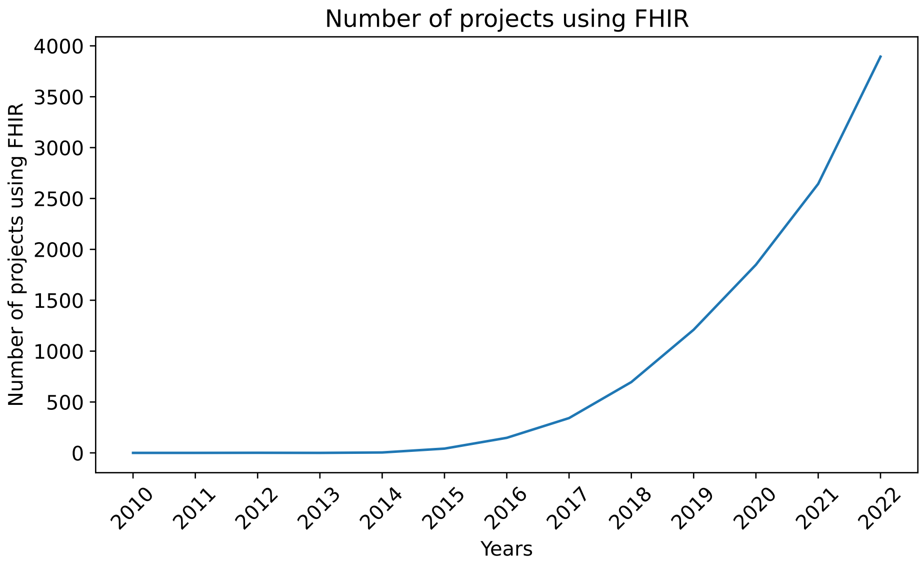


Figure 1: Popularity of FHIR in GitHub

2.3 Public and private clouds

Centering on accessing the information anytime and anywhere, encourages moving the healthcare information towards the cloud. Although the cloud offers several benefits, it also poses threats to health data in terms of privacy and security [1]. Here, we go through the pros and cons that cloud solutions provide taking into account the telemedicine use-cases.

Paper [20] presents a detailed comparison of public and private cloud solutions, highlighting the benefits of the former such as accessibility, scalability, availability, and reliability. Public cloud typically delivers the so-called pay-as-you-go model in

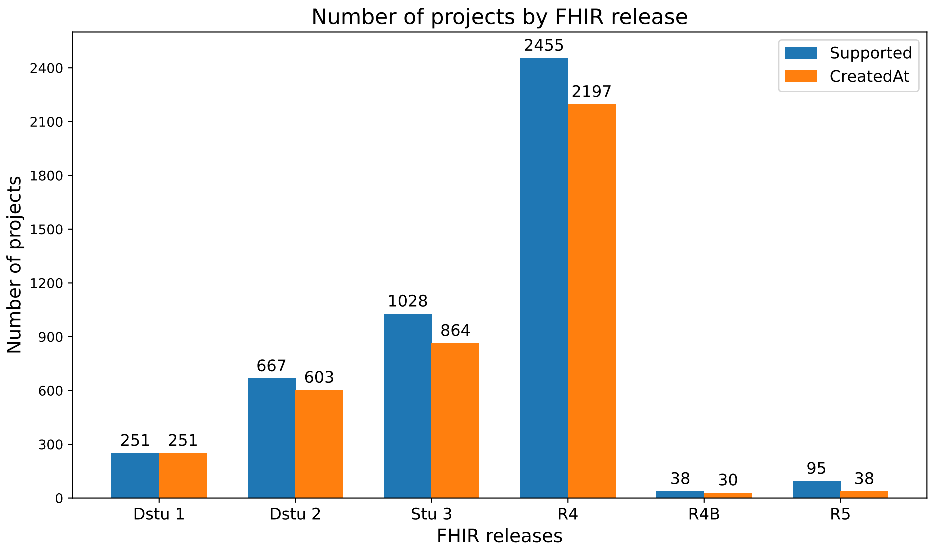


Figure 2: Popularity of FHIR in GitHub considering the version

which you pay after using the resources. Since public cloud services are ready-to-use, developers do not have to take care about the time consuming configurations and the infrastructure below the services. Public cloud services are ready-to-use, which means that developers do not need to worry about configurations and infrastructure. However, public cloud providers such as Amazon, Google, and Microsoft offer a lower level of security, so it is not recommended to store sensitive data in them.

Private cloud is usually dedicated to a single organization and it operates within the network of the organization or company. Thus it is required to buy, build and manage the cloud infrastructure that needs experts and comes with a higher cost. In terms of productivity, the development processes are longer in case of private clouds, but designed services are more use-case specific and they can operate more efficiently in given circumstances. Moreover, the level of security is higher in private clouds.

Chen et al. [12] introduced a solution that uses a hybrid cloud approach. Healthcare records stored in public clouds are encrypted with Symmetric Key Algorithm (SKE) and can be decrypted only through the private content key. Their solution is a redundant hybrid cloud service that is offered at the cost and scale benefits of public clouds, while also offering the security and control of private clouds. Nowadays, this approach is more and more common in telemedicine projects but there is no available library that supports hybrid cloud approach with interchangeable cloud background.

2.4 Serverless development

Startups and smart tech companies have begun to take advantage of serverless scalability, reliability, and performance for rapid growth - and now serverless development is more popular than ever. Moreover, it is also found that the developer productivity can be increased as well. Here, we consider the word "serverless" as a service in which the infrastructure is maintained by the service provider. There are various serverless services, such as databases, storages, runtime environments that can be used to run computational tasks and host Web applications. Vadym Kazulkin [11] collected the main effects that serverless development can have on productivity. The main advantages are the followings:

- no infrastructure maintenance
- auto-scaling and built-in fault tolerance
- less engineers required
- less code written
- bigger focus on business value and innovation
- shorter time-to-market procedures

ip.labs has been following the concepts of serverless development for years but in a hybrid form. They still have monolithic Java applications, but two teams are developing completely serverless. Before they went serverless, they had a central administration team and prioritized the tasks that lead to increased waiting time. Serverless development needs no low level administration at networking level, there are no servers and no operating systems (OS) that developers have to take care about and there is less interaction between developer and administration teams. Thus, the development processes become faster, developers write less code and use managed services.

2.5 Productivity

In Information Communication Technology (ICT) productivity is measured in different fields. Here, we focus on developers' productivity, show which metrics are used to measure it and how new techniques and technologies affect productivity in software development.

Shake [19] collected the most important metrics that can describe the workflows and measure the productivity of development. Firstly, the code quality is a big hit to productivity. There are several metrics to measure code quality and reduce quality defects. Shake is a good tool to create automatic reports that help deal with bugs efficiently.

Code coverage is a valuable metric for monitoring the development team's testing activities. It is easy to measure since it has a concrete formula and it gives feedback about how much of the source code is not covered by test cases. These metrics do not measure productivity explicitly, they have only effects on it.

Cycle time can tell a team a lot about the productivity of developers. It measures the time taken for a task to move from one phase to another. Cycle time is broken down into more stages and it gives information about which stage is problematic and where bottlenecks are. Most of the issue and project tracking softwares provide data about cycle time, so it is a more and more common metric used to measure productivity.

Lead time is very similar to cycle time, but it is a more comprehensive metric that measures productivity from task creation until delivery. So it is a metric for not individuals but for the development team.

Deployment frequency is measured within a specific period of time and it is one of the most valuable metrics in terms of productivity. There are 4 software delivery performance levels: low, medium, high and elite. This level is specified by the deployment frequency. Software delivery performance is low if deployment frequency is fewer than once per six months and it is elite if there are multiple deploys per day. Flickr reported an average of 10 deployments a day in 2009, while Etsy had 11,000 deployments in 2011 [18]. At Facebook each developer released an average of 3.5 software updates into production per week. These numbers prove that Continuous Integration (CI) and Continuous Delivery (CD) can significantly improve productivity.

The DevOps Research and Assessment (DORA) group published their platform and introduced 4 key metrics for measuring DevOps performance. These metrics are deployment frequency, lead time for changes, change failure rate and mean time to recover. Using this platform, Fin500 was able to increase the number of releases to production from 40 to over 800 [6].

The above mentioned tools can show only an approximation for the productivity, based on the committed source codes and the logged work hours, so they do not perform in a way that produces exact results. We found that using our SDK can significantly improve developers' productivity in telemedicine applications and it is confirmed with metrics too that are measured based on the developers' activity in the preferred integrated development environment (IDE).

3 Actuality of using FHIR

To prove that using FHIR is a trend, we analyzed the publicly available telemedicine projects in the world and measured the presence of the standard. We used GitHub as the main source of our research and using its API, we have collected the repositories having telemedicine purposes and repositories using FHIR.

We started using general expressions (e.g. health) to find the most repositories of our interest, but we realized that only a very small part of the results would be useful for us. Due to the limitations of GitHub API, we had to choose the terms carefully and analyze the repositories focused on the results.

On GitHub, there are thousands of empty or almost empty repositories that can be easily found using an expression that is present in its name or description or in a readme file. The valuable repositories may contain source codes too that can be analyzed and later compared to each other by using code metrics. GitHub has several categorizations for the repositories, and most of them are supported by the API, too.

Based on our experiences, we searched for FHIR-related repositories on GitHub using search terms listed in Table 2 and filtered by 5 main programming languages: Java, C#, Python, JavaScript, and TypeScript. Table 3 shows the most popular packages for these languages that have references to FHIR. HAPI[11] is an open-source FHIR server written in Java, Firely SDK[12] is the official .NET SDK written in C#, FHIR Resources[13] is a Python package for creating and validating FHIR objects, and `ts-fhir-types` [2] is a TypeScript package for FHIR resources.

Table 2: GitHub crawling terms and the number of found repositories

Term	**Number of repositories**
telemedicine	780
e-health	167
ehealth	703
telehealth	350
teledermatology	5
teleradiology	20
teleeducation	2
healthcare	1795

Table 3: Number of repositories retrieved from a product-based GitHub crawling

Programming language	**Product**	**Number of repositories**
Java	HAPI server	780
C#	Firely .NET SDK	167
Python	fhir.resources	703
JavaScript/TypeScript	@ahryman40k/ts-fhir-types	62

Further analysis was made by filtering the results using the selected 5 programming languages. Firstly, we have inspected how popular FHIR is on GitHub. If we check the repository names, descriptions and readme files, we can find 5,931 repositories. Comparing this number to the number of retrieved results if we make a code-based search, it is very few. The number of GitHub code-based search results

[11]Hapi FHIR — The Open Source FHIR API for Java. URL: https://hapifhir.io/hapi-fhir/
[12]Firely .NET SDK —– The Official .NET SDK for Hl7 FHIR. URL: https://fire.ly/products/firely-net-sdk/
[13]URL: https://pypi.org/project/fhir.resources/

shows how many files were found on GitHub containing the term. Usually, it is much more than the number of repositories, so these results need further analysis. We made a code search for the FHIR term, but due to the high number of results (2,011,239 occurrences found), we limited them using the 5 language filters. When TypeScript is selected as the main language, GitHub returns thousands of files that contain FHIR, but after forming a set from the repositories, only 277 repositories were left in the end. C# shows similar behavior because code search found about 50,000 files on GitHub but there are only 257 repositories. Python and JavaScript seem to be more popular in project development using FHIR. There are more than 500 public repositories that have Python and JavaScript as the main languages. Java is the most popular with 1,679 different repositories. Figure 3 shows the ratio of used main programming languages in FHIR repositories. There were 3,392 repositories found using FHIR and the 5 selected languages. It is also found that using techniques, frameworks and components like search terms can produce more focused and more valuable results in such data mining.

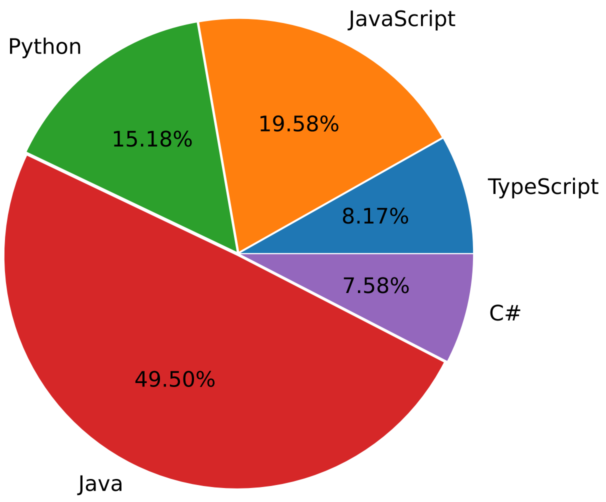


Figure 3: Presence of FHIR on GitHub by programming language categories

It is observed that FHIR is popular in application development but it is not clear how many telemedicine projects use FHIR. Telemedicine repositories were selected by using code search with the 8 search terms listed in Table 2 and filtered by the 5 chosen languages. Figure 4 depicts that the expression ”healthcare” is present in most of the repositories. We found projects from specific areas of telemedicine too but they do not exceed 1% of the total together.

We have also collected the repositories that use the most common products listed in Table 3. After seeing that most of the telemedicine-related projects use Java, it

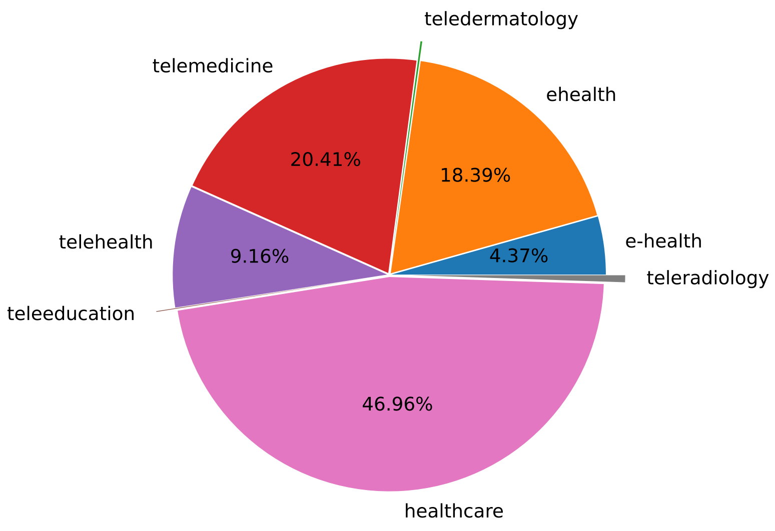


Figure 4: Presence of telemedicine on GitHub based on search terms

seemed to be obvious that the search term "HAPI" will return the most records. After HAPI, the official Python package is the most commonly used resource. Firely SDK showed a surprisingly significant popularity with 19.46%. Compared to the number of available public repositories, only 62 (4.68% of) repositories use the ts-fhir-types package written in TypeScript. The total number of repositories using the 4 packages was 1,326. Figure 5 shows the ratios.

To see how many telemedicine-related projects use helper packages, libraries and official solutions, we have inspected the intersections of these sets. We found that 35% of TypeScript projects using FHIR rely on the ts-fhir-types package. However, this typed version is not so popular in JavaScript-based projects. Unsurprisingly, HAPI and fhir.resources are used in more than 50% of projects in which Java or Python are the main programming languages and FHIR is present, too. The results are presented in Figure 6.

Finally, we found it is important to see how FHIR affects the lifetime of projects. We measured the freshness of the projects by applying a threshold for the last commit date. We filtered out repositories with a last commit date older than 3 months and an average size below 68,852 KB. This approach helped us identify repositories with a high maturity level. Naturally, in order not to distort the statistics, we applied the language filters, too. It came out that in mature projects FHIR is really popular, almost 50% of the projects use it as a standard (Figure 7). Here, we found only 326 repositories. Based on this experience, we evaluated the intersection of the FHIR set and all other telemedicine sets. It is seen that in the retrieved telemedicine projects with high levels of maturity, FHIR is commonly

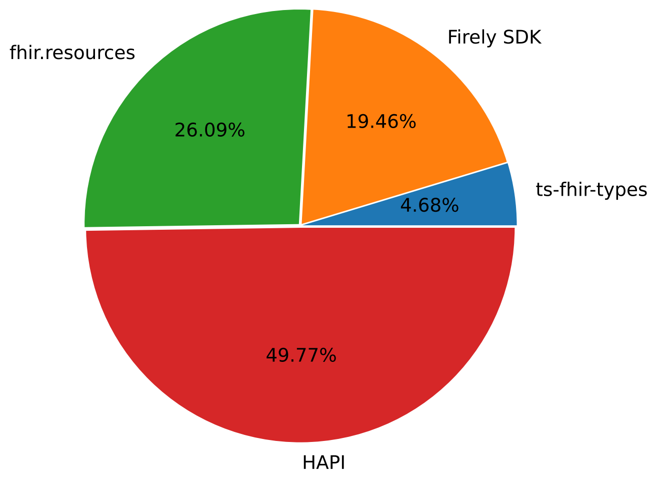


Figure 5: Presence of most common FHIR products on GitHub

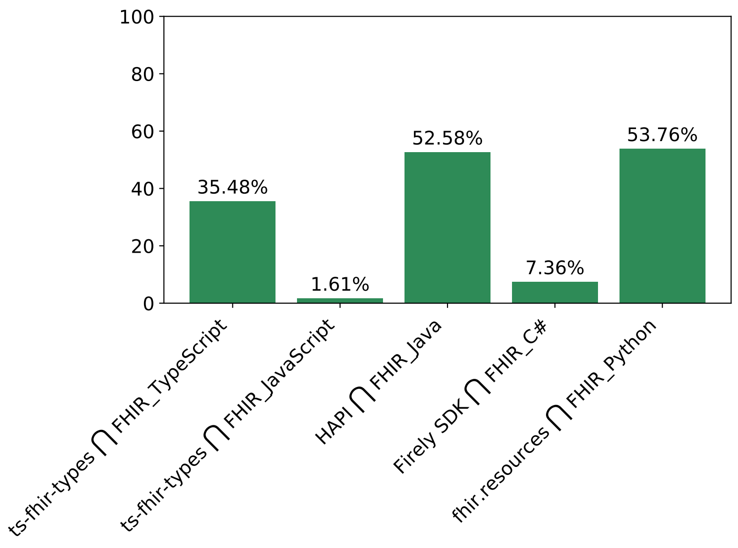


Figure 6: Intersection of most common products and projects using FHIR with main programming languages

applied (Figure 8), but some projects have unique data models while others show similarities to the FHIR model.

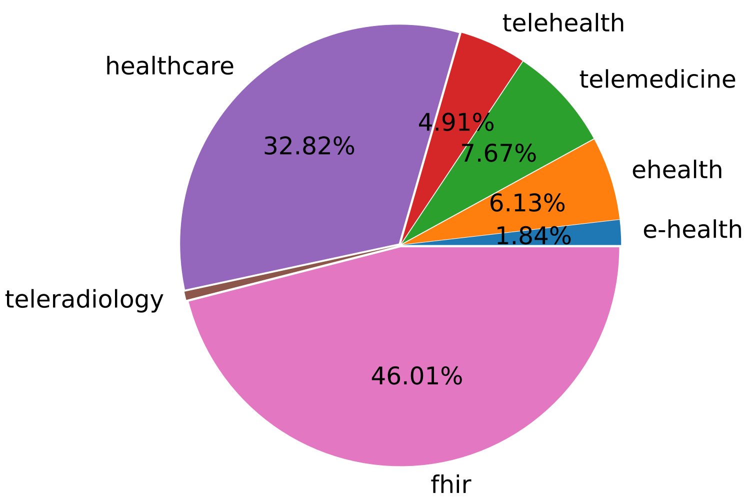


Figure 7: Presence of telemedicine repositories filtered by threshold values

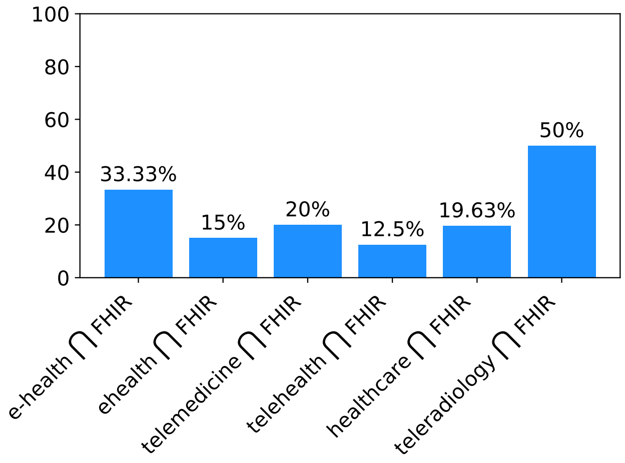


Figure 8: Ratio of telemedicine projects using FHIR filtered by threshold values

4 Challenges in telemedicine application development

Here, we collected the challenges that we were facing during the development of telemedicine applications and the Inclouded platform. To ease the development processes, we have elaborated and implemented an SDK that supports new technologies, provides extra features that help developers and solves complex problems as well. In this section, we list the challenges that our SDK provides solutions for. We will present our solutions in detail later.

- Inclouded supported FHIR since the foundation of the platform but only relational database systems were used prior. As public cloud solutions became more and more popular, we found that Not-only Structured Query Language (NoSQL) databases can perform better in many situations. FHIR offers a relational data model for handling healthcare resources, so it is a challenge to have a compatible NoSQL solution, too.
- Using a pre-created domain model, it is not trivial to find the proper entities and fields to store all necessary data. Standards are sometimes too generic even if they are practical. FHIR provides a lot of healthcare resources with a well-defined data model, but in many real telemedicine cases, it is hard to find the place to store the data. FHIR offers an extension mechanism for such cases but the extensions must contain a precise description of what they contain. Our SDK was extended with a Natural Language Toolkit-based (NLTK) recommendation system that helps to find the best matching extension for the data to be stored.
- FHIR provides a well-defined domain model but there is no recommendation on what technologies to use. Since a telemedicine application can contain not only metadata about healthcare records but binary files too, we decided to pursue the idea of polyglot persistence where we use different database systems, but each of them is used for what they are best at.
- In 2016, Google introduced Angular 2 framework that brought a big change after AngularJS [22]. As they recommended using TypeScript programming language, everyone felt the lack of a typed version of FHIR client library. `fhir.js`[14] offers an official solution for using FHIR in JavaScript but it was not extended to handle interfaces and classes.
- In many telemedicine projects it is limited where data can be stored and what path data can be transferred through. These limitations can be stated by owners, organizations or a project. To meet these requirements and run into less legal issues, a private cloud is recommended to establish. Inclouded SDK supports hybrid cloud solutions, so developer can choose to use public or private cloud to store the data.

[14]URL: https://github.com/FHIR/fhir.js/

5 Our solution

This paper introduces our telemedicine SDK called Inclouded SDK, its importance in telemedicine application development and all of its features. The basic concept of our SDK is to provide the capabilities of WebDAO for telemedicine application developments. Using WebDAO analogy, SDK offers Data Transfer Objects (DTOs) in form of classes to apply the entire DAO design pattern. Since it is shown that serverless development increases productivity, we decided to start telemedicine developments in a public cloud. Google Cloud Firebase platform and its services were used as a basis, so we built an SDK that can handle FHIR and can conform to the solutions of Firebase. Here, we used Google Cloud Firestore for storing metadata, Google Cloud Storage for storing binary files and the Google Authentication service for managing users. The SDK is publicly available and installable via Node Package Manager (NPM).

5.1 SDK structure

As it is shown in Figure 9, Inclouded SDK consists of resource-based application programming interfaces (APIs) that provide fully FHIR-compatible typed document classes, so-called DTOs and a list of queries that implements the necessary FHIR search parameters in form of independent functions. The basic Create, Read, Update, Delete (CRUD) operations are implemented in the FhirApi class and all the FHIR resources are inherited from this class. Besides the CRUD operations there is an additional id-based query function that is suitable for all resources. Hence, Inclouded SDK fully implements the DAO design pattern for FHIR and can be used as a part of the data layer of Clean Architecture [16]. To make Figure 9 more clear, we have only drawn a part of the whole library but the rest of it uses the same idea.

During the design of SDK, we took into account that FHIR has never had NoSQL support, however, Firebase provides only NoSQL database systems. FHIR was designed for RDBMS, but today NoSQL is gaining more and more space. FHIR defines search parameters that describe the necessary filters if one uses the standard. Using Google Cloud Firestore we were facing issues that made it hard to filter data that FHIR supports. Four main problems were:

1. finding a value of a field if it is in an object inside an array,
2. filtering by substrings of a field,
3. extending queries based on access-control rules,
4. and obtaining results that are gathered from multiple collections.

In Section 6, we show our algorithms for these problems and present a logical model for their specifications and verifications.

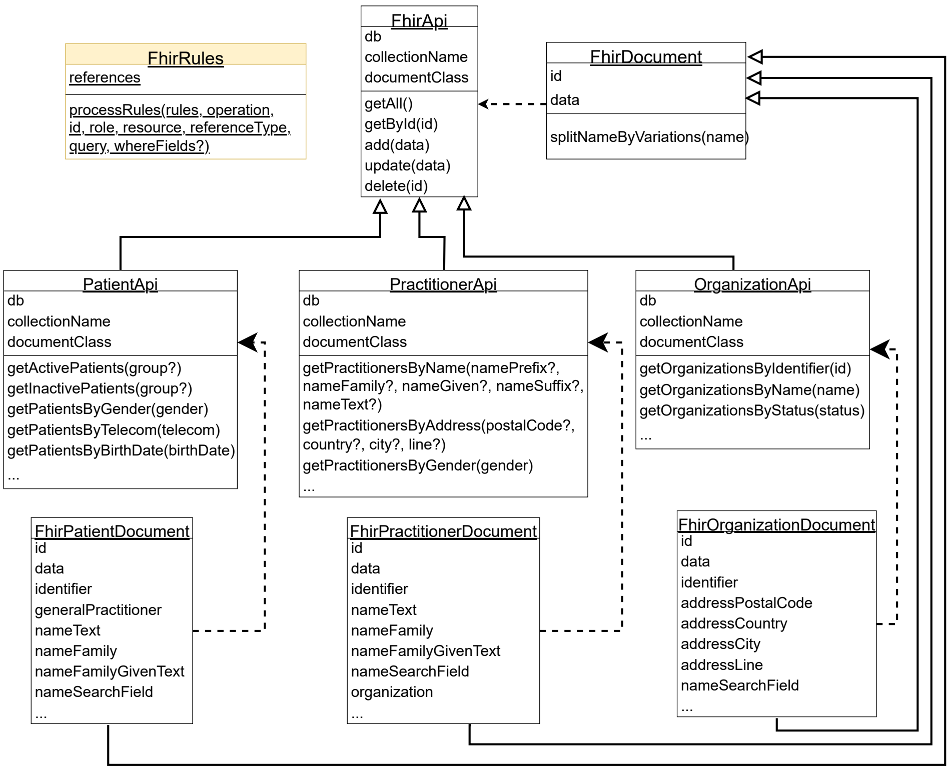


Figure 9: SDK structure

5.2 SDK architecture

Since FHIR requires endpoints that return back standard data, there is no restriction on the format in which the data is stored. So, the data is stored in a format compatible with NoSQL and Firestore concepts, but the data that is retrieved by the client is fully FHIR compliant. The above mentioned intelligent capabilities are implemented in the document classes. FhirDocument is the base class of all resource entities. Here, we have two attributes, one is id, the other is data. To solve the 4 main challenges, document classes use helper functions that convert data to a format that makes it searchable in any database system.

Figure 10 shows an architectural map about how SDK takes place in the data path. It is installed on the client side and processes the data sent by the client and forwards it to the cloud. If the client operation is an insertion, SDK waits for an FHIR-compatible object and creates an extended object that makes the original data NoSQL compatible. If the request is a query, then SDK waits for search parameters and adds the necessary filters to the query object. Since access control techniques are different in different database systems, SDK also has an optional rule processing function that checks the grants based on the database settings. Since the rule system of Firestore does not work as a filter, it is required to add extra filters to the query to retrieve the needed objects.

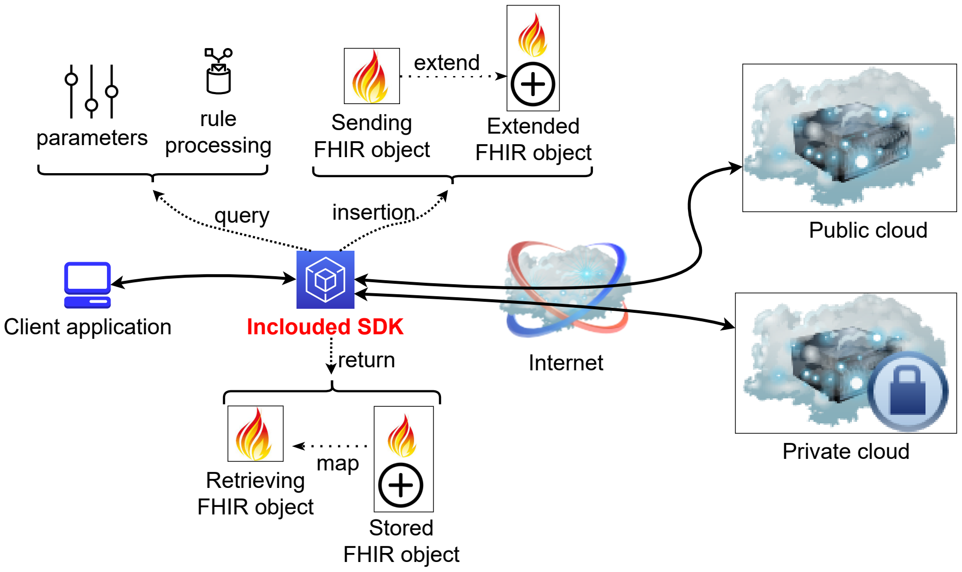


Figure 10: SDK architecture

6 Results

In this section, we will present the main components of our SDK, the formal descriptions of the algorithmic solutions, and the productivity results achieved.

6.1 Main components

Inclouded SDK consists of six core components that support developers in telemedicine application development. Here, we provide a detailed description about these solutions.

6.1.1 Outsourcing non-searchable fields

It is shown that in Google Cloud Firestore, if a field can be found in an object inside an array, it cannot be filtered. It is also problematic in other NoSQL systems if only a field value is known, not the whole object. SDK manages such cases by outsourcing these values into new so-called search-fields created at the top level. The standard form of the data is also kept, so REST endpoints conform to the FHIR standard.

6.1.2 Rule processor

Access control mechanism of Google Cloud Firestore is so simple that only requested collections or one document of a collection can be controlled by them. Rules can use the requester's sent data, requested data or a preset date to check if the resource can be given to the client. Since a rule controls the whole request, it is not an option to retrieve only those documents for which we have permission. To do so, requests must contain additional filters to start requests only for those documents that we have permission. If we construct our queries using this structure, we can execute queries in any NoSQL system. Thus, we elaborated a rule processor algorithm. Since it is known which FHIR resources can contain data referring to users, groups or permissions, we made a built-in resource descriptor object that contains the resources and their fields that may contain references to such entities. The rule processor waits for a query object, the currently active rule set and the logged in user's id, roles and groups. Optionally, the FHIR resource fields that may refer to permissions can be explicitly set up. By default, SDK uses the basic FHIR knowledge. After starting a request, – calling an SDK function, – SDK will extend the query based on the active rules, the user data, the requested resource and the FHIR resource references. So, developers do not have to take care about how data can be retrieved under given access control settings because SDK will resolve this issue and build up a perfect query. Naturally, the rule processor functionality can be turned off if there are no rules set up in the project. This solution can be applied in private cloud solutions, too, e.g. in RESTHeart[15] with MongoDB.

6.1.3 Collected system codes

FHIR stores quantitative values and values from enumerations in coding systems. SDK collects the most important and most common codes from the LOINC database that helps to describe stored data. This component not only provides the codes and their descriptions, but also creates the necessary FHIR format of the object that will contain the value. Developers can waste a lot of time by searching for these codes and finding the best description.

6.1.4 Extension finder

FHIR has an extension mechanism to give the opportunity to place data at a resource if there is no given field for the data. However, it is not easy to use because extensions must be well-defined using a FHIR profile that describes the stored data. We have extended our SDK with a public REST endpoint that can return a suggestion for data that developers could not find a field in the standard. The idea came after analyzing open-source projects using FHIR and applying extensions in the wrong way. Our approach uses NLTK to find the best matching extension that can define the data. In [10], we have shown our component in detail and demonstrated that our solution can achieve an 89% success rate.

[15]SoftInstigate. Restheart - ready to use backend for web and mobile apps. `https://restheart.org/`

6.1.5 Support for offline capability

In [8] and [9], we have presented a taxonomy to help design distributed telemedicine systems. Based on our elaborated taxonomy, we showed how consistency and data quality changes if the data path is complex and how systems can be configured to maintain consistency, availability and partition-tolerance at a high level. Inclouded SDK took a part in that model, and it has the option to easily tune a telemedicine system so that it remains offline capable without significant data staleness. It was measured that by allowing data up to 1 version older to be used in the cache, the system can still provide 83% consistency.

6.1.6 Hybrid cloud support

One of the biggest advantages of using Inclouded SDK is the hybrid cloud capability. It is a requirement in many projects to store data in a private cloud. However, public clouds can perform better. The development of Inclouded SDK started in 2016 and was introduced first in our former paper [7]. That study examined the concept of WebDAO and highlighted the importance of DAO in telemedicine application development. Based on our experiences in using Google Cloud, we have implemented a DAO layer that can substitute Google's document classes. It is a modern implementation of the classic DAO layer. Since Google Cloud Firestore is a document-oriented NoSQL database system, we found MongoDB as the closest open-source alternative to build a private cloud. Comparing their features, they operate very similarly, only technical differences can be found. Google is a bit more limited in filtering and setting up rules to access resources.

RESTHeart is an open-source cloud platform that provides REST API for MongoDB but cannot notify clients about data changes in real time via REST API. Since RESTHeart HyperText Transfer Protocol (HTTP) endpoints close the connection between the client and the server after responding, a WebSocket connection is needed to keep the connection alive. We have integrated a WebSocket module into Inclouded SDK that can establish WebSocket connection to a server. To follow up real time changes of a MongoDB collection, a Change Stream must be opened but Change Stream requires a MongoDB Replica Set that is not part of the basic RESTHeart platform. Hence, we have extended the original RESTHeart project with a MongoDB Replica Set and added a WebSocket server that can open Change Streams to collections. The WebSocket connection initiated by the SDK is used only for notifying the subscribed clients about changes. Every request goes through the original RESTHeart API.

Inclouded SDK can transform all type of queries that Google Cloud Firestore can handle including CRUD operations, filterings, ordering and paging. To have an interchangeable solution, we have created a MongoDBCollection, a MongoDBDocument and a MongoDBQuery classes that have the same functions as Firestore's TypeScript classes have, with the same input arguments and return values. Thus, a configured Firestore database object and a configured MongoDB DAO object can be interchangeably used. The database object is an input of FHIR API classes, so developers can decide if a resource should be stored in a private cloud or in a public cloud.

6.2 Formal definitions of algorithms

In this section, we present our algorithmic solutions for the four main issues that we were facing by using NoSQL database systems. For three of them, we provided an algorithmic solution. In the fourth case, if data can be queried only from multiple collections, developers can start multiple queries to get the needed data but it can produce a huge load on the client side. A better approach is to collect the data based on use-cases in result tables. SDK supports two types, these are used for creating result tables and charts. These are implemented in independent functions, so here we do not provide an algorithmic solution.

Since FHIR was designed for relational database systems, its applicability in public cloud databases is limited due to their predominant use of NoSQL database systems. Glenn Pepito collected the challenges and strategies of RDBMS to NoSQL migration in [17]. In a recent ScyllaDB guide [21], it is detailed what trade-offs must be taken when changing from relational to non-relational database system. Alachisoft[16] collected the key steps for adapting an existing schema to non-relational databases. All in all, it is commonly recommended that during the data model transformation process, denormalization and embedding of referenced objects into the reference location should be employed in most cases. However, in some instances, a hybrid model may be more effective. Similarly, our algorithmic approach also follows a hybrid principle in structuring data by preserving the standard part intact but outsourcing specific fields due to limitations in filtering capabilities.

Three algorithmic solutions were modeled in Temporary Logic of Actions (TLA) using its TLA+ language [14] to provide formal definitions as well. We have also verified the correctness of algorithms with Temporary Logic of Components (TLC) model checker. The algorithms and their formal definitions for the mentioned issues are as follows:

- If a field that must be searchable by the standard is hidden in an object that is in an array, the field is outsourced to an independent field at the top level (Algorithm 1).

- If a field containing a string must be filtered by substrings, SDK generates all the possible variations of the string that may occur and place them in an independent array field at the top level (Algorithm 2).

- If a rule exists for a given resource, it must be extended in the query to return back data (Algorithm 3).

[16] Alachisoft. Migration from sql to nosql databases. `https://www.alachisoft.com/resources/whitepapers/sql-to-nosql-migration.html#json-collections`

Algorithm 1 Formal definition of "checking if object is in an array" algorithm

```
$CheckObjInArray(x)$
1:  if $num\_op[x] < Len$(INPUT_OBJECT_FOR_OBJ_IN_ARRAY) then
2:      $num\_op' = [num\_op$ EXCEPT $![x] = num\_op[x] + 1]$
3:      $head' = Head$(check_arr)
4:      if $head'! =$ "elementary" and $Head(head')! =$"object" then
5:
6:          if $Len(head') > 0$ and $Head(Head(head')) =$"object" then
7:              $found\_arrays' =$ TRUE
8:              $array\_counter' = array\_counter + 1$
9:              UNCHANGED $substr\_filter\_vars$
10:         else
11:             UNCHANGED << $array\_counter$, $substr\_filter\_counter$,
                $found\_arrays$, $substr\_filter\_needed$, $check\_substr$ >>
12:         end if
13:     else
14:         UNCHANGED << $array\_counter$, $substr\_filter\_counter$, $found\_arrays$,
            $substr\_filter\_needed$, $check\_substr$ >>
15:     end if
16:     $check\_arr' = Tail$(check_arr)
17: else
18:     UNCHANGED $vars$
19: end if
```

Algorithm models were verified if they work properly. We have developed a Generator API to all the FHIR Resource APIs and these generators produced inputs that were passed to TLC Model Checker. We have evaluated the state graph of the algorithms but none of them produced error or deadlock, and returned the expected values, so we can conclude that algorithms are working as expected.

6.3 Productivity

In addition to many features that Inclouded SDK carries, we have also taken measurements on how it influences the development productivity. We have seen that there are code and project metrics that can be used to measure productivity. Here, we introduce another technique that measures specifically the coding and its progress. After testing various tools, we found an open-source, cross-platform time tracker for operating systems that can profile to output only the time used for development. Automatic, rule-based time tracker (ARBTT[17]) is a completely automatic time tracker that can collect statistics about how users spend their time. It runs in the background and monitors the computer and saves statistics about

[17]Breitner, Joachim and et al. arbtt: the automatic, rule-based time tracker. https://arbtt.nomeata.de/#what

Algorithm 2 Formal definition of ”substring filtering needed” algorithm

```
$CheckIfSubStrFilterNeeded(x)$
1: if $num\_op[x] < Len(\text{INPUT\_OBJECT\_FOR\_SUBSTR\_FILTER})$ then
2:    $num\_op' = [num\_op \text{ EXCEPT } ![x] = num\_op[x] + 1]$
3:    $head' = Head(\text{check\_substr})$
4:    if $head'! = \text{NEEDED}$ and $Head(head') > 0$ then
5:
6:       if $Head(head') = \text{NEEDED}$ then
7:          $substr\_filter\_needed' = \text{TRUE}$
8:          $substr\_filter\_counter' = substr\_filter\_counter + 1$
9:          UNCHANGED $obj\_in\_array\_vars$
10:      else
11:         UNCHANGED $<<$ $array\_counter$, $substr\_filter\_counter$,
            $found\_arrays$, $substr\_filter\_needed$, $check\_arr$ $>>$
12:      end if
13:   else
14:      UNCHANGED $obj\_in\_array\_vars$
15:   end if
16:   $check\_substr' = Tail(\text{check\_substr})$
17: else
18:   UNCHANGED $vars$
19: end if
```

Algorithm 3 Formal definition of ”rule processor needed” algorithm

```
$CheckRules(x)$
1: if $num\_op[x] < rules\_length$ then
2:    $num\_op' = [num\_op \text{ EXCEPT } ![x] = num\_op[x] + 1]$
3:    $check\_rules' = \text{RULES[RESOURCE]}$
4:    $head' = Head(\text{check\_rules'})$
5:    if $Len(head') > 0$ then
6:       $query' = query \text{ o } << head' >>$
7:    else
8:       FALSE
9:    end if
10:   UNCHANGED $obj\_in\_array\_vars$
11:   UNCHANGED $substr\_filter\_vars$
12:   UNCHANGED $rules\_length$
13: else
14:   UNCHANGED $vars$
15: end if
```

what windows were open, which one was the most active one in a given interval. The interval can be configured before starting the tracker.

In our study, we involved 10 university students who have not met FHIR yet, but completed a Web-development frameworks course where they learnt about Angular 2+ framework and Google Cloud Firebase platform and its services. With this study, our goal was to measure how productivity can be increased if developers use Inclouded SDK instead of start using the documentation of FHIR and the original Firestore SDK. The development phases were the followings:

1. Create an example TypeScript object for the selected FHIR resource.
2. Implement a list of Firestore queries without using Inclouded SDK (CRUD operations and other queries taking into account the FHIR search parameters).
3. Implement a list of Firestore queries using Inclouded SDK (same list of functions).

ARBTT was started with *-r 10* argument, so after every 10 seconds a log was created in the log file containing the opened windows and puts a flag to the most active one. Every developer used Visual Studio Code as IDE and installed the Angular 13 Snippets extension in advance. In the developer's ticket, it was specified what name they have to use by creating the file for the functions, so after analyzing the logs it was easy to determine how much time they spent editing a given file in the project. Everyone started to work on the same Angular 13 project that contained the necessary packages with fixed version numbers. The task list and the order of the tasks were identical, only the operating system was permitted to choose after the preferences. To verify the accuracy of the ARBTT capture logs, we analyzed the work logs added to the tickets as part of our quality control measurements.

In Figure 11, it can be seen that in all 4 scenarios the development time is reduced if developers used SDK. The development time in hours is presented as an average for each FHIR resource. The time tracker puts a flag to the window with the highest activity within the last 10 seconds, so it is not obvious how long the development time really took. We have also validated the time tracker results with logged work hours, and we found similar ratios between the two types of development form. After the final analysis of ARBTT logs, we found that the average activity time of a window in a 10 seconds long interval is 3.51 seconds. Since Patient was the first resource that developers had to work with, it needed the most time. Moreover, Patient has the most search parameters as well, so it needs the longest development time period. Comparing the development times, we can say that the development with SDK can be at least 2 times better than using only official documentation with no helper functions. Thus, we found that the Inclouded SDK can be an important key element not only in ours, but also in other telemedicine architectures. Moreover, these measurements validate the importance of DAO pattern from the point of view of productivity as well.

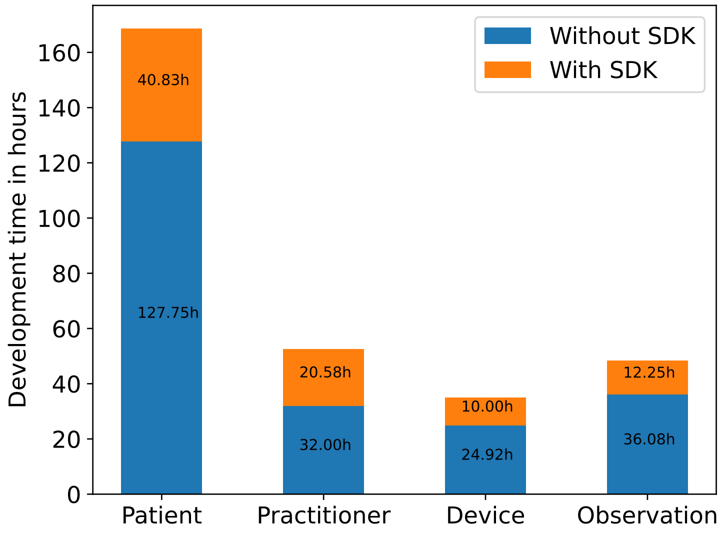


Figure 11: Average development time measured using SDK and without SDK

7 Future plans

Inclouded SDK is constantly updated and it is following the innovations of the dependencies. It is planned to integrate more public cloud solutions to support various systems. With these integrations, more novelties can be added to the package. After comparing our Google-based solution to Amazon Web Services and Azure, we found that all three platforms have common key points that make it possible for our SDK to be compatible with all public cloud platforms. Naturally, we would like to provide further support for private clouds as well. Regarding private cloud solutions, our solution primarily supports NoSQL databases. Since the filtering capabilities are more limited compared to a relational database, our solution can be clearly adapted to support relational databases as well. Nevertheless, FHIR defines a relational domain model, so such a solution can be implemented without the algorithmic solutions we proposed. Our solution and its significance came up with the idea to support other standards and may focus on other areas as well, not only on telemedicine. We have already started to develop a SDK with similar capabilities supporting telecommunication projects and using an acknowledged standard called TM Forum. In summary, there is a planned effort to assess the productivity of GitHub projects, supporting the importance of the WebDAO pattern as presented in [3].

8 Conclusions

This paper presented Incloouded SDK that can be a key component of any telemedicine system. It acts as a link between client-side and server-side in a way that the backend system can be easily changed. Both private and public clouds are supported, furthermore it contains several functionalities that help developers to work efficiently. Here, we support the most popular telemedicine standard, and we made various statistics to show its actuality. We presented the five main components of the SDK, in which the algorithmic solutions were formally defined and verified as well. The significance of Incloouded SDK in telemedicine application development is proved with different measurements. In terms of productivity, we have shown that the required development time can be at least two times less with SDK than without using SDK. Our results based on GitHub analysis and productivity showed that it is a promising solution. It is open-source and publicly available in NPM, and the increasing number of downloads denotes that there is a growing demand on packages and libraries like this.

References

[1] Abbas, A. and Khan, S. A review on the state-of-the-art privacy preserving approaches in e-health clouds. *IEEE Journal of Biomedical and Health Informatics*, 18(4):1431–1441, 2014. DOI: 10.1109/JBHI.2014.2300846.

[2] Baudin, G. Handle FHIR objects with TypeScript (and JavaScript). URL: https://medium.com/@ahryman40k/handle-fhir-objects-in-typescript-and-javascript-7110f5a0686f. [Accessed: 2022-09-29].

[3] Choudhary, S., Bogart, C., Rosé, C. P., and Herbsleb, J. D. Modeling coordination and productivity in open-source GitHub projects, 2018. URL: http://reports-archive.adm.cs.cmu.edu/anon/isr2018/CMU-ISR-18-101.pdf.

[4] Ferrer-Roca, O. *Standards in Telemedicine*. In *E-Health Systems Quality and Reliability: Models and Standards*, pages 220–243. Medical Information Science Reference, 2011. DOI: 10.4018/978-1-61692-843-8.ch017.

[5] FHIR version history and maturity. Technical report, The Office of the National Coordinator for Health Information Technology. URL: https://www.healthit.gov/sites/default/files/page/2021-04/FHIR%20Version%20History%20Fact%20Sheet.pdf.

[6] Forsgren, N., Tremblay, M., Vander Meer, D., and Humble, J. DORA platform: DevOps assessment and benchmarking. In *Proceedings of the International Conference on Design Science Research in Information System and Technology*, pages 436–440, 2017. DOI: 10.1007/978-3-319-59144-5_27.

[7] Jánki, Z. R. and Bilicki, V. Full-stack FHIR-based MBaaS with server- and client-side caching capable WebDAO. In *Proceedings of the 11th Conference*

of PhD Students in Computer Science, pages 179–183, 2018. URL: https://www.inf.u-szeged.hu/~cscs/cscs2018/pdf/cscs2018.pdf.

[8] Jánki, Z. R. and Bilicki, V. Crosslayer cache for Telemedicine. In *Proceedings of the 12th Conference of PhD Students in Computer Science*, pages 159–163, 2020. URL: https://www.inf.u-szeged.hu/~cscs/cscs2020/proceedings.php.

[9] Jánki, Z. R. and Bilicki, V. Taxonomy for trade-off problem in distributed Telemedicine systems. *Acta Cybernetica*, 25(2):285–306, 2021. DOI: 10.14232/actacyb.290352.

[10] Jánki, Z. R. and Bilicki, V. Domain specific semantic data model integration. In *Proceedings of the 13th Conference of PhD Students in Computer Science*, pages 197–201, 2022. URL: https://www.inf.u-szeged.hu/~cscs/cscs2022/pdf/cscs2022.pdf.

[11] Kazulkin, V. Measure and increase developer productivity with help of Severless. URL: https://www.slideshare.net/VadymKazulkin/measure-and-increase-developer-productivity-with-help-of-severless-by-kazulkin-and-bannes-sla-the-hague-2020-238115659. [Accessed: 2022-09-29].

[12] Kruse, C. S., Smith, B., Vanderlinden, H., and Nealand, A. Security techniques for the electronic health records. *Journal of Medical Systems*, 41(8):127–136, 2017. DOI: 10.1007/s10916-017-0778-4.

[13] Kulakiewicz, A., Parkin, E., and Powell, T. Patient health records: Access, sharing and confidentiality. Technical report, House of Commons Library, UK Parliament, 2022. URL: https://researchbriefings.files.parliament.uk/documents/SN07103/SN07103.pdf.

[14] Lamport, L., Matthews, J., Tuttle, M., and Yu, Y. Specifying and verifying systems with TLA+. In *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop*, pages 45–48, 2002. DOI: 10.1145/1133373.1133382.

[15] Maia, R., Von Wangenheim, A., and Nobre, L. A statewide telemedicine network for public health in Brazil. In *Proceedings of the IEEE Symposium on Computer-Based Medical Systems*, Volume 2006, pages 495–500, 2006. DOI: 10.1109/CBMS.2006.29.

[16] Martin, R. C. *Clean Architecture: A Craftsman's Guide to Software Structure and Design.* Prentice Hall Press, USA, 1st edition, 2017. ISBN: 0134494164.

[17] Pepito, G. RDBMS to NoSQL migration: Challenges and strategies, 2018. URL: https://www.researchgate.net/publication/341294540_RDBMS_to_NoSQL_Migration_Challenges_and_Strategies.

[18] Savor, T., Douglas, M., Gentili, M., Williams, L., Beck, K., and Stumm, M. Continuous deployment at Facebook and OANDA. In *Proceedings of the 38th International Conference on Software Engineering Companion*, pages 21–30. ACM, 2016. DOI: 10.1145/2889160.2889223.

[19] Shake Technologies, I. Metrics for measuring the productivity of your development team. URL: https://www.shakebugs.com/blog/measuring-developer-productivity/. [Accessed: 2022-09-29].

[20] Solanke, V., Kulkarni, G., Vishnu, M., and Kumbharkar, P. Private vs public cloud, 2013. URL: https://www.researchgate.net/publication/258253155_Private_Vs_Public_Cloud.

[21] SQL to NoSQL: Architecture differences and considerations for migration. Technical report, ScyllaDB, 2020. URL: https://www.scylladb.com/wp-content/uploads/wp-sql-to-nosql-architectur-differences-considerations-migration-1.pdf.

[22] Sultan, M. Angular and the trending frameworks of mobile and web-based platform technologies: A comparative analysis. In *Proceedings of the Future Technologies Conference*, pages 928–936, 2018. https://saiconference.com/Downloads/FTC2017/Proceedings/128_Paper_264-Angular_and_the_Trending_Frameworks_of_Mobile.pdf.

Quadratic Displacement Maps for Heightmap Rendering[*]

Mátyás Kiglics[ab], Gábor Valasek[ac], Csaba Bálint[ad], and Róbert Bán[ae]

Abstract

We present a higher-order representation of heightfields by constructing unbounding revolved parabolas about every texel of the height texture. These surfaces of revolution do not intersect the interior of the volume defined by the heightfield. We present a simple generation algorithm and show that these maps can be rendered by computing intersections between lines and parabolas in the plane. We compare its quality and performance with cone step mapping.

Keywords: computer graphics, parallax mapping, cone step mapping, quadric tracing

1 Introduction and Related Work

Heightmaps are two-dimensional textures that store elevation values at each sample position. These textures are mapped onto simplified base geometries, and the base shapes are transformed by displacing their points by the corresponding elevations along a direction. This direction is usually the unit normal of an interpolated tangent frame over the surface. Geometrically, the heightfield describes a variable radius offset of the coarse geometry, as shown in Figure 1.

There are two main approaches to the implementation of the above transformation [8]. A geometric one takes the vertices of the simplified base shape and translates them according to the heightmap. This mesh-based heightmap requires a sufficiently dense base geometry to accommodate the heightmap resolution. It also necessitates carefully crafted level-of-detail (LOD) variations of the base shape

[*]Supported by the ÚNKP-22 New National Excellence Program of the Ministry for Innovation and Technology from the source of the National Research, Development and Innovation Fund.

[a]Eötvös Loránd University, Budapest, Hungary
[b]E-mail: kiglics@caesar.elte.hu, ORCID: 0000-0003-4957-7495
[c]E-mail: valasek@inf.elte.hu, ORCID: 0000-0002-0007-8647
[d]E-mail: csabix@inf.elte.hu, ORCID: 0000-0002-5609-6449
[e]E-mail: rob.ban@inf.elte.hu, ORCID: 0000-0002-8266-7444

DOI: 10.14232/actacyb.299736

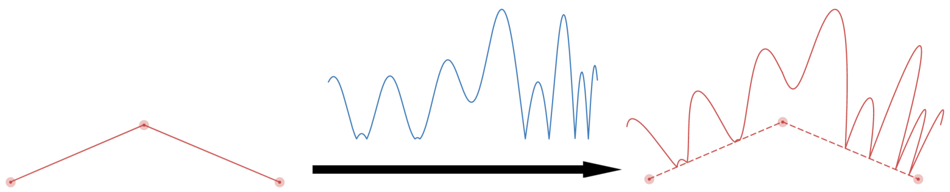


Figure 1: Heightfield (center, in blue) applied on top of a coarse geometry (left) yields a continuous, higher detail surface (right).

so that GPU performance is not wasted by rendering micro-triangles at a distance. Moreover, the transitions between the LODs should also be seamless.

The screen-space or per-fragment approach does not alter the raw geometry; instead, it casts a ray through each pixel of the base shape and alters the shading parameters according to a ray trace against the heightfield. Here, the displaced geometry is not stored explicitly; it only exists procedurally during ray traversal.

Initial screen-space techniques relied on the linear search along the ray to identify the ray-heightfield intersection [8]. Dummer proposed a conservative empty space skipping technique called cone step mapping to accelerate this process [4] with a different heightmap representation. Unbounding cones replaced the elevation values. These are the widest cones that are disjoint from the heightfield, have their apex on the heightfield surface, and their axes of symmetry are the tangent-space normals. Usually, these cones are stored with two scalars: the height of the apex and the tangent of the half-cone angle. Other numerical representations have been proposed as well that improve various numerical properties [5].

The current state-of-the-art in rendering performance is the relaxed cone map technique of Policarpo and Oliveira [7]. They extended Dummer's cone step mapping by replacing strictly conservative cones with relaxed cones that do not allow for more than one outside-inside transition between the ray and the heightmap. This approach guides the tracing inside the volume of the heightfield, so it requires a robust root refinement process, e.g., binary search, to find the surface point of the intersection.

Our paper proposes a generalization of cone maps by assigning a surface of revolution based on a parabola to each heightmap sample. This approach is a generalization of quadric maps [2]. The resulting surfaces are conservatively unbounding in the same sense as Dummer's, and we refer to them as unbounding revolved parabolas. Figure 2 illustrates several parabola cross-sections.

Section 2 introduces our proposed representation and specifies our cone-parabola hybrid model mathematically. We present a construction algorithm in Section 3 and a new tracing method for our data structure in Section 3.2. We propose generation and render time optimizations in Section 4. Section 5 contains our empirical results. We compared our proposed method with cone step mapping. Section 6 concludes this paper.

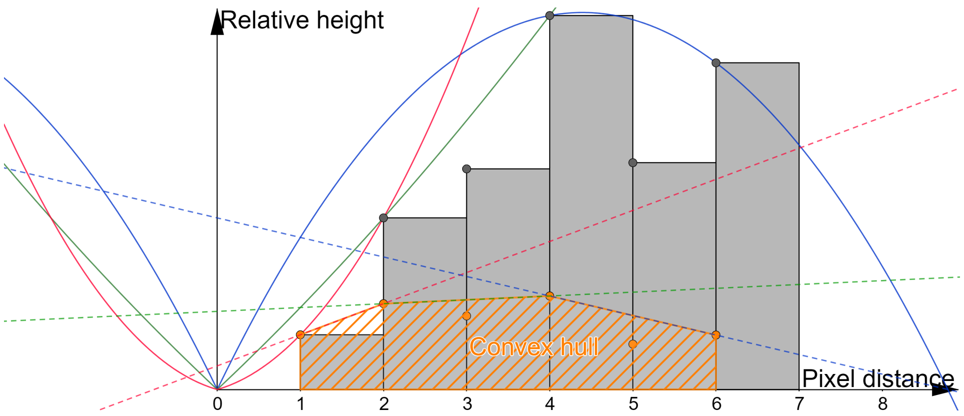


Figure 2: Distance–maximum height histogram for a given texel. The orange points are the height differences divided by the distance. The blue, red, and green lines form a convex boundary, and each line corresponds to a bounding parabola.

2 Quadratic displacement maps

The previously mentioned cone step mapping algorithms excel at rendering heightmaps in real-time; their most significant slowdown results from areas with high tangent slopes in the heightfield. Generally, the generated cones at these texels have a narrow opening angle, limiting the volume that can be skipped during ray-marching, therefore, increasing the number of iterations required. While more steps taken per ray does not necessarily mean worse performance, the cost of multiple texture read queries on the GPU cause a significant amount of idle processing time, making the algorithm less effective.

We aim to reduce the number of iterations by generalizing the cones to conservative parabolic surfaces, thereby increasing the unbounding volume size where possible. Quadratic surfaces have a non-constant tangent that allows them not to be defined only by the closely surrounding height values.

However, surfaces defined by implicit quadratic equations may not be sufficient, as they cannot generally provide the necessary improvement in step size extension; hence we complicate the surface to consist of two parts. First, we define a cone with similar characteristics to Dummer's cones, although limiting the height of the cone to a predefined value which we specify as a ratio between the height of the texel and the maximal heightmap value. Then, we connect a revolution of the parabola to the edge of the cone to create a continuous surface, allowing the parabola to be defined by an independent parameter from the cone.

Our representation of the described surface consists of three parameters denoted by $a, b, c \in \mathbb{R}$. The first two values represent the coordiantes of a point on the plane relative to the position of the texel, defining a line segment as one side of the cone. The two-dimensional description is sufficient here due to the radial symmetry of the surface. The third value, c, specifies a parabola starting from (a, b) defined by

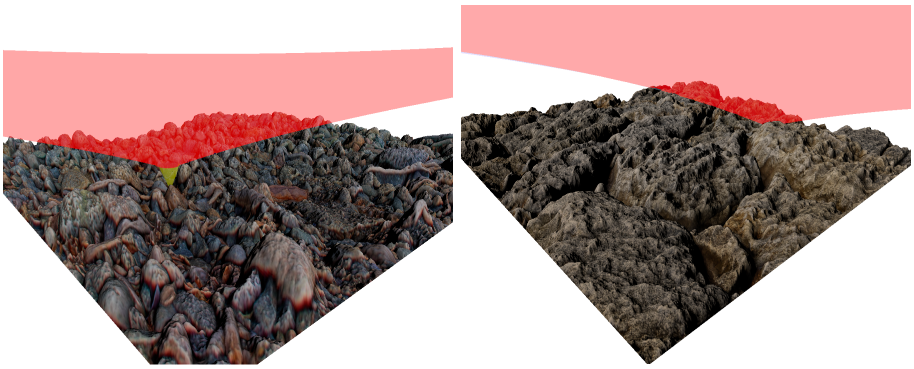


Figure 3: Generated quadratic surfaces with 0.5 (left) and 0.2 (right) height ratio for sample heightfields. The conic part on the left picture is colored yellow, and the quadratic parts are red on both sides.

the implicit equation

$$y = -(x-c)^2 + b + (c-a)^2 \qquad (a \geq 0, b \geq 0)\ .$$

Examples of such surfaces are shown in Figure 3. This representation requires storing three floating point values in addition to the height value for rendering. Thus, we equip the texture with four channels in our implementation.

3 Proposed algorithms

Our method traverses the empty space between higher heightmap elevations differently than similar techniques. We propose an algorithm for constructing unbounding parabolas and an efficient way to render heightmaps.

3.1 Generation of quadratic maps

Similarly to cone tracing, our ray tracing technique requires the revolved parabolas to be defined for every texel of the heightfield. For large textures, satisfying this condition demands time-efficient, parallelized construction of the unbounding surfaces with a small storage footprint.

First, for each (u, v) texel of the heightmap, we generate a radial function m_{uv} that returns the maximal relative height at a given distance from (u, v). An example of this function is shown in Figure 4. Let this function be defined for each $d \in \mathbb{N}$ integer pixel distance by

$$m_{uv}(d) = \max_{d \leq \left\|(x,y)-(u,v)\right\|_2 < d+1} \{h(x, y) - h(u, v)\} \qquad (x, y) \in \mathcal{D}_h.$$

After this transformation, we only need m_{uv} to find the optimal a, b parameters for a texel by advancing along the maximal height function, and in each step, we

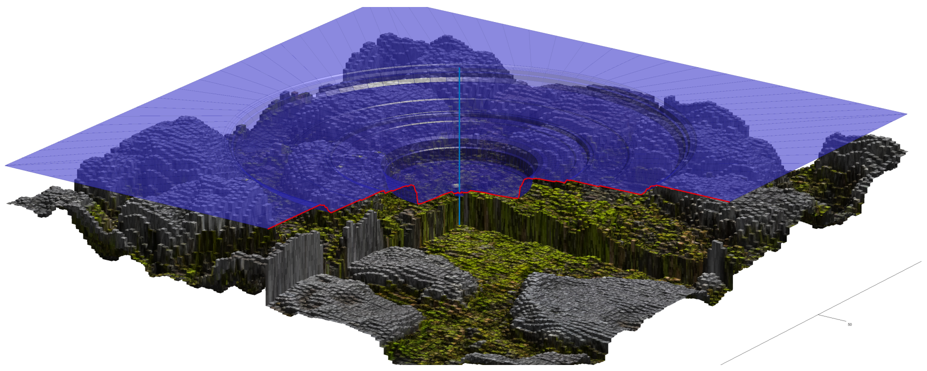


Figure 4: Generated radial function m_{uv} (red line) for a heightfield around the axis of the current texel (blue line). The value of the function at a given distance from (u, v) is the maximum height of the texels with the same distance.

choose $a \leftarrow i$ and $b \leftarrow \max\{i \cdot \frac{b}{a}, m_{uv}(i)\}$, where $i \in \mathcal{D}_{m_{uv}}$. We keep progressing until $i \cdot \frac{b}{a}$ exceeds the value of the height ratio parameter or i reaches the maximal distance. These steps can be pictured as searching for a cone which is a revolution of the line segment to (a, b) and does not intersect the heightfield but has the smallest slope possible. The result yields a finite-sized cone defined by the (a, b) point relative to the texel, as visualized in Figure 5.

With given values of a and b, we connect a revolved parabola to the top of the cone. Since these parabolas can be defined sufficiently in many ways, we choose a single equation to reduce the number of required parameters to one. This representation allows us to store all data for a parabola efficiently in four channels of a single texture. Let the implicit equation of the parabola be

$$y = -(x - c)^2 + b + (c - a)^2, \tag{1}$$

where $c \in \mathbb{R}^+$ remains to be determined. The global maximum of this curve is at c. Increasing this value guarantees that every point on the parabola between a and c will rise; satisfying our initial condition of avoiding intersection with the heightfield is trivial. This also means that the optimal value of c can be found by fitting a parabola to each point of m_{uv} and finding their global maximum. Thus, we solve the quadratic equations

$$m_{uv}(j) = -(j - c_j)^2 + b + (c_j - a)^2 \qquad j \in (a + 1, a + 2, \dots) \cap \mathcal{D}_{m_{uv}}$$

for c_j and let $c = \max c_j$.

After determining the a, b, and c parameters for each (u, v) texel, we include $h(u, v)$ and store the four floating point values in a texture to accelerate ray tracing. The complexity of the algorithm for a texture of size $N \times N$ is $\Theta(N^4)$ because the generation of m_{uv} requires checking all texels. However, since these calculations are independent, they can be parallelized.

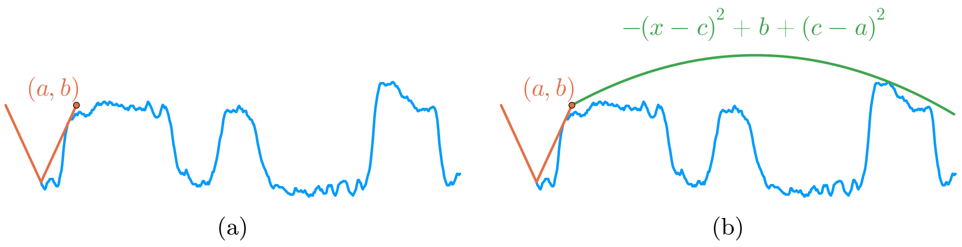


Figure 5: The two parts of generation of parabolas. First, we find values a and b, which define a cone (orange). Then, we calculate a parabola from (a, b) above the heightfield, defining an additional c parameter (green).

3.2 Rendering

During rendering, we cast rays from the geometry surface toward the heightfield, and our goal is to find their intersection using the quadratic maps described above. The raymarching algorithm is identical to the cone step mapping apart from the ray-geometry intersection calculations.

Let us define a ray starting from $\mathbf{p}_0$ and direction $\mathbf{v}$ by $\mathbf{p}(t) = \mathbf{p}_0 + t\mathbf{v}$, where $t \geq 0$. In each step of raymarching, we load the four values (a, b, c, and h) from the texel below $\mathbf{p}(t_i)$ (t_i being the current ray parameter), and find t where $\mathbf{p}(t)$ intersects the quadratic surface. We can simplify the problem into two dimensions due to the symmetry of the revolved geometries while obtaining the same results. Additionally, we only have to account for one parabola because the travel direction of the ray is known within the plane.

The entire curve is stitched together from two curve segments. The line segment and the parabolic curve share a single (a, b) point in the two-dimensional representation. When looking for an intersection with the ray, we separate these two parts and look at the line first.

Let us define $\mathbf{s}(\alpha) = \alpha(a, b)$, where an $\mathbf{s}(\alpha)$ point of the line is on the segment only if $\alpha \in [0, 1]$. Then, let the intersection of the two lines $\mathbf{p}(t)$ and $\mathbf{s}(\alpha)$ be (x, y), we have $\alpha = \frac{x}{a}$. If $\alpha \leq 1$, we have $t = \left\| \mathbf{p}(t_i) - (x, y) \right\|$.

If $\alpha > 1$, then substituting $\mathbf{p}_0 + t\mathbf{v}$ into Equation (1), we get

$$\mathbf{p}_{0_y} + t\mathbf{v}_y = -(\mathbf{p}_{0_x} + t\mathbf{v}_x - c)^2 + b + (c-a)^2 \tag{2}$$

a quadratic equation of t. If there is no real solution, then we terminate with no intersection; otherwise, let t_1, t_2 be two, not necessarily different, roots, thus $t = \min\{t_1, t_2\}$. The correctness of choosing the smaller value is because $\mathbf{v}_y < 0$ and $\mathbf{p}(t_i) > -c^2 + b + (c-a)^2$ holds by definition. For the latter inequality, it is important that it only holds if the previous $\alpha > 1$ is also true, although solving (2) would be unnecessary. Algorithm 1 formalizes this method, and Figure 6 shows two examples.

Algorithm 1 Tracing of quadratic map

```
Input: $\mathbf{p}_0$ ray origin, $\mathbf{v}$ ray direction, $a_{ij}, b_{ij}, c_{ij}, h_{ij}$ parabola parameters stored in a texture
Output: $t$ distance along the ray

$\mathbf{p} \leftarrow \mathbf{p}_0; s \leftarrow 0$
while $s < steps \wedge t > \epsilon \wedge \mathbf{p}$ above heightfield do
    $(i, j) \leftarrow$ texel coordinates of $\mathbf{p}$
    $(x, y) \leftarrow$ intersection point of the ray and the line to $(a_{ij}, b_{ij})$
    $\alpha \leftarrow \frac{x}{a}$
    if $\alpha \leq 1$ then                                  ▷ Intersected the line
        $t \leftarrow \|\mathbf{p} - (x, y)\|$
    else                                                     ▷ Check intersection with parabola
        $A \leftarrow \mathbf{v}_x^2$
        $B \leftarrow \mathbf{v}_y - 2 \cdot c_{ij} \cdot \mathbf{v}_x$
        $C \leftarrow \mathbf{p}_y - h_{ij} - b_{ij} - a_{ij}^2 + 2 \cdot a_{ij} \cdot c_{ij}$
        $t \leftarrow solveQuadratic(A, B, C)$
    end if
    $\mathbf{p} \leftarrow \mathbf{p} + t \cdot \mathbf{v}$                                ▷ Step along the ray
    $s \leftarrow s + 1$                                     ▷ Increase step count
end while
return $t$
```

4 Further optimizations

The introduced algorithms above perform similarly to the classic cone step mapping method in both runtime and error metrics. Naturally, there is always room for improvement, and we have made some optimizations that we deemed necessary.

4.1 Convex bounds

We have proposed an algorithm for constructing conservative parabolas for a texel on a heightfield in Section 3. The generation consists of two steps: first, we calculate the values of the radial function m_{uv}, and while this is the more costly of the two parts in terms of performance, it requires further research. Second, we compute the a, b, c parameters of the parabola by iterating through every possible value of this function.

Since this iteration is linear in texture size, it can be more efficient to reduce the number of values using the same method with less repetition. The mentioned reduction is made by computing the upper convex bound of the m_{uv} values, excluding a significant number of possible parameters from the search. Upper convex bounds can be constructed in linear time according to [1].

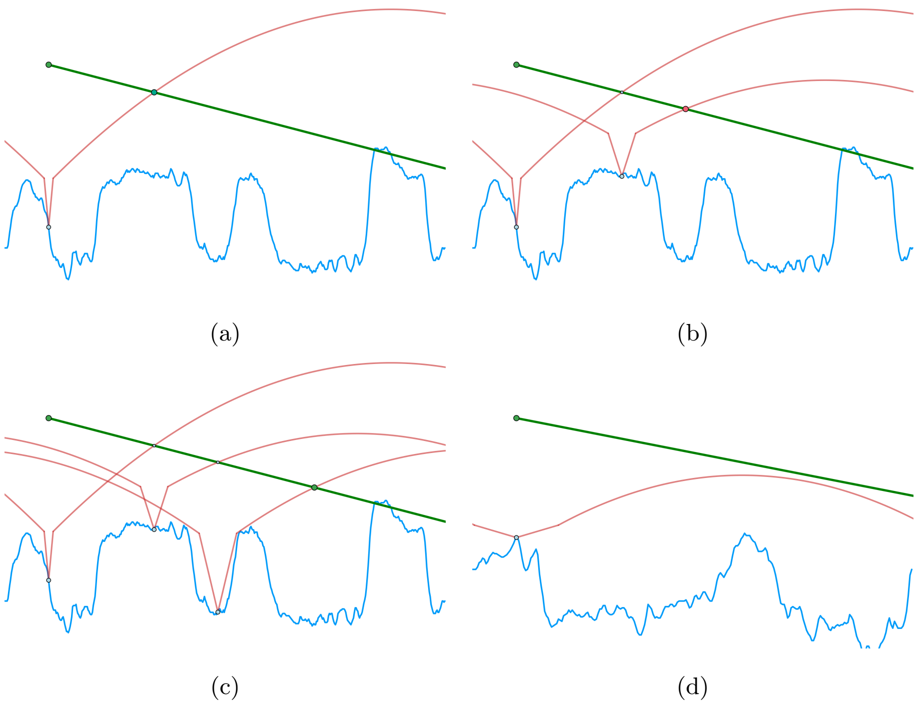


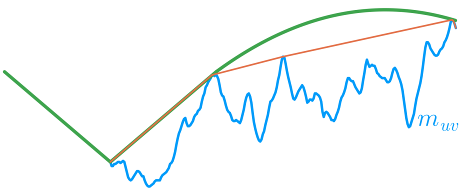
Figure 6: Ray tracing using quadratic maps. The first three pictures (a-c) show three tracing steps by reading the stored parabola in the current texel and calculating its intersection with the ray, resulting in the position of the next iteration. In picture (d), the ray misses the parabola, thus taking an arbitrarily large step.

This optimization, though not changing the overall complexity of the solution, allows to more efficiently separate the two phases of generation and reduces the required number of memory access queries. Additionally, the construction of the convex bound can be further accelerated and even performed directly from the heightfield, thus skipping the costly m_{uv} generation.

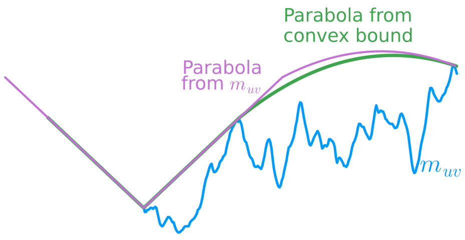
Note that with these changes, we will not always have the same parameters as a result, as demonstrated in Figure 7; however, it is guaranteed to preserve the conservative property of the parabolas.

4.2 Numerical stability

Time efficiency and numerical precision are critical during rendering to have the best results in the shortest possible time. It is known that finding the roots time-efficiently with minimal numerical error is a difficult task. Since we solve a quadratic equation in every iteration, we have to ensure that we do so in a numerically stable way.



(a) Example for parabola (green) generated using upper convex bound (orange) of the heightfield.



(b) Difference between a parabola generated from the radial m_{uv} function (purple) and from convex bound (green).

Figure 7: Parabola generated from convex bound

Blinn [3] has published a method to solve a general quadratic equation using homogeneous coordinates. Though robust and has a low error rate, it relies on several condition elevations, which can be time-consuming for real-time rendering. However, we can restrict the coefficients by considering how the values are computed.

Using the notations of Algorithm 1, it is guaranteed that $A \geq 0$ and $B < 0$, since $\mathbf{v}_x > 0, \mathbf{v}_y < 0$ and $c_{uv} > 0$ by definition. These inequalities allow writing a single conditional operator to yield the sufficient root of the equation, that is, checking if the root is real or not. The optimized quadratic equation solver is in Algorithm 2.

Algorithm 2 Numerically stable quadratic equation solver for parabolic maps

```
**Input:** $A \geq 0, B < 0, C \in \mathbb{R}$ coefficients
**Output:** smaller real root of $Ax^2 + Bx + C = 0$

$B \leftarrow \frac{B}{2}$
$M_1 = C$
$M_2 = -B + \sqrt{B^2 - AC}$
$x = \frac{M_1}{M_2}$

**if** $x \in \mathbb{R}$ **then**
    **return** $x$
**end if**
**return** $\infty$
```

5 Testing and results

The proposed method aims to reduce the number of steps taken along the rays during real-time heightfield rendering, thus lowering the GPU processing time of a single image. In this section, we compare the algorithms to Dummer's cone step mapping by distance taken per iteration and runtime of frame rendering. While the current state-of-the-art method is the relaxed cone stepping, it fundamentally differs from our proposed conservative technique. Quadric steps do not require refinement, as we are not entering the surface during rendering. Thus, the preferred choice of comparison is the cone step mapping.

These algorithms were implemented and tested in the Falcor framework by NVIDIA [6] with 1920 × 1080 screen resolution on a GeForce GTX 1060M GPU. The listed results are the average values of renders on 9 different 1024 × 1024 sized heightmaps from Figure 8.

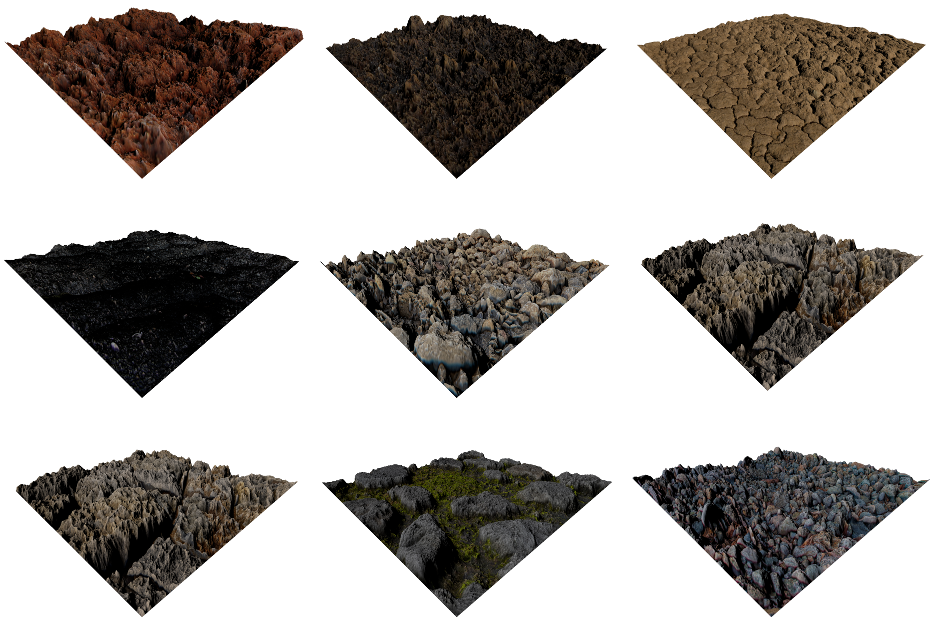


Figure 8: Heightmap samples used for testing the algorithms. Every texture has the same resolution of 1024 × 1024 pixels.

The height ratio parameter of the constructed quadratic surfaces is 0.2 in the following sections since this value seems to provide the best performance across our testing.

5.1 Step size and error

Both methods were analyzed by their performance compared to the same *ground-truth* image, a result of 200 iterations of linear search corrected with 20 steps of refinement. The absolute error for a ray is measured as the difference from this value. The rendered images are viewed from the same 8 camera positions that differ in incidence angle.

As shown in Figure 9, by increasing the number of maximal iterations, the total absolute error decreases for both algorithms as it is expected. In tests where the camera angle was below 45°, taking quadric steps usually gave significantly longer advancements along the ray because of the broad upper sections of the parabolas. Viewing the scene from higher than 45°, we lose some of this improvement; hence, the rays quickly reach the bottom part of the cones, which are similar in the two methods. Above 48 iterations, both algorithms seem to halt by reaching the heightfield surface or leaving the geometry, so the difference between their errors becomes insignificant.

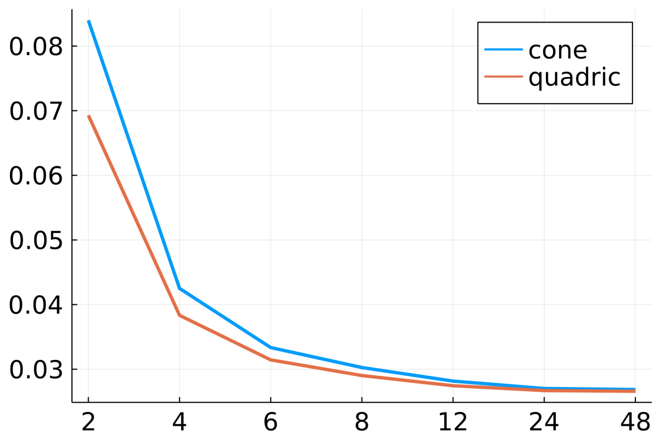


Figure 9: Average error in the distance taken on the ray (vertical axis) by iteration count (horizontal axis). Quadric stepping has generally lower error than cone step mapping, more significantly for fewer steps.

Figure 10 compares the average number of steps taken. Both algorithms follow a decreasing tendency by increasing the angle of the camera. Below 45° degrees, quadric mapping generally requires fewer iterations to converge. However, above 45° degrees, the method slows down as it approaches the surface.

5.2 Render time

We compared the two methods by average rendering speed for various heightfields from several camera angles and iteration numbers. Due to the composite nature of our representation and the fact that we have to resolve intersections with two different geometries, a single step of parabola tracing is computationally more expensive than that of cone step mapping. However, faster convergence properties allow for taking fewer steps, making our method more performant.

Our test results indicate that we can achieve better performance for view angles below 45° on all textures and maximal iterations. Although less noticeable, the overall mean rendering time for all angles is also reduced according to Table 1.

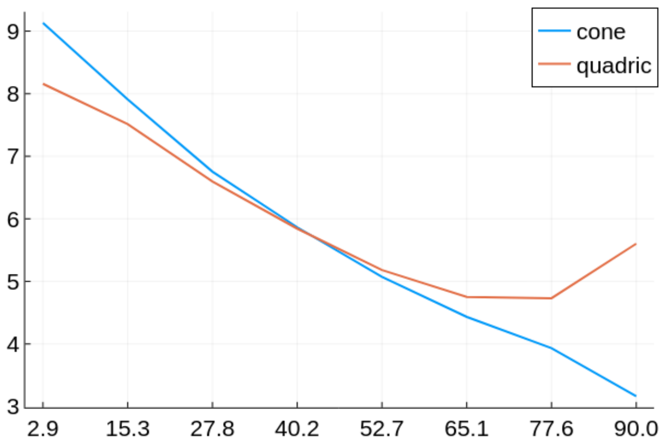


Figure 10: Average number of iterations before terminating of cone step mapping and quadric stepping (vertical axis). Up until 45°, quadric stepping performs better in general. The horizontal axis shows the angle in degrees.

Table 1: Average difference of render times between a single cone step and quadric steps in milliseconds. Negative values (highlighted) mean faster rendering for a quadric step. For angles below 45°, the sum of the values is -0.667, while for all values, it is -0.32, which means a faster average render time of a single image.

Angle	Cone minus Quadric render time (ms)						
	4 iters	6 iters	8 iters	12 iters	24 iters	48 iters	200 iters
2.9	**-0.023**	**-0.100**	**-0.047**	**-0.107**	**-0.093**	**-0.123**	**-0.260**
15.3	0.010	0.000	**-0.020**	**-0.010**	**-0.017**	**-0.023**	**-0.043**
27.8	0.027	0.017	**-0.003**	0.020	0.013	0.007	**-0.007**
40.2	0.023	0.017	0.013	0.027	0.023	0.013	0.000
52.7	0.020	0.013	0.010	0.023	0.020	0.010	0.010
65.1	0.013	0.010	0.010	0.017	0.020	0.010	0.000
77.6	0.010	0.007	0.010	0.017	0.010	0.000	0.000
90.0	0.010	0.010	0.013	0.027	0.027	0.010	0.010
Total:	0.090	**-0.027**	**-0.013**	0.013	0.003	**-0.097**	**-0.290**

There is less than 1% difference in the average runtime ratio between the two methods, where cone step mapping performed better as in Figure 11.

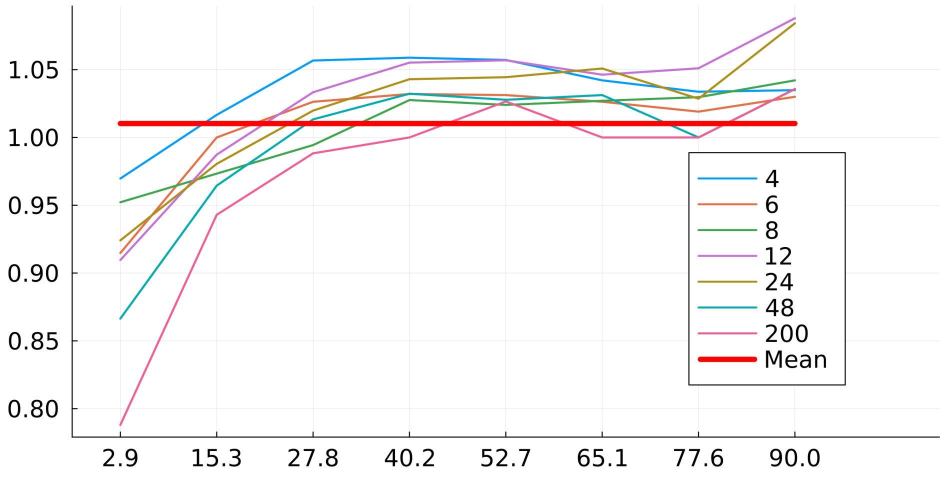


Figure 11: The average render speed of quadric steps compared to the cone stepping. The ratio (vertical axis) is below 1 for smaller angles (faster by $5-20\%$), and above $30°$, it becomes $1-5\%$ slower. The lines represent different measures with varying numbers of maximum iterations.

6 Conclusion

We proposed a method for efficient real-time ray tracing of heightfields, utilizing revolved parabolas stored in a four-channeled texture. We introduced algorithms for generating these parabolas and rendering the surface with additional optimizations.

The algorithms were compared to the cone step mapping technique in extended testing by convergence speed and frame render time. The tests showed that our method performed better in both metrics when the camera view angle was low and produced similar results otherwise. The slowdown can be originated from the arithmetic costs of a single ray-parabola intersection computation that we plan to optimize in the future.

The main improvement of our method showed in faster convergence of rays, which is achieved by taking longer steps in most of the iterations. This indicates that it can be efficient for rendering heightfields with more expensive queries such as procedural textures. We plan to explore these possibilities in the future.

Currently, the generation algorithm of the parabola maps demands high memory and computing capacity, which requires further optimization. We are currently experimenting with alternative methods for construction that could radically reduce the resources needed.

References

[1] Andrew, A. Another efficient algorithm for convex hulls in two dimensions. *Information Processing Letters*, 9(5):216–219, 1979. DOI: 10.1016/0020-0190(79)90072-3.

[2] Bálint, C. and Kiglics, M. A geometric method for accelerated sphere tracing of implicit surfaces. *Acta Cybernetica*, 25(2):171–185, 2021. DOI: 10.14232/actacyb.290007.

[3] Blinn, H. How to solve a quadratic equation? *IEEE computer Graphics and Applications*, 25(6):76–79, 2005. DOI: 10.1109/MCG.2005.134.

[4] Dummer, J. Cone step mapping: An iterative ray-heightfield intersection algorithm, 2006. URL: https://www.scribd.com/document/57896129/Cone-Step-Mapping.

[5] Halli, A., Saaidi, A., Satori, K., and Tairi, H. Per-pixel displacement mapping using cone tracing. *International Review on Computers and Software*, 3(5):1–11, 2008.

[6] Kallweit, S., Clarberg, P., Kolb, C., Davidovič, T., Yao, K.-H., Foley, T., He, Y., Wu, L., Chen, L., Akenine-Möller, T., Wyman, C., Crassin, C., and Benty, N. The Falcor rendering framework, 2022. URL: https://github.com/NVIDIAGameWorks/Falcor.

[7] Policarpo, F. and Oliveira, M. Relaxed cone stepping for relief mapping. *GPU Gems 3*, 3:409–428, 2007. URL: https://developer.nvidia.com/gpugems/gpugems3/part-iii-rendering/chapter-18-relaxed-cone-stepping-relief-mappingl.

[8] Szirmay-Kalos, L. and Umenhoffer, T. Displacement mapping on the GPU — state of the art. *Computer Graphics Forum*, 27(6):1567–1592, 2008. DOI: 10.1111/j.1467-8659.2007.01108.x.

Acta Cybernetica **26** (2024) 543–562.

The Influence of the Nonfunctional Requirements on the Data Model*

Grácián Kokrehel[ab] and Vilmos Bilicki[ac]

Abstract

During the design and development of real-world telemedicine applications, the data model evolves significantly along the datapath. The model itself, the storage technique, and the user interface are the most common contributors. This relates to non-functional requirements. The size and complexity of the domain model may also be significantly influenced by standards. This phenomenon is distinct from data model erosion, which occurs when the data model changes due to a software developer's fault and non-properly defined interfaces. This is occurring by design. We are unaware of any technique, including OMG's Unified Modeling Language (UML), that focuses on this aspect of complex systems: the change of the data model along the datapath. In this article, we investigate this phenomenon and, in addition to identifying the locations where this change may occur, we classify the modifications depending on the possible influence a specific model change may have on the system's overall properties. This paper presents a novel methodology for complex system datapath analysis and demonstrates its application to a selection of telemedicine-related applications. This technique illustrates the possible effect of non-functional requirements on the datapath and the potential consequences of these modifications.

Keywords: FHIR, telemedicine, GUI, Firebase, Angular, modeling

*This work was supported by the EU-funded Hungarian grant GINOP-2.2.1-15-2017-00073; project no. TKP2021-NVA-09 has been implemented with the support provided by the Ministry of Innovation and Technology of Hungary from the National Research, Development and Innovation Fund, financed under the TKP2021-NVA funding scheme; project no. II-NKFIH-1528-1/2021 has been implemented with the support provided by the Ministry of Innovation and Technology of Hungary from the National Research, Development and Innovation Fund, financed under the II-NKFIH-1528-1 funding scheme. The research was also supported by the Ministry of Innovation and Technology NRDI Office within the framework of the Artificial Intelligence National Laboratory Program (RRF-2.3.1-21-2022-00004).

[a]Department of Software Engineering, University of Szeged, Hungary
[b]E-mail: kokrehel@inf.u-szeged.hu, ORCID: 0000-0002-5074-6033
[c]E-mail: bilickiv@inf.u-szeged.hu, ORCID: 0000-0002-7793-2661

DOI: 10.14232/actacyb.299253

1 Introduction

In the late 1950s, renting an IBM 704 digital mainframe computer cost hundreds of dollars per minute. Recently, cloud computing as a service with on-demand pay-per-use is a widely used Information Technology (IT) phenomenon that offers great economies of scale. In order to make the platform as a service more accessible and affordable, serverless computing has attracted the interest of both industry and academia.

Another important trend is the widespread use of Internet of Things (IoT) devices. The Function as a Service (FaaS) and Platform as a Service (PaaS) solutions provide the de facto backend for IoT solutions. The integration of the IoT to the cloud/edge node is governed in most cases by the traditional Representational state transfer (REST) paradigm, implemented on top of the Hyper Text Transfer Protocol (HTTP). The data is typically serialized in JavaScript Object Notation (JSON). On the datapath, data travels through a variety of technology stacks.

Domain model erosion is a phenomenon in which the information model of an application becomes separated from its actual implementation. When applications shift from one technology stack to another, causing the information model to change, or when software engineers contribute changes that are not accurately reflected in the information model, this might occur. When the data model of an application is expressed in JSON format, but the actual implementation of the data model changes without corresponding modifications to the JSON representation, domain model erosion can occur. This might result in incompatibilities between the data model and its representation, leading to unanticipated application behavior.

Consider, for instance, a web application that employs a JSON representation to store user information. If a software engineer adds a new field to the user data model, such as a new email address, but does not update the JSON representation to incorporate this information, the program may continue to work but will not keep the new email address for users.

In addition to the data erosion, which may be viewed as a design flaw (lack of strong, typed interfaces), the data model change along the datapath may be a real but little understood phenomenon. A widespread system integration approach is based on the REST architectural style, therefore there is no absolute domain model, but rather a given representation on a given portion of the data path. If we consider the MVVM (Model-ViewModel) architectural pattern, we can observe that the domain model in a given location/layer may differ dramatically from other locations. All of these alterations are possible in software systems; a comprehensive system integration is not required to meet these issues.

Non-functional requirements may also influence this domain model. When an application needs to grow to accommodate greater traffic, this is a classic example of non-functional requirements resulting in data model modifications. To accommodate the increasing load, it may be necessary to modify the data model for performance by adding additional indices, denormalizing the data, or sharding it across different servers. Another illustration is when privacy and security needs change, resulting in data model modifications. For instance, a new rule may man-date that sensitive data be encrypted at rest, which may necessitate alterations to the manner in which it is kept within the data model.

In order to conform to non-functional requirements such as standard interfaces or system extensibility, the data format is governed by the standards of a given domain. E.g.: In the field of telemedicine the Fast Healthcare Interoperability Resources (FHIR) [4] standard is widely used. When implementing a system that is intended to conform to a standard, such as FHIR, the domain model may need to be extended or updated to comply with the standard. To describe the numerous healthcare concepts, the domain model may need to incorporate FHIR resources, such as patients, medications, conditions, and procedures, in the case of FHIR. Additionally, the domain model may need to be extended to include data elements.

2 Research questions

As stated in the introduction, both functional and non-functional requirements influence the evolution of data along the datapath. While functional requirements determine the necessary data transformations, non-functional requirements also play a significant role in shaping the data evolution process. This is a frequent practice among software developers, although it has not been properly investigated. This is a gray area from both a qualitative and quantitative sense. In addition to defining our study, we posed research questions. The first question discusses the potential ramifications of the modification. As observed, change occurs, but what are its consequences? The second research topic concerns the FHIR standard's effects on the domain model. FHIR is one of the few practical standards, and as a result, it is frequently used, making it an excellent candidate for study. The third study topic focuses on the technical environment's effects on the data model. In our situation, we chose Google's serverless Firebase solution, which is also a good contender due to its popularity. Within Firebase, the impact of using Firestore as a database is studied.

- RQ1: What are the dimensions which enable us to measure the impact of non-functional requirements?
- RQ2: Based on the defined dimensions, what are the impacts of the FHIR standard and using the Firebase API

3 State-of-the-art

There are numerous articles about software architecture and FaaS best practices. Wen et al. presented the first comprehensive study on understanding the challenges in developing serverless-based applications from the developers' perspective. They mine and analyze 22,731 relevant questions from Stack Overflow (a popular Q&A website for developers), and show the increasing popularity trend and the high difficulty level of serverless computing for developers. Through manual inspection of

619 sampled questions, they constructed a taxonomy of challenges that developers encounter, and report a series of findings and actionable implications [1].

In a different publication, Wen et al. presents a comprehensive study on characterizing mainstream commodity serverless computing platforms, including AWS Lambda, Google Cloud Functions, Azure Functions, and Alibaba Cloud Function Compute. Specifically, they conduct both qualitative analysis and quantitative analysis. Based on the results of both qualitative and quantitative analysis, they derive a series of findings and provide insightful implications for both developers and cloud vendors [2].

Grogan et al. showed the impact of the FaaS on the software architecture. The analysis of the data path is missing from these articles [3].

On another hand, there is a community focusing on cloud modeling e.g.: Bergmayr et al. investigated the diverse features currently provided by existing Cloud Modeling Languages (CMLs). They classified and compared them according to a common framework with the goal to support Cloud Service Customers (CSCs) in selecting the CML which fits the needs of their application scenario and setting. As a result, not only features of existing CMLs are pointed out for which extensive support is already provided but also in which existing CMLs are deficient, thereby suggesting a research agenda [4].

Software architectures allow identifying confidentiality issues early and in a cost-efficient way. Information Flow (IF) and Access Control (AC) are established confidentiality mechanisms, so modeling and analysis approaches should support them. Because confidentiality issues often trace back to data usage, data-oriented approaches are promising. However, Seifermann et al. could not identify a data-oriented approach to handling both, IF and AC. Therefore, they present a unified data-oriented modeling and analysis approach supporting both, IF and AC. They demonstrated the integration into an existing architectural description language and evaluated the resulting expressiveness and accuracy by a case study considering 22 cases [5].

The main theoretical results of Stunkel et al. are proofs of the facts that comprehensive systems are an admissible environment for (i) applying formal means of consistency verification (diagrammatic predicate framework), (ii) performing algebraic graph transformation (weak adhesive HLR category), and (iii) that they generalize the underlying set of graph diagrams and triple graph grammars [6]. The data path aspect and the evolution of the data are not studied.

From a data modeling perspective there are articles about the effect of the different storage formats that have been studied [7] or about different binary serialization formats [8]. There are also studies focusing on single system persistence issues [9] but the impact of the standards and the technical environment has not been studied.

Ulrich et al. presents a Metadata Repository (MDR) prototype that allows for the linking and mapping of data elements, enabling relations to be defined semi-automatically. The system enables the management of all registered data elements and metadata, allowing for comfortable queries within classified data elements. The architecture has technical advantages such as fast response times and the ability to search across clinical coding systems. The MDR allows for the reuse of data elements, simplifying cooperation among research groups. The system was evaluated with positive results using two methods: usability testing and cross-validation [10].

4 Typical software in telemedicine stack

A typical software stack consists of an IoT or mobile client and a FaaS backend. In our case, Firebase is used as a FaaS service while the client side is implemented in an Angular environment. The data model is implemented in a FHIR conformant way.

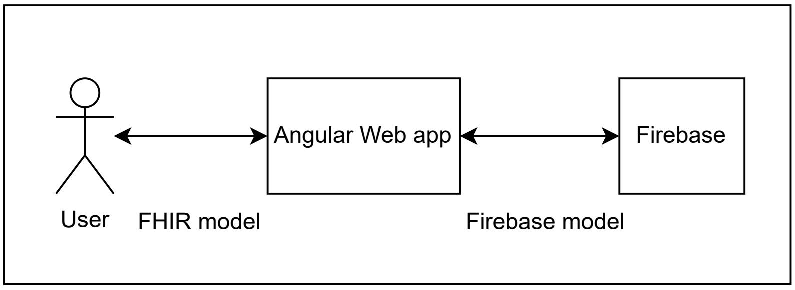


Figure 1: Cloud software stack

4.1 FHIR

The philosophy behind FHIR is to build a base set of resources that, either by themselves or when combined, satisfy the majority of common use cases. FHIR resources aim to define the information contents and structure for the core information set that is shared by most implementations. There is a built-in extension mechanism to cover the remaining content as needed [11].

4.2 Firebase

Cloud Functions is another integration of the existing Google product into Firebase. It is a tool for running back-end code from the cloud on an event-driven basis. The way Cloud Functions suggests running our app is what is usually called a serverless architecture. This type of architecture means building applications as a set of separate functions, isolated in the cloud, and connected between each other via APIs [12].

Usually, we will use the Realtime Database as our main storage. The main problem is limited querying capabilities. We cannot query for more than one key at a time and the service does not provide a way to filter our data. The format also excludes the option to model the data. We do not host the data, all data is hosted on Firebase and it is a major problem of using BaaS platforms as our app backend. Unless Firebase provides a migration tool to enable easy transfer of user's data, it strongly limits data migration. It makes users dependent on the platform and there is no easy way to transfer the app to another source.

5 Methodology

The CAP theorem was utilized as a starting point for defining the dimensions of impact. We expanded it with security; defining the taxonomy where security should be associated is not straightforward. The "C" (Consistency) component is the first aspect of security to consider within the context of the CAP theorem. Access restrictions, encryption, and authentication are examples of security controls and protocols that can be used to maintain data consistency and ensure that only authorized users have access to critical information. Next, security considerations can be included into the "A" (Availability) component by designing redundant systems and disaster recovery methods that assure the ongoing availability of vital systems and data in the event of a security breach or other calamity. This may involve backing up important data to secure off-site locations and deploying network segmentation to mitigate the impact of any security compromise. Last but not least, security can be integrated into the "P" (Partition Tolerance) component by providing security controls that can detect and respond to network partitions and other disruptions that may jeopardize the system's security. Therefore, we decided to treat it as a distinct dimension. As a consequence, we settled on four major dimensions or categories for measuring the impact:

- Consistency
- Availability
- Partition tolerance
- Security

We examined the source code of the selected systems in order to determine the patterns in which a domain model change could affect a certain dimension. We chose ten sample pilot projects from our telemedicine portfolio as the focus of our research. From a functional standpoint, these projects adequately address the requirements of a certain medical field of interest. The target field might be deemed representative since, from both a modality (e.g., imaging, CT, vital signals, lifestyle, etc.) and a health sciences perspective, it encompasses a vast array of topics (e.g.: dermatology, Otology and rhino-laryngology, diabetes care, cohrea surgical planner, etc).

This was the qualitative portion of our analysis, in which we identified patterns within the source code that influence system properties along a specific dimension (CAP + Security).

To examine the influence of the technology stack and FHIR (RQ2), we conducted an analysis centered on the domain model. We took the source code and, beginning with the user interface, traced all affected source code sections until they reached the backend. This allowed us to extract the data pathways. Then, we extracted entity-specific subpaths from these datapaths (instances of a given part of the domain model). We integrated the data paths along the entities along the dimension, incorporating the discovered pattern-based concerns. Thus, a summary of the impact of technical infrastructure and FHIR on a given dimension at the entity level was obtained.

In the case of availability, we extended this analysis to include the average influence of the standard and the technological environment by collecting the data (from the databases of the running applications) at the entity level and deleting each subsequent layer until just the core information remained. The average size of each entity was then determined for each domain model level.

In the section under "Threats to Validity," the statistical importance of our data collection methods will be examined in detail. Here, we would like to emphasize that the analyses presented represent simply the first "sampling approach." There is a continuing effort to crawl and automatically analyze a subset of GitHub projects. The patterns detected by manually analyzing the code will likely be extended, but the current patterns will remain valid. The results on the data volume inflation may also be slightly impacted, but we are certain that our results will accurately reflect the magnitude of change.

6 Results

In this section, we define the dimensions and then show what the overall impact is based on real world examples. Following that, we will analyze the size problems and then we will present the structure of the data at a given analysis point.

6.1 Dimensions

RQ1: What are the dimensions which enable us to measure the impact of non-functional requirements?

As indicated in the methodology section, we decided to choose CAP+ Security as the primary dimensions. We gathered code patterns that have an effect on one or more of these dimensions. The patterns are classified into categories, which are then linked to dimensions. The categories were identified using a pattern-based grouping strategy. As indicated in the methodology section, we opted to choose CAP+ Security as the primary dimension. We gathered coding patterns that influence at least one of these dimensions. The patterns are divided into categories, and the categories are then linked to dimensions. As an appropriate grouping technique, the categories were determined based on the patterns. The following categories have been identified:

1. T(transformation): refers to situations where the domain model transforms online. Typically, this is the result of employing the MVVM design pattern or a comparable one. However, this may also occur on the backend, where some aggregation is required.
2. S(security): this indicates a data leak may occur (e.g.: for a given screen there could be an aggregated screen specific data containing sensitive information)
3. C(consistency): in this instance, consistency may be at risk (e.g.: aggregation of data and if the trigger is not executed then the data becomes inconsistent)
4. Pe(performance): data manipulation or access with increased overhead (e.g: deep data structures).

We can link these categories tha CAP in the following way:

- Consistency: C (Consistency), T (Transformation)
- Availability: Pe (Performance)
- Security: S (Security)

Due to Firebase, there is strong partition tolerance and availability, but in exchange, there are significant issues with consistency. During the analysis, we did not encounter any partition tolerance errors that would affect the data. We developed a tool for identifying the portion of the data path affected by the given effect. Table 1 displays the places and categories, with the names of the most significant patterns identified in each cell.

Table 1: Dimension with patterns

T	sr/mr (single/multi row)	c(client)	s(server)	
T	1 - string	2 - number	3 - date	4 - others
S	1 - gdpr			
C	1 - id reference	2 - embedded data	3 - many extra data	4 - data after delete
Pe	1 - deep data	2 - multiple promise	3 - handle null/und	

Then, we developed a coding method to enable the efficient and compact coding of all relevant information. The processed projects were developed using the Angular framework, therefore the code snippets below will be presented with JavaScript or TypeScript syntax. Table 1 helps in interpreting the following code snippets. These are the identified patterns:

T SM/MR C/S 1 String transformations happen including the modification of one or more records on the client or server side. In Firebase there is no way to order by based on the value of the object that is in the array, so it must be outsourced to a separate field. In Firebase there is no %like% search option, so another method can be used to achieve the same effect, which results in having to organize the text into a string array. The complex FHIR data structure can be simplified by merging it into one text.

```
function splitNameByVariations(nameString: string): Array<string> {
   const allSubstrings = new Set<string>();
   const start = 0;
   const splittedName = nameString.split(' ');

   for (let j = 0; j < splittedName.length; j++) {
       for (let i = start + 1; i <= splittedName[j].length; i++) {
           allSubstrings.add(splittedName[j].substring(start, i));
       }
   }

   let a = 0;
   while (a < splittedName.length) {
       let possibleSubStrs = '';
       for (let i = a; i < splittedName.length; i++) {
           possibleSubStrs += splittedName[i];
           if (i < splittedName.length - 1) {
               possibleSubStrs += ' ';
           }
       }
       for (let i = start + 1; i <= possibleSubStrs.length; i++) {
           allSubstrings.add(possibleSubStrs.substring(start, i));
       }
       a++;
   }

   for (let i = start + 1; i <= nameString.length; i++) {
       allSubstrings.add(nameString.substring(start, i));
   }

   return Array.from(allSubstrings.values());
}
```

T SM/MR S 2 Often happens that the smart device sends the data in the form of a string or with an incomplete value (undefined/null). Firebase cannot handle undefined data. Number transformations happen including the modification of one or more records on the server side.

```
getPulse(notFhirFormatData: any) {
    // for(data in [pulse, hearthrate, bp])
    let pulse = notFhirFormatData.pulse;
    if (pulse === undefined) {
        return null;
    }
    pulse = pulse.trim() as unknown as number;
    pulse = pulse.toFixed(2);
    return this.convertToFHIRFormat(pulse);
}
```

T SM /MR C/S 3 Date transformations happen including the modification of one or more records on the client or server side. Date transformation must be performed before insertion and modification, because Firebase used a unique date solution. timestamp = nanoseconds: 0, seconds: 0. If this transformation does not take place and we try to use the data, the client side will fail with an error. The localization required by the user also takes place here.

```
function formatPeriodLocal(start: Date, end?: Date) {
 const localStartDate = DateTime.fromISO(start.toISOString(),
   { zone: 'Europe/Budapest'});

 let formattedLocalDate = localStartDate.year + '. ' +
   localStartDate.month.toString().padStart(2, '0') + '. ' +
   localStartDate.day.toString().padStart(2, '0') + '. ' +
   localStartDate.hour.toString().padStart(2, '0') + ':' +
   localStartDate.minute.toString().padStart(2, '0');
 if (end) {
   const localEndDate = DateTime.fromISO(end.toISOString(),
     { zone: 'Europe/Budapest'});
   formattedLocalDate += ' - ' + localEndDate.hour.toString()
     .padStart(2, '0') + ':' +
     localEndDate.minute.toString().padStart(2, '0');
  }

 return formattedLocalDate;
}
```

C 2/3/4 Due to Firebase, numerous attributes and objects must be stored in the data; if the original data is modified, Cloud Function must be used to update/delete all embedded data, otherwise the data becomes inconsistent.

```
export const onEntity1UpdateOrDelete =
 functions.firestore.document('/Entity1/{cpid}').onUpdate/onDelete(
    async (snap: any) => {
        if (snap.after.exists === true) {
          const entity1: any = snap.after.data();
          const entity2: any = this.get(entity1.referenceToEntity2);
          // update/delete the embedded data in entity2        }
    });
```

Pe13 The deep data structure and their non-mandatory attributes place additional burdens on the client side. So that the application does not stop, we handle all non-mandatory data elements with the optional chain operator (?).

```
// reading the patient name
{{ patient?.name?.[0]?.given?.[0] }} &nbsp;
{{ patient?.name?.[0]?.family }}
```

Pe2 Occasionally, a piece of data contains many references that must all be accessed on the client side. It can have many sources of error, and it must be treated as a transaction, if it stops during any step, it must be started again. This can be extremely burdensome for the client side.

```
async function getPatient(): Promise<IPatient>
{ /* Handle error, reset on fail, process data... */ }

async function getDevice(): Promise<IDevice> { /* ... */ }

async function getParent(): Promise<IPractitioner> { /* ... */ }

addApointment(){
    const [patient, device, parent] =
    await Promise.all([getPatient(), getDevice(), getParent()]);
    // Do something, create an appointment .....
}
```

As a summary, based on the patterns discovered in the source code files along the datapath, we divided the effects into four groups and, within each category, determined the exact code patterns responsible for the observed effect. Dimensions were also tied to the categories.

6.2 Overall impact

RQ2: Based on the defined dimensions, what are the impacts of the FHIR standard and using the Firebase API?

In order to assess the impact among the dimensions we collected the handled entities. We analyzed the datapath for each entity in different applications and created an abstract datapath based on the metrics. The table below contains the result of this process.

In Table 2, all the models that have been used till now were subjected to the evaluation points defined in Table 1. In the other columns, three main problem sources were defined: FHIR, Firebase(CRUD) and GUI. Then the representation of common errors at given points. Based on the above results, it can be noticed that the FHIR model alone is not enough, but several transformations have to be applied during development to get the proper results. If we observe carefully we can see that the incorrect combinations are repeated column by column, so for e.g. under Firebase Create we get the same errors, which means that the problems can be well delimited and thus they can be solved as a group, there is no need to deal with it specifically by model. The development of solutions to individual problems can be defined and reused for other models as well. Explanation for Patient row:

1. String transformation must be performed on the server before insertion and modification, because in Firebase there is no %like% search option, so another method can be used to achieve the same effect, which results in having to organize the text into a string array.

2. When reading data, consistency problems arise because of the embedded data. Because of Firebase, many attributes and objects must be stored in the data, if the original data is changed, all embed data must also be updated using Cloud Function.

3. It is a security problem if the patient's sensitive data (name, social institute number, birthdate) are also displayed when the view tables are created.

4. It is an extra task to ensure that no reference to the entity remains anywhere after deletion.

For example, in the Appointment row:

1. Date transformation must be performed before insertion and modification, because Firebase used a unique date solution. timestamp = nanoseconds: 0, seconds: 0

2. Because of more than three references and embed data, extra data has to be retrieved and displayed, which burdens the performance of the GUI.

Table 2: Analysis of FHIR models

Model	FHIR	Firebase				GUI
		C	R	U	D	
Device	C2 C3 Pe1	T4s	C2 S1	T4s	C4	
Patient	C3 Pe1	T1srs	C2 S1	T1srs T1mrs	C4	C4 Pe1
Practitioner	C3 Pe1	T1srs	C2 S1	T1srs T1mrs	C4	C4 Pe1
Appointment	C1 Pe1	T3s	C2	T3s		Pe2
Questionnaire	C1 Pe1					Pe1
Questionnaire Response	C2 Pe1					Pe1
Group	Pe1	T1s	C2	T1s		C4 Pe2
CarePlan	Pe1		C2			Pe1
Condition	Pe1		C2			Pe2
Communication	Pe1		C1			Pe1
Goal	Pe1		C2			Pe1
Medication Request	C2 Pe1		C2			Pe1
ServiceRequest	C2 C3 Pe1		C2			

6.3 Data size evolution

RQ2: Based on the defined dimensions, what are the impacts of the FHIR standard and using the Firebase API?

The last dimension of the analysis is the data overhead caused by the elements on the datapath. Out of the 25 deployed telemedicine solutions, we extracted data of the selected entities. The starting point was the fully extended data stored in the Firebase, with the help of scripts we were able to remove the firebase specific field in order to get the simple FHIR conform data, with the help of another script we were able to extract the core data in order to remove the FHIR overhead, now we have what is called the basic data. Figure 2 shows the size difference of 16.67 times between the simple and the final state.

Table 3 shows the data overhead for both the clear text and compressed data. We can conclude that FHIR adds a very significant overhead, while the Firebase specific data fields also doubles the FHIR data volume. So it is important to remove this overhead before sending it to the IoT client, the DAO layer has its place in this case.

In Table 3, we compared the 3 forms of data (Simple, after FHIR conversion and after transformations required by Firebase) in JSON file size (bytes) and zipped (bytes) file size. We can observe is that the size of the json files without repetition does not change as much as expected after compression. Furthermore, it can be observed that the ratio between the columns is the same or very similar for each row.

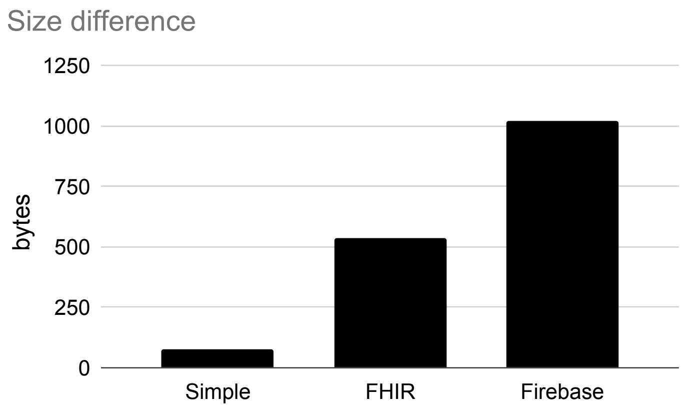


Figure 2: Size difference

Table 3: Size difference(bytes)

Model	Basic	Basic zip	FHIR	FHIR zip	Firebase	Firebase zip
Device	94	36	980	499	1560	602
Patient	220	56	1920	700	3666	937
Practitioner	130	42	1430	628	2080	708
Appointment	84	35	992	518	1410	574
Questionnaire	72	31	985	503	1160	523
Questionnaire Response	145	46	1653	526	2400	805
Group	47	28	700	412	799	440
CarePlan	49	31	809	426	865	459
Condition	39	23	589	375	659	398
Communication	37	22	581	369	643	388
Goal	44	26	646	388	705	411
Medication Request	43	25	623	384	716	430
Service Request	74	37	1180	586	1320	623

6.4 Details of the data expansion

In the previous section, we presented the effect of certain parts of the system, now we will explain step by step how the models look. The comparison is based on JSON file sizes. In this section, the different stations are introduced and explained, based on a simple example. Listings 1, 2, and 3 can be found in the Appendix section. The four analysis condition are as follows:

1. Simple (file size: 75 bytes) [Listing 1]
2. Effects of FHIR (file size: 536 bytes) [Listing 2]
3. Effects of Firebase (file size: 1022 bytes) [Listing 3]

6.5 Effects of Firebase

Due to the use of Firebase, the model went through extra changes in order to be able to implement certain filtering interfaces. Description of data fields:

- `data` — The data must be stored according to the FHIR standard and returned to the user in this form.
- `email, name` — In Firebase there is no way to order by based on the value of the object that is in the array, so it must be outsourced to a separate field.
- `nameText, emailText` — In Firebase there is no `%like%` search option, so another method can be used to achieve the same effect, which results in having to organize the text into a string array.

7 Discussion

As indicated in the introduction, our motivation was to observe/study the life/evolution of the domain model along the datapath as influenced by non-functional requirements such as technology, standards, and even the use of predefined design patterns. As we were unable to find appropriate measures in the literature, we established both the dimension along which we wanted to assess the impact and, within that dimension, the exact metrics that assist us understand the nature of the change and its impact. Instead of using a theoretical approach, we began to examine the source code for patterns that could be utilized as metrics. We were able to discover categories and groupings of patterns. We uncovered thirteen patterns, which we presented in the findings section. We supplied a fundamental datapath map indicating where and why these patterns could be discovered (backend vs. frontend, which data subpath e.g. CRUD). With this method, it is possible to discover potential points that are not ”bad smells” from a software quality viewpoint, but rather key portions of the datapath that require special attention. We believe that this datapath-oriented perspective could provide important insight into a system's inherent features. Table 2 provides a summary of this map's typical entity level pathways. The true potential of an approach will become apparent when defined patterns could be detected automatically. Our ongoing effort is now centered on this issue. We have also demonstrated how implementing a standard or utilizing a technology stack affects the data size of the domain model. We agree that we cannot alter the standard itself, but we would like to provide a suggestion for future standard's data structure architecture. If one consults the FHIR documentation, it is evident that modularity/extensibility, not simplicity, was the driving force behind the domain model. One could argue that with 4G, 5G, and XG technologies, the amount of data to be transmitted across the line is insignificant due to the large bandwidth. Here, we would like to emphasize the delay caused by the transfer of substantially more data. As end users are not patient, delay is a crucial part of the design of actual user-facing technologies.

8 Threats to Validity

The objective of our research was to determine the domain model's response to non-functional needs. We adopted an analytical strategy by identifying and assessing a code basis. We selected 10 telemedicine initiatives (consisting of more than 200 screens, and 300 modules) from our portfolio. Functionally, these projects satisfy the needs of the medical industry. The target field is representative from both modality (e.g., imaging, CT, vital signals, lifestyle) and health sciences vantage points (e.g., imaging, CT, vital signals, lifestyle) (e.g.: dermatology, Otology and rhino-laryngology, diabetes care, cohrea surgical planner, etc). We agree that further examples from the open-source community should be included. This is likely to increase the number of sample patterns and create a more complex depiction of the problem, but our conclusion and fundamental patterns will remain unchanged. Consequently, it may be anticipated as the initial result in this field. Concerning the size-related findings, it is difficult to construct a valid database for a particular open-source application; thus, we believe that our results are statistically significant because our system is utilized in routine medical work.

During our research, we made the following decisions:

- The FHIR was used as an example for analyzing the impact. As one of the most prevalent criteria, we believe this to be a suitable option.

- We chose the Angular - Firebase technology stack to investigate the impact of technology on the domain model. In this case, we consider that the selection of technological stacks does not reduce the statistical significance of the study. In the case of web application frameworks (e.g., React, Vue, etc.), general design patterns (e.g., MV*) may have similar effects. From a backend perspective, Firestore has the same constraints as other serverless document stores, therefore it was also a strong contender. We agree that it would be interesting in the future to categorize the various persistence options and evaluate the impact of each category separately.

9 Conclusions

It is evident from the literature review that there are numerous techniques to investigate complex software stacks, but there are very few articles that account for the entire data lifecycle. By evaluating significant issues encountered in the creation of Telemedicine applications utilizing the FHIR standard, we identified key evaluation criteria for modern systems. With our methodology, we can determine how architectural and component-level design patterns are applied. This strategy will demonstrate its effectiveness if it is accompanied by an automated data life cycle analysis tool. In our ongoing effort, we have already located more than 9k GitHub projects, and we are currently creating an NLP-based method for identifying the patterns that match to the requirements. Massive standard objects present issues when a small IoT device transmits data; in this instance, it is important to consider a simpler model than FHIR. FHIR adds a large amount of overhead, and the addition of Firebase-specific data fields increases the FHIR data capacity. Before providing data to the IoT client, it is crucial to eliminate this overhead; the DAO layer has a place in this scenario. Even if the system must be standardized, the DAO layer makes it possible for devices to provide a minimal quantity of data while the ultimate outcome is still standardized. In the subsequent essay, we will describe this concept's capabilities.

References

[1] Altexsoft. What is Firebase: Review, pros and cons, alternatives. URL: https://www.altexsoft.com/blog/firebase-review-pros-cons-alternatives/. Accessed: 2022-09-26.

[2] Bergmayr, A., Breitenbücher, U., Ferry, N., Rossini, A., Solberg, A., Wimmer, M., Kappel, G., and Leymann, F. A systematic review of cloud modeling languages. *ACM Computing Surveys*, 51(1):1–38, 2018. DOI: 10.1145/3150227.

[3] Grogan, J., e. a. A multivocal literature review of Function-as-a-Service (FaaS) infrastructures and implications for software developers. In *European Conference on Software Process Improvement*, pages 58–75. Springer, 2020. URL: https://link.springer.com/chapter/10.1007/978-3-030-56441-4_5.

[4] HL7. Overview — FHIR v4.0.1. URL: https://www.hl7.org/fhir/overview.html. Accessed: 2022-03-26.

[5] Petković, D. SQL/JSON standard: Properties and deficiencies. *Datenbank Spektrum*, 17(3):277–287, 2017. DOI: https://doi.org/10.1007/s13222-017-0267-4.

[6] Seifermann, S., Heinrich, R., Werle, D., and Reussner, R. A unified model to detect information flow and access control violations in software architectures.

In *Proceedings of the 18th International Conference on Security and Cryptography*, Volume 1, pages 26–37. SCITEPRESS — Science and Technology Publications, 2021. DOI: 10.5220/0010515300260037.

[7] Stünkel, P., König, H., Lamo, Y., and Rutle, A. Comprehensive systems: A formal foundation for multi-model consistency management. *Formal Aspects of Computing*, 33:1067–1114, 2021. DOI: 10.1007/s00165-021-00555-2.

[8] Swami, D. and Sahoo, B. Storage size estimation for schemaless big data applications: A JSON-based overview. In *Intelligent Communication and Computational Technologies*, Volume 19 of *Lecture Notes in Networks and Systems*, pages 315–323. Springer, Singapore, 2018. DOI: 10.1007/978-981-10-5523-2_29.

[9] Ulrich, H., Kock, A.-K., Duhm-Harbeck, P., Habermann, J. K., and Ingenerf, J. Metadata repository for improved data sharing and reuse based on HL7 FHIR. *Studies in Health Technology and Informatics*, 281:160–164, 2021. DOI: 10.3233/978-1-61499-678-1-162.

[10] Viotti, J. C. and Kinderkhedia, M. A survey of JSON-compatible binary serialization specifications, 2022. DOI: 10.48550/arXiv.2201.02089.

[11] Wen, J., Chen, Z., Liu, Y., Lou, Y., Ma, Y., Huang, G., Jin, X., and Liu, X. An empirical study on challenges of application development in serverless computing. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 416–428. Wiley, 2021. DOI: 10.1145/3468264.3468558.

[12] Wen, J., Liu, Y., Chen, Z., Chen, J., and Ma, Y. Characterizing commodity serverless computing platforms. *Journal of Software: Evolution and Process*, 35(10), 2021. DOI: 10.1002/smr.2394.

Appendix

Listing 1: `Simple.json`

```
{
   "id": "1",
   "email": "jon@doe.mr",
   "name": "Mr. Jon Doe"
}
```

Listing 2: `Fhir.json`

```
{
   "id": "1",
   "telecom": [
       {
           "system": "email",
           "value": "jon@doe.mr",
           "use": "home",
           "rank": 1,
           "period": ""
       }
   ],
   "name": [
       {
           "use": "official",
           "text": "Mr. Jon Doe",
           "family": "Doe",
           "given": [
               "Jon"
           ],
           "prefix": [
               "Mr."
           ],
           "suffix": [],
          "period": ""
       }
   ]
}
```

Listing 3: `Firebase.json`

```
{
   "id": "1",
   "data": {
       "id": "1",
       "telecom": [{
               "system": "email",
               "value": "jon@doe.mr",
               "use": "home",
               "rank": 1,
               "period": ""
           }],
       "name": [{
               "use": "official",
               "text": "Mr. Jon Doe",
               "family": "Doe",
               "given": ["Jon"],
               "prefix": ["Mr."],
               "suffix": [],
               "period": ""
           }]
},
   "email": "jon@doe.mr",
   "name": "Mr. Jon Doe",
   "nameText": [
       "m", "mr", "mr.", "j", "jo", "jon", "d", "do", "
          doe", "mr. j",
       "mr. jo", "mr. jon", "mr. jon ", "mr. jon d", "mr
          . jon do", "mr. jon doe"
   ],
   "emailText": [ "j", "jo", "jon", "jon@", "jon@d",
       "jon@do", "jon@doe", "jon@doe.", "jon@doe.m", "
          jon@doe.mr"] }
```

Acta Cybernetica **26** (2024) 563–579.

Integer Programming Based Optimization of Power Consumption for Data Center Networks

Gergely Kovásznai[a] and Mohammed Nsaif[b]

Abstract

With the quickly developing data centers in smart cities, reducing energy consumption and improving network performance, as well as economic benefits, are essential research topics. In particular, Data Center Networks do not always run at full capacity, which leads to significant energy consumption. This paper experiments with a range of optimization tools to find the optimal solutions for the Integer Linear Programming (ILP) model of network power consumption. The study reports on experiments under three communication patterns (near, long, and random), measuring runtime and memory consumption in order to evaluate the performance of different ILP solvers. While the results show that, for near traffic pattern, most of the tools rapidly converge to the optimal solution, CP-SAT provides the most stable performance and outperforms the other solvers for the long traffic pattern. On the other hand, for random traffic pattern, Gurobi can be considered to be the best choice, since it is able to solve all the benchmark instances under the time limit and finds solutions faster by 1 or 2 orders of magnitude than the other solvers do.

Keywords: integer programming, optimization, power consumption, Data Center Network, solvers

1 Introduction

Data Centers Networks (DCNs) are becoming increasingly significant in daily routine because of the fast growth of modern information technologies such as the Internet of Things, Big Data, Cloud Computing, and Mobile Sensing Networks [17, 16]. DCNs aim for high reliability and stability with several redundant links and enough capacity. The network devices usually work at full capacity 24 hours a day, consuming much energy. However, network equipment is underutilized most of the time, causing extremely low network energy efficiency. As a result, this problem attracts

[a]Department of Computational Science, Eszterházy Károly Catholic University, Eger, Hungary, E-mail: kovasznai.gergely@uni-eszterhazy.hu, ORCID: 0000-0001-8455-0218

[b]Department of Information Technology, University of Debrecen, Hungary, E-mail: mohammed.nsaif@mailbox.unideb.hu, mohammed.nsaif@uokufa.edu.iq, ORCID: 0000-0001-6768-4644

DOI: 10.14232/actacyb.299115

many researchers to figure out techniques that save energy while maintaining network performance. For the current DCNs, there are two techniques to save power consumption: device sleep [1, 7] and adaptive link rate [19]. The device sleeping technique is based on turning on/off the switches and links that are under a utility value in a dynamic manner. Whereas the adaptive link rate technique is based on assigning the fair bandwidth value for each flow passing through the links to control the ports' clock rate (operating frequency), leading to lower power consumption.

Because switches are considered to be one of the major devices in DCNs, this paper focuses on devices' sleep techniques to save power. This technique appeared in 2010; Heller et al. [7] introduced three types of optimizers to save power consumption: formal model, greedy bin-packing, and topology-aware heuristic. The topology-aware heuristic shows good results in saving up to 50%. It is based on elastic topology, which increases/decreases the size of the topology according to the size of the traffic.

Our current paper builds upon our last contribution in [14], which proposed an Integer Linear Programming (ILP) model for traffic and energy-aware routing in Software-Defined Networking (SDN) based on link utility information, and could decide many pathways simultaneously. Additionally, it proposed a link utility-based heuristic algorithm called FPLF, which had the ability to save energy up to 10% when the traffic load is high (e.g., during rush hour) and 63.3% when the load is low (e.g., at night). Our current paper aims to explore and examine other ILP solving tools that can solve convex and non-convex optimization problems, which we can use in real-time action to find an optimal solution for a large number of injected flows, instead of FPLF-heuristic solutions.

The rest of the paper is arranged as follows. In Section 2, we review recent papers and studies that use an ILP formulation to optimize power consumption. The power optimization problem for DCNs is explained in Section 3. Our ILP model is described in Section 3.1. Our experiments, benchmarks, ILP solving tools, our portfolio solver, and the experimental results are detailed in Section 4. Finally, we summarize all our key contributions and outline some future directions in Section 5.

2 Related Work

This section outlines the robustness and limitations of recent Integer Linear Programming (ILP) approaches that address the power consumption decrease challenge for network routing algorithms.

The authors in [7] developed three methods to calculate a minimum-power network subset; one of them uses a formal model. The objective function consists of two binary variables for each switch and link. The constraints represent Multi-Commodity Network Flow, Power Minimization, and Flow Split. At the same time, the model's input parameters include the traffic matrix, the switch power model, and the topology. The model outputs a subset of the original topology and per-flow route information. While the study focuses on the number of nodes that the model can manage in the topology, network performance is not taken into consideration.

In contrast, our current model calculates the minimum number of links that satisfy the traffic matrix, based on the utilization matrix.

The authors in [18] proposed a 0-1 ILP model to minimize the power of idle line cards and integrated chassis by switching them to sleep, under link utilization and packet delay constraints. Meanwhile, the study proposes two heuristic algorithms for the same purpose and reports on experiments executed in two scenarios: synthetic topology and real-life topology named CERNET. In the same context, another study in [2] proposes an ILP model with a multi-objective function to minimize the sum of the energy consumption of switches and links. The study limits the links' maximum and minimum utility to manage the trade-off between the power consumption and the network performance. Nevertheless, neither study presents any experimental result with the ILP model, and the algorithms were experimented outside of any DCN.

[11] proposed a data center scheduling algorithm called FLOWP, besides an ILP formula, with the aim of optimizing power consumption and Quality of Service (QoS). The formula considers a minimum threshold for the efficiency of links and switches. The results show that QoS is improved compared to the approach in [7]. However, similar to [18], the study does not show any experimental result with the ILP model. Experiments are conducted on a heuristic algorithm only.

Our contribution in [14] presents an ILP model that has the ability to manage multiple paths to save power consumption and to balance the load at overloaded times. The study experiments with the model using the optimization tool LINGO [10]. The results show that LINGO could not find the optimal solution in a reasonable time for high number of flows sent simultaneously. Our current study shows that more powerful optimization tools can find solutions in a reasonable time.

Finally, we mentioned that various ILP formulations have been proposed to address the traffic-aware energy consumption challenges. Nevertheless, in many of them, the results of optimal solutions do not scale to a large number of links, nodes, line cards, switches, or *flows* [14]. Some of these ILP formulations are designed for appointed DCN topologies, i.e., fat-tree and bicubic. On the other hand, some of them are independent of topology structures. All those facts encouraged the authors to explore a wide range of state-of-the-art optimization tools and to compare their experimental results in the current study.

3 Problem Statement and Proposed Solutions

Although the ILP model in [14] can calculate optimal multi-path ways and can manage the current status of the network, the model was solved by using the optimization toolkit LINGO, which was costly when the network size and number of flows were large. It took more than 160 minutes to accommodate only 120 bursts of flows in the topology (a fat-tree topology with $k = 4$) at one time. This fact motivated the authors to propose a heuristic routing algorithm called Fill Preferred Link First (FPLF) to find feasible solutions.

Figure 1 shows part of the results from [14]. Based on the results, the left figure shows that LINGO cannot find a solution in reasonable time when sending more than 100 simultaneous flows, and the runtime dramatically increases. On the other hand, the right figure shows how the number of active links increases proportionally to the flows. Therefore, the authors of the current study think that it might be possible to find more powerful optimization tools that can find the optimal solution for a higher number of flows in reasonable time.

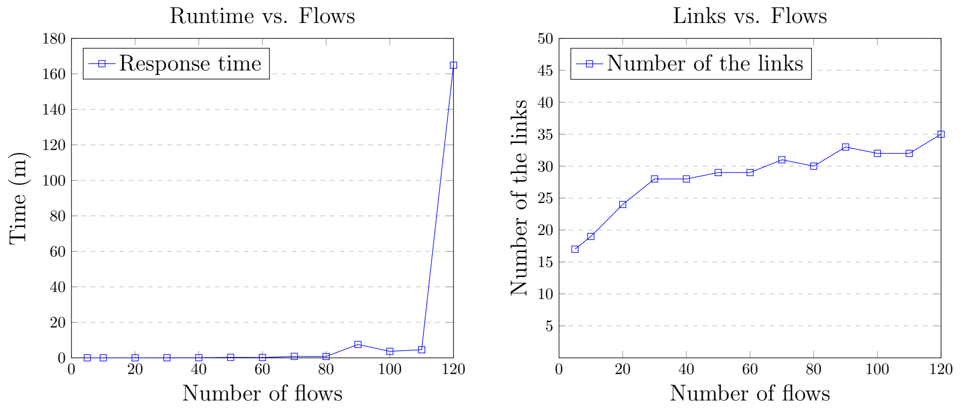


Figure 1: Correlation between the number of flows and the runtime of LINGO, and between the number of flows and the number of active links, respectively.

Our contributions in the current paper are summarized as follows:

- Developing NEO-DCN, our network optimizer tool for DCNs.
- Generating more realistic benchmarks to experiment with the proposed ILP model for three different traffic patterns.
- Evaluating the performance of several solvers with the ILP model on three different benchmarks.

3.1 DCN Models and Constraints

The DCN is modeled as an undirected graph G = $(\mathbb{S}, \mathbb{E})$, where $\mathbb{S} = \{s_1, s_2, \ldots, s_n\}$ is a set of switches and $\mathbb{E} \subseteq \{e_{ij} \mid s_i, s_j \in \mathbb{S}\}$ is a set of links. The traffic is represented as a set of flows $\mathbb{F}$, where each flow $f = (f.S, f.D, \lambda_f) \in \mathbb{F}$ consists of a source $f.S \in \mathbb{S}$, a destination $f.D \in \mathbb{S}$ and a bit rate $\lambda_f \in \mathbb{N}$.

The power consumption of a DCN is based on the SDN network equipment $\mathbb{S}$ and $\mathbb{E}$. Therefore, the Network Power Consumption (NPC) model is directly related to the number of active switches and the number of links. The computation formula for NPC is shown in (1).

$$\text{NPC} = P_{\text{switch}} \sum_{s_i \in \mathbb{S}} B_i + P_{\text{link}} \sum_{e_{ij} \in \mathbb{E}} L_{ij}. \tag{1}$$

B_i and L_{ij} denote the state of a corresponding switch and link, respectively, where the value 1 represents the active state, and 0 the passive state. The base power consumption of switches and links are denoted by P_{switch} and P_{link}.

Links and traffic correlation constraint: This constraint considers the correlation between the traffic volume and the links. Therefore, the constraint defines the relationship between the traffic volume T_{ij} and the link state L_{ij} to increase the utility of the link as much as possible.

Links and flows correlation constraint: This constraint represents the correlation between links and flows, such that a link should be active if and only if a flow passes through it.

Utility constraint: This constraint computes the utility of all the topology's links, and limits the link utility to less than or equal to the link's bandwidth BW_{ij}.

Path conservation constraint: This constraint installs the path from the source $f.S$ to the destination $f.D$ for each flow f.

Flow conservation constraint: This constraint guarantees for any flow f that the incoming and outgoing flows of the intermediate switches between the source $f.S$ and the destination $f.D$ should be equal, in order to avoid packet loss.

Network loop avoidance constraint: Since this model computes an acyclic graph in this context of routing, it is impossible to start at a switch s and to follow a directed path that returns to s. Thus, this constraint helps to avoid looping between switches.

3.2 ILP Formulation

This section describes how the DCN optimization model specified in Section 3.1 can be formulated as an ILP model, which computes the minimum number of links for a given traffic utilization, under the following conditions:

- The optimization model's parameters refer to a snapshot of the network state. This means that the model considers the case of the network state in a specific unit of time.
- The model starts with a standard multi-commodity flow problem. The constraints include flow conservation, link capacity, demand satisfaction, and the total number of active links.
- Splitting a single flow into packets across multiple links in the topology could save energy by increasing overall link utilization. However, due to varied path delays, reordered packets at the destination can degrade the performance. As

a result, we incorporate restrictions into our formulation based on the entire flow.

In this ILP model, all the variables are Boolean, i.e., they are restricted to the range $\{0, 1\}$. We will use the following Boolean variables:

- L_{ij} denotes if the link e_{ij} is active;
- $FR(f, i, j)$ denotes if the flow f passes through the link e_{ij}.

Additional integer constants are used in the model:

- BW_{ij} represents the bandwidth of the link e_{ij};
- T_{ij} represents the input traffic volume over the link e_{ij}.

The model in [14] employed only the second operand from (1), to minimize the number of the links as shown in (2).

$$\min\left(\sum_{i=1}^{n}\sum_{j=1}^{n} L_{ij}\right). \tag{2}$$

The above objective function works against the following constrains:

Links and traffic correlation constraint:

$$\frac{T_{ij}}{BW_{ij}} \leq L_{ij}, \quad \forall e_{ij} \in \mathbb{E}, \tag{3}$$

expressing that the traffic volume must not exceed the bandwidth of a link.

Links and flows correlation constraint:

$$FR\,(f, i, j) \leq L_{ij}, \quad \forall f \in \mathbb{F},\ \forall e_{ij} \in \mathbb{E}, \tag{4}$$

meaning that flows should pass only through active links.

Utility constraint:

$$\sum_{f \in \mathbb{F}} \big(FR\,(f, i, j) + FR\,(f, j, i)\big) \cdot \lambda_f \leq BW_{ij} - \mathrm{T}_{ij}, \quad \forall e_{ij} \in \mathbb{E}, \tag{5}$$

where a flow's packet rate is counted according to the undirected nature of the network graph.

Path conservation constraint:

$$\sum_{i=1}^{n} FR\,(f, f.S, i) = 1, \quad \sum_{i=1}^{n} FR\,(f, i, f.D) = 1, \quad \forall f \in \mathbb{F}. \tag{6}$$

Flow conservation constraint:

$$\sum_{\substack{i=1\\ i\neq f.S}}^{n} FR\,(f,i,j) = \sum_{\substack{i=1\\ i\neq f.D}}^{n} FR\,(f,j,i)\,, \quad \forall\; f \in \mathbb{F},\; j \in \mathbb{S}. \tag{7}$$

Network loop avoidance constraint:

$$FR\,(f,i,j) + FR\,(f,j,i) \leq 1, \quad \forall f \in \mathbb{F},\; i,j \in \mathbb{S}. \tag{8}$$

4 Implementation and Experiments

4.1 Benchmarks

The communication patterns affect performance and power consumption [9]. Based on the fact that the traffic in a data center swings between peak traffic (e.g., at daytime) and low traffic (e.g., at nighttime) [3], the traffic matrix for a DCN follows the sine wave in (9).

$$\text{Traffic Rate } = \frac{1}{2} \cdot \text{max rate} \cdot (1 + \sin(t)) \tag{9}$$

This paper explores three types of the sine-wave traffic matrix: near, long, and mixed (i.e., random). The benchmarks build upon what we described in Section 3.2. Each benchmark is a snapshot of the DCN at the time interval t_i , $1 \leq i \leq n$. This means that each benchmark captures the state of the traffic matrix at a specific time.

Near traffic pattern: The traffic is restricted between the servers that reside in the same PODs (Point of Delivery) of the topology, i.e., the servers that are connected through the edge layer switches only. The benchmarks aggregate the flows to a minimum number of links inside the same POD.

Long traffic pattern: The traffic is restricted between the servers that reside in different PODs of the topology, i.e., the servers that are connected through the edge, aggregation, and core layer switches. The model saves less power due to balancing the load between switches and using multiple paths to keep the QoS at an acceptable level, depending on the utility of links at that time.

Random traffic matrix: The traffic matrix for this pattern is a mixture of the above patterns, in order to explore how many links we can save with a random sine-wave pattern.

4.2 ILP Solving Tools

LINGO provides a collection of built-in solutions to handle a wide range of optimization problems. Unlike many modeling products, all of the LINGO solvers are directly connected to the modeling environment. Instead of using slower intermediary files, this seamless connection enables LINGO to transmit the issue to the right solver immediately in memory. This direct connection also reduces the compatibility issues between the solver and the modeling language components. LINGO is supported by LINDO Systems Inc. [10]. It is free and available for students and interested researchers.

Google's OR-Tools [15] is an open-source toolkit for solving optimization problems in general. Via the Python package `ortools`, one can access several optimization tools, including the following ones. Gurobi [6] provides one of the most powerful commercial solvers for a wide range of optimization problems, including ILP problems, and it is free to use for academics and students. CP-SAT[1] is a constraint programming solver that uses SAT methods, and it is part of the OR-Tools package. SCIP [4] is one of the fastest non-commercial solvers for Mixed Integer Programming (MIP) and, also, an open-source framework for constraint integer programming. CBC [5] (Coin-or Branch and Cut) is an open-source MIP solver. We will run Gurobi, CP-SAT, SCIP, and CBC from Python code inside our portfolio solver neO-DCN, as detailed in Section 4.3.

4.3 neO-DCN Portfolio Solver for DCN Optimization

The proposed ILP model has been implemented in our tool neO-DCN, which is a variant of our open-source tool neO [8] that we adapted to our DCN model. neO-DCN is publicly available at https://github.com/kovasz/neO-DCN.

neO-DCN is a portfolio solver, meaning that it can execute different ILP solvers, which were mentioned in Section 4.2, in parallel. The parallel execution is implemented by instantiating `ProcessPool` from the `pathos.multiprocessing` [12] Python module, which can run jobs with a non-blocking and unordered map.

The OR-Tools package provides two solver interfaces that we can use for ILP solving: (1) the `MPSolver` interface for MIP solvers such as Gurobi, SCIP and CBC, and (2) the `CPSolver` interface for Google's Constraint Programming solver CP-SAT. Both interfaces allow adding ILP constraints in the form

`solver.Add(`w_1 `*` x_1 `+ ... +` w_n `*` x_n `<=` c`)`

where each w_i and c is an integer, and x_i a Boolean variable. Note that although the interface allows to use relational operators other than $\leq$, neO-DCN translates all constraints to "AtMost" constraints for normalization purposes.

For neO-DCN, we introduced a JSON input format to read data about the configuration of the network as well as the current configuration of flows. The benchmark files that we used in our experiments apply this input format and can be found in the repository of neO-DCN.

[1] https://developers.google.com/optimization/cp/cp_solver

4.4 Experimental Results

In our experiments, we are dealing with the real-world DCN topology same DCN topology as in [14]. Figure 2 shows the topology containing 20 switches and 16 hosts, and numerous links between those nodes. The bandwidth of each link is uniformly set to 1 Gbps.

The ILP solvers that we mentioned in Section 4.2 were run on the benchmark instances with a wall clock time limit of 1200 seconds. LINGO was run as a Windows desktop application, while the other ILP solvers as part of NEO-DCN on Linux. During the experiments, we measured the runtime of the solvers and, also, monitored the memory consumption of solvers by using the memory profiler `mprof`. While applying `mprof` to NEO-DCN was successful, we were not able to apply it to LINGO. This is why we will not provide memory consumption data for LINGO in the subsequent sections.

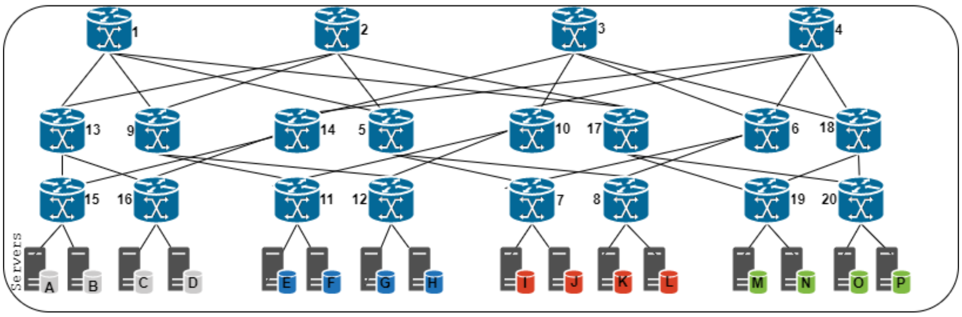


Figure 2: DCN topology (fat tree) for experiments.

4.4.1 Near Traffic Pattern

All mentioned solvers converge to the optimal solution for this traffic pattern very fast. They reduce the topology in Figure 2 to a minimum-tree DCN topology in Figure 3 by setting all unneeded links to off-state. Figure 3 shows the four scenarios of the near sine-wave pattern. In the first scenario, we burst roughly 1 Gbps distributed over 20 flows from servers A, B to C, D, i.e, unidirectional traffic. The optimizer aggregates all flows in six links. On the other hand, in the second scenario, the number of flows are increased to 30 and the traffic to 1.3 Gbps. The topology changes because the model balances the traffic over the links, and the number of links becomes 8 instead of 6. In the third scenario, part of the traffic is bidirectional, while we keep the traffic volume at 1.3 Gbps. We burst the same number of flows, 30, distributed as follows: (1) 10 flows from servers A to C, (2) 10 flows from B to D, and (3) 10 flows in a reverse direction, from server C to A. The optimizers output the minimum number 10 of the 12 links for the sake of aggregating as many flows as possible in one path. In the fourth scenario, we keep all the characteristics of the third scenario, except that we increase the traffic volume to roughly 2 Gbps by sending new traffic from server D to B. As a result, the minimum number of active links becomes 40.

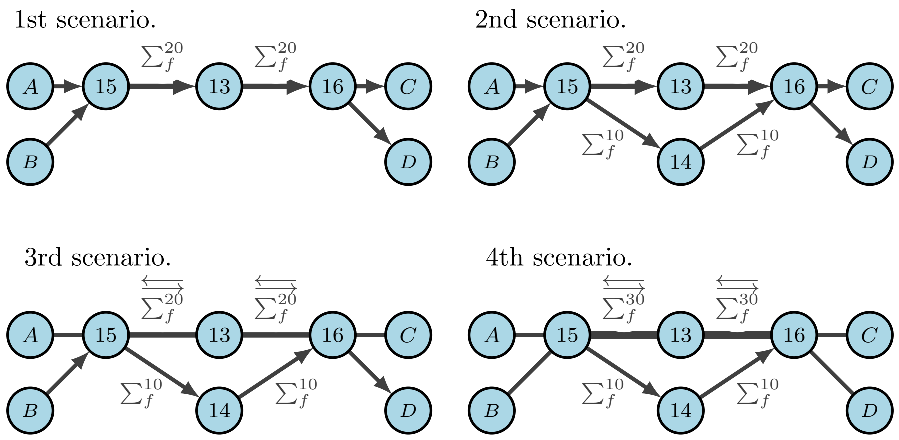


Figure 3: Four near-traffic scenarios, where directed links indicate flow direction. Undirected links represent bidirectional flows.

Table 1 shows some of the results reported by the ILP solvers, including the optimum value (e.g., minimum number of active links). In the table, the runtime of each solver is given in seconds. The results show that the benchmark instances in all scenarios can be solved in reasonable time by any of the solvers. However, Gurobi and CP-SAT outperform the other solvers by almost 1 order of magnitude on this benchmark.

Table 1: Runtimes (s) of ILP solvers for near traffic pattern

Scenario	**Flows**	**Opt.**	LINGO	Gurobi	CP-SAT	SCIP	CBC
1	20	6	2.11	0.44	0.42	0.63	0.53
2	30	8	4.21	0.53	0.53	0.73	3.03
3	30	10	3.21	0.53	0.52	0.93	0.62
4	40	14	5.01	0.73	0.72	1.33	3.43

4.4.2 Long Traffic Pattern

In this benchmark, we separate the servers from different PODs into two groups: sender and receiver. Besides that, we set the number of flows to a constant value of 24. Then, we gradually increase the traffic volume roughly from 1 Gbps to 5 Gpbs, which were captured in 14 benchmark instances, and we burst them into the DCN. The idea behind this benchmark is to demonstrate how the subset of active links changes according to the traffic demand when the number of flows is constant.

Table 2 shows the traffic volume in Gbps for each benchmark instances, and the corresponding optimum values (e.g., minimum number of active links).

While 6 active links are sufficient to use for the initial time interval (representing low traffic), one needs to activate all the 48 links in the topology for the last time interval (representing high traffic). In the table, one can definitely see higher runtimes than those in the near-traffic experiment. LINGO, SCIP, and CBC even timed out (TO) for some of the benchmark instances, due to the higher level of difficulty of solving, caused by a high load of traffic.

Table 2: Runtimes (s) of ILP solvers for long traffic pattern

Time Interval	**Traffic**	**Opt.**	LINGO	Gurobi	CP-SAT	SCIP	CBC
1	0.91	6	2.81	0.53	0.42	0.62	0.63
2	0.92	12	2.74	0.43	0.43	0.62	0.63
3	1.5	18	2.93	0.53	1.93	0.62	1.73
4	2.15	26	2.96	0.53	28.88	1.32	3.13
5	2.3	28	2.97	0.53	22.57	25.28	19.56
6	2.4	30	2.85	0.53	17.45	54.63	5.53
7	2.7	32	2.73	0.53	12.25	14.76	11.35
8	3	36	27.73	1.74	16.76	146.02	101.21
9	3.4	36	10.83	2.44	11.35	29.31	80.16
10	3.6	40	597.18	141.96	17.69	122.61	406.6
11	3.9	40	566.44	162.26	13.05	951.43	TO
12	4.2	44	TO	425.5	36.6	291.98	TO
13	4.5	44	TO	144.51	28.28	TO	TO
14	4.8	48	TO	575.49	45.01	TO	TO

The solvers' runtimes are visualized in Figure 4. Notice that the vertical axis, that represents the runtimes in seconds, is log-scaled. Up to the 7th time interval, when the traffic volume is 2.7 Gbps, all the solvers can find the optimum in a short time and, in particular, Gurobi and LINGO seem to provide a stable performance. For a higher volume of traffic, however, all the solvers loose efficiency very rapidly, except for CP-SAT, which keeps a surprisingly stable performance all the way to the very last time interval.

Figure 5 visualizes the memory consumption of the different ILP solvers. Recall that we could not apply memory profiling to LINGO. As the chart shows, all the solvers consume a moderate amount of memory for each benchmark instance. Most importantly, for CP-SAT, which was proved to be the fastest solver on this benchmark, memory consumption seems to be constant-like.

4.4.3 Random Traffic Matrix

In the benchmark with random traffic matrix, we inject different burst sizes and random flows into the DCN. The generated benchmark instances consist of 5, 10,

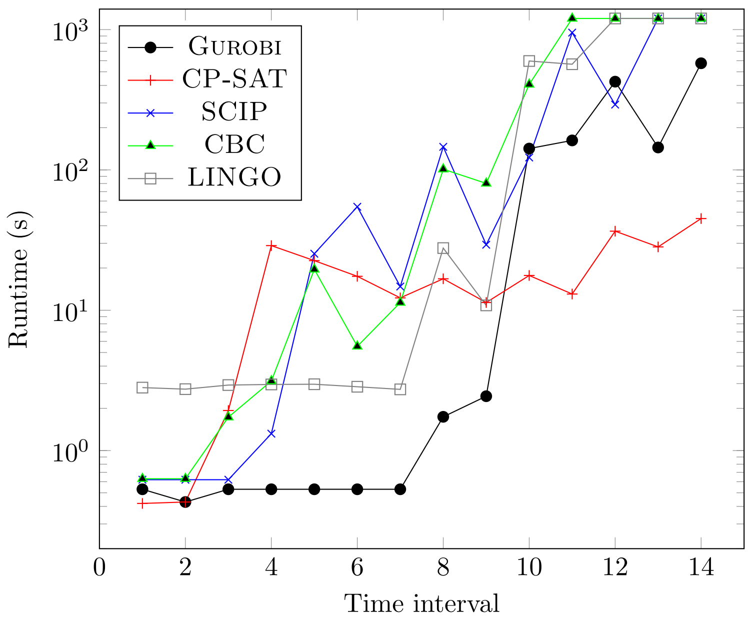


Figure 4: Runtimes of ILP solvers for long traffic pattern.

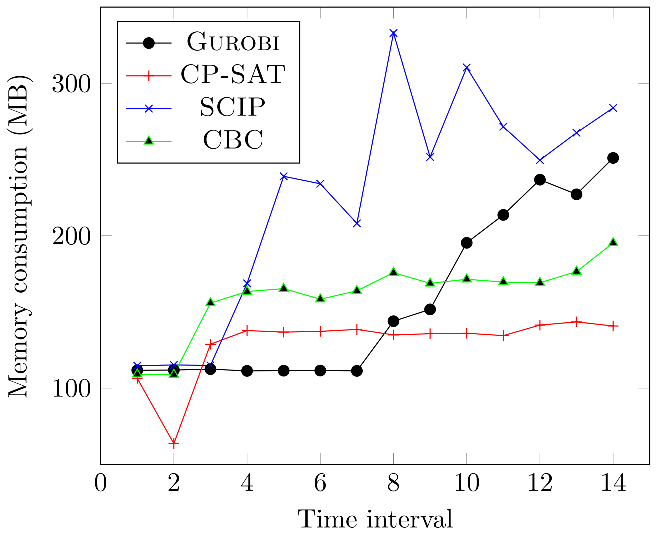


Figure 5: Memory consumption of ILP solvers for long traffic pattern.

20, . . . , 200 flows, respectively, where each flow f is given by a random source host $f.S$, a random destination hosts $f.D$ and a random packet rate λ_f.

Table 3 gives several details for each benchmark instance. The number of flows gradually increases from 5 to 200. Note that benchmark instances up to 110 flows are satisfiable (SAT), while the ones above that are unsatisfiable (UNSAT), meaning that there does not exist any solution for them. For the SAT instances, the table shows the optimum values (e.g., minimum number of active links).

Table 3: Runtimes (s) of ILP solvers for random traffic

Flows	**Traffic**	**Result**	**Opt.**	LINGO	GUROBI	CP-SAT	SCIP	CBC
5	0.08	SAT	15	0.79	0.24	0.24	0.22	0.24
10	0.21	SAT	19	1.34	0.23	0.33	0.32	0.44
20	0.49	SAT	24	2.22	0.33	0.63	0.53	0.74
30	0.61	SAT	28	3.22	0.53	1.23	0.83	0.83
40	0.92	SAT	28	4.37	0.63	1.63	1.13	1.14
50	1.06	SAT	29	16.89	1.13	3.13	25.68	11.75
60	1.35	SAT	29	12.38	1.54	5.54	7.74	158.31
70	1.35	SAT	31	49	4.54	22.47	185.03	206.83
80	1.67	SAT	30	45.53	3.85	62.25	97.99	158.70
90	1.86	SAT	33	452.84	32.2	83.36	TO	TO
100	2.42	SAT	32	219.28	6.85	45.02	1160.16	329.72
110	2.25	SAT	33	275.24	34.92	TO	TO	481.98
120	2.71	UNSAT		TO	1.93	TO	7.44	42.51
130	8	UNSAT		TO	2.14	TO	8.25	41.3
140	7.71	UNSAT		TO	2.24	TO	10.25	69.56
150	7.84	UNSAT		TO	2.34	TO	9.15	78.88
160	8	UNSAT		TO	2.64	TO	7.44	63.04
170	8	UNSAT		TO	2.83	TO	10.55	116.14
180	7.81	UNSAT		TO	3.04	TO	9.35	136.88
190	7.99	UNSAT		TO	3.14	TO	16.16	87.49
200	8	UNSAT		TO	4.65	TO	13.66	115.13

Table 3 and Figure 6 show the solvers' runtimes for each benchmark instance. Notice that the vertical axis of the chart is log-scaled.

Only GUROBI is able to solve all the benchmark instances under the time limit. LINGO and CP-SAT times out on all the UNSAT instances, while all the other solvers are able to recognize the UNSAT case in quite a reasonable timeframe. CBC and SCIP time out on 1 and 2 SAT instances, respectively, which consist of a high number of flows.

Regarding runtime, GUROBI outperforms all the other solvers by 1 or 2 orders of magnitude, especially when comparing to LINGO that was used as an underlying solver in [14].

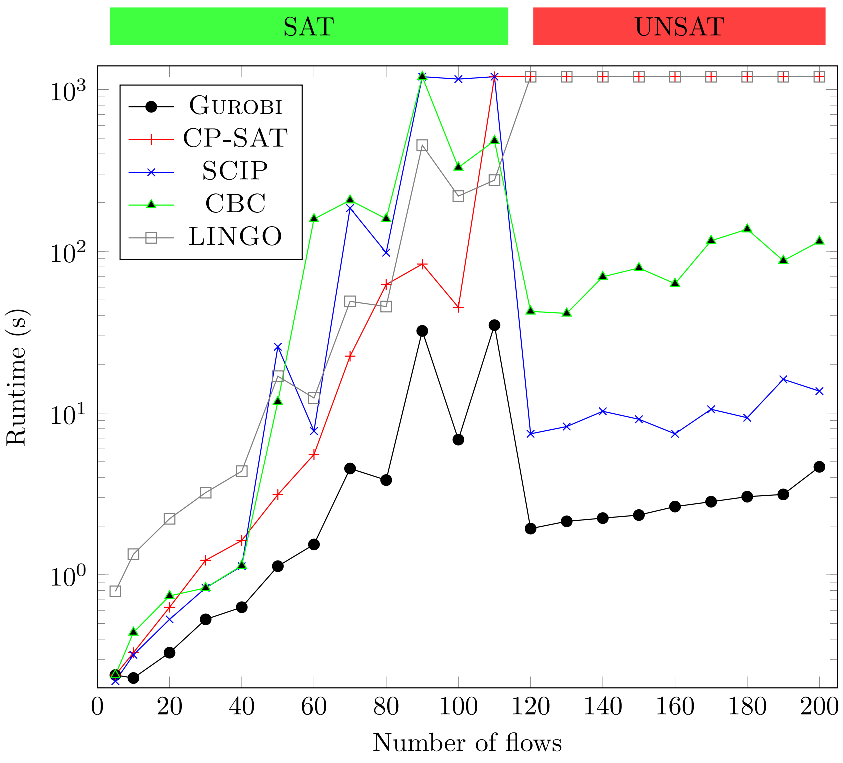


Figure 6: Runtimes of ILP solvers for random traffic.

The memory consumption we have recorded is shown in Figure 7. Gurobi, as the fastest solvers for the current benchmark, consumes a moderate amount of memory.

5 Conclusion and Future work

With the aim of optimizing the power consumption of a real DCN topology called the fat tree, we proposed an ILP model in our previous paper [14] and reported on experiments with the optimization toolkit LINGO. For our current paper, we have implemented the same model for other ILP solvers. We report on comparative experiments with them on a wide range of traffic benchmarks for three different communication patterns: (1) the near traffic pattern results show that Gurobi and CP-SAT outperform the other solvers regarding runtime for most traffic instances; (2) the long traffic pattern results show that above a traffic volume of 2.7 Gbps all the solvers dramatically loose efficiency, except for CP-SAT, which keeps a good performance and roughly constant memory consumption; (3) the random

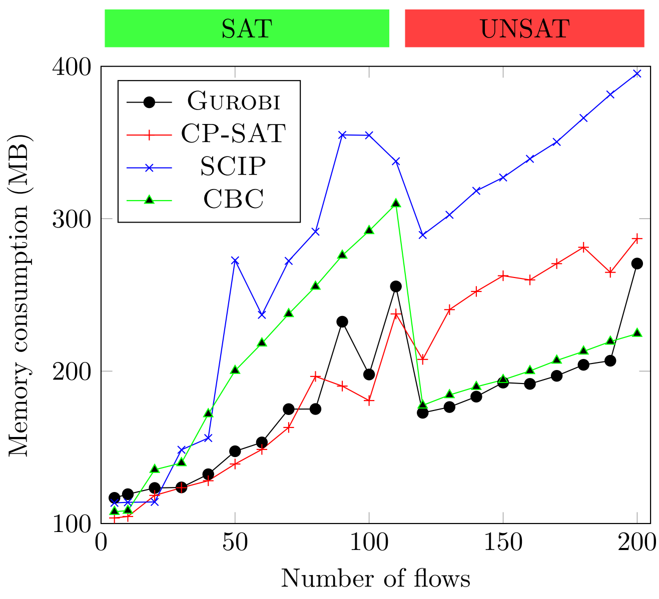


Figure 7: Memory consumption of ILP solvers for random traffic.

traffic results show that, for most of the traffic instances, GUROBI outperforms the other solvers regarding both runtime and memory. We can conclude that, for most of the benchmark instances, most of the solvers outperform LINGO regarding runtime. Consequently, it was definitely worth experimenting with those solvers, with GUROBI and CP-SAT in particular, as part of our new contribution.

As future work, it would be worth investigating how much ILP solvers scale for certain generalizations of the DCN model, such as using heterogeneous power consumption values for switches and links. In an ongoing work, we are upgrading the ILP model to save as much power as possible by adding more parameters, such as flow type [13]. Additionally, we are planning to experiment with pseudo-Boolean solvers as well.

References

[1] Al-Tarazi, M. and Chang, J. M. Performance-aware energy saving for data center networks. *IEEE Transactions on Network and Service Management*, 16(1):206–219, 2019. DOI: 10.1109/TNSM.2019.2891826.

[2] Assefa, B. G. and Ozkasap, O. Framework for traffic proportional energy efficiency in software defined networks. In *Proceedings of the IEEE International Black Sea Conference on Communications and Networking (BlackSeaCom)*, pages 1–5. IEEE, 2018. DOI: 10.1109/BlackSeaCom.2018.8433618.

[3] Assefa, B. G. and Özkasap, Ö. A survey of energy efficiency in SDN: Software-based methods and optimization models. *Journal of Network and Computer Applications*, 137:127–143, 2019. DOI: 10.1016/j.jnca.2019.04.001.

[4] Bestuzheva, K. et al. The SCIP Optimization Suite 8.0. ZIB-Report 21-41, Zuse Institute Berlin, 2021. URL: http://nbn-resolving.de/urn:nbn:de:0297-zib-85309.

[5] Forrest, J. et al. coin-or/cbc: Release releases/2.10.7, 2022. DOI: 10.5281/zenodo.5904374.

[6] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual. URL: https://www.gurobi.com/documentation/9.5/refman/, 2022.

[7] Heller, B., Seetharaman, S., Mahadevan, P., Yiakoumis, Y., Sharma, P., Banerjee, S., and McKeown, N. Elastictree: Saving energy in data center networks. In *NSDI'10: Proceedings of the 7th USENIX conference on Networked systems design and implementation*, Volume 10, pages 249–264, 2010.

[8] Kovásznai, G., Gajdár, K., and Kovács, L. Portfolio SAT and SMT solving of cardinality constraints in sensor network optimization. In *Proceedings of the 21st International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, pages 85–91. IEEE, 2019. DOI: 10.1109/SYNASC49474.2019.00021.

[9] Li, X., Lung, C.-H., and Majumdar, S. Green spine switch management for datacenter networks. *Journal of Cloud Computing*, 5(1):1–19, 2016. DOI: 10.1186/s13677-016-0058-8.

[10] LINDO Systems Inc. Lingo the modeling language and optimizer. URL: http://www.lindo.com, 2020.

[11] Luo, J., Zhang, S., Yin, L., and Guo, Y. Dynamic flow scheduling for power optimization of data center networks. In *Proceedings of the Fifth International Conference on Advanced Cloud and Big Data (CBD)*, pages 57–62. IEEE, 2017. DOI: 10.1109/CBD.2017.18.

[12] McKerns, M. M., Strand, L., Sullivan, T., Fang, A., and Aivazis, M. A. Building a framework for predictive science. *arXiv preprint arXiv:1202.1056*, 2012. DOI: 10.48550/arXiv.1202.1056.

[13] Nsaif, M., Kovásznai, G., Abboosh, M., Malik, A., and Fréin, R. d. ML-based online traffic classification for SDNs. In *Proceedings of the IEEE 2nd Conference on Information Technology and Data Science (CITDS)*, pages 217–222, 2022. DOI: 10.1109/CITDS54976.2022.9914138.

[14] Nsaif, M., Kovásznai, G., Rácz, A., Malik, A., and de Fréin, R. An adaptive routing framework for efficient power consumption in software-defined datacenter networks. *Electronics*, 10(23), 2021. DOI: 10.3390/electronics10233027.

[15] Perron, L. and Furnon, V. OR-Tools. URL: https://developers.google.com/optimization/, 2019.

[16] Rabee, F., Al-Haboobi, A., and Nsaif, M. R. Parallel three-way handshaking route in mobile crowd sensing (PT-MCS). *Journal of Engineering and Applied Sciences*, 14:3200–3209, 2019. DOI: 10.36478/jeasci.2019.3200.3209.

[17] Rabee, F., Nsaif, M., and Al-Haboobi, A. Reliable compression route protocol for mobile crowd sensing (RCR-MSC). *Journal of Communications*, 14:170–178, 2019. DOI: 10.12720/jcm.14.3.170-178.

[18] Wang, R., Jiang, Z., Gao, S., Yang, W., Xia, Y., and Zhu, M. Energy-aware routing algorithms in software-defined networks. In *Proceedings of IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks 2014*, pages 1–6. IEEE, 2014. DOI: 10.1109/WoWMoM.2014.6918982.

[19] Xu, G., Dai, B., Huang, B., Yang, J., and Wen, S. Bandwidth-aware energy efficient flow scheduling with SDN in data center networks. *Future Generation Computer Systems*, 68:163–174, 2017. DOI: 10.1016/j.future.2016.08.024.

Extracting Line Parameters of Woven Wire Mesh in Images under Directional Illumination*

László Körmöczi[ab] and László G. Nyúl[ac]

Abstract

Localizing the wires of a mesh in an image is important in various image processing applications. This task can be difficult if the wires cannot be detected with simple line detectors, e.g. if corrugated wires of a woven mesh appear as dark and bright segments under directional illumination. Template matching is insufficient if the appearance of the wires varies throughout the image, depending on the viewing angle, and neural networks require computationally expensive training on a well-prepared dataset. We propose an efficient way to extract the line parameters (position and orientation) of the wires of a regular mesh from an image by finding meaningful local minima of a cost function, followed by RANSAC-controlled robust outlier filtering.

Keywords: image processing, wire detection, cage mesh detection, line parameter extraction

1 Introduction

Detecting wire mesh in an image and extracting the line parameters (i.e. the location and orientation of the wires' projection in the image) is an important task in various image processing applications. Localising the wires can help in 2D-3D camera pose estimation, or for inpainting. This is useful when the relative pose of the camera and the wire mesh (e.g. an animal cage) can change and has to be known throughout a series of images or a video stream. In such applications, if a wire mesh (e.g. the front mesh of the cage) is visible in the image, it can be used for reliable, unsupervised camera pose estimation if a subject (e.g. rodent) is to be localized in a coordinate system in which the mesh is fixed.

A mesh made of straight wires can easily be detected with line detectors, e.g. using Canny edge detector [1] followed by Hough transform [2]. Detection of thick

*Project no. TKP2021-NVA-09 has been implemented with the support provided by the Ministry of Innovation and Technology of Hungary from the National Research, Development and Innovation Fund, financed under the TKP2021-NVA funding scheme.

[a]Department of Image Processing and Computer Graphics, University of Szeged, Hungary

[b]E-mail: kormoczi@inf.u-szeged.hu, ORCID: 0000-0002-3833-0609

[c]E-mail: nyul@inf.u-szeged.hu, ORCID: 0000-0002-3826-543X

DOI: 10.14232/actacyb.299046

lines can be achieved by either downscaling the image or by using thick line detectors [8, 3]. However, these methods fail when the wires are not seen as lines in the image, like the wires of a welded corrugated mesh that appear as wavy lines with alternating darker and brighter segments due to directional reflection of their material.

Different parts of a mesh of corrugated wires are seen from different angles, as in the examples in Fig. 1, thus have different shape throughout the image, so template matching cannot be used.

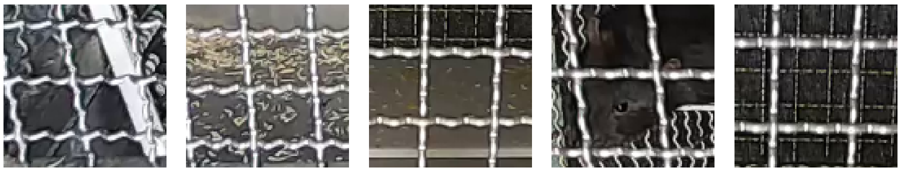


Figure 1: Examples of corrugated wires seen from different angles.

Quasi-periodic wire meshes can be detected in the frequency domain [5]. There are methods that use machine learning and neural networks to detect and segment meshes [10], but training neural networks requires a lot of carefully prepared training data (e.g. precisely segmented samples) and is computationally expensive.

Wire mesh detection can be easily achieved if there is camera motion between frames [9] or the focusing distance of the lens can be varied [12], but having a fixed-focus camera in a fixed position requires a different approach.

If the wire mesh fills a large part of the image, we can find "bands" that can be macroscopically recognised by an average intensity, although having a large local variation. The regularity of the mesh (i.e. the distance between wires is constant and the mesh is a rectangular grid) can be utilized without the need to rectify the image. In this work, we extend the procedure described in [7] and show quantitative results on test images.

Detection and localisation of a wire mesh is needed in several image processing applications. In medical experiments, rodents are often used as models for human diseases. For behavior analysis, the animals are placed in a cage and observed with cameras outside the cage [6]. Localisation in the image is possible but the subject's location is of interest in the cage coordinate system. Reliable pose estimation of the camera with respect to the cage is needed and can be achieved using the front wire mesh of the cage.

2 Line parameter extraction for wires

In order to find the line parameters of the wires' projection in the image, we calculate a cost function in a 2-dimensional parameter space, then find strong local minima (negative spikes) of the cost function as candidates for detected lines, and finally apply a robust filtering on the candidates.

2.1 Preprocessing

The algorithm can find projections of wires in a grayscale image that fit in a rectangle ("band") of arbitrary position and orientation. If the source image suffers from lens distortion (e.g. barrel or pincushion distortion), an undistortion step has to be performed. Perspective effect does not have to be compensated, since straight lines remain straight after perspective projection.

The image should be normalized so that the pixel values are in the $[0, 1]$ range.

2.2 Cost function calculation

First, an average intensity value $\overline{I}$ is to be defined that macroscopically represents the wires in the image, along with a band width w that covers a wire. We compute the absolute difference of pixel intensity and $\overline{I}$ for each pixel of the image I as:

$$J(x, y) = |I(x, y) - \overline{I}|$$

Then the image space of J is transformed into a (ρ, θ) parameter space with a cost function, where ρ denotes the distance of a line from the image center and θ denotes the rotation of the line. The calculation is performed separately for finding the projection of vertical and horizontal wires, and θ represents the deviation from the vertical or horizontal direction, respectively.

The cost function C is calculated for (ρ, θ) pairs, ρ ranging from $\rho_{\min}$ to $\rho_{\max}$ with step size ρ_{step} and θ ranging from $\theta_{\min}$ to $\theta_{\max}$ with step size θ_{step}. The $C(\rho, \theta)$ value of C at given ρ and θ is calculated as the sum of the intensity of the pixels of J that are covered by a w wide band around a line that is at ρ distance from the image center, rotated with θ (denoted as $B_{w,\rho,\theta}$), divided by the area of the image covered by the band ($|B_{w,\rho,\theta}|$):

$$C(\rho, \theta) = \frac{\sum\limits_{(x,y) \in B_{w,\rho,\theta}} J(x, y)}{|B_{w,\rho,\theta}|}$$

In this approach, the possible values of the cost function lie within the $[0, 1]$ range, 0 represents a band that has $\overline{I}$ intensity in each pixel. Furthermore, we do not consider bands that cover less than half of the area of a vertical band (for the vertical case) or a horizontal band (for the horizontal case). The function value of such points of the parameter space are set to 1.

2.3 Finding candidates for lines

Good candidates are negative peaks of C. A negative peak can be defined as being a local minimum in its (large enough) neighbourhood and deeper than a threshold compared to its neighbouring baseline.

As C is calculated in a given finite range with a finite step size for both parameters, this ordered set of calculated values can be treated as an image, with axes ρ and θ, and the pixel intensities are values of C at given ρ and θ values.

We declare the neighbourhood in which the local minimum search is performed as a rectangle $\mathcal{N}_{d\rho,d\theta}$ with axes along the parameter space dimensions. For $\mathcal{N}_{d\rho,d\theta}(\rho,\theta)$, the maximal distance between the (ρ,θ) point and any point in $\mathcal{N}_{d\rho,d\theta}(\rho,\theta)$ is $d\rho$ and $d\theta$ along the two dimensions:

$$\mathcal{N}_{d\rho,d\theta}(\rho,\theta) = \{\, (\rho',\theta') \mid |\rho' - \rho| \le d\rho \wedge |\theta' - \theta| \le d\theta \,\}$$

To fulfill the aforementioned two criteria (i.e. having local minimum and sufficiently small value compared to a baseline), we choose (ρ,θ) pairs by two conditions. We select those in a set $\mathcal{L}$ that have local minima in $\mathcal{N}_{d\rho,d\theta}$ neighbourhood:

$$\mathcal{L} = \{\, (\rho,\theta) \mid C(\rho,\theta) = \min_{(\rho',\theta') \in \mathcal{N}_{d\rho,d\theta}(\rho,\theta)} C(\rho',\theta') \,\}$$

We independently perform a bottom-hat (black-hat or black top-hat) transform on C treated as an image (as described above) with a rectangle as structuring element defined by $\mathcal{N}_{d\rho,d\theta}$, followed by thresholding. Bottom-hat transform performs closing on an image, then subtracts the original image from the closed image, thus extracts the baseline and transforms negative peaks into positive [11]. For simplicity, denote the structuring element by $\mathcal{N}$ and the image representation of C values by C. Let C' be the result of the bottom-hat transform:

$$C' = (C \bullet \mathcal{N}) - C = ((C \oplus \mathcal{N}) \ominus \mathcal{N}) - C$$

(where $\bullet$ denotes closing, $\oplus$ is dilation and $\ominus$ is erosion).

Thresholding (with a c_t threshold) applied to the result of the bottom-hat transform selects (ρ,θ) pairs in a set $\mathcal{V}$ that are part of a valley:

$$\mathcal{V} = \{\, (\rho,\theta) \mid C'(\rho,\theta) > c_t \,\}$$

Points of the parameter space that fulfill both criteria are selected as candidates in a set $\mathcal{P}$:

$$\mathcal{P} = \mathcal{L} \cap \mathcal{V}$$

2.4 Robust filtering

As there is only a perspective projection present in the image after correcting for optical distortion, and a regular mesh consists of two (usually perpendicular) sets of parallel wires that are to be detected independently, the lines for the image of the parallel wires are either parallel or intersect at a vanishing point $V(u,v)$. For lines that intersect at $V(u,v)$, the following is true for every line, if ρ denotes the distance of a line from the image center and θ denotes the rotation of the line:

$$\rho = u\cos\theta + v\sin\theta$$

If a vanishing point exists for a set of lines, solving a linear system for two (ρ,θ) value pairs referring to two of these lines gives an explicit result, except in the case when $|\theta_1 - \theta_2| = 180°$ (where θ_1 and θ_2 belong to the first and second selected line, respectively) and the equation system has infinite solutions. In this configuration, the two lines coincide and the vanishing point can be any point on that line.

If the lines in the image are parallel, there is no solution for the equation system, as there is no vanishing point and the (ρ, θ) pairs representing these lines lie on a line in the parameter space.

Although the above function is highly non-linear, for many practical applications, when the vanishing point is outside the image, the (ρ, θ) points representing the visible lines in the image fit on a line with a small tolerance. Line fitting also works for those configurations when the equation system does not have one exact solution.

We use RANSAC [4] to fit a line on the candidates. RANSAC is widely used in numerous applications for fast and robust selection of inliers, because it can reliably filter out outliers and is robust for low inlier ratio. We assume that outliers do not fit another line (e.g. no other strongly visible grid-like structure is present in the image).

Algorithm 1 Line parameter extraction for wires

```
Input: $\overline{I}$ average intensity of the wires in the image
  $w$ band width
  $I$ grayscale image
  $H, W$ height and width of the image
  $\theta_{\min}$, $\theta_{\max}$, $\theta_{\text{step}}$ as minimal, maximal rotation angle and rotation step size
  $\rho_{\min}, \rho_{\max}, \rho_{\text{step}}$ as minimal and maximal signed distance from image center and distance step size
Output: $\rho$, $\theta$ line parameters of the wires in the image
  for $x \in [0, W]$, $y \in [0, H]$ do
      $J(x, y) = |I(x, y) - \overline{I}|$
  end for
  for $\theta \in [\theta_{\min}, \theta_{\max}]$ with $\theta_{\text{step}}$ step size do
      for $\rho \in [\rho_{\min}, \rho_{\max}]$ with $\rho_{\text{step}}$ step size do
          Let $B_{w,\rho,\theta}$ be a $w$ pixel wide band that is at $\rho$ distance from the image center and rotated with $\theta$
          $C(\rho, \theta) = \frac{\sum_{(x,y) \in B_{w,\rho,\theta}} J(x,y)}{|B_{w,\rho,\theta}|}$
      end for
  end for
  Search for candidates with $\mathcal{N}_{d\rho,d\theta}$ neighbourhood:
  $\mathcal{L} = \{ (\rho, \theta) \mid C(\rho, \theta) = \min_{(\rho',\theta') \in \mathcal{N}_{d\rho,d\theta}(\rho,\theta)} C(\rho', \theta') \}$
  $\mathcal{V} = \{ (\rho, \theta) \mid C'(\rho, \theta) > c_t \}$, where $C' = (C \bullet \mathcal{N}) - C = ((C \oplus \mathcal{N}) \ominus \mathcal{N}) - C$
  $\mathcal{P} = \mathcal{L} \cap \mathcal{V}$
  Line fitting and outlier filtering with RANSAC
```

3 Experimental results

In the experiments described in [6] we are working with images of 60×60×60 cm rodent home cages acquired with consumer grade IP cameras having 1/2.7" class sensors and wide angle lens with 3.6 mm focal length. The resolution of the images is 1920×1080 px and the cage almost fills the whole image. Each cage is observed by two cameras in a non-standard vertical stereo configuration, facing the front of the cage. The angle between the cameras' optical axis and the axis perpendicular to the front wire mesh is around 20-30°. The cages are placed on a movable platform so that the relative pose of the cameras and the cage can change. The cameras are mounted on rotatable heads, thus the stereo configuration can also change. The cameras have 83° vertical and 44° horizontal angle of view and a strong barrel distortion can be observed in the images. We calibrated our cameras and undistorted the images so that barrel distortion is eliminated. Fig. 2 shows that only a perspective effect remains.

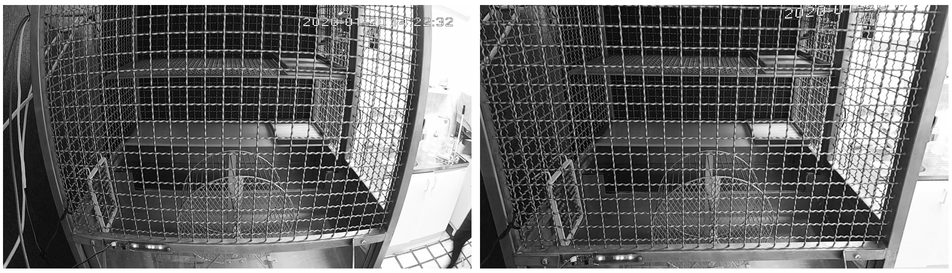


Figure 2: Sample original (left) and undistorted (right) image showing the rodent home cage

The home cages have a skeleton of square steel tubes and a mesh of corrugated wires is welded to the inner side of the tubes.

We ran the proposed algorithm on 25 images, 13 taken in daylight conditions with artificial directional illumination and 12 at night, with the cameras set to night vision mode and inbuilt infrared LEDs illuminated the scene. The expected width w was set to 9 px and we observed the effect of varying $\overline{I}$ in the range from 0 to 1.0 with 0.05 step size. The resolution of the parameter space was 1 px for ρ and 0.2° for θ, ρ swept through the image in both dimensions and θ was limited between -15° and 15°. $d\rho$ was set to 40 px and $d\theta$ was set to 8° for $\mathcal{N}_{d\rho,d\theta}$ used in Section 2.3. Tolerance in RANSAC filtering for a candidate to fit on a line was set to 2° and 10 px.

Filtered candidates (predicted positives) were compared against ground truth values that were computed from manually assigned lines for each wire in the image. A predicted positive is considered true positive if θ error is no more than 1° and ρ error is no more than 5 px.

Precision, recall and F_1 score (average ± standard deviation) are shown in Tables 1 and 2, for vertical and horizontal lines, respectively. Although the test images originate from 4 cameras, we aggregated the results because the conditions were similar for each camera. With $\overline{I} < 0.6$ there were no correctly detected wires in most images, so we excluded these results from the table.

We can see from the results that the algorithm is robust for a large variation of $\overline{I}$, and it can detect almost all wires of the mesh if $\overline{I}$ is set correctly. The total number of vertical lines to be detected was 24, and the number of horizontal lines to be detected was 12 or 14 in most cases. Results for the horizontal wires are better, as some of the vertical wires near the side of the cage were not detected correctly due to the background and the side meshes. False positives generally come from the skeleton of the cages.

Table 1: Precision, recall and F_1 score for finding vertical lines.

	daylight			night		
$\overline{I}$	precision (avg ± std)	recall (avg ± std)	F_1 score (avg ± std)	precision (avg ± std)	recall (avg ± std)	F_1 score (avg ± std)
0.60	0.32 ± 0.03	0.48 ± 0.06	0.38 ± 0.04	0.25 ± 0.05	0.42 ± 0.07	0.31 ± 0.06
0.65	0.51 ± 0.04	0.66 ± 0.06	0.58 ± 0.04	0.35 ± 0.06	0.48 ± 0.08	0.41 ± 0.07
0.70	0.67 ± 0.06	0.76 ± 0.03	0.71 ± 0.03	0.48 ± 0.06	0.56 ± 0.05	0.51 ± 0.06
0.75	0.76 ± 0.05	0.81 ± 0.04	0.78 ± 0.04	0.61 ± 0.05	0.69 ± 0.05	0.65 ± 0.05
0.80	0.80 ± 0.04	0.82 ± 0.03	0.81 ± 0.03	0.65 ± 0.09	0.72 ± 0.06	0.69 ± 0.08
0.85	0.85 ± 0.04	0.84 ± 0.01	0.84 ± 0.03	0.68 ± 0.06	0.76 ± 0.05	0.72 ± 0.06
0.90	0.83 ± 0.03	0.82 ± 0.01	0.83 ± 0.01	0.69 ± 0.06	0.77 ± 0.05	0.73 ± 0.06
0.95	0.82 ± 0.04	0.82 ± 0.01	0.82 ± 0.02	0.70 ± 0.07	0.78 ± 0.05	0.74 ± 0.06
1.00	0.82 ± 0.05	0.81 ± 0.02	0.81 ± 0.03	0.70 ± 0.06	0.78 ± 0.04	0.74 ± 0.05

Table 2: Precision, recall and F_1 score for finding horizontal lines.

	daylight			night		
$\overline{I}$	precision (avg ± std)	recall (avg ± std)	F_1 score (avg ± std)	precision (avg ± std)	recall (avg ± std)	F_1 score (avg ± std)
0.60	0.46 ± 0.06	0.70 ± 0.03	0.55 ± 0.04	0.38 ± 0.05	0.83 ± 0.12	0.52 ± 0.07
0.65	0.53 ± 0.03	0.71 ± 0.01	0.61 ± 0.03	0.47 ± 0.03	0.87 ± 0.09	0.61 ± 0.04
0.70	0.62 ± 0.08	0.81 ± 0.06	0.71 ± 0.07	0.62 ± 0.07	0.89 ± 0.09	0.73 ± 0.06
0.75	0.77 ± 0.09	0.89 ± 0.04	0.83 ± 0.07	0.84 ± 0.09	0.94 ± 0.05	0.88 ± 0.05
0.80	0.84 ± 0.12	0.90 ± 0.04	0.87 ± 0.08	0.92 ± 0.11	0.92 ± 0.06	0.91 ± 0.06
0.85	0.88 ± 0.07	0.90 ± 0.03	0.89 ± 0.03	0.94 ± 0.07	0.92 ± 0.06	0.93 ± 0.03
0.90	0.90 ± 0.03	0.90 ± 0.03	0.90 ± 0.02	0.96 ± 0.07	0.91 ± 0.06	0.93 ± 0.01
0.95	0.93 ± 0.00	0.90 ± 0.03	0.92 ± 0.02	0.97 ± 0.04	0.90 ± 0.06	0.94 ± 0.02
1.00	0.93 ± 0.00	0.90 ± 0.03	0.92 ± 0.02	0.97 ± 0.04	0.89 ± 0.05	0.93 ± 0.01

An example visualization can be seen in Figs. 3 and 4. Fig. 3 shows the values of the cost function over the selected (ρ, θ) range for horizontal lines of a test image, with $\overline{I} = 0.95$. Brighter color means higher value, 0 is black and 1 is white. Negative peaks can be observed as dark spots. After filtering the candidates, inliers that fit on a line are marked with green, while outliers are marked with blue. Fig. 4 shows found lines painted over an example image, horizontal lines are red, vertical lines are green.

We use the proposed method as part of a video processing pipeline, and intersection points of the lines are used for 2D-3D relative pose estimation (i.e. the transformation between the camera coordinate system and the cage coordinate system). For that application, not all wires are required to be detected, but the accuracy of the line parameters is crucial.

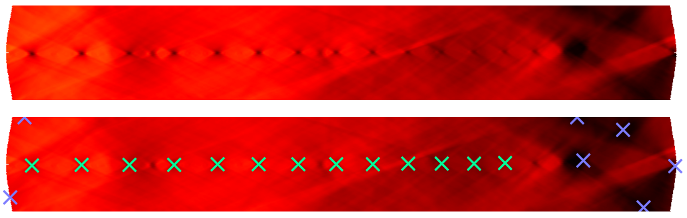


Figure 3: Example visualization of C cost function for horizontal wires. Horizontal axis is ρ and vertical axis is θ. Brighter color means higher function value. Top: Negative peaks can be seen as dark spots. Bottom: All candidates as described in Section 2.3, filtered as in Section 2.4. Inliers are marked with green, outliers are marked with blue.

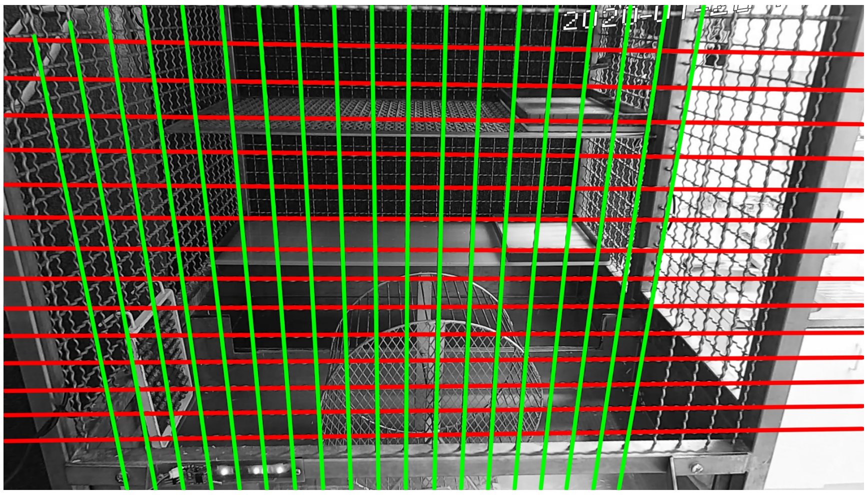


Figure 4: Found lines painted over the original image, vertical lines in green and horizontal lines in red.

The parameter w has to be determined for a given experimental setup. The proposed method is not sensitive to this parameter and can tolerate deviations that occur in most cases. A problem can arise when there is a large angle between the image plane and the plane of the mesh, resulting in significantly different distances between the camera and the two edges of the mesh. As a result, the thickness of the wires in the image varies within a wide range. In such situations, a band with a given width can cover more than one wire in some parts of the image while covering only a small part of a single wire in other areas.

As described in Section 1, other methods cannot be used reliably and easily on our image set. Line detectors cannot be applied, as the wires do not appear as lines in the image. For Hough transform, precise binarization of the image would be needed with a carefully set threshold. However, on our images, especially in night conditions, a strict threshold results in almost all wires except at the center of the image not being detected at all. Conversely, by using a more permissive threshold, the wires of the side and back of the cage, along with other objects, make detection of the front wires impossible. An example visualization for a night image is presented in Fig. 5. Pre-trained models of neural network-based methods are trained on wire mesh samples that differ from the ones present in our images, and cannot detect the mesh. Training would require a huge number of manually segmented samples.

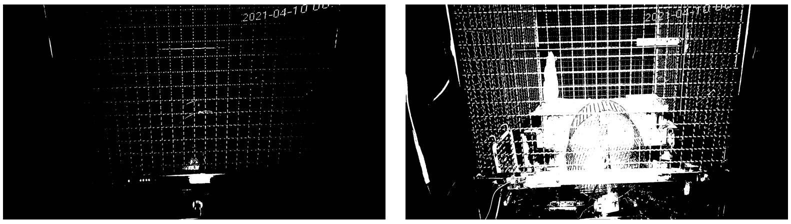


Figure 5: Illustration of problems with binarizing for Hough transform shown on a night image. Left: wires except the central ones disappear with a restrictive threshold. Right: wires of the side and back of the cage, along with other objects, make the detection of the front wires impossible with a more permissive threshold.

4 Conclusion

We presented an efficient method for extracting line parameters of the projection of wires of a woven mesh in an image by transforming the image space into a 2D parameter space and finding and robustly filtering local minima of the resulting cost function. Experimental results show that the algorithm is able to accurately detect wires and filter out false detections in general experimental setups where the vanishing point is outside the image.

Although the presented method works robustly for the desired application, several improvement possibilities could be investigated. An optimizer could be utilized to find local minima so that the computational cost could be reduced if the cost function is not calculated for the entire parameter space. We plan to examine the effect of running the method on gradient images instead of the original images, so that $\overline{I}$ would not have to be defined. We also plan to incorporate a solver for the linear equation system described in Section 2.4 to make the algorithm usable for meshes that are seen from a low angle.

References

[1] Canny, J. A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 8(6):679–698, 1986. DOI: 10.1109/tpami.1986.4767851.

[2] Duda, R. and Hart, P. Use of the Hough transformation to detect lines and curves in pictures. *Communications of the ACM*, 15(1):11–15, 1972. DOI: 10.1145/361237.361242.

[3] Even, P., Ngo, P., and Kerautret, B. Thick line segment detection with fast directional tracking. In *Image Analysis and Processing*, pages 159–170. Springer International Publishing, 2019. DOI: 10.1007/978-3-030-30645-8_15.

[4] Fischler, M. and Bolles, R. Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography. *Communications of the ACM*, 24(6):381–395, 1981. DOI: 10.1145/358669.358692.

[5] Hettiarachchi, R., Peters, J., and Bruce, N. Fence-like quasi-periodic texture detection in images. *Theory and Applications of Mathematics and Computer Science*, 4(2):123–139, 2014. URL: http://cs.umanitoba.ca/~bruce/papers/periodic.pdf.

[6] Körmöczi, L., Kalmár, G., Adlan, L., Büki, A., Kékesi, G., Horváth, G., and Nyúl, L. Rágcsálók viselkedésmintázatának kutatása automatizált videóelemzéssel. In *Képfeldolgozók és Alakfelismerők Társaságának 13. konferenciája*, 2021. 17 pages, URL: https://kepaf.njszt.hu/kepaf2021/submissions/submission_00047.pdf.

[7] Körmöczi, L. and Nyúl, L. Detecting corrugated wire mesh in images. In *Proceedings of the 13th Conference of PhD Students in Computer Science*, pages 116–120. University of Szeged, 2022. URL: https://www.inf.u-szeged.hu/~cscs/cscs2022/pdf/cscs2022.pdf.

[8] Lo, R.-C. and Tsai, W.-H. Gray-scale Hough transform for thick line detection in gray-scale images. *Pattern Recognition*, 28(5):647–661, 1995. DOI: 10.1016/0031-3203(94)00127-8.

[9] Lueangwattana, C., Mori, S., and Saito, H. Removing fences from sweep motion videos using global 3d reconstruction and fence-aware light field rendering. *Computational Visual Media*, 5(1):21–32, 2019. DOI: 10.1007/s41095-018-0126-8.

[10] Matsui, T. and Ikehara, M. Single-image fence removal using deep convolutional neural network. *IEEE Access*, 8:38846–38854, 2020. DOI: 10.1109/access.2019.2960087.

[11] Serra, J. *Image analysis and mathematical morphology.* Academic Press, 1982. ISBN: 0126372403.

[12] Yamashita, A., Matsui, A., and Kaneko, T. Fence removal from multi-focus images. In *Proceedings of the 20th International Conference on Pattern Recognition.* IEEE, 2010. DOI: 10.1109/icpr.2010.1101.

Acta Cybernetica **26** (2024) 593–619.

Overlaying Control Flow Graphs on P4 Syntax Trees with Gremlin[*]

Dániel Lukács[ab] and Máté Tejfel[ac]

Abstract

Our overall research aim is to statically derive execution cost and other metrics from program code written in the P4 programming language. For this purpose, we extract a detailed control flow graph (CFG) from the code, that can be turned into a full, formal model of execution, to extract properties – such as execution cost – from the model. While CFG extraction and analysis is well researched area, details are dependent on code representation and therefore application of textbook algorithms (often defined over unstructured code listings) to real programming languages is often non-trivial. Our aim is to present an algorithm for CFG extraction over P4 abstract syntax trees (AST). During the extraction we create direct links between nodes of the CFG and the P4 AST: this way we can access all information in the P4 AST during CFG traversal. We are utilizing Gremlin, a graph query language to take advantage of graph databases, but also for compactness and to formally prove algorithm correctness.

Keywords: control flow graph, static analysis, P4, Gremlin, graph database, proof of correctness

1 Introduction

Our long-term research goal – that also motivates this current work – is to develop an adaptable, scalable, and efficient static cost analysis tool for programs written in the P4 programming language [7]. P4 is a new domain-specific programming language running on programmable network switches. P4 programs describe network communication protocols: more specifically, a P4 programs tells the switch how process (transform, forward, or drop) an incoming network packet. Static cost

[*]Supported by the ÚNKP-21-4 New National Excellence Program of the Ministry for Innovation and Technology from the source of the National Research, Development and Innovation Fund. This research is in part supported by the project no. FK_21 138949, provided by the National Research, Development and Innovation Fund of Hungary.

[a]Faculty of Informatics, Eötvös Loránd University, Budapest, Hungary
[b]E-mail: dlukacs@caesar.elte.hu, ORCID: 0000-0001-9738-1134
[c]E-mail: matej@inf.elte.hu, ORCID: 0000-0001-8982-1398

DOI: 10.14232/actacyb.298770

analysis tools – that can estimate performance, energy needs, and other metrics of a P4 program automatically and without actually executing the program code – have several industrial use cases. Unfortunately, cost analysis is NP-hard (it solves the halting problem). While algorithms exist, they do not scale well for large, industrial size P4 programs: the time it takes to compute the solution exceeds the bounds of what is considerable usable in the industry.

As we attempted to realise a cost analysis tool for P4, we found that control flow analysis has a central role in all our efforts. Control flow analysis [1, Chapter 8.4.3.] concerns discovering the order of execution of the program statements in compile-time (i.e. based on the source code, or its equivalent representation, the syntax tree). Due to branching structures, there are usually multiple possible selections of executable statements, so the appropriate representation of the results is a graph, called control flow graph (CFG). We refer to paths in the CFG as execution paths.

Cost analysis requires a representation where implementation-dependent information (abstractions of the implementation, for example, cost formulas) can be easily inserted. CFGs turned out to be such representations. In our earlier works [10, 9], we discussed an approach based on enumerating all possible execution paths in the CFG of a P4 program to produce the average (or minimum or maximum) cost for that program. Our preliminary measurements have also shown that CFG path enumeration scales up for CFGs having as much as one hundred-thousand execution paths.

CFGs can also be considered Kripke-structures or transition systems [5, Chapter 2.1.], with the program counter being the only visible variable in the state, while the actual program state stays implicit in the start state and its subsequent transformations by the program instructions. A clear consequence of this is that CFGs – as traditionally understood – are not full program representations, but graphs that connect program points to program points, mostly with no formal references to the actual data that is being processed during execution. For example, it may happen that during execution, we update a variable in a way such that one branch of a conditional is never executed. Taking the cost of this branch into account in the average cost then possibly leads to a significant overestimation of the true program cost.

For this reason, in our latest work [11], we decided to try a new approach instead of CFG path enumeration and relied on probabilistic model checking to cost analyse P4 code. Here, we translate P4 code into model checkable representation, and – together with the specification of cost requirements – we delegate this to a model checker tool. Yet, even in this approach, we rely on CFGs in order to generate the model checkable representation. And here as well, we need more information about the program points than what textbook definition CFGs store.

Thus, our first aim in this paper is to develop a variant of CFG representations that is also capable of meaningfully representing program data and instructions over this data. To achieve this, we will define control flow analysis over the abstract syntax tree (AST) [1, Chapter 2.5.1.] of P4 programs. The AST is a hierarchical representation of the program sources, describing how various program structures and expressions are nested into each other. As AST is a full program representation, we can exploit the fact that the AST already has every information about data and instructions, and to make this information available during control flow analysis, we just have to establish the links between the corresponding nodes in the AST and the CFG. As a result, when we traverse the CFG, any further information about the current program point is just one link away, in the form of an AST subtree.

As it can be seen, interconnected graph representations will be central to our efforts. To make sure we use graphs in the most efficient way possible, we host these graphs (AST and CFG) in a graph database (GDB). A GDB is a database that stores data in graph data structure and provides a query language with graph semantics. Compared to relational databases (storing data in tables), GDBs eliminate the need for expensive join-operations, making them more efficient (both in terms of computation and usage) for storing and traversing heavily interconnected data. An extensive meta-analysis on the concept can be found in Angles et al. [3].

Then, our second aim is to address the problem of implementing a CFG algorithm in the form of a Gremlin query. By doing so, we can leverage built-in optimisations in Gremlin-compatible GDBs, such as parallelisation and bulking (see Rodriguez [14]). As Gremlin is a domain-specific language with somewhat unusual syntax and semantics, we found this task challenging enough to deem it necessary to discuss it in depth. We also hope that this discussion will be helpful for all future Gremlin programmers aiming to implement non-trivial algorithms in Gremlin.

Contributions In this work, we present an approach to intraprocedural CFG extraction from ASTs (formalised as a Gremlin query), and prove its correctness. At the same time, we explore the expressive power of Gremlin for specifying fairly complex static analysis procedures. We define both ASTs and CFGs in Section 3.1, in a way that can handle most of the P4 language control flow, and makes extending the AST for the rest is a straightforward process. Then, in Section 3.2, we describe the extraction algorithm in pseudocode. In Section 4.1 we formalised the semantics of a subset of Gremlin. Section 4.2 contains the extraction algorithm in Gremlin. We use the formal description to prove the correctness of the algorithm in Section 4.3. Finally, we conclude the paper with a few words about limitations and future work.

2 Related work

Recently, Dumitrescu et al. [8] introduced *Bf4*, a program verification tool for P4 programs, that also builds heavily on the CFG representation of P4. They rely on a preceding instrumentation step, and extend the CFG with "bug nodes" (nodes, guarded with a condition that can only be satisfied if there is a bug), and then perform program slicing (using SSA and various dependency analyses) to compute reachability of the bug nodes using an SMT solver. They do not discuss their internal CFG representation, but they do tell that they realised the tool as a P4C backend, and the size of the implementation is around 25000 lines of C++ code (not counting the P4C infrastructure). We suspect GDB-integrated deep CFGs could complement the Bf4 implementation in order to reach all necessary the information more easily than what visitors over the P4C intermediate program representation can currently provide.

The work of Amighi et al. [2] shares some goals with ours. They extract CFGs from Java bytecode, which is a more difficult problem since it involves handling stack (implicit the bytecode) and exception flow as well. First, they translate bytecode to an intermediate representation that makes the stack explicit. Then, for each instruction they declaratively define the transition relation between program points. The final CFG is simply the union of the transitions resulting from evaluating the relation over program points and instructions of a given bytecode. Like us, they also prove the correctness of the extraction. In their proof, they establish the existence of a simulation relation between states induced by the bytecode instructions and states induced by the extracted CFG.

An important application of CFGs is that it is the representation on which data flow analysis (DFA) [1, Chapter 9.2.] operates. The results of DFA can be represented e.g. in the form of a definition-use graph, establishing links between the definitions of variable names and the usages of these names. In turn, it should be possible to store such a definition-use graph in our GDB, interlinked to AST and CFG, in order to enable even more applications. For example, Birnfeld et al. [6] combine CFG and definition-use graphs to discover potential faults in P4 code, to detect e.g. that there are execution paths where the P4 program processes invalid packets.

In our work, we extract CFGs from a structured AST, not from unstructured code (with features such as `gotos`, no nesting, etc.). This is also the approach of Söderberg et al. [15], who – analysing Java – recognise that by superimposing the CFG on the AST, "high-level abstractions are not compiled away during the translation to intermediate code". The authors utilise elegant reference attribute grammars for control flow and data flow analysis. In this approach control-related AST nodes get a reference attribute (e.g. *successor*) that points to another AST node where control is supposed to flow from the previous node. Another interesting feature of this work is that it is easily extensible to handle new language elements in novel versions of Java, by incrementally adding new grammar rules.

Another inspiring example for this concept is the RefactorErl framework, that also superimposes control flow (and many other static analysis results) on the AST [16] for the Erlang programming language.

3 Problem and solution idea

3.1 Basic definitions

In this section, we define what we mean by ASTs and control flow graphs in the following sections. In the correctness proof in Section 4.3, these definitions constitute the precondition and postcondition.

An illustration of these concepts is depicted by Figure 1. One the left, there is a simplified excerpt from `basic_routing-bmv2.p4`, a P4 program describing an L2/L3 routing protocol, used in the testing of the P4 reference compiler, P4C [13]. This code describes a control declaration named `ingress`. It declares a few match-action tables (their external definition is linked at compile-time), and then it specifies the actual control flow determining the (sometimes conditional) invocation of these tables. In the middle is the corresponding AST with `d`, `b`, `c`, `s` labelled nodes denoting control declaration, block, conditional, and statement nodes respectively. (We do not analyse control flow inside expressions.) On the right is the corresponding CFG with matching node names. The thin lines are the association edges between AST nodes and CFG nodes.

```
control ingress(inout headers hdr,
                [...]) {
 table bd {[...]}
 table ipv4_fib {[...]}
 table ipv4_fib_lpm {[...]}
 table nexthop {[...]}
 table port_mapping {[...]}
 [...]
 apply {
   if (hdr.ipv4.isValid()) {
     port_mapping.apply();
     bd.apply();
     if ([...]) {
       ipv4_fib_lpm.apply();
     } else {}
     nexthop.apply();
   } else {}
 }
}
```

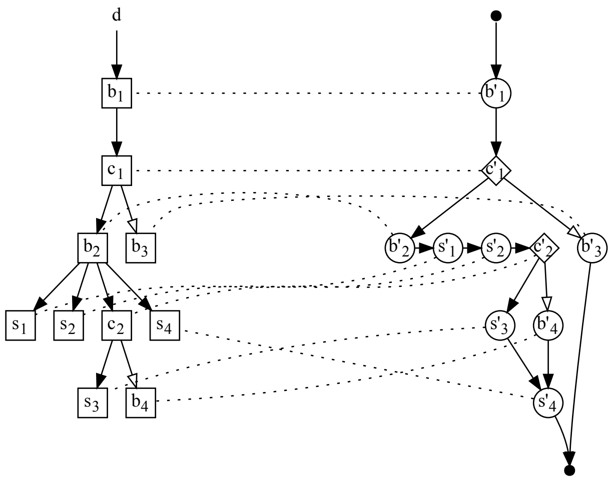


Figure 1: Source code, AST, and CFG of a control declaration

In the rest of this section, we define these concepts based on labelled graphs and related notations. GDBs support sophisticated attribute-based labelling, but for ease of understanding we use a simpler scheme in this paper.

Definition 1. *A labelled graph is a* (V, E, l) *tuple of a* V *node set, an* E *edge set, and an* $l : (V \cup E) \to L$ *labelling function (where* L *is an arbitrary set of labels).*

Notations. In case a distinction must be made between multiple graphs, we write e.g. (V_g, E_g, l_g) to denote the components of a particular graph g. We use underlined lowercase letters to denote a node with a specific label: for example $\underline{x}$ denotes a node $n \in V$ for which $l(n) = \mathtt{x}$ (a node with label `x`). We use indexes to distinguish between multiple nodes: for example, $\underline{x}_1$ and $\underline{x}_2$ denotes n_1, n_2 nodes for which $l(n_1) = l(n_2) = \mathtt{x}$. We write $n_1 \xrightarrow{\mathtt{x}} n_2$ to denote an edge $(n_1, n_2) \in E$ for which $l((n_1, n_2)) = \mathtt{x}$. In the case graph g is a tree, $root_g$ denotes the root of tree g, and $children_g(n, \mathtt{x})$ denotes those child nodes of node n in g whose incoming edge has label `x`.

Notations. In the AST, we use labels `d`, `b`, `c`, and `s` to denote control declarations, blocks, conditionals, and statements, respectively (so e.g. $\underline{d} \in V$ denotes any $n \in V$ node that is a control declaration). Syntactical edges between the AST nodes are appropriately labelled with labels to distinguish from other edges introduced into G, e.g. `assoc` (see later). These edge labels are `body` (between a control declaration node and the top-level block forming its body), `nest` (between a block and its nested blocks), `statement` (between a block and its statements), `true`, and `false` (between a conditional node and its branches).

Definition 2. *A (V, E, l) graph is an abstract syntax tree (AST), if it is a tree, and $\forall n \in V : l(n) \in \{\mathtt{d}, \mathtt{b}, \mathtt{c}, \mathtt{s}\}$, and*

1. *If $(n_1 \xrightarrow{\mathtt{body}} n_2) \in E$, then $l(n_1) = \mathtt{d}$, and $l(n_2) = \mathtt{b}$*
2. *If $(n_1 \xrightarrow{\mathtt{nest}} n_2) \in E$, then $l(n_1) = \mathtt{b}$, and $l(n_2) = \mathtt{b}$*
3. *If $(n_1 \xrightarrow{\mathtt{statement}} n_2) \in E$, then $l(n_1) = \mathtt{b}$, and $l(n_2) = \mathtt{s}$,*
4. *If $(n_1 \xrightarrow{\mathtt{true}} n_2) \in E$, then $l(n_1) = \mathtt{c}$, and $l(n_2) = \mathtt{b}$*
5. *If $(n_1 \xrightarrow{\mathtt{false}} n_2) \in E$, then $l(n_1) = \mathtt{c}$, and $l(n_2) = \mathtt{b}$*
6. *If $\underline{s} \in V$, then $children(\underline{s}) = \varnothing$*
7. *If $\underline{d} \in V$, then $\exists!\ \underline{b} \in V : (\underline{d} \xrightarrow{\mathtt{body}} \underline{b}) \in E$,*
8. *If $\underline{c} \in V$, then $\exists!\ \underline{b}_1, \underline{b}_2 \in V : ((\underline{c} \xrightarrow{\mathtt{true}} \underline{b}_1) \in E \ \wedge\ (\underline{c} \xrightarrow{\mathtt{false}} \underline{b}_2) \in E)$,*

The definition asserts that all P4 control declaration has an AST made of blocks (containing 0,1 or more ASTs), conditionals (containing exactly two ASTs, one per branch), and statements (primitives, along with empty blocks).

In addition, we assume (without explicitly featuring) that all AST nodes have a unique identifier incremented in depth-first order. Such identifier attributes (labels) can be inserted in the graph straightforwardly (preferably directly after graph construction). The recursive top-down traversal of ASTs guarantees termination and allows us to use a common idea in correctness proofs of functional programs. By inductively assuming that the previous elements were correctly processed, and applying a proven correct procedure to the current element, we only have to assure that the previous and the current elements are aggregated in an appropriate manner.

In later sections, we will define our algorithm over ASTs of P4 control declarations. For this reason, we restrict our discussion here to these, and omit discussing `goto`-like flows in the P4 packet parser declarations. In P4, packet parsers are defined in the form of state machines. Their control flow analysis is straightforward, so omit this for simplicity. P4 has no construct for user-defined loops except for match-action tables (lookup tables that match packet headers to actions). The implementation of these table algorithms is not part of the language, only the node of the table application appears in the syntax tree (and similarly, table applications will be featured as single nodes in the control flow graph). For languages with loops, extending the definition is straightforward, albeit cumbersome. P4 also has one-way and multi-way conditionals, but those are processed similarly to two-way conditionals, and so we also omit them for simplicity.

Now, we prepare for defining deep CFGs. The main problem we have to solve as we translate an AST to a CFG is that ASTs give no explicit clue about the order of execution of its elements, while CFGs aims to describe precisely that. It is well-known [4], that by transforming CFGs to static single-assigment form, blocks can be treated as functions, and directed flows as calls between these functions: the result is a functional program, where the continuation (i.e. the rest of program) at each program point explicitly appears as a function. While we aim for a more direct definition, this gives us a clue that a recursive approach can be successful. Specifically, we will define CFGs as compositions of sub-CFGs, with each sub-CFG relating to a subtree of the control declaration AST.

Definition 3. *Let $G = (V, E)$ contain syntax subtree t and subgraph c. We say that c is the sub-CFG corresponding to t, with source $n \in V_c$ and return points $R \subseteq V_c$ given the following conditions are satisfied in G:*

1. *If $root_t = \{\underline{s}\}$, then $(n \xrightarrow{assoc} \underline{s}) \in E, \quad R = \{n\}$*
2. *If $root_t = \{\underline{b}\}$ and $children_t(\underline{b}) = \varnothing$, then $(n \xrightarrow{assoc} \underline{b}) \in E, \quad R = \{n\}$*
3. *If $root_t = \{\underline{b}\}$ and $children_t(\underline{b}) = \{t_1, \dots, t_k\}$ and c_i is the sub-CFG with source n_i, return points R_i corresponding to the subtree rooted in t_i $(\forall i = 1, \dots, k)$, then*

$$\begin{aligned}
&(n \xrightarrow{assoc} \underline{b}) \in E,\\
&(n \xrightarrow{flow} n_1) \in E_c,\\
&(r \xrightarrow{flow} n_i) \in E_c \ \ (\forall i = 2 \dots k, \ \ \forall r \in R_{i-1}),\\
&R = R_k
\end{aligned}$$

4. *If $root_t = \{\underline{c}\}$ and $children_t(\underline{c}) = \{t_1, t_2\}$, then*

$$\begin{aligned}
&(n \xrightarrow{assoc} \underline{c}) \in E,\\
&c_i \textit{ is the sub-CFG corresponding to } t_i \textit{ with source } n_i,\\
&\quad \textit{return points } R_i \ \ (\forall i = 1, 2),\\
&(n \xrightarrow{flow} n_i) \in E_c \ \ (\forall i = 1, 2),\\
&R = R_1 \cup R_2
\end{aligned}$$

According to the definition, the sub-CFG of statements and empty blocks is a single node, relating to the syntax node of the statement or empty block itself. The sub-CFG of non-empty blocks is composed of a CFG node relating to the syntax node of the block, and the sub-CFGs of its children. We set up flow from the block to the first child CFG, and also between the siblings. The sub-CFG of a conditional is composed of a CFG node relating to the syntax node of the conditional, and of the sub-CFGs of the children. Here, we omitted true and false labels on the flows, but these can be easily identified by querying the incoming edge (labelled with `true` or `false`) of the associated node in the AST.

We may note that control declaration nodes (`d`-labelled nodes) are missing from the sub-CFG definition. This is because the CFG corresponding to such a node is a top-level CFG, that we call deep CFG. This node does not require much analysis compared to its descendants, it simply identifies the entry and exit nodes of the CFG.

Definition 4. *Let $G = (V, E)$ contain control declaration AST u and subgraph g. We say that g is the deep CFG corresponding to u, with entry $e \in V_g$ and exit $f \subseteq V_g$ given the following condition is satisfied in G:*
If $root_u = \underline{d}$, $child(\underline{d}) = \{t\}$ and g is the sub-CFG corresponding to the tree rooted in t, with source n and return points R, then

$$(\underline{d} \xrightarrow{entry} e) \in E, \quad (\underline{d} \xrightarrow{exit} f) \in E, \quad (e \xrightarrow{flow} n) \in E_g, \quad (r \xrightarrow{flow} f) \in E_g \quad (\forall r \in R)$$

Edges labelled with `entry` and `exit` are similar to `assoc`, linking the AST declaration node to the CFG entry and exit points. Control will flow from the entry node to the first block (the source of the sub-CFG corresponding to the declaration body). From the return points of this first block, control flows into the exit node. Flows inside the CFG are determined by Definition 3.

3.2 CFG extraction

First, we present the idea of our CFG extraction algorithm by translating the CFG definition into an informal, imperative description. Later on we formalise this as a Gremlin traversal. We split the operation in two.

Algorithm 1 iterates over the control declaration nodes in the AST, creates one entry and one exit CFG node (e and f) for each in graph G, and then calls Algorithm 2 on b (the top-level block of the declaration). An edge will be sent from e to b by that other procedure (as b's CFG is the subsequent continuation of e). Finally, we send an edge from R (the return nodes returned by that procedure) to f (as the exit is the final continuation).

Algorithm 2 creates a CFG for the AST of some b, either a block or a conditional. The algorithm expects a set of predecessor CFG nodes: while ProcNode is initially called with (a set of) just a single predecessor, later it is called again recursively with the R set of return points. The resulting CFG is the subsequent continuation of whatever is in R, and so right after we create its starting point n, we link the contents of R to n. In case b is a block or a statement, we recursively apply the procedure to the children. R is initially set to n, since the continuation after node b is the CFG of the first child. After the child CFG was produced, we assign the return points of that CFG to R as this needs to be linked to the next continuation (that is either a sibling, or a sibling of an ancestor). Note that in case b is an empty block or a statement, it has no children and $\{n\}$ is the returned return point (this is the node that will be followed by the sibling of an ancestor).

In case b is a conditional, n has to be linked to the two possible subsequent continuations, and we collect into R the return points of both branch CFGs, as anything that follows will be the subsequent continuation of both branches. Algorithm 2 always terminates because it is progressing from child to child, and for any of its calls, the longer the call's stack trace, the shorter the distance between b and the AST leaves.

```
Procedure CFG($G$):
    Input: $G$ is graph, includes AST
    Result: CFG of each control declaration is added to $G$
    begin
        forall $v \in V_G$ do
            if $v$ is control declaration then
                $e :=$ new CFG entry node
                $f :=$ new CFG exit node
                $V_G := V_G \cup \{e, f\}$
                $b := child_G(v, \texttt{body})$
                $R :=$ ProcNode($G$, $b$, $\{e\}$)
                forall $r \in R$ do
                    $E_G := E_G \cup \{r \xrightarrow{\texttt{flow}} f\}$
                end
            end
        end
        return
    end
```

Algorithm 1: Control declarations

4 Formalising CFG extraction in Gremlin

4.1 Semantics of Gremlin

To formally prove our CFG extraction operation in Section 4.2, we have to have a formal semantics for the Gremlin Traversal Machine (GTM). In the white paper [14], the authors give a mathematical description of the GTM, but one of their goals is to keep it general and enable adapters to implement many possible – including parallel – evaluation strategies. In this section, we intend to reiterate this description with two important modifications: we formalise it as an axiomatic

```
Procedure ProcNode(G,b,P):
    Input: $G$ is graph, includes AST and CFG
    Input: $b$ is block, statement or conditional in the AST
    Input: $P$ is set of predecessor CFG nodes
    Output: Return points of CFG of $b$
    Result: CFG of $b$ is added to $G$
    begin
        $n :=$ new CFG node
        $V_G := V_G \cup \{n\}$
        forall $p \in P$ do
            $E_G := E_G \cup \{p \xrightarrow{flow} n\}$
        end
        if $b$ is block $\vee$ $b$ is statement then
            $R := \{n\}$
            forall $c \in children_G(b, \{\texttt{nest}, \texttt{statement}\})$ do
                $R :=$ ProcNode($G$, $c$, $R$)
            end
            return $R$
        else
            if $b$ is conditional then
                $R := \varnothing$
                forall $c \in children_G(b, \{\texttt{true}, \texttt{false}\})$ do
                    $R := R \cup$ ProcNode($G$, $c$, $\{n\}$)
                end
                return $R$
            end
        end
    end
```

Algorithm 2: Blocks, conditionals

semantics so that we can use it in proofs (see Section 4.3), and we restrict the traverser set (see later) to be an ordered set (or list). This restriction ensures that sibling nodes are processed sequentially, and in a fixed order. In our experience, the default evaluation strategy in the native Gremlin Java graph implementation (called `TinkerGraph`) satisfies this restriction.

The GTM executes a program called graph traversal, which is effectively a sequence of instructions (with some higher-order instructions also accepting graph traversal programs as parameters). While "traversal" is a notion employed in the Gremlin-literature to denote programs, it may cause some confusion that it is also used colloquially to denote the execution of such programs. Therefore, in the formal treatment we use the notion of traversal to denote the program text, and use other phrases (semantics, state, etc.) to characterise execution. We now proceed first to define the state of the GTM. The global memory state of the GTM consists of the graph contents and other auxiliary storages. There are stores in the memory that can store both graph objects (e.g. nodes, edges) and other objects (e.g. collections of graph objects).

Definition 5. *The* ***global memory*** *of the GTM state is a* $\Gamma = (V, E, H, K)$ *tuple, where* $(V,\ E)$ *is the graph itself,* H *is an object heap for processing non-graph objects during the traversal, and* K *is a global key-value store (often, referred to as side-effect store).*

Besides the global memory, graph traversals have a current local state called traverser. A GTM state stores multiple traversers at the same time. Informally, we can imagine each traversers as a worker (in Gremlin materials often depicted as the green little monster) that jumps from node to node (as per the instructions of the traversal program), and collects data about the nodes into various data structures. In addition, the worker can "clone" itself when the traversal program branches: the clone starts with the same data as the original, but they will move around following the instructions of a different program branch.

Definition 6. *A* ***traverser*** *is a* (p, k, s) *triple, where* p *is a pointer to a graph element or an object in* H*,* k *is a local key-value store, and* s *is the sack, a local store for sum-like operations (aggregation).*

The white paper [14] lists some other components of traversers as well, but we will not use those in this work.

Finally, a traversal is effectively a program, whose instructions (called *steps* and denoted by σ) transform a Γ global memory (e.g. by changing the graph) and list of (p, k, s) traversers (e.g. by moving the individual traversers forward in the graph).

Definition 7. *A* ***traversal*** *is a program defined by the following simple grammar:*

$$\begin{aligned} \Psi &::= \varepsilon \mid \sigma \rightsquigarrow \Psi \\ \sigma &::= \sigma_1 \mid \sigma_n(\Psi, \ldots, \Psi) \end{aligned}$$

A traversal Ψ may be empty (ε), or it may consist of a step σ, sequentially followed by the rest of the traversal. Steps are $2^\Gamma \times 2^T \longrightarrow 2^\Gamma \times 2^T$ functions, and come in two variants: higher-order steps ($\sigma_n(\Psi, \ldots, \Psi)$) are parameterised by a number of different subtraversals and transform the GTM state by executing these subtraversals in the current GTM state, while first-order and zeroth-order steps (σ_1) take only ordinary parameters to transform the state. In these terms, we can think of $\rightsquigarrow$ as forward function composition.

Notations. In the grammar of Definition 7, σ, σ_1, σ_n and Ψ are non-terminals. Later in the text, we use them to denote concrete traversals and steps. In particular, we will use σ to denote steps in the Gremlin language, e.g. $\sigma_{\texttt{flatMap}}$ will denote the `flatMap` step of Gremlin.

Gremlin defines over 30 type of steps, and we have no place here – nor do we find it indispensable – to include a formal definition for each of them. After defining machine state and semantics, we will include in Table 1 short informal descriptions for the ones we used in our CFG extraction traversal (e.g. $\sigma_{\texttt{outE}}$, $\sigma_{\texttt{inV}}$, $\sigma_{\texttt{flatMap}}$, $\sigma_{\texttt{sideEffect}}$), and hope that our readers can reconstruct a formal definition in case needed.

Definition 8. *The state of the GTM is a triple* $(\Gamma,\ T,\ \Psi)$*, where* Γ *is the state of the global memory,* T *is a traverser list (storing the current local state of multiple traversals), and* Ψ *is a traversal.*

Below, we formalise the evaluation of Gremlin traversals as an axiomatic semantics similar to Hoare-logic. A thorough introduction on defining language semantics with inferential systems and proving it using inferential trees can be found in Nielson & Nielson [12, Chapter 6.2].

Note that we intend Rules (4), (5), and (6) as templates that highlight and help in formalizing the three main categories of concrete steps.

$$\frac{}{\{\Gamma;\varnothing\}\quad \Psi \quad \{\Gamma;\varnothing\}} \tag{1}$$

$$\frac{}{\{\Gamma;T\}\quad \varepsilon \quad \{\Gamma;T\}} \tag{2}$$

$$\frac{\{\Gamma;T\}\ \sigma\ \{\Gamma_1;T_1\} \qquad \{\Gamma_1;T_1\}\ \Psi\ \{\Gamma_2;T_2\}}{\{\Gamma;T\}\quad \sigma \rightsquigarrow \Psi \quad \{\Gamma_2;T_2\}} \tag{3}$$

$$\frac{\sigma_1(\Gamma;\ t_1) \mapsto (\Gamma_1;\ R_1) \qquad \cdots \qquad \sigma_1(\Gamma_{n-1};\ t_n) \mapsto (\Gamma_n;\ R_n)}{\{\Gamma;\ t_1,\dots,t_n\}\quad \sigma_1 \quad \{\Gamma_n;\ R_1 \cup \ldots \cup R_n\}} \tag{4}$$

$$\frac{\{\Gamma;t_1\}\ \Psi'\ \{\Gamma_1;R_1\} \qquad \cdots \qquad \{\Gamma_{n-1};t_n\}\ \Psi''\ \{\Gamma_n;R_n\}}{\{\Gamma;\ t_1,\dots,t_n\}\quad \sigma_n(\Psi) \quad \{\Gamma_n;\ R_1 \cup \ldots \cup R_n\}} \tag{5}$$

$$\frac{\{\Gamma;T\}\ \Psi_1'\ \{\Gamma_1;R_1\} \qquad \cdots \qquad \{\Gamma_{n-1};T\}\ \Psi_n''\ \{\Gamma_n;R_n\}}{\{\Gamma;\ T\}\quad \sigma_n(\Psi_1,\dots,\Psi_n) \quad \{\Gamma_i;\ R_i\}} \quad i = \min_{\substack{j=1\ldots n \\ R_j \neq \varnothing \vee j=n}} j \tag{6}$$

Axioms (1) and (2) correspond to termination in case the traversal mapped to an empty traverser list or in case all steps were executed in the traversal.

Rule (3) corresponds to the classic sequencing rule: the first step and the rest of the traversal is executed in the program state resulting from the first step. What may not be evident at first glance, is that this step prescribes a **breadth-first traversal**: step σ is applied to all traversers in T (possibly modifying the global state) before the following steps are applied to any of them.

Rule (4) is a template rule for describing how first-order steps are evaluated. This rule also emphasizes our restriction that traversers are processed in some fixed order.

Note that some steps may map to any number of traversers (including zero). Rule (5) is a template rule for higher-order traversals: it goes through the traversers in some fixed order, and applies the subtraversal one-by-one to each traverser (possibly modifying the global state in the mean time). Additionally, we allow instances of this rule to apply some modifications to the Ψ traversal (denoting the resulting traversal as Ψ') so a large variation of traversals (such as conditional traversals) can be expressed.

Finally, Rule (6) formalises special higher-order traversals knowns as branching. This rule executes its subtraversals in a way such that if Ψ_1 terminates with an empty traverser list, then Ψ_2 is executed, and so and so on either until one of the traversers terminates with a non-empty traverser list, or until Ψ_n is executed. While failed traversers may modify the state, only the traverser list of the first successful traverser will be returned. For generality, we also allow instances of this rule to modify their Ψ_i subtraversals. Note that higher-order traversals enable us to define **depth-first traversals** as well, since subtraversal has to be completely executed to complete the step.

In Table 1, we informally describe individual Gremlin steps we use in this paper. For space reasons, we omit formally defining the semantics of each step, and just highlight some of them to familiarise our readers with the notation. Based on the white paper [14], the online Gremlin documentation, and our algorithm description in our algorithm description in Section 4.2, we believe it is not too difficult to reconstruct the semantics of the steps.

4.2 CFG extraction in Gremlin

It is common for graph procedures (and as such, static analysis procedures as well) to have a short, intuitive textual specification, that – when implemented in executable code – blows up into an entangled web of nested loops, custom data structures, and reliance on language built-in dispatch mechanisms for distinguishing involved objects.

To avoid the gap between presentation and implementation, we formalised CFG extraction as a Gremlin (v3.4.4) traversal. The traversal in Figure 2 – consisting of around 60 steps – can be typed into any Gremlin language variant (e.g. Gremlin Java) with minimal amount of language-specific modifications (for example defining a function expression for σ_{clear}, using function expressions to enable lazy recursive calls of subtraversals, using type hints, etc.). In fact, we automatically generated the formulas in this paper directly from our implementation code, and then manually simplified the aforementioned elements. (Regarding recursion, see limitations in Section 5.) Our Gremlin Java implementation – that was extended to handle a slightly more elaborate graph schema that distinguishes between AST overlays and CFG overlays –, is slightly below 110 lines of code with each line having one or two steps. Beyond those mentioned, the requirement of navigating the labels and properties in the more complex graph schema was solely responsible for having to add in additional steps.

We show in Section 4.3 that this compact representation combined with Gremlin's simple semantics is very effective for formally deriving its correctness proof by hand. The alternative – proving a less abstract, less compact executable rep-

Table 1: Informal description of selected Gremlin steps

Step	**Description**
σ_{outE}	"Moves the traverser forward", i.e. it replaces the nodes in the traverser list with their outgoing edges.
σ_{inV}	"Moves the traverser forward", i.e. it replaces the edges in the traverser list with the nodes they enters
$\sigma_{\text{flatMap}}(\Psi)$	Applies Ψ to all $t \in T$ (see Definition 8) as described by Rule (5).
$\sigma_{\text{sideEffect}}(\Psi)$	Abbreviated as $\sigma_{\text{sEffect}}(\Psi)$. Also a Rule (5) step; it may modify the global state Γ, but it discards the traverser list produced by Ψ and returns the original T.
$\sigma_{\text{coalesce}}(\Psi_1, \dots, \Psi_n)$	A Rule 6 step, that returns the traversers from the first of traversals $\Psi_1, \dots \Psi_n$ which is successful (has non-empty result).
$\sigma_{\text{aggregate}_x}$	Adds the current traverser list into a collection assigned in K to name x.
σ_{cap_x}	Loads the content stored in x into the traverser list.
σ_{unfold}	Replaces collections in the traverser list with the elements of the collections.
$\sigma_{\text{sEffect}}(\text{clear})$	Empties a collection. Useful for discarding the traversers stored earlier in x, before adding a new element. *clear* is a custom Java function expression that empties the collection.
$\sigma_{\text{sack}_{\text{in}}}$	Applied to (p, k, s) will put p inside store s. This step enables additional sum-like operations, that we will not use.
$\sigma_{\text{sack}_{\text{out}}}$	Applied to (p, k, s) will replace p with the content of store s.
$\sigma@x$	Given that some σ step produces a (p, k, s) traverser, $\sigma@x$ will assign p to name x in local store k.
σ_{tail_1}	Removes all elements of the traverser list except for the last one.

resentation (such as direct Java code) with the same level of formality – would have likely required machine assistance, and in that case it likely would have been impossible to fit the complete proof of such a representation into this paper.

We now describe how this traversal implements the previously described CFG extraction procedure. The procedure is a traversal consisting of five subtraversals depicted in Figure 2. Ψ_{control} and Ψ_{icontrol} are corresponding to Algorithm 1 in Section 3.2. Ψ_{control} selects nodes in the AST that correspond to a P4 control declaration d, and then executes Ψ_{icontrol} (the internal part of Ψ_{control} we separated for readability). Here, we make use of Rule 5 semantics to make sure that steps in

Ψ_{icontrol} affect just one control declaration subtree at a time. Ψ_{icontrol} clears global register `r`, stores the entry CFG node of d into `r`, invokes subtraversal Ψ_{node} over the body block of d, and finally sends a `flow`-edge from the contents of `r` (supposedly filled by Ψ_{node}) to the exit CFG node of d. The second and third side-effects are simply there because we do not need the results from those subtraversals, and instead we want to continue from the same place they started. The side-effect with Ψ_{node} simply performs the invocation of the subtraversal. $\sigma_{\text{flatMap}}(\sigma_{\text{cap}_{\texttt{r}}} \rightsquigarrow \sigma_{\text{unfold}})$ is an idiom that makes the content of `r` the current traversal. σ_{flatMap} here is an implementation detail: σ_{cap} loses the traverser data, but σ_{flatMap} reattaches it to the new traversers.

$$\text{control} = \begin{pmatrix} V_{\texttt{ControlDeclaration}} \\ \rightsquigarrow\ \text{sEffect(icontrol)} \end{pmatrix}$$

$$\text{icontrol} = \begin{pmatrix}
\text{sEffect}(\text{clear}_{\texttt{r}}) \\
\rightsquigarrow\ \text{sEffect}\begin{pmatrix} \text{outE}_{\texttt{entry}} \\ \rightsquigarrow\ \text{inV} \\ \rightsquigarrow\ \text{aggregate}_{\texttt{r}} \end{pmatrix} \\
\rightsquigarrow\ \text{sEffect}\begin{pmatrix} \text{outE}_{\texttt{body}} \\ \rightsquigarrow\ \text{inV} \\ \rightsquigarrow\ \text{sEffect(node)} \end{pmatrix} \\
\rightsquigarrow\ \text{outE}_{\texttt{exit}} \\
\rightsquigarrow\ \text{inV@}\texttt{exit} \\
\rightsquigarrow\ \text{flatMap}\begin{pmatrix} \text{cap}_{\texttt{r}} \\ \rightsquigarrow\ \text{unfold} \end{pmatrix} \\
\rightsquigarrow\ \text{addE}\ \begin{matrix} \mathit{label} = \texttt{flow}, \\ \mathit{to} = \texttt{exit} \end{matrix}
\end{pmatrix}$$

$$\text{node} = \begin{pmatrix}
\text{\_\_@}\texttt{synB} \\
\rightsquigarrow\ \text{addV}_{\texttt{block}}\text{@}\texttt{newB} \\
\rightsquigarrow\ \text{sEffect}\begin{pmatrix} \text{addE}\ \begin{matrix} \mathit{label} = \texttt{assoc} \\ \mathit{from} = \texttt{synB} \\ \mathit{to} = \texttt{newB} \end{matrix} \\ \rightsquigarrow\ \text{flatMap}\begin{pmatrix} \text{cap}_{\texttt{r}} \\ \rightsquigarrow\ \text{unfold} \end{pmatrix} \\ \rightsquigarrow\ \text{addE}\ \begin{matrix} \mathit{label} = \texttt{flow} \\ \mathit{to} = \texttt{newB} \end{matrix} \end{pmatrix} \\
\rightsquigarrow\ \text{coalesce}\begin{pmatrix} \text{cond} \\ \text{block} \\ \begin{pmatrix} \text{sEffect}(\text{clear}_{\texttt{r}}) \\ \rightsquigarrow\ \text{aggregate}_{\texttt{r}} \end{pmatrix} \end{pmatrix}
\end{pmatrix}$$

$$\text{cond} = \begin{pmatrix}
\text{sack}_{\text{in}} \\
\rightsquigarrow\ \text{select}_{\texttt{synB}} \\
\rightsquigarrow\ \text{has}_{\mathit{label}=\texttt{Conditional}} \\
\rightsquigarrow\ \text{outE}_{\{\texttt{true},\texttt{false}\}} \\
\rightsquigarrow\ \text{order}_{\{\mathit{in}=\texttt{asc},\mathit{by}=\texttt{id}\}} \\
\rightsquigarrow\ \text{inV} \\
\rightsquigarrow\ \text{flatMap}\begin{pmatrix} \text{sEffect}\begin{pmatrix} \text{sack}_{\text{out}} \\ \rightsquigarrow\ \text{aggregate}_{\texttt{r}} \end{pmatrix} \\ \rightsquigarrow\ \text{flatMap(node)} \end{pmatrix} \\
\rightsquigarrow\ \text{sEffect}(\text{clear}_{\texttt{r}}) \\
\rightsquigarrow\ \text{aggregate}_{\texttt{r}}
\end{pmatrix}$$

$$\text{block} = \begin{pmatrix}
\text{sEffect}(\text{clear}_{\texttt{r}}) \\
\rightsquigarrow\ \text{aggregate}_{\texttt{r}} \\
\rightsquigarrow\ \text{select}_{\texttt{synB}} \\
\rightsquigarrow\ \text{outE}_{\{\texttt{nest},\texttt{statement}\}} \\
\rightsquigarrow\ \text{order}_{\{\mathit{in}=\texttt{asc},\mathit{by}=\texttt{id}\}} \\
\rightsquigarrow\ \text{inV} \\
\rightsquigarrow\ \text{flatMap}\begin{pmatrix} \text{flatMap(node)} \\ \rightsquigarrow\ \text{fold} \\ \rightsquigarrow\ \text{sEffect}(\text{clear}_{\texttt{r}}) \\ \rightsquigarrow\ \text{sEffect}\begin{pmatrix} \text{unfold} \\ \rightsquigarrow\ \text{aggregate}_{\texttt{r}} \end{pmatrix} \end{pmatrix} \\
\rightsquigarrow\ \text{tail}_1 \\
\rightsquigarrow\ \text{unfold}
\end{pmatrix}$$

Figure 2: CFG extraction in Gremlin

Ψ_{node}, Ψ_{cond}, and Ψ_{block} are corresponding to Algorithm 2 in Section 3.2. Here, `synB` and `newB` are just arbitrary variables (names of local store keys). Ψ_{node} assigns the local `synB` name to the AST node in its traverser (there is always just one) and creates a new CFG node corresponding to the AST node (again with a name). It links the two with an `assoc`-edge, and sends `flow`-edges from the traversers stored in `r`. This is the CFG entry node when Ψ_{node} is called initially, and later in the recursion `r` stores those CFG nodes that directly precede `newB` (see later). Finally,

Ψ_{node} calls σ_{coalesce} (see Rule 6): first, Ψ_{cond} is called, if terminates early (`newB` is not a conditional), then Ψ_{block} is called, and if that also terminates early (`newB` is an empty block), then we simply store `newB` in register `r`. This means that `newB` will be the direct predecessor of a CFG node created later (or of the CFG exit). The returned node will be that of σ_{coalesce} (`newB` in case both subtraversals terminate early).

Ψ_{cond} first stores its traverser (a CFG node) into a path-local store, and makes `synB` the current traverser. In case this node is not a conditional, Ψ_{cond} terminates early, otherwise it processes its branches. Ordering the branches ensures that we can retrace later which CFG flows correspond to the true and false branches (as a lengthier alternative we could store the labels and use them to label the flow appropriately). For each branch, we copy the CFG node (of the conditional) from the local store to the global `r` and recursively invoke Ψ_{node}: this means that Ψ_{node} will create a CFG of the branch, and send a flow from the conditional to the source-node of the branch CFG. Finally, Ψ_{cond} stores into `r` the traversers returned from applying Ψ_{node} to the branches, and also returns these.

Finally, Ψ_{block} iterates over all the children (specifically statements and nested blocks) of the syntactic blocks and sets up the flows between them. It first stores the current CFG node into `r`, as this will be the predecessor node for the CFG of the first child. In case there are no children (the current block is an empty block), the subtraversal terminates early. Children are traversed in ascending order and per Rule 5 σ_{flatMap} ensures that each child is fully processed before we start processing the next. After Ψ_{node} was invoked for a child, we store all returned traversers into `r` as these will be the preceding CFG nodes for the next child. The only traversers we want to return are the traversers returned from the processing of the last child (these will be the predecessors of the continuation CFG). For this reason, each child is mapped to a collection of all its return nodes (using σ_{fold}): the traversers after σ_{flatMap} will be collections of CFG nodes (not just nodes). Then, σ_{tail_1} will keep only the last collection, which is then unfolded so that we return the CFG nodes instead of a collection. Note that these are also stored in `r`.

4.3 Proving the algorithm

$$\dfrac{\dfrac{\vdots}{\{\Gamma_0;\ c_1\}\quad \text{icontrol}\quad \{\Gamma_1;\ \Box\}} \qquad \dots \qquad \dfrac{\dfrac{\vdots}{\{\Gamma_{n-1}^{\mathtt{r}=[e_n]};\ b_n\}\quad \text{node}\quad \{\Delta_n^{\mathtt{r}=R};\ \Box\}} \qquad \dfrac{\vdots}{\{\Delta_n^{\mathtt{r}=R};\ c_n\}\quad \text{outE}_{\mathtt{exit}} \rightsquigarrow \dots\quad \{\Gamma_n;\ \Box\}}}{\{\Gamma_{n-1};\ c_n\}\quad \text{icontrol}\quad \{\Gamma_n;\ \Box\}}}{\{\Gamma_0;\ \varnothing\}\quad V_{\mathtt{ControlDeclaration}} \rightsquigarrow \text{sEffect}(\text{icontrol})\quad \{\Gamma_n;\ \Box\}}$$

Figure 3: Structure of the proof tree

In this section, we describe the idea of proving the correctness of our CFG extraction algorithm in Section 4.2. Using our formalisation of Gremlin semantics in Section 4.1, we can formally prove that each traversals in the algorithm will result in a state, that can be shown to guarantee correctness inductively. Despite operating with non-linear data structures, such as graphs, the recursive top-down traversal of ASTs guarantees termination and allows us to use structured correctness proofs. We include a complete, semi-formal proof tree in Appendix A, together with an informal description.

The claim of correctness is that in any Γ_0 initial state – containing the correct AST of each control declaration c_i –, the algorithm ultimately produces a Γ_n state, that contains also the correct CFG of each c_i control declaration.

Figure 3 formally depicts the beginning of the proof, including an inductive $(n-1) \longrightarrow n$ step. To prove the claim, we construct a proof tree, which consist of explicitly (formally) writing out the preconditions and postconditions (effects) of each steps of the algorithm, as dictated by the formal Gremlin semantics.

In the root of the proof tree, we insert the claim itself: starting from Γ_0 and an empty traverser list, the algorithm should lead to Γ_n and an arbitrary traverser list (denoted using the wildcard or placeholder symbol $\square$). As per Rule 5, during this step the c_i syntax nodes of each P4 control declarations are stacked into the traverser list, and Ψ_{icontrol} is applied to each of them one by one. Since this step is using induction, the proof tree depicts only the processing of the last declaration (c_n). The inductive assumption is that the former $n-1$ applications of traversal Ψ_{icontrol} result in a Γ_{n-1} state that is already correct, apart from missing the CFG of the nth declaration. For the last Ψ_{icontrol} to be correct, we need to prove that Ψ_{node} is correct, starting from $\Gamma_{n-1}^{r=[e_n]}$ (i.e. the state we get from Γ_{n-1} by storing CFG entry e_n in registry $\texttt{r}$ as per the first steps of Ψ_{icontrol}). Given that Ψ_{node} results in some state $\Delta_{n-1}^{r=R}$ (where R is a variable, denoting – in case Ψ_{node} is correct – the set of return points), we also need to prove that in this state, the last steps of Ψ_{icontrol} (i.e. the steps that link return points to the exit node) result in the the expected Γ_n. In turn, both propositions can be proved by building the proof tree further. Due to its technical nature, we do not include complete proof tree here, but for our readers interested in the details, we include it – together with an informal description – in Appendix A.

5 Conclusion

Summary In this work, we defined deep CFGs, and presented a CFG extraction algorithm that superimposes CFGs over ASTs inside a Gremlin graph database: this way all information of the AST is at hand during CFG traversals, and can be readily accessed through a uniform graph interface. We already rely on deep CFGs for code generation in our latest paper on P4 cost analysis [11]. There, we used the CFG to transform P4 code to a custom, low-level instruction language which is then passed to a sophisticated probabilistic model checker tool. In the implementation of that transformation, we simply traverse the CFG, and at each node, we follow the association links in the graph to collect further information needed to create instructions from the node. Our other goal in the current paper was to explore how the expressive power of Gremlin can be used for specifying and proving fairly complex static analysis procedures. For this reason, we formalised our algorithm as an executable Gremlin query, and used the formal semantics of Gremlin to formally prove the correctness of our CFG extraction algorithm.

Limitations We highlight two limitations of this CFG extraction algorithm. First, our Gremlin formalisation relies on (shallow) recursion to realise depth-first traversal. Recursive queries are made possible only by host language features such as lambdas, and these are not serialisable. This means that in a client-server environment this query has to be stored on the server-side as a stored procedure. Second, complex Gremlin traversals are – at least in our experience – not easy maintain. For example, the graph query in Figure 2 has to keep track of program state as it stores return points in the global register `r`. If a developer intends to extend the algorithm for handling further P4 language elements, they have to understand how this register is used in the algorithm, in order to make sure not to cause unintended side effects. For production environments, a simpler – although less efficient – approach is to iterate over the AST in multiple stages, e.g. first discovering the R return points of each node n, and in a second iteration link R to the continuation of n. This way the developer can either store global state in the host language (arguably more suitable for handling global state) or eliminate it entirely by storing intermediate information in the graph. Even without considering state and side effects, a sequence of simple, small queries is usually much easier to read, test, and debug.

Future work We started to utilise the deep CFG concept in this paper in the code generation phase of our model checking-based cost analysis [11]. One missing element for this is intraprocedural control flow (i.e. translating function calls), which we currently handle with an ad-hoc solution. One advantage of the CFG being superimposed on the AST is that this element can be added in as a separate analysis on the AST, and we can reach this information in every traversal. We would also like to explore more of the possibilities this opens up, including graphs resulting from data flow analysis (see Section 2) as well.

References

[1] Aho, A., Lam, M., Sethi, R., and Ullman, J. *Compilers: Principles, Techniques, and Tools (2nd Edition).* Addison Wesley, 2006.

[2] Amighi, A., de C. Gomes, P., Gurov, D., and Huisman, M. Sound control-flow graph extraction for Java programs with exceptions. In Eleftherakis, G., Hinchey, M., and Holcombe, M., editors, *Software Engineering and Formal Methods*, pages 33–47, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. DOI: 10.1007/978-3-642-33826-7_3.

[3] Angles, R. and Gutierrez, C. Survey of graph database models. *ACM Computing Surveys*, 40(1), 2008. DOI: 10.1145/1322432.1322433.

[4] Appel, A. SSA is functional programming. *SIGPLAN Not.*, 33(4):17–20, 1998. DOI: 10.1145/278283.278285.

[5] Baier, C. and Katoen, J.-P. *Principles of Model Checking (Representation and Mind Series)*. ISBN: 978-0-262-02649-9. The MIT Press, 2008.

[6] Birnfeld, K., da Silva, D., Cordeiro, W., and de França, B. P4 switch code data flow analysis: Towards stronger verification of forwarding plane software. In *Proceedings of the IEEE/IFIP Network Operations and Management Symposium*, page 1–8. IEEE Press, 2020. DOI: 10.1109/NOMS47738.2020.9110307.

[7] Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G., and Walker, D. P4: Programming Protocol-independent Packet Processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, 2014. DOI: 10.1145/2656877.2656890.

[8] Dumitrescu, D., Stoenescu, R., Negreanu, L., and Raiciu, C. Bf4: Towards bug-free P4 programs. In *Proceedings of SIGCOMM'20*, page 571–585, New York, NY, USA, 2020. Association for Computing Machinery. DOI: 10.1145/3387514.3405888.

[9] Lukács, D., Pongrácz, G., and Tejfel, M. Are graph databases fast enough for static P4 code analysis? In *Proceedings of the 11th International Conference on Applied Informatics*, pages 213–223. CEUR Workshop Proceedings, 2020. URL: http://ceur-ws.org/Vol-2650/#paper22.

[10] Lukács, D., Pongrácz, G., and Tejfel, M. Control flow based cost analysis for P4. *Open Computer Science*, 11:70–79, 2020. DOI: 10.1515/comp-2020-0131.

[11] Lukács, D., Pongrácz, G., and Tejfel, M. Model checking-based performance prediction for P4. *Electronics*, 11(14), 2022. DOI: 10.3390/electronics11142117.

[12] Nielson, H. and Nielson, F. *Semantics with Applications: A Formal Introduction*. John Wiley & Sons, Inc., USA, 1992.

[13] P4 Language Consortium. basic_routing-bmv2.p4, a small test case for the official P4 reference compiler, P4C, 2018. URL: https://github.com/p4lang/p4c/blob/master/testdata/p4_16_samples/basic_routing-bmv2.p4.

[14] Rodriguez, M. The gremlin graph traversal machine and language (invited talk). *Proceedings of the 15th Symposium on Database Programming Languages*, 2015. DOI: 10.1145/2815072.2815073.

[15] Söderberg, E., Ekman, T., Hedin, G., and Magnusson, E. Extensible intraprocedural flow analysis at the abstract syntax tree level. *Sci. Comput. Program.*, 78(10):1809–1827, 2013. DOI: 10.1016/j.scico.2012.02.002.

[16] Tóth, M. and Bozó, I. Building dependency graph for slicing Erlang programs. *Periodica Polytechnica Electrical Engineering*, 55(3-4):133–138, 2011. DOI: 10.3311/pp.ee.2011-3-4.06.

Appendix A Proof of correctness

In this section, we give a semi-formal proof for the correctness of our CFG extraction algorithm in Section 4.2. Using our formalisation of Gremlin semantics in Section 4.1, we formally prove that each traversals in the algorithm will result in a state, that can be shown to guarantee correctness inductively. Despite operating with non-linear data structures, such as graphs, the recursive top-down traversal of ASTs guarantees termination and allows us to use structured correctness proofs. We depict these formal proofs as proof trees in Figures 4, 5, 6, and 7. We describe the contents of these proof trees, and give the informal induction steps in the course of this section.

In the proof trees, we heavily rely on the notation introduced in the previous sections (especially the semantics of Gremlin). In addition to that, we utilize a small number of additional notations: these are related to the semantics of individual Gremlin steps, or to the proof tree syntax.

Notations. Earlier, we used Γ and its subscripted variants to denote the state of global memory: in the proof, we also use Δ and $\tilde{\Delta}$ to denote intermediate memory states. In the proofs, we rarely write out the state explicitly, rather we treat it as a set of statements which are true for the state. For example, we write $\Gamma^{\,\mathtt{r}=R}$ to highlight the assumption that in Γ, the value of register `r` is R. `synB`, `newB`, `exit` are referring to the local store keys assigned in CFG algorithm. In order to not to waste variable names, we use the "wildcard" variable $\Box$ as a placeholder, to denote intermediate values (usually traverser lists) that are not important (not used elsewhere). Since it is a wildcard symbol, two occurrences of $\Box$ may contain different values.

In the proof tree leafs, ✓ means that an axiom has been reached (the tree branch has been proved); numbers between parentheses (e.g. (1), (2), (3)) mean that the proof is continued in another proof tree (with the number in its root); dots (...) mean some steps were omitted (we elaborate these in the explanations). Note that in some cases we compressed multiple steps into one movement, specifically where effects of the individual steps were simple (e.g. in case of $\sigma_{outE} \rightsquigarrow \sigma_{inV}$).

As noted before, σ_{coalesce} (Rule 6) is a higher-order step (it is parameterised with sub-traversals), that returns the traversers from the first successful sub-traversal. We use the $\underset{\rightarrow}{\vee}$ shorthand to denote its semantics: given Ψ,Ψ' traversals, the proposition $(\{\Delta_0; T_0\}\Psi\{\Delta_n; T_n\}) \underset{\rightarrow}{\vee} (\{\Delta_0; T_0\}\Psi'\{\Delta_n; T_n\})$ is solved for $(\Delta_0,\, T_0,\, \Delta_n,\, T_n)$ either by solving the left-hand side operand of $\underset{\rightarrow}{\vee}$, or in case this would result in $T_n = \varnothing$, then by solving the right-hand side operand.

In the rest of this section, we go through each of the traversals defined in Section 4.2, state its correctness and prove that claim. In each of the proofs, we refer to the corresponding formal proof tree, include a descriptive commentary to explain and clarify the proof tree contents, and then finish the proof with an informal argument about satisfaction of the requirements posed by the proof tree. Our first claim concerns Ψ_{control} and Ψ_{icontrol}. The precondition and postcondition posed by this claim refers to our definition of ASTs and control flow graphs in Section 3.1.

Claim 1. *Given that Γ_0 contains a correct AST, together with previously prepared entry and exit nodes (e_i and f_i) for each control declarations c_i in the tree, Γ_n contains for each of c_i in the tree the CFG of the control declaration, rooted in the entry node e_i and all its return points linked to the exit node f_i.*

Proof. Figure 4 formally depicts the inductive $((n-1) \longrightarrow n)$ step of this proof. The rest of this paragraph is commentary for that diagram. In short, we process each control declaration one after the other, and for each, we run Ψ_{node} and link its returned return points to the exit node. In detail, the syntax nodes of each P4 control declarations are stacked into the traverser list, and Ψ_{icontrol} is applied to each of them one by one (as per Rule 5). Each application is expected to fill Γ_0 with the CFGs of each declaration c_i, ultimately reaching Γ_n containing all the CFGs. The final contents of the traverser list is irrelevant (we denote it with the "wildcard" variable $\Box$ that can match anything). Since this is the inductive step, the proof tree depicts only the processing of the last declaration (c_n). To process the last declaration, we store the corresponding CFG entry (e_n) into the global variable r (used later in n), and move the traverser to the first syntax block (b_n) of the declaration. We expect side effect traversal Ψ_{node} to solve the rest of the problem, resulting in global state Δ_n, in which global variable r is set to the set of those CFG nodes (R) that are the return points of the CFG built by Ψ_{node}. After that we move the traverser from c_n to the corresponding exit node f_n, save it to path-local name `exit`, replace the traverser list with R (path-local names are preserved) and send links from R to f_n, and the processing ends.

From this, we can see that the inductive step has two requirements. The first requirement is that Ψ_{node} produces the correct CFG of the nth declaration apart from missing the links between the return points and the exit node. The other requirement (base case) is that the former $n-1$ applications of traversal Ψ_{icontrol} result in a Γ_{n-1} state that is already correct, apart from missing the CFG of the nth declaration.

We give the rest of this proof informally. The first requirement (namely that *node* produces $\Delta_n^{\mathtt{r}=R}$ from Γ_{n-1} by creating the CFG of c_n rooted at entry e_n with R containing the return nodes of this CFG in R) is satisfied by Claim 2. To see that the inductive hypothesis (Γ_{n-1} has the correct CFGs for $c_1, \ldots, c_{n-1}$) is satisfied, recognize that c_1 is applied in state Γ_0, and state Γ_0 already has the 0 correct CFGs for $\varnothing$. (For a non-degenerate case, we can build a very similar proof tree for c_1, and see that Γ_1 has the correct CFG for $\{c_1\}$.)

Then, it follows that $\Gamma_n = \Delta_n^{\forall r_1 \ldots r_k \in R:\ ((r_1 \to f_n) \in E, \ldots, (r_k \to f_n) \in E)}$, s.t. r_i are

$$\dfrac{\dfrac{\vdots}{\{\Gamma_0;\ c_1\}\quad \text{icontrol}\quad \{\Gamma_1;\ \Box\}} \qquad \dots \qquad \dfrac{\dfrac{(1)}{\{\Gamma_{n-1}^{\mathtt{r}=[e_n]};\ b_n\}\quad \text{node}\quad \{\Delta_n^{\mathtt{r}=R};\ \Box\}} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\checkmark}{\{\Delta_n^{\forall r_1 \dots r_k \in R:\ ((r_1 \to f_n) \in E, \dots, (r_k \to f_n) \in E)};\ \Box\}\quad \varepsilon \quad \{\Gamma_n;\ \Box\}}}{\{\Delta_n;\ R \times \{\mathtt{exit} = f_n\}\}\quad \text{addE}_{\substack{label\, =\, \mathtt{flow} \\ to\, =\, \mathtt{exit}}}\quad \{\Gamma_n;\ \Box\}}}{\{\Delta_n^{\mathtt{r}=R};\ (c_n,\ \mathtt{exit} = f_n)\}\quad \text{flatMap}(\dots) \rightsquigarrow \dots \{\Gamma_n;\ \Box\}}}{\{\Delta_n^{\mathtt{r}=R};\ c_n\}\quad \text{outE}_{\mathtt{exit}} \rightsquigarrow \dots\quad \{\Gamma_n;\ \Box\}}}{\{\Gamma_{n-1};\ c_n\}\quad \text{icontrol}\quad \{\Gamma_n;\ \Box\}}}{\{\Gamma_0;\ \varnothing\}\quad V_{\mathtt{ControlDeclaration}} \rightsquigarrow \text{sEffect(icontrol)}\quad \{\Gamma_n;\ \Box\}}$$

Figure 4: Proof of Claim 1

$$\dfrac{\dfrac{\dfrac{\dfrac{(2)}{\{\Delta_1; (x, \{\mathtt{synB} = b\})\}\ \text{cond}\ \{\tilde{\Delta}^{\mathtt{r}=R};\ R\}} \quad \underset{\to}{\vee} \quad \dfrac{(3)}{\{\Delta_1; (x, \{\mathtt{synB} = b\})\}\ \text{block}\ \{\tilde{\Delta}^{\mathtt{r}=R};\ R\}} \quad \underset{\to}{\vee} \quad \dfrac{\checkmark}{\{\Delta_1^{\mathtt{r}=[x]}; x\}\ \varepsilon\ \{\tilde{\Delta}^{\mathtt{r}=R};\ R\}}}{\{\Delta_1;\ (x, \{\mathtt{synB} = b\})\}\quad \text{coalesce}(\dots)\quad \{\tilde{\Delta}^{\mathtt{r}=R};\ R\}}}{\{\Delta^{\mathtt{r}=U, x \in V};\ (x, \{\mathtt{synB} = b, \mathtt{newB} = x\})\}\quad \text{sEffect}(\dots) \rightsquigarrow \dots\quad \{\tilde{\Delta}^{\mathtt{r}=R};\ R\}}}{\{\Delta^{\mathtt{r}=U};\ b\}\quad \text{node}\quad \{\tilde{\Delta}^{\mathtt{r}=R};\ R\}} \quad \Delta_1 = \Delta\left(\begin{matrix} x \in V, \\ \forall u \in U : (u \xrightarrow{\mathtt{flow}} x) \in E, \\ (b \xrightarrow{\mathtt{assoc}} x) \in E \end{matrix}\right)$$

$$(1)$$

Figure 5: Proof of Claim 2

return nodes, and f_n is exit node, and so Γ_n is the correct graph for $c_1, \ldots, c_n$. $\square$

We now continue with stating the correctness of traversal Ψ_{node}. While we needed this claim in the proof of Claim 1, we will also use this in proving Claim 3 and 4. For this reason, we state it in a more general form than what is used in Claim 1. We make use of a heuristic, namely the recognition that Ψ_{node} is always called with one traverser. (On the other hand, r may contain multiple nodes: Ψ_{block} loads its results into r and may call Ψ_{node}, and Ψ_{node} can call Ψ_{cond} which is guaranteed to have multiple results.)

The returned traversers of Ψ_{node} are used in Ψ_{block} to return the return points of its last child. The contents of R are used by Ψ_{block} to correctly process the right sibling of b (we also use it in Ψ_{icontrol} to link the exit nodes).

Notice as well that while, at first glance, the proof of Ψ_{node} requires using mutual induction with the proofs of Ψ_{cond} and Ψ_{block} (because of mutually recursive calls), this can be easily eliminated. To linearise the induction, we just have to inline traversals Ψ_{cond} and Ψ_{block} in traversal Ψ_{node}. (Indeed, we only introduced these traversals to increase readability.)

Claim 2. *Given that U contains the predecessor nodes, then Ψ_{node} produces $\tilde{\Delta}^{\mathtt{r}=R}$ from Δ a correct CFG, rooted at some node x of a given syntax block b, links the predecessor nodes to x, and returns the return nodes of this CFG in R, and returns these as traversers as well.*

Proof. Figure 5 formally depicts the inductive $((n-1) \longrightarrow n)$ step of this proof. The rest of this paragraph is commentary for that diagram. Here, we expect that by calling Ψ_{node} in some state $\Delta^{\mathtt{r}=U}$ (that is, a global state where global variable r is set to a set of nodes U) with a traverser pointing to a syntax block b, Ψ_{node} produces the expected $\tilde{\Delta}$ state, with global variable r storing the expected R. We assign the path-local name `synB` to b, create a new CFG node x in the graph, and move the traverser to x in order to assign it to the path-local name `newB`. Then, in a σ_{sEffect}, we link all the predecessors in U to our new x, and set up the association between x and b as well, resulting in global state Δ_1. (Notice that both *block* and *cond* resets r, and *node* never directly calls itself, so since its contents are not used anymore, we can omit it from Δ_1.) Then, we call the σ_{coalesce} step in state Δ_1. As per the description of the $\underset{\rightarrow}{\vee}$ symbol in Rule 6, we call Ψ_{cond} which either correctly creates the rest of this CFG (if b is a conditional), or terminates without side-effects, in which case we call Ψ_{block}. Again, Ψ_{block} either correctly creates the rest of this CFG (if b is a non-empty block), or terminates without side-effects, in which case we expect b to be an empty block, and so our new x is the appropriate return point. In this last case, we modify Δ_1 by storing x in r, it also stays in the traverser list, and the traversal ends.

From this, we can see that the inductive step has three requirements. The first two requirement is that both Ψ_{cond} and Ψ_{block} continues the correct processing of b according to its type, and they result in the expected state $\tilde{\Delta}^{\mathtt{r}=R}$. The third requirement (posed by Ψ_{cond} and Ψ_{block}) is the satisfaction of the base step, i.e. that the last element of the chain is processed correctly. In more precise term, given b is an empty block such that it is the syntactic child of a previously processed syntactic node, and U contains the return points of the CFG resulting from this, then Ψ_{node} ends in the expected state.

We give the rest of this proof informally. The first requirement (namely that in case b is a conditional, *cond* produces $\tilde{\Delta}$ by including in Δ_1 the nodes and edges of the branches and sets their return nodes as return nodes in R, and returns these as traversers as well) is satisfied by Claim 3. Then, the claim is true, since $\{x \in V, \forall u \in U : u \xrightarrow{\mathtt{flow}} x \in E, b \xrightarrow{\mathtt{assoc}} x \in E\} \subset \Delta_1$. The second requirement (namely that in case b is a non-empty block, *block* produces $\tilde{\Delta}$ by including all remaining nodes and edges of the nested statements and blocks in Δ_1, and sets the last nest as the return node in R, and returns these as traversers as well) is satisfied by Claim 4. Then, the claim is true, since $\{x \in V, \forall u \in U : (u \xrightarrow{\mathtt{flow}} x) \in E, (b \xrightarrow{\mathtt{assoc}} x) \in E\} \subset \Delta_1$.

The third requirement (base case) is satisfied, because in every step we get closer to the bottom of the AST (containing only empty blocks or statements), and since in case b is not a conditional – as Ψ_{cond} terminated before doing any side effect –, and b is an empty block – as Ψ_{block} terminated before doing any side effect –, then $\tilde{\Delta} = \Delta^{\mathtt{r}=[x],\ x \in V,\ \forall u \in U:(u \xrightarrow{\mathtt{flow}} x) \in E,\ (b \xrightarrow{\mathtt{assoc}} x) \in E}$, and the returned traverser is x. That is, one CFG node x is created, it is linked from each $u \in U$, and associated with syntax node b, and finally, $R = [x]$ contains the correct return node since the only node x should be the return node. $\square$

With that, we now proceed to state the correctness of Ψ_{cond} and Ψ_{block}, starting with the former one. As mentioned, these proofs seemingly have mutual dependence with the proof of Ψ_{node}, but we can easily reduce mutual induction to linear induction by inlining traversals Ψ_{cond} and Ψ_{block} into traversal Ψ_{node}.

Claim 3. *In case b is a conditional, then cond produces $\tilde{\Delta}^{\mathtt{r}=R}$ by including in Δ the nodes and edges of the branches, sets their return nodes as return nodes in R, and returns these as traversers as well. Otherwise terminates without side-effects.*

Proof. Figure 6 formally depicts the inductive $((n-1) \longrightarrow n)$ step of this proof. The rest of this paragraph is commentary for that diagram. We expect that in some state Δ, with one traverser standing on a CFG x (created earlier by Ψ_{node} for syntactical node b), Ψ_{cond} produces the expected state $\tilde{\Delta}^{\mathtt{r}=R}$. We treat the termination requirement in the informal part of the proof. First, we store in x in the path-local store, also known as sack $(\varsigma(\cdot))$ and set the current traverser to b. (We have to use a path-local store, as the processing of the branches may modify the global stores (e.g. r), and in that case we would lose our reference to x after the processing the branch.) We continue the processing in case *label*$(b) =$ `Conditional` (otherwise the algorithm terminates). In this case, it is always true (by our definition of the AST) that b has two branches: one true-branch starting in syntactical node y, and one false-branch starting in syntactical node n. We move a traverser to each of these nodes (the sack is preserved). In the σ_{flatMap} step, we process the two branches (first the branch of y, then the branch of n) by traversal Ψ_{node} (storing x in r, as Ψ_{node} will have to link x to the produced CFG as predecessor), with the expectation that Δ_1 contains the CFG produced for y, and Δ_2 contains the CFGs both for y and n, and that Ψ_{node} returns y's CFG return points in T_y and n's CFG return points in T_n. (The processing order of the two branches does not matter as long as it is consistent across all applications of the traversal.) After that, we store the return points of both of T_y and T_n in r, and we also return this return points as traversers.

From this, we can see that the inductive step has two very similar requirements regarding the correct processing of y and n by Ψ_{node}. It is worth noting the analysis of the two branches never interferes with each other: r is reseted and *node* will traverse different subtrees of the tree. A possible third requirement of a base case (it is possible that the called Ψ_{node} may call Ψ_{cond} again, but ultimately Ψ_{node} will stop when statements or empty blocks are encountered) was already discussed in the proof of Ψ_{node}.

We give the rest of this proof informally. The first requirement (namely that *node* produces Δ_1 from Δ by creating the CFG of the true branch y, linking it to the (conditional) CFG node x, and returns in traverser list T_y all the return nodes of the CFG, is satisfied by Claim 2. Similarly, the second requirement (namely that *node* produces Δ_2 from Δ_1 by creating the CFG of the false branch y, linking it to the (conditional) CFG node x, and returns in traverser list T_n all the return nodes of the CFG is also satisfied by Claim 2.

Then it follows, that Δ_2 contains CFGs of the branches rooted at x, and $R = T_y \cup T_n$ contains the return nodes of the branches, and returns these as traversers as well.

Regarding the termination requirement of the claim, in case b is not a conditional, the *has* filter fails and the traversal terminates without side-effects. □

Finally, we state the correctness of Ψ_{block}. This is a case where we use induction in two axes: Ψ_{block} as part of a chain initiated from Ψ_{node}, and at the same time, use induction in proving that a sequence of sibling nodes are processed correctly.

Claim 4. *In case b is a non-empty block, assume that block produces $\tilde{\Delta}^{\mathtt{r}=R}$ by including all remaining nodes and edges of the nested statements and blocks in Δ, and sets the last nest as the return node in R, and returns these as traversers as well. Otherwise terminates without side-effects.*

Proof. Figure 7 formally depicts the inductive ($(n-1) \longrightarrow n$) step of this proof. The rest of this paragraph is commentary for that diagram. We expect that in some state Δ, with one traverser standing on a CFG x (created earlier by Ψ_{node} for syntactical node b), Ψ_{block} produces the expected state $\tilde{\Delta}^{\mathtt{r}=R}$. We treat the termination requirement in the informal part of the proof.

First, we store in x in the global store r (we expect this to be modified as children of b are processed and use its contents to create links between the children), and set the traverser list to S, denoting the blocks and statements nested in b (in case there is non, the traversal terminates). Through ordering S we ensure that σ_{flatMap} processes these nested nodes from left-to-right (guaranteed by definition of

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{(1)}{\{\Delta^{\mathtt{r}=[x]}; y\}\ \text{node}\ \{\Delta_1;\ T_y\}}}{\{\Delta; (y, \varsigma(x))\}\ \text{sEffect}(\ldots) \rightsquigarrow \ldots\ \{\Delta_1;\ T_y\}} \quad \dfrac{\dfrac{(1)}{\{\Delta_1^{\mathtt{r}=[x]}; n\}\ \text{node}\ \{\Delta_2;\ T_n\}}}{\{\Delta_1; (n, \varsigma(x))\}\ \text{sEffect}(\ldots) \rightsquigarrow \ldots\ \{\Delta_2;\ T_n\}} \quad \dfrac{\checkmark}{\{\Delta_2^{\mathtt{r}=T_y \cup T_n}; T_y \cup T_n\}\ \varepsilon\ \{\tilde{\Delta}^{\mathtt{r}=R};\ R\}}}{\vdots \qquad\qquad \vdots}}{\{\Delta; (y, \varsigma(x)), (n, \varsigma(x))\}\ \text{flatMap}(\ldots) \rightsquigarrow \ldots\ \{\tilde{\Delta};\ R\}}}{\{\Delta; (b, \{\mathtt{synB} = b\}, \varsigma(x))\}\ \text{has}_{label=\mathtt{Conditional}} \rightsquigarrow \ldots\ \{\tilde{\Delta};\ R\}} \begin{cases} label(b) = \mathtt{Conditional}, \\ y \text{ is true branch}, \\ n \text{ is false branch} \end{cases}}{\{\Delta; (x, \{\mathtt{synB} = b\})\}\ \text{cond}\ \{\tilde{\Delta}^{\mathtt{r}=R};\ R\}}
$$

(2)

Figure 6: Proof of Claim 3

$$
\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\{\Delta_0; s_1\}\ \text{node} \rightsquigarrow \ldots\ \{\Delta_1^{\mathtt{r}=R_1};\ F_1\}} \quad \ldots \quad \dfrac{\dfrac{(1)}{\{\Delta_{m-1}^{\mathtt{r}=R_{m-1}}; s_m\}\ \text{node}\ \{\Delta'_m;\ T_m\}} \quad \dfrac{\dfrac{\checkmark}{\{{\Delta'}_m^{\mathtt{r}=T_m};\ \{T_m\}\}\ \varepsilon\ \{\Delta_m^{\mathtt{r}=R_m};\ F_m\}}}{\vdots}}{\{\Delta_{m-1}; s_m\}\ \text{node} \rightsquigarrow \ldots\ \{\Delta_m^{\mathtt{r}=R_m};\ F_m\}}}{\vdots} \quad \dfrac{\dfrac{\checkmark}{\{\Delta_m^{\mathtt{r}=R_m}; T_m\}\ \varepsilon\ \{\tilde{\Delta}^{\mathtt{r}=R};\ R\}}}{\{\Delta_m; F_1 \cup \ldots \cup F_m\}\ \text{tail}(1) \rightsquigarrow \text{unfold}\ \{\tilde{\Delta};\ R\}}}{\{\Delta_0; S\}\ \text{flatMap}(\ldots) \rightsquigarrow \ldots\ \{\tilde{\Delta};\ R\}}}{\{\Delta^{\mathtt{r}=[x]}; b\}\ \text{outE}_{\{\mathtt{nest},\mathtt{statement}\}} \rightsquigarrow \ldots\ \{\tilde{\Delta};\ R\}} \begin{cases} \Delta_0 = \Delta^{\mathtt{r}=[x]} \\ S \neq \varnothing \\ S \text{ is ordered by id} \end{cases}}{\{\Delta; (x, \{\mathtt{synB} = b\})\}\ \text{block}\ \{\tilde{\Delta}^{\mathtt{r}=R};\ R\}}
$$

(3)

Figure 7: Proof of Claim 4

the AST). σ_{flatMap} starts out in state Δ_0 when it starts processing the first nested node, and finishes in state Δ_m after processing the last nested node.

We will check that the σ_{flatMap} step processes S correctly by induction, and so we only included the proof tree corresponding to the last child (s_m). Here, we expect that given all previous siblings have their CFGs in global state Δ_{m-1} with r storing the return points (R_{m-1}) of the closest previous sibling's CFG, and Ψ_{node} produces Δ'_m containing (and properly linking) all the CFGs of the children, and returns the return points (T_m) of the last child of b as the traverser list. Then this nested traversal stores T_m into r and at the same folds T_m into a single collection-element $F_m = \{T_m\}$ that is returned in the traverser list. After σ_{flatMap} processed everyone, we are in state Δ_m with r storing the return points of the last child of b, and the traverser list containing all the collection-elements return from processing the children. Now, keep only the last traverser (a collection), and unfold this collection again into the individual traversers (still pointing to the return points of the last child's CFG), and return these.

From this, we can see that the inductive step has two requirements. The first – about Δ'_m – is that Ψ_{node} creates a correct node CFG for any child of b, links this CFGs node from the return points of the previous child's CFGs, and returns the return points of the currently processed child's CFG as the traverser list, while also storing it in r. The other requirement (base case) is that the former $n-1$ applications of traversal Ψ_{node} result in a Δ_{m-1} state contains the correct CFGs of all the previous children.

We give the rest of this proof informally. The first requirement (namely that *node* produces Δ'_m from $\Delta_{m-1}^{\mathtt{r}=R_{m-1}}$ by creating the CFG of s_m, linking it to the nodes in R, and returns in traverser list T_m all the return nodes of the CFG) is satisfied by Claim 2. The next requirement is that the inductive hypothesis ($\Delta_{m-1}^{\mathtt{r}=R_{m-1}}$ contains the chain of CFGs produced from the first $m-1$ blocks of b, and R_{m-1} contains the return points of the last CFG in this chain) is satisfied. Again, we may consider the degenerate case (Δ_0, with $R = [x]$ satisfies the hypothesis) or alternatively, we can build a very similar proof tree for s_1, and from the first requirement it follows that Δ_1 has the correct CFG for $\{s_1\}$, with r storing its T_1 return points, and F_1 in the traverser list is T_1 folded.

Then, the claim follows, since $\tilde{\Delta}^{\mathtt{r}=R} = \Delta_m^{\mathtt{r}=R_m} = {\Delta'}_m^{\mathtt{r}=T_m}$, with $R = R_m = T_m$ also being returned.

Regarding the termination requirement of the claim, in case b is an empty block, *outE* maps to $\varnothing$ and the traversal terminates without side-effects. $\square$

Acta Cybernetica **26** (2024) 621–636.

Corner-Based Implicit Patches[*]

Ágoston Sipos[a]

Abstract

Free-form multi-sided surfaces are often defined by side interpolants (also called ribbons), requiring that the surface has to connect to them with a prescribed degree of smoothness. I-patches represent a family of implicit surfaces defined by an arbitrary number of ribbons. While in the case of parametric surfaces describing ribbons is a well-discussed problem, defining implicit ribbons is a different task.

In this paper, we introduce a new representation, *corner I-patches*, where implicit corner interpolants are blended together. Corner interpolants are usually simpler, lower-degree surfaces than ribbons. The shape of the patch depends on a handful of scalar parameters; constraining them ensures continuity between adjacent patches. Corner I-patches have several favorable properties that can be exploited for design, volume rendering, or cell-based approximation of complex shapes.

Keywords: implicit surfaces, multi-sided patches, volumetric data

1 Introduction

Computer Aided Geometric Design focuses on the mathematical representation of complex surface geometries. There is a wide variety of side interpolating multi-sided free-form surfaces in the literature, including both parametric [4, 11, 7, 17, 18] and implicit [1, 6, 16] patches. They are popular in curvenet-based design, as a patchwork of smoothly connected complex N-sided patches can be automatically created from simple ribbon surfaces.

The common concept behind these patches is that *ribbons* are introduced for each side, then *blending functions*, that satisfy prescribed continuity constraints at the boundaries, mix those together. In the case of parametric multi-sided patches, ribbons in most cases are tensor-product surfaces. The blend functions are usually (not always) defined on a polygonal domain and the surface points are calculated as a weighted sum of the well-parametrized points of the ribbons.

[*]This work was supported by the Hungarian Scientific Research Fund (OTKA, No. 124727).

[a]Department of Control Engineering and Information Technology, Faculty of Electrical Engineering and Informatics, Budapest University of Technology and Economics, Hungary, E-mail: asipos@edu.bme.hu, ORCID: 0000-0002-5562-2849

DOI: 10.14232/actacyb.299598

In the case of implicit surfaces, ribbons and blending functions are represented by implicit functions, defined on the whole 3D space. This, however, requires careful construction. An implicit surface, including the ribbons themselves, might interpolate the desired patch boundary, and have a very uneven shape inside the relevant space region at the same time. Sometimes it can have disconnected branches or self-intersections. However, many operations are computationally less expensive when using implicit surfaces, including ray tracing, intersections, point classification, or joining trimmed patches.

For this reason, the polynomial degree of ribbons and blend functions is preferred to be as low as possible. This poses limitations when used in design, as very detailed surfaces cannot be represented with a single patch. Locally defined surface elements are therefore important.

This paper explores the capabilities of a corner-based implicit surfacing scheme, where the patch is constructed by blending corner interpolants.

2 Previous work

The precursor of research on ribbon-based implicit surfaces is Liming's work on interpolating curves [10] and the functional splines [9]. Formally, in the surface equation, the product of the surfaces to be interpolated is used, which, in the case of implicit surfaces, means taking their union. Thus, the information we had while they were separate surfaces is lost. The improved version of the functional spline, the *symmetric functional spline* [6] collects the interpolated surfaces into two categories in its equation, but similarly can take the union of a higher number of them.

In the case of I-patches [16] the surface equation has an arbitrary number of sides appearing separately in the equation. This helps ensure that the surface is consistently oriented and the appropriate sides of the ribbons are joined together. For details, see Section 3. I-patches were applied for polyhedral design [14] and approximating triangular meshes [13] while ensuring geometric continuity.

A different way to approach the problem of interpolating implicit surfaces is to directly solve equations to get a minimal degree surface adhering to point, normal, curve, and normal fence constraints [2]. Doing this on the whole space may lead to high-degree, poor-quality surfaces. A scheme similar to the current work, A-patches [1], prevents this by constraining each surface inside a tetrahedron.

Another aspect of current work relates to the extraction of isosurfaces from discrete data on a regular grid. Generating a mesh is usually performed by derivative methods of Marching Cubes [12]. Direct rendering generally goes through interpolation methods, smooth surfaces can be acquired by tricubic interpolation [8]. There is also a list of enhanced approaches like using a modified (BCC or FCC) grid structure [15] or storing gradient values and approximating the isosurface by Taylor polynomials [3].

The potential representation of isosurfaces extracted on a grid by patches was investigated in [5]. First, a boundary curve network with Hermite interpolation is created, then the surface is represented using multi-sided parametric patches.

3 Preliminaries

Implicit surfaces are constant isosurfaces of real-valued functions defined on the 3D space. Usually the zero-isosurface is used: for a function $f : \mathbb{R}^3 \to \mathbb{R}$ the surface is $\{(x, y, z) \in \mathbb{R}^3 \mid f(x, y, z) = 0\}$. In the following, capital letters in the formulae will mean implicit functions ($\mathbb{R}^3 \to \mathbb{R}$), but the function arguments (x, y, z) will be omitted for readability.

Ribbon-based implicit surfaces are usually described in the following way. For each side, there is a given surface R_i, that the surface should smoothly connect to, with a given order of continuity. Then, there is a fixed equation of the patch, combining the R_i-s and other defining surfaces.

In the case of I-patches, which are the basis of the current research, the equation is

$$\sum_{i=1}^{n} \left(w_i R_i \prod_{\substack{j=1 \\ j \neq i}}^{n} B_j^k \right) + w_0 \prod_{j=1}^{n} B_j^k = 0, \tag{1}$$

where

- n is the number of sides
- R_i are the ribbons, one for each side, to which the patch connects
- B_i are the *bounding surfaces*, whose intersection curves with the corresponding R_i define the boundaries of the patch
- $0 \neq w_i \in \mathbb{R}$ are scalar parameters and $2 \leq k \in \mathbb{N}$ is an integer parameter determining the degree of continuity

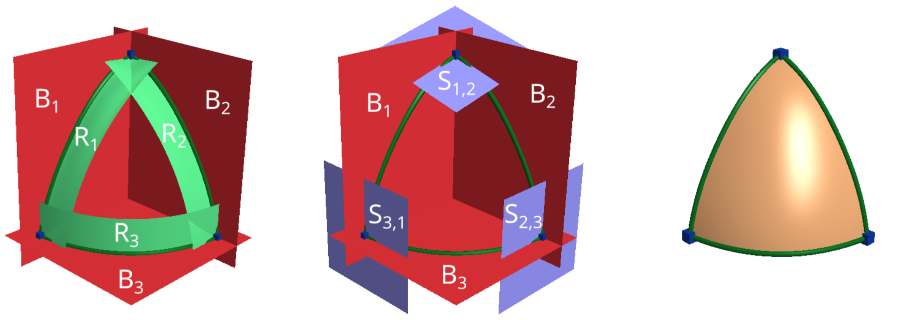
The patch connects with G^{k-1} continuity to the ribbons along the bounding surfaces, as shown in [16]. It has also been proven that the I-patch represents a consistent distance function with inside and outside, in case of well-chosen signs of w_i-s [14]. See Figures 1a and 1c for an example.

In the following, we will use $k = 2$ to keep the polynomial degree as low as possible, and in this paper, we discuss patches with G^1 continuity.

4 Corner I-patch

4.1 Basic equation

A corner I-patch is composed of corner interpolants $S_{1,2}, S_{2,3}, ..., S_{n,1}$ and bounding surfaces $B_1, B_2, ..., B_n$ (neither of them coincides with another one), such that $S_{i,i+1}$ denotes the corner interpolant between the ith and the $(i + 1)$th boundaries.



(a) Input (ribbons and bounding surfaces) for an **I-patch**

(b) Input (corners and bounding surfaces) for a **corner I-patch**

(c) Approximate shape of resulting patches

Figure 1: I-patch and corner I-patch

Then, the equation of the corner I-patch is

$$\sum_{i=1}^{n}\left(w_{i,i+1}\cdot S_{i,i+1}\cdot\prod_{\substack{j=1\\ j\neq i, j\neq i+1}}^{n} B_j^2\right)+\sum_{i=1}^{n}\left(w_i\cdot\prod_{\substack{j=1\\ j\neq i}}^{n} B_j^2\right)+w\prod_{i=1}^{n} B_i^2=0, \qquad (2)$$

where the $w_{i,i+1}$ scalars can be merged into $S_{i,i+1}$, as multiplying with a nonzero number does not change the implicit isosurface, only its distance metric. See an example of corner interpolants and bounding surfaces in Figure 1b.

Some important properties of this representation are:

- In each corner, the patch connects with G^1 continuity to the corner interpolants. (This means that the gradient vectors of the surface have the same direction as the gradients of the interpolants there.)
- Along the ith boundary, the shape of the surface does not depend on w and w_j for $j \neq i$.

The $w_{i,i+1}$ coefficients will be called corner coefficients, w_i-s are the side coefficients and w is the central coefficient.

4.2 Comparison to (side-based) I-patches

A disadvantage of I-patches is that their gradient is a zero vector in the corner points. This may lead to poor surface quality and generally should be avoided. However, the gradient of the corner I-patch can easily be proven to be the gradient of the corner interpolant times a nonzero number.

The corner I-patch along the ith boundary connects smoothly to the implicit surface

$$S_{i-1,i} \cdot B_{i+1}^2 + S_{i,i+1} \cdot B_{i-1}^2 + w_i \cdot B_{i-1}^2 \cdot B_{i+1}^2 = 0. \tag{3}$$

This itself is a 2-sided I-patch. Corner I-patches are thus similar (but not equivalent) to I-patches defined by ribbons that are themselves I-patches. (Such surfaces were described in [14].) This is because the I-patch defined by the ribbons in Equation 3 would be

$$\begin{aligned}\sum_{i=1}^{n}\left(S_{i,i+1}(B_{i+1}^2 B_{i-1}^2 + B_i^2 B_{i-2}^2)\prod_{\substack{j=1\\ j\neq i, j\neq i+1}}^{n} B_j^2\right) +\\ +\sum_{i=1}^{n}\left(w_i B_{i-1}^2 B_{i+1}^2 \prod_{\substack{j=1\\ j\neq i}}^{n} B_j^2\right) + w\prod_{i=1}^{n} B_i^2 = 0,\end{aligned} \tag{4}$$

which is not equivalent to Equation 2. Indeed, the factor $(B_{i+1}^2 B_{i-1}^2 + B_i^2 B_{i-2}^2)$ causes the I-patch's gradient to be zero at the corner points. Accordingly, corner I-patches have a lower degree of $2n$, as opposed to the $2n + 2$ for this kind of I-patches.

4.3 Setting coefficients

The w_i and w parameters can be set in a process similar to I-patches [16] forcing the patch to interpolate one point on each boundary and one point in the interior of the patch. As the shape of the surface on the ith boundary depends only on w_i, each of those can be set separately, and finally, w can be set to interpolate an interior point. I.e.:

$$w_i := -\frac{S_{i-1,i}(\mathbf{p}_i) \cdot B_{i+1}^2(\mathbf{p}_i) + S_{i,i+1}(\mathbf{p}_i) \cdot B_{i-1}^2(\mathbf{p}_i)}{B_{i-1}^2(\mathbf{p}_i) \cdot B_{i+1}^2(\mathbf{p}_i)} \tag{5}$$

$$w := -\frac{\sum_{i=1}^{n}\left(S_{i,i+1}(\mathbf{p}_0) \cdot \prod_{\substack{j=1\\ j\neq i, j\neq i+1}}^{n} B_j^2(\mathbf{p}_0)\right) + \sum_{i=1}^{n}\left(w_i \cdot \prod_{\substack{j=1\\ j\neq i}}^{n} B_j^2(\mathbf{p}_0)\right)}{\prod_{i=1}^{n} B_i^2(\mathbf{p}_0)}, \tag{6}$$

where $\mathbf{p}_i$ is a point on the ith side, $\mathbf{p}_0$ is a point in the interior to interpolate. The parameters can be computed in this order, i.e. first all w_i, then w.

See examples later, in Figure 4.

4.4 Limitations

When connecting neighboring patches with geometric continuity, we need them to coincide at their common boundary in both a positional and a differential sense. As the corner I-patch along B_i connects to the surface defined by Equation 3, the patch on the other side of B_i also has to connect to it. This, however, only happens if B_{i-1} and B_{i+1} are identical to the corresponding bounding surfaces for the other patch.

This is not easily fulfilled when creating a general topology patchwork, but it is straightforward if the space is subdivided by planes, creating finite volume cells. Any such cell structure could theoretically work with corner I-patches, however, the most practical and useful is a regular grid of cubes.

5 Corner I-patch with multiple loops

5.1 Motivation and equation

In some cases, an isosurface must be represented by several disjoint surface elements. In Marching Cubes [12] for example, 7 of the 15 basic configurations result in a surface represented by more than one polygon. These surface elements could be represented separately, but in an implicit representation, it is advantageous to have the same implicit function on a well-defined 3D volume, otherwise, the piecewise implicit function would likely have discontinuities.

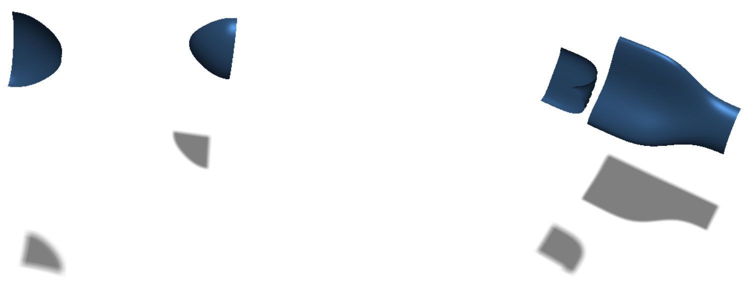
Fortunately, corner I-patches are capable of achieving this with a little modification. Consider m separate boundary loops where the lth of those is n_l-sided and for each of them the previously defined corner interpolants $S^l_{1,2}, S^l_{2,3}, ..., S^l_{n_l,1}$ and the bounding surfaces $B^l_1, B^l_2, ..., B^l_{n_l}$. Then the new equation is

$$\begin{aligned}\sum_{l=1}^{m}\sum_{i=1}^{n_l}\left(w_{l,i,i+1}\cdot S^l_{i,i+1}\cdot\prod_{k=1}^{m}\prod_{\substack{j=1\\ k\neq l\vee(j\neq i\wedge j\neq i+1)}}^{n_k}(B^k_j)^2\right)+\\ +\sum_{l=1}^{m}\sum_{i=1}^{n_l}\left(w_{l,i}\cdot\prod_{k=1}^{m}\prod_{\substack{j=1\\ k\neq l\vee j\neq i}}^{n_k}(B^k_j)^2\right)+w\prod_{l=1}^{m}\prod_{i=1}^{n_l}(B^l_i)^2=0.\end{aligned} \tag{7}$$

The meaning of this is that for each corner interpolant, the bounding surfaces not multiplied to it are the two ones beside it, corresponding to the next and previous boundaries of the patch. A simple multiloop surface can be seen in Figure 2a, with two loops each composed of three corners.

5.2 Coinciding bounding surfaces

In a multiloop setting, especially if working in a grid of cubes, some bounding surfaces will likely coincide. Consider, for example, the configuration in Figure 2b



(a) Two 3-sided loops (b) A 3- and a 4-sided loop

Figure 2: Multiloop patches

(a configuration of Marching Cubes), where one of the boundings for the 3-sided loop coincides with one of those of the 4-sided loop. The problem with that is that when two bounding surfaces coincide, they will be in all the members of the weighted sum and thus can be factored out from the equation.

With regard to I-patches, a modified equation for this problematic case has been proposed in [14], however, it takes advantage of the 1-to-1 relation between ribbons and bounding surfaces which is not applicable to corner patches.

In this paper, the following solution is proposed. When computing the product of the bounding surfaces not neighboring the respective corner, omit those as well which coincide with one of the neighboring ones.

The formalized equation is:

$$\begin{aligned}&\sum_{l=1}^{m}\sum_{i=1}^{n_l}\left(w_{l,i,i+1}\cdot S^l_{i,i+1}\cdot\prod_{k=1}^{m}\prod_{\substack{j=1\\B^k_j\neq B^l_i,\,B^k_j\neq B^l_{i-1}}}^{n_k}(B^k_j)^2\right)+\\&+\sum_{l=1}^{m}\sum_{i=1}^{n_l}\left(w_{l,i}\cdot\prod_{k=1}^{m}\prod_{\substack{j=1\\B^k_j\neq B^l_i}}^{n_k}(B^k_j)^2\right)+w\prod_{l=1}^{m}\prod_{i=1}^{n_l}(B^l_i)^2=0.\end{aligned}\tag{8}$$

What this means is that each corner is multiplied with the product of all bounding surfaces (regardless of which loop they are in) unless they coincide with one of its neighbors. Side components for a given side are the product of all bounding surfaces, other than those coinciding with the bounding surface representing that side. This formulation keeps the properties highlighted about the original equation (G^1 continuity to corner interpolants, and sides being only affected by the corresponding coefficient).

A practical implementation for evaluating this is to store all bounding surfaces in an array, and only store indices for them in the loops. When computing the products, we only have to check for the non-equality of the indices.

6 Use in cell structures

When used in regular cell structures, the $S_{i,j}$ and B_i surfaces are all planes. Thus, the patch itself is a polynomial surface, with a degree of twice the number of sides (see Equation 2).

In Figures 3 and 4 corner patches are defined inside the unit cube. Figure 3a is a 3-sided surface near a corner of the cube. Figure 3b is a 6-sided patch that intersects all faces of the cube.

In Figure 4, three patches with the same corners but different coefficients can be seen. They are set using the algorithm presented in Section 4.3, i.e. on each side and in the interior one point is fixed and respective coefficients are calculated from them. Between Figures 4a and 4b, two side points; between Figure 4a and 4c, the interior point is changed. The numerical data for these patches can be found in Table 1.

The possible topological configurations are similar to those of Marching Cubes [12]. The multiloop scheme also works for topologically disjoint isosurfaces (Figure 3c).

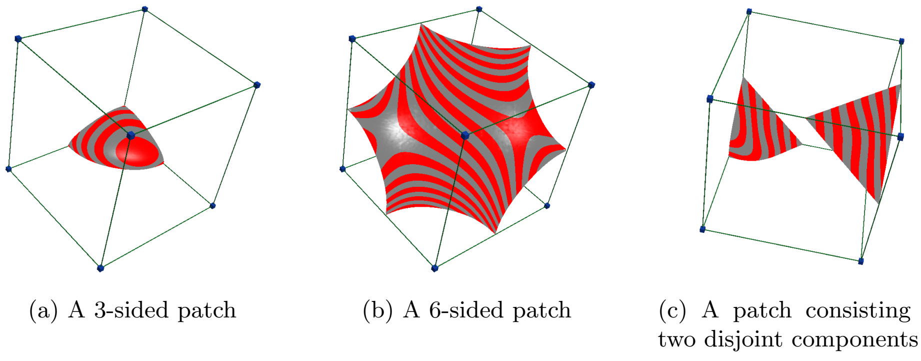
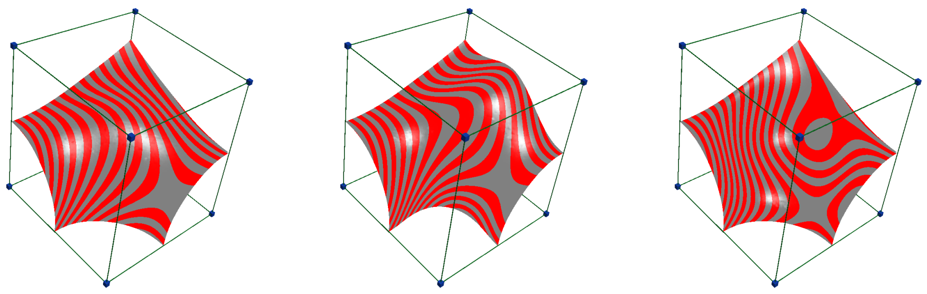

(a) A 3-sided patch (b) A 6-sided patch (c) A patch consisting of two disjoint components

Figure 3: Corner I-patches inside the unit cube



(a) Base patch (b) Changing boundaries (c) Changing the interior

Figure 4: Corner I-patches with the same corner interpolants and different coefficients

Table 1: Points and coefficients for the patches in Figure 4. All patches are in the $[0;1]^3$ cube. Differences from Patch #1 are **bold**.

Corner points				
$[0, 0.5, 1]$	$[0, 0.8, 0]$	$[1, 0.5, 0]$	$[1, 0, 0.5]$	$[0.5, 0, 1]$
Patch #1				
Side points				
$[0, 0.6, 0.5]$	$[0.5, 0.6, 0]$,	$[1, 0.35, 0.35]$	$[0.65, 0, 0.65]$	$[0.35, 0.35, 1]$
Side coefficients				
0.6	0.6	-2.34	2.34	-2.34
Interior point:	$[0.5, 0.5, 0.5]$	Central coefficient:		-7.77
Patch #2				
Side points				
$[0, 0.6, 0.5]$	$[0.5, \mathbf{0.8}, 0]$	$[1, 0.35, 0.35]$	$[\mathbf{0.55}, 0, \mathbf{0.55}]$	$[0.35, 0.35, 1]$
Side coefficients				
0.6	**2.2**	-2.34	**0.45**	-2.34
Interior point:	$[0.5, 0.5, 0.5]$	Central coefficient:		**−8.94**
Patch #3				
Side points				
$[0, 0.6, 0.5]$	$[0.5, 0.6, 0]$,	$[1, 0.35, 0.35]$	$[0.65, 0, 0.65]$	$[0.35, 0.35, 1]$
Side coefficients				
0.6	0.6	-2.34	2.34	-2.34
Interior point:	$[0.5, \mathbf{0.2}, 0.5]$	Central coefficient:		**55.83**

7 Discussion

7.1 Methodology

The examples were generated in the following way. The input was a voxel array of floating point isovalues. Then, for each cell, based on the eight isovalues in the corner, polyline loops were generated from the estimated edge intersections. Ambiguities (when on a face all four edges have an intersection) were resolved by minimizing the sum of distances between the two pairs. This resulted in one or more loops of closed polylines.

Then, in each of the isovertices, a plane was introduced, with its normal pointing towards the positive cell corner. From those, and the cube's faces, corner I-patches were generated. Coefficients for the corner components were set to 1, and side coefficients were calculated with the triangle rule or the tetragon rule (see Appendix A). The central coefficient was set to 0.

Rendering was done using raycasting, neighboring patches have an alternating texture color. Phong shading is used, and normal vectors are calculated from the exact gradients of the surface.

7.2 Single cell examples

The flexibility of the corner I-patch representation can be shown in Figure 5 where patches are generated from the corner sign settings corresponding to the 14 non-

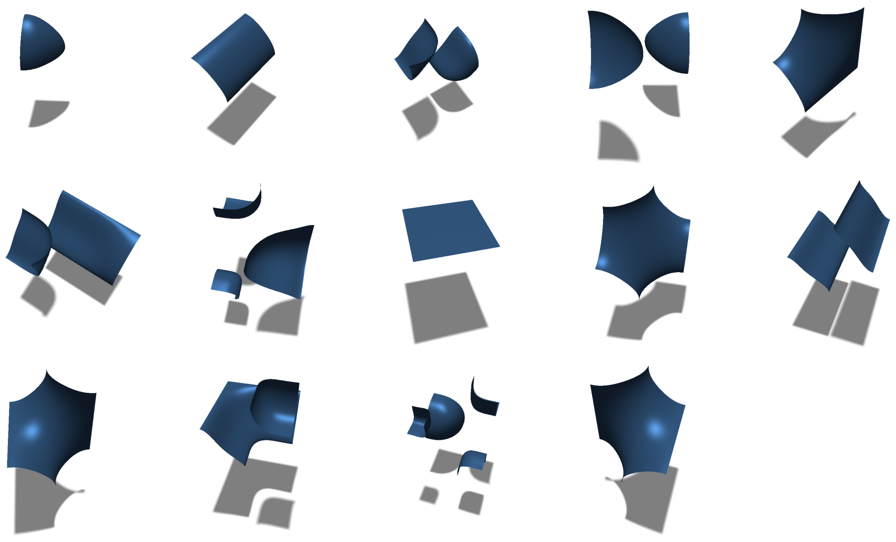


Figure 5: The 14 non-empty basic configurations of Marching Cubes represented with corner I-patches (in the same order as in [12])

empty basic configurations of Marching Cubes. The patches were created using the formulation in Equation 8, as in cases where there were two boundary curves on the same side of the cube, boundary surfaces coincided.

7.3 Multi-cell examples

In these examples, a sparse (9×9×9) voxel array was created and the patches were automatically generated from it. In the first example (Figure 6) two sphere-like objects can be seen which are close to each other. Notice that the brown surfaces at their closest points are represented with the same corner I-patch. In the second example (Figure 7) a voxel value was modified, extending the volume of the bigger object. In Figure 8 a hole was put into the object by modifying isovalues in a line.

7.4 Examples with exact vertices and gradients

Here, the scheme was modified so that when introducing the corner planes, an exact implicit function is used for both exactly calculating the isovertex and using an exact gradient. This can be useful in cases where evaluating the original functions would be very costly but approximating them with piecewise polynomial surfaces could bring a reduction in both storage and computation costs.

In Figures 9 and 10 an ellipsoid can be seen with a lower and a higher resolution cell structure. It can be observed that although the boundary curves approximate

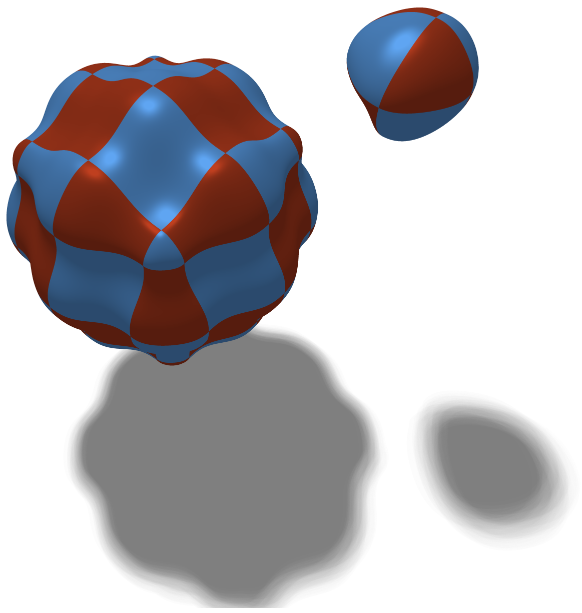


Figure 6: Disjoint surfaces generated from voxel data

the original surface well, the interior of some surfaces is less smooth. Optimization of the coefficients is therefore an important area of future research.

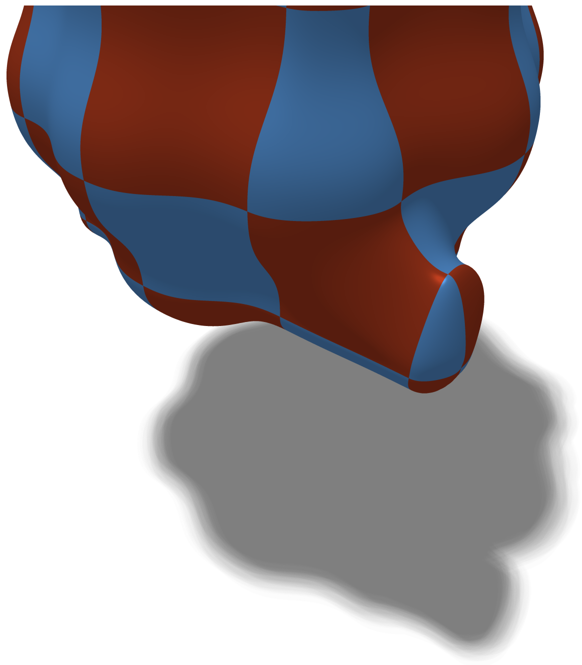


Figure 7: Enlarging the object by modifying a voxel value

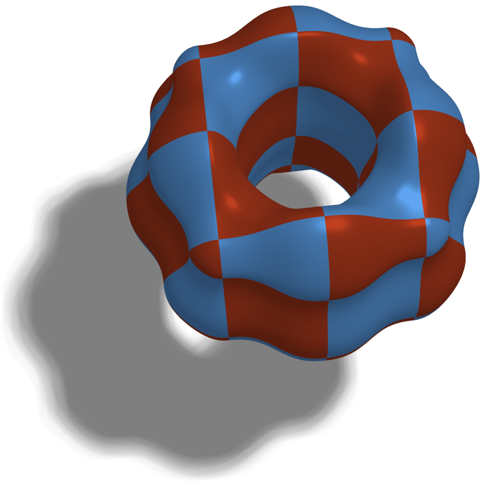


Figure 8: Putting a hole inside the object by modifying voxel values

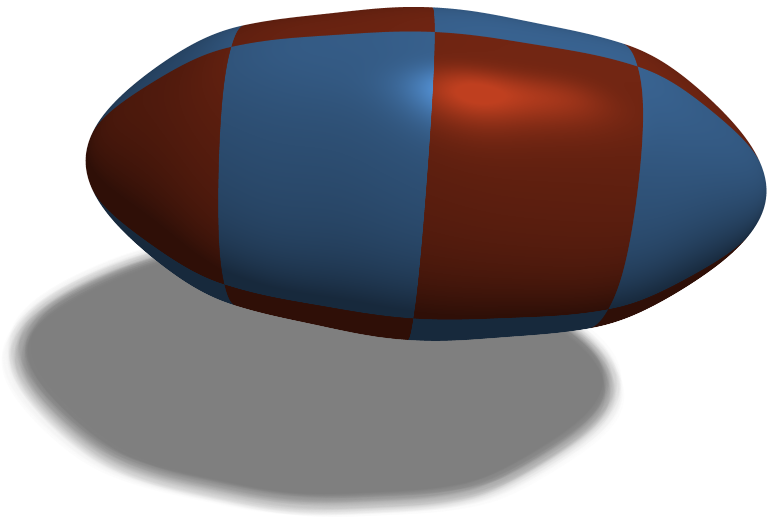


Figure 9: Ellipsoid with lower resolution

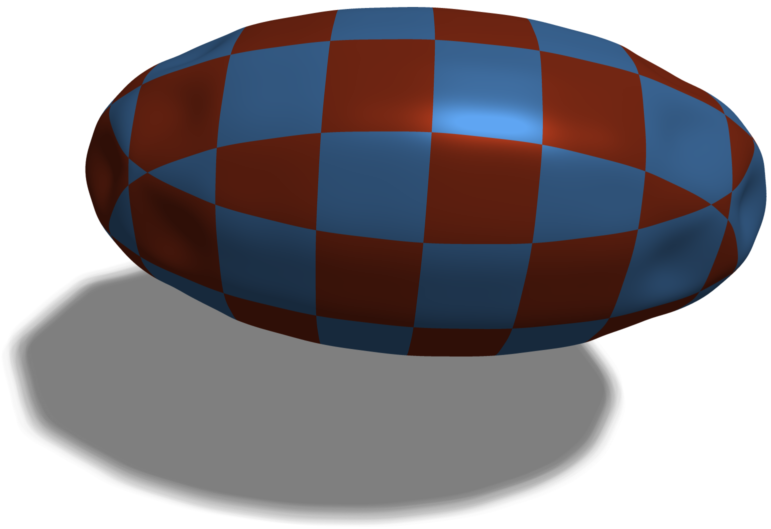


Figure 10: Ellipsoid with higher resolution

8 Conclusion

We have presented corner I-patches, a class of implicit surfaces with several advantages over existing representations. Patches can be defined by combining only planes, in contrast to the relatively more complicated ribbons needed to define I-patches. They can be used to create complex piecewise surfaces. Unlike I-patches, corner I-patches have no singularities in the corners. Several scalar coefficients can be used to optimize a target function on the patch for approximation or surface fairing purposes.

The figures in the paper have been produced by raytracing, however, an effective implementation (possibly a GPU one) can be an important goal. Finding good target functions for optimizing the coefficients is a possible area of improvement. Detecting poor-quality patches and automatically adjusting the surface coefficients would also enhance the scheme.

Acknowledgements

This research would not have been possible without the help of Tamás Várady who guided the author for several years and raised many important issues. The author would like to also thank the interesting discussions with Alyn Rockwood, Péter Salvi, and Márton Vaitkus as well as the interesting questions raised by the audience of the CSCS 2022 conference.

References

[1] Bajaj, C. L., Chen, J., and Xu, G. Modeling with cubic A-patches. *ACM Transactions on Graphics (TOG)*, 14(2):103–133, 1995. DOI: 10.1145/221659.221662.

[2] Bajaj, C. L. and Ihm, I. Algebraic surface design with Hermite interpolation. *ACM Transactions on Graphics (TOG)*, 11(1):61–91, 1992. DOI: 10.1145/102377.120081.

[3] Bán, R. and Valasek, G. First order signed distance fields. In *Eurographics (Short Papers)*, pages 33–36, 2020. DOI: 10.2312/egs.20201011.

[4] Charrot, P. and Gregory, J. A. A pentagonal surface patch for computer aided geometric design. *Computer Aided Geometric Design*, 1(1):87–94, 1984. DOI: 10.1016/0167-8396(84)90006-2.

[5] Chávez, G. and Rockwood, A. Marching surfaces: Isosurface approximation using G^1 multi-sided surfaces. *arXiv preprint arXiv:1502.02139*, 2015. DOI: 10.48550/arXiv.1502.02139.

[6] Hartmann, E. Implicit G^n-blending of vertices. *Computer Aided Geometric Design*, 18(3):267–285, 2001. DOI: 10.1016/S0167-8396(01)00030-9.

[7] Krasauskas, R. Toric surface patches. *Advances in Computational Mathematics*, 17(1):89–113, 2002. DOI: 10.1023/A:1015289823859.

[8] Lekien, F. and Marsden, J. Tricubic interpolation in three dimensions. *International Journal for Numerical Methods in Engineering*, 63(3):455–471, 2005. DOI: 10.1002/nme.1296.

[9] Li, J., Hoschek, J., and Hartmann, E. G^{n-1}-functional splines for interpolation and approximation of curves, surfaces and solids. *Computer Aided Geometric Design*, 7(1-4):209–220, 1990. DOI: 10.1016/0167-8396(90)90032-M.

[10] Liming, R. A. Conic lofting of streamline bodies: The basic theory of a phase of analytic geometry applicable to aircraft. *Aircraft Engineering and Aerospace Technology*, 19(7):222–228, 1947. DOI: 10.1108/eb031528.

[11] Loop, C. T. and DeRose, T. D. A multisided generalization of Bézier surfaces. *ACM Transactions on Graphics*, 8(3):204–234, 1989. DOI: 10.1145/77055.77059.

[12] Lorensen, W. E. and Cline, H. E. Marching Cubes: A high resolution 3D surface construction algorithm. *ACM SIGGRAPH Computer Graphics*, 21(4):163–169, 1987. DOI: 10.1145/37402.37422.

[13] Sipos, Á., Várady, T., and Salvi, P. Approximating triangular meshes by implicit, multi-sided surfaces. *Computer-Aided Design and Applications*, 19(5):1015–1028, 2022. DOI: 10.14733/cadaps.2022.1015-1028.

[14] Sipos, A., Várady, T., Salvi, P., and Vaitkus, M. Multi-sided implicit surfacing with I-patches. *Computers & Graphics*, 90:29–42, 2020. DOI: 10.1016/j.cag.2020.05.009.

[15] Vad, V., Csébfalvi, B., Rautek, P., and Gröller, E. Towards an unbiased comparison of CC, BCC, and FCC lattices in terms of prealiasing. *Computer Graphics Forum*, 33(3):81–90, 2014. DOI: 10.1111/cgf.12364.

[16] Várady, T., Benkő, P., Kós, G., and Rockwood, A. Implicit surfaces revisited – I-patches. In *Geometric Modelling*, pages 323–335. Springer, 2001. DOI: 10.1007/978-3-7091-6270-5_19.

[17] Várady, T., Rockwood, A., and Salvi, P. Transfinite surface interpolation over irregular n-sided domains. *Computer Aided Design*, 43(11):1330–1340, 2011. DOI: 10.1016/j.cad.2011.08.028.

[18] Várady, T., Salvi, P., and Karikó, G. A multi-sided Bézier patch with a simple control structure. *Computer Graphics Forum*, 35(2):307–317, 2016. DOI: 10.1111/cgf.12833.

A Auxiliary point calculation

Side coefficients in the examples are calculated such that the boundary curve interpolates a given point. That point is computed with either the triangle rule or the tetragon rule.

There are two cases: the two corner planes and the bounding face either have an intersection point inside the face, or it does not. If that point exists, we have a triangle (Figure 11a) and we take its centroid as the point to interpolate. Otherwise, we have a tetragon (Figure 11b), by intersecting each corner plane with the opposite corner's cube edge. We then take the centroid of this polygon.

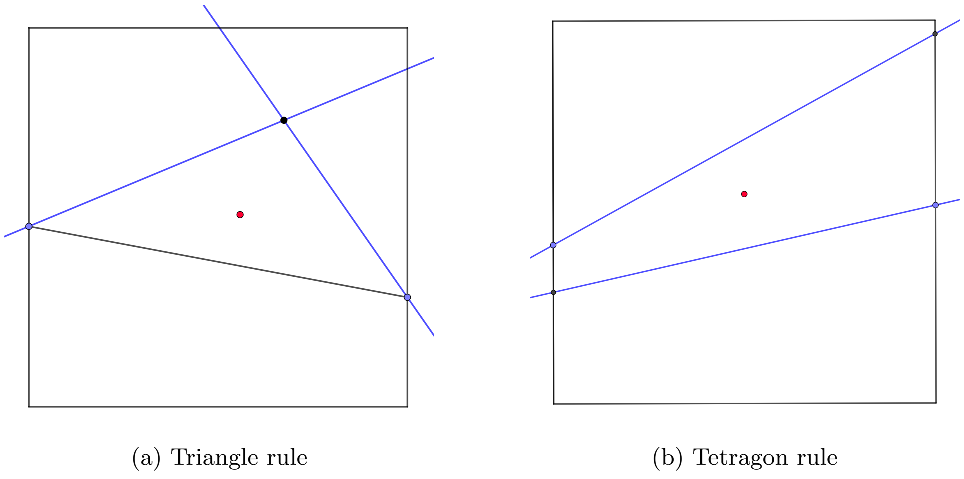

(a) Triangle rule (b) Tetragon rule

Figure 11: Rules for computing interpolated points. Blue points and lines represent corner points and planes.

The side coefficient can then easily be calculated by evaluating Equation 5 for each w_i, as the other side coefficients do not affect the current boundary.

Acta Cybernetica **26** (2024) 637–669.

Patient Flow Analysis with a Custom Simulation Engine[*]

Zoltán Szabó[ab], Emőke Adrienn Hompoth[ac], and Vilmos Bilicki[ad]

Abstract

Patient flow simulation and analysis is one of the oldest IT -based methods used to optimize patient care processes and hospital management. During the pandemic, interest in this domain suddenly increased due to the various constraints and recommendations to reduce the likelihood of further infections in the hospital. Suddenly, metrics such as the number of patients waiting in the same area, the maximum time a patient could stay in a single room, and the minimum distance between patients became important issues to monitor and optimize. Using data and modelling concepts from various hospitals, our team developed a simulation tool that used bpmn models to define an emergency department. We then modified a single day's usual patient flow with various real-world inspired edge cases to evaluate how the simulated flow would change and which stations would become bottlenecks, where the quality of patient care would deteriorate and rooms would become overcrowded. To execute the models, we developed our own tool based on the open-source Camunda modeling tool and the Business Process Model Notation (BPMN) file format. To execute the generated models, we use our own Python-based execution environment based on the SpiffWorkflow library, which permits extensive logging and extensive customization of the attributes analysed. In addition, the modelling toolkit of Camunda was narrowed down and compiled so that it could be easily used by researchers who are not programmers. In the paper, we present both the modeling process and the scenario design process, as well as the results obtained through the runs, including the maximum waiting times during the model runs and the maximum number of

[*]This research was supported by the EU-funded Hungarian grant GINOP-2.2.1-15-2017-00073; project no. TKP2021-NVA-09 has been implemented with the support provided by the Ministry of Innovation and Technology of Hungary from the National Research, Development and Innovation Fund, financed under the TKP2021-NVA funding scheme; project no. II-NKFIH-1528-1/2021 has been implemented with the support provided by the Ministry of Innovation and Technology of Hungary from the National Research, Development and Innovation Fund, financed under the II-NKFIH-1528-1 funding scheme. The research was also supported by the Ministry of Innovation and Technology NRDI Office within the framework of the Artificial Intelligence National Laboratory Program (RRF-2.3.1-21-2022-00004).

[a]Department of Software Engineering, University of Szeged, Hungary

[b]E-mail: szaboz@inf.u-szeged.hu, ORCID: 0000-0003-3863-7595

[c]E-mail: hompothemoke@gmail.com, ORCID: 0000-0002-7085-3901

[d]E-mail: bilickiv@inf.u-szeged.hu, ORCID: 0000-0002-7793-2661

DOI: 10.14232/actacyb.299391

patients waiting at once, which allowed us to validate the effectiveness of the framework.

Keywords: healthcare, telemedicine, patient flow, simulation, covid, modeling

1 Introduction

As can be read in the NEJM Catalyst short article [18], patient flow technically defines the total time frame that patients spend in and move through the healthcare system from arrival to discharge. In general, we want this time to be as minimal as possible, apart from the time required for the actual examination, diagnosis, and care processes, without compromising patient and provider quality and satisfaction. Improving the flow is essential as it can reduce the workload of medical staff and patient waiting times, but otherwise overcrowding can occur, patient health can deteriorate, while readmission and mortality rates can increase [18, 9]. Improving patient flow was also an area of research in the 1990s. The World Health Organization (WHO) published a study using patient flow analysis (PFA), which helps researchers examine staff utilization, key patient flow characteristics, resource and financial needs, and emerging problems [20]. Another approach has been variability analysis, which involves dividing variables into groups and then determining how to measure them (e.g., severity of illness can be described as the deviation from a perfectly healthy state). The next step is to reduce or even eliminate any variability that is artificial, as it usually arises from dysfunctional processes. This should already lead to an improvement in patient flow. Further progress can be expected if natural variability is also measured and optimally managed [17]. Our approach was to create a patient flow simulation framework that could account for different variables to calculate and measure potential patient flow. To do this, we collected information on commonly used patient flow measures and the variables that may affect patient numbers in the Emergency Department (ED).

In the State of Art section of this study, we provide a summary of numerous research papers that investigate techniques for quantifying the efficacy of ED. Then, in the Motivation section, we review the modeling methodologies and requirements that have arisen throughout the COVID-19 pandemic, summarize the parameters and elements that have been investigated, and highlight those that are pertinent to our simulation. This is followed by the Methodology section, which describes the operation of the ED we aim to represent as well as the modeling and simulation tools produced. In the Simulation section, we first list the scenarios that we planned and then present and analyze their running results, in each case based on the longest waiting times, the greatest number of patients waiting simultaneously, the trends in patient numbers at each station, the average waiting time per triage level, and the relative duration of each waiting time per triage level. The results of the research are reviewed in the Discussion section. Finally, we present an outlook on future goals for the framework in the Conclusion section.

2 State of Art

According to the literature, different patient flow patterns occur under different circumstances. Kang and Park [12] studied the hourly visit pattern and found a bimodal distribution: the peak flow was from 10:00 to 11:00 and from 20:00 to 21:00. The lowest number of visits was between 02:00 and 08:00. In one Hungarian hospital, patient volumes increase from 8:00 and peak around 12:00. Late night hours are the least visited times, but the workload for staff is fairly constant [26]. When daily visit patterns were the focus, Hitzek et. al. [9] found that the peak in patient numbers occurred on weekends (starting on Fridays, with the highest numbers on Saturdays), holidays, and school vacations. The authors suggest that the explanation may be that people tend to engage in risky activities at these times. Varga et al. [26] also examined the difference between patient numbers on weekdays and weekends: They found similar trends, except that weekend nights were slightly more demanding.

There are also seasonal patterns of visits: Hitzek et. al [9] found the highest numbers of patients in spring and the lowest in fall. In contrast, Won, Hwang, Roh, and Chung [27] found the highest number of asthma patients in the fall, especially in September and October, and the lowest from June to August. They also found that more patients visit the ED in spring from year to year.

Linked to seasonality, but with more focus on the actual temperature Otsuki, Murakami, Fujino, Matsumura and Eguchi [19] found that during cold winters less non-urgent patients visited the ED, suggesting that people are less active in the cold weather. In contrast the warmer summer weather raised the patient numbers.

Heat waves can also impact visits to ED. Schramm et. al [22] published a study of the likely impact of a June 25-30, 2021 heat wave, affecting 10 regions of the U.S. that contain 4% of the population but accounted for 15% of heat-related ED visits. From May to June, there were 3,504 heat-related cases at the ED, 79% of which occurred during the heat wave. The peak was on June 28, when 1,038 patients arrived. In comparison, 2 years earlier on the same day, 9 patients had heat-related problems at the ED.

The usual measures of patient flow are bed occupancy rate (it is also suggested to consider the number of outgoing and incoming patients) [13, 8], transfer time (i.e., the time to prepare the bed for a new patient), and patient transfer (how many patients had to be transferred, how much time and phone calls were required to transfer, etc.). Other ED related measures may include: the time a patient spends in the department from admission to discharge, the actual time it takes to discharge a patient and/or refer them to another department, how many patients were treated in a given time interval, the wait time to see a physician or receive treatment, the number of ambulances transferred to another ED, etc. [8].

For example, Varga et. al. [26] measured how much time elapsed before medical care was initiated between different triage levels. The results showed 3.6 ± 5.8 minutes at the first triage level, 7.0 ± 11.8 and 23.2 ± 26.1 minutes at the second and third triage levels, and 37.8 ± 38.3 and 44.2 ± 43.5 minutes at the fourth and fifth triage levels.

Patient flow analysis has also been used as the basis for many research projects using genetic algorithms and in some cases, machine learning, to solve or optimize scheduling issues at various parts of the hospital process.

Yousefi et al. [28] conducted an evaluation of 38 simulation-based optimization experiments for the ED, published between 2007 and 2019. They have given a bibliographic foundation on the topics discussed, compiled data on the methodologies and tools used, and identified significant trends in the area of simulation-based optimization. They have stated that future research should concentrate on improving the effectiveness of multi-objective optimization problems by reducing their time and labor requirements.

In their study, Yang-Kuei Lin and Yin-Yi Chou [16] examined the difficulty of allocating a set of surgical procedures to many multipurpose operating rooms. They have suggested a redesigned mathematical model and four simple heuristics that ensure the efficient discovery of viable solutions to the examined issue. In addition, they provided four local search processes that may greatly enhance a given solution and used a hybrid genetic algorithm (HGA) that combines initial solutions, local search procedures, and an elite search technique to the examined issue.

El-Bouri et al. [7] conducted a literature study on the use of Artificial Intelligence (AI) to hospital patient scheduling. They addressed the many AI strategies described in the literature, such as rule-based systems, decision trees, artificial neural networks, and evolutionary algorithms. In addition, they have examined the many sorts of patient scheduling challenges that have been investigated, including surgery scheduling, appointment scheduling, and emergency department scheduling.

Seunghoon Lee and Young Hoon Lee [14] have suggested using reinforcement learning (RL) to schedule emergency department (ED) patients. They have developed a mathematical model and a Markov decision process (MDP). Then, they developed an RL algorithm based on deep Q-networks (DQN) to identify the ideal scheduling strategy for patients. In the provided cases, they have shown that deep RL outperforms dispatching rules in terms of reducing the weighted waiting time of patients and the penalty score for emergency patients.

Haya Salaha and Sharan Srinivas [21] investigated the usage of a hybrid artificial intelligence system to solve the issue of hospital patient scheduling. To enhance patient scheduling, they have presented a mix of genetic algorithms and an Artificial Neural Network (ANN). They have shown that their hybrid approach can find superior schedules than either Generic Algorithm (GA) or ANN alone, and it has been applied to actual hospital data.

3 Motivation

Research, optimization, and various IT solutions played an important role during the COVID-19 pandemic. While various impacts and metrics of the pandemic itself are still being researched and evaluated by various research teams, another very active area is focused on preparing existing systems to work better and more efficiently in the event of another pandemic.

One need that most research teams agree on is the need for modeling and simulation tools. Currie et. al [4] in their work emphasized the importance of simulations to reduce the impact and severity of the epidemic COVID. They identified the following decision areas as appropriate for optimizing their effectiveness through simulations: the selection of quarantine and isolation strategies, the development of social distancing rules, the construction of lockdown release scenarios, the appropriate method for test distribution and transport, the identification of the most critical demographic groups for vaccine distribution, and the appropriate expansion and allocation of hospital resources.

Similar comments were made by Dieckmann et al. [5], whose work focused on the resources needed for effective simulation and how they can be used. In their view, simulations should focus on three main areas: educating workers about the epidemic, optimising the process of care at the system level, and assessing the needs and mental health workload of health care workers.

Improving hospital systems and patient flow to provide faster patient treatment, efficient resource allocation, and the development of techniques to avoid future infections lies at the junction of the two fields of study. Tavakoli et al.[24] recently published their results on a simulation methodology similar to ours. Although the model and triage levels are much simpler than they should be to prove accurate in simulations of Hungarian hospitals, the metrics and principles established can serve as a model for similar simulations. Terning et al.[25] had similar elements in mind, and although the simulation from their published work is still relatively rudimentary, the formulas and conditions used to evaluate their results provide a very good basis for initial validation of a similar simulation.

One of these key parameters, perhaps the easiest to follow in simulations, is to avoid overcrowding, i.e., to avoid the kind of patient flow where many patients are waiting in an area at the same time. Dinh et al. [6] specifically focused on this importance in their work, attempting to establish principles and rules to avoid unnecessary hospitalizations during an epidemic and to reduce the length of stay in the hospital. In their brief review, Janbabai et al. [10] focused on protecting hospital staff in addition to patients, focusing on preoperative, intraoperative, and postoperative processes within the patient flow. Of course, other approaches have been explored in addition to simulation-based patient flow study and analysis. For example, Arnaud et al, [2] have attempted to use machine learning based on patient flow metrics to determine how to optimise the number of hospital beds and expedite the triage process, to name a few examples.

In the development of various healthcare applications for hospitals and research teams, our team has used the work of Prof. Jose L. Jimenez & Dr. Zhe Peng [11] who, based on various peer-reviewed research, developed an easy-to-use tool to measure the likelihood of COVID infection in different environments based on the size and type of the area in question, as well as the number, behaviour, and condition of the people in it. Based on these results, and taking into account the fact that patients and staff wear masks in the hospital and hospitals use various distancing measures and restrictions, including a strong emphasis on ventilation, our team calculated that the probability of infection for a number of 10 to 20 patients in the area was only 4.39 % after one hour, and even after six hours it only increased to 5.96 %. This means that one of the most important aspects of optimising patient flow for COVID prevention is to keep the number of patients in a given range around or below 10 while trying to speed up the flow itself to avoid congestion.

4 Methodology

4.1 Introduction of the ED

During the early phases of our research, we used the ED model of Leva and Sulis [15], as it proved to be the model most similar to the Hungarian ones based on comparisons between this model and our team's experience and knowledge of the structure and functioning of the ED. Differences include minor changes in terms of which station is served by which staff member, and the introduction of an additional fifth triage level as mandated in the Hungarian system. The model includes 7 different lanes and a sub-process for handling the complex visit process if required by the patient's condition. The first lane is the registration process, where patients are admitted to the hospital and treated according to the severity of their condition upon arrival. They are then admitted to the triage lane where they await initial assessment. For less urgent cases, they may voluntarily leave the process at this stage if they wait too long.

After leaving triage, the triage nurse may decide to refer the patient to an internal clinic; otherwise, the visit process takes place, where the nurse or physician takes a history, takes blood, performs a radiology referral, and then decides the patient's fate based on the results. The outcome of the process may be referral to an internal clinic, admission to the emergency department, referral to an outside facility, discharge, or in a small percentage of cases, death of the patient.

4.2 Modeling and Simulation Tools

To create the hospital simulation, we chose the open-source Camunda Modeler [1], which allows us to create arbitrary processes in a parameterizable, editable, and executable format. The output of Camunda modelling is the BPMN (Business Process Model and Notation) file [3], a text file based on the XML standard that is displayed by Camunda-compatible tools and execution environments with a visual representation.

However, Camunda and BPMN modelling do not always prove suitable. In our various healthcare projects, the issue of clarity and complexity of modelling has often arisen in similar cases, especially when some clinicians and researchers wanted to model and describe processes in a way that was transparent to them, but the Camunda elements were considered too broad and complex. Our goal, therefore, was not only to accurately model the model we created, but also to make it understandable to researchers outside of IT and be able to create similarly simple processes, leaving the more complex parts to scripts and programmers running in the background.

While developing the simulation, we considered using the official Camunda simulation tools and Visual Paradigm, among others. However, we ultimately decided to create our own simulation environment using open source tools to ensure that the simulation settings, configurations, and types of metrics collected were customizable for us.

Our custom simulation is based on the Python library SpiffWorkflow [23], which can process and run bpmn models created with Camunda, among many other inputs. Scalability and robustness were key elements of the hospital workload modelling operating environment. The principle is based on the idea that each patient is a parallel running SpiffWorkflow thread sharing common resources for which we implemented waiting, handover and reservation using semaphores. The measurement and logging of wait and turnaround times in the system is differential, with each thread regularly logging its timestamps as it arrives at and departs from the stations. Our approach was initially based on the naive assumption that the bottleneck is the availability of staff in the ED, and that if the required physician, nurse, or nursing staff is available to perform the task, then the space and equipment are available as well.

Figure 1 illustrates the components of the framework and their precise relationships. The model defining the ED is provided in two bpmn files: Visit.bpmn for the visit subprocess and EDAsIs.bpmn for the ED architecture, which references Visit in the correct place. The framework's starting point is runner.py, which specifies how many patients must be allowed into the system for the simulation, what stop condition must be satisfied to terminate the simulation, and also manages the extraction of the various metrics at the conclusion of the simulation (the latter activity is expected to be handled by a separate module in a future version). The runner.py parses the contents of the bpmn files and utilizes them to generate runner threads for each patient that will execute the steps specified in the bpmn files. The simulations employ playbook.py to execute the simulation of each step and simulation.py to indicate when a shared resource (e.g., doctor, nurse) is required, lock it using semaphore, or set a triage level-based queue if there are no available instances of that resource.

4.3 Modeling

The following section presents the model and elements of the Camunda workflow based on the combination of the Leva and Sulis paper with elements from the Hungarian hospital system. Table 1 shows the content of the first two lanes, registration and triage, the first stations that are the same for every patient in the hospital. The elements of the simulation script are handled either as events, where patients must acquire a shared resource and then perform some processing before proceeding, or as end states, which, when reached, terminate the patient's simulation thread. The required resource in the simulation is a member of the ED staff: Generic Nurse (GN), Hospital Employee (HE), Specialized Nurse (SN), Doctor (DR) and Generic

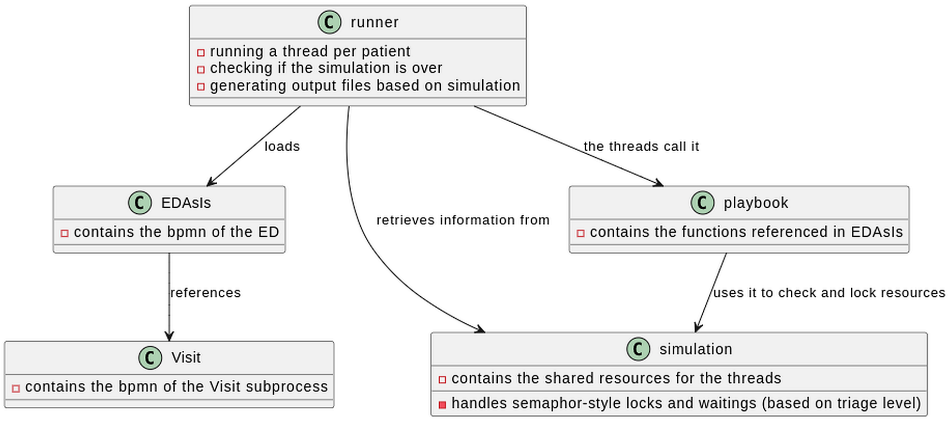

runner
running a thread per patient
checking if the simulation is over
generating output files based on simulation
loads
the threads call it
EDAsIs
contains the bpmn of the ED
retrieves information from
playbook
contains the functions referenced in EDAsIs
references
uses it to check and lock resources
Visit
contains the bpmn of the Visit subprocess
simulation
contains the shared resources for the threads
handles semaphor-style locks and waitings (based on triage level)

Figure 1: Flowchart of the simulation framework

Operator (GO).

Note that due to the deterministic nature of the simulation, these stations and steps model how a patient is admitted to the hospital and then treated, with the environment assigning almost the entire pathway to the patient at the beginning of the simulation with all the important attributes. Our research team had two main reasons for this: On the one hand, the methodology gives researchers who might use our tools in the future the ability to analyze and debug the expected runtime of the simulation without having to wait for the entire simulation to run. On the other hand, it also gives us the ability to manually enter patients into tables in order of arrival with their severity, and even to examine specific cases in minute detail using the tools. The modeling of these pathways raised a serious research question at the beginning, as the international literature and the original source of this model indicated that the necessary personnel for these pathways are the general nurses, and in order to keep the simulation accurate, we decided to stick with this version. However, in Hungarian hospitals it is much more common to have at least one physician present during these phases. In our further research, collecting more specific information from Hungarian hospitals, including those we have conducted research with, we want to test different modifications of this trace and see how they might change some of the results and trends we have obtained during our previous research. The next major step is the Visit, which is modeled as a separate subprocess. The elements of the Visit model can be seen in Table 2. The main difference from the main lanes is the need for specialized nurses and doctors, and the many optional pathways depending on whether blood tests or radiological examinations are required.

After the Visit subprocess, the only step left in the simulation is the processing of the outcome, which is usually performed by a specialised nurse (SN). There are five possible outcomes defined both in the paper containing the basic version of this model and in the papers analysing Hungarian hospitals: Death, Hospitalisation on

Table 1: Major events of the main ED flow

Name	ID	Description	Type	Req
Patient Registration	register_patient	Admitting the patient to the hospital	event	HE
Evaluate Urgency	evaluate_urgency	The urgency, as a binary information is determined for the patient. Urgent cases are immediately forwarded to pre-visit	event	GN
Abandon	abandon	The patient decides to leave the hospital before pre-visit	end_state	-
Pre-visit	pre_visit	Quick measurements and examination by a nurse to determine the triage level and severity of the case.	event	GN
Assign ESI	assign_esi	Emergency severity index is assigned to the patient, who is either transferred to internal clinic or sent to the full visit process	event	GN
Manage outcome	manage_outcome	Management of results and outcomes resulting from the visit process.	event	SN

Ward, Discharge, Transfer to External Facility, or Transfer to Internal Clinique.

We also achieved the desired simplification in modelling. Since the model was not overly complex, we used only four elements that were visually and practically comprehensible: the start point, the end point, the event, and the decision point. These were simply augmented during design with information about which event gave the patient which additional attributes, and the decision points were then used to select exactly what criteria the patient should use to choose the direction of travel in the simulation. The entire modelling process thus consisted of a total of four elements, plus a few lines of pseudocode description for the events, which is not only simple, but also compatible and interoperable with many other modelling tools. The timer event was considered as a fifth element type, but it was ultimately

Table 2: Elements of the Visit subprocess

Name	ID	Description	Type	Req
Collect History	collect_history	The nurse collects and organizes the healthcare history of the patients via direct interview with the patient and/or database queries	event	SN
Hypothesize Diagnosis	hyp_diag	The doctor evaluates the results so far, examines the patient and either establishes a diagnosis or may require further tests (blood tests and X-rays) before doing so.	event	DR
Take Blood Sample	take_blood	Taking a blood sample as prescribed by the doctor.	event	SN
Laboratory	laboratory	The blood sample is transferred to laboratory examinations. If there is any information available quickly, it is sent back to the doctor.	event	DR
Transfer to radiology	trans_rad	The patient is transferred to radiology and prepared for the X-ray recordings.	event	GO
Radiology	radiology	Usage of diagnostic imaging procedures, such as ultrasound, CT, MR.	event	SN
Establish diagnosis	estab_diag	Based on the results and the information, diagnosis is established	event	DR
Define therapy	def_therap	If possible, therapy is defined by the doctor	event	DR

ruled out due to the system's standardized time management. All wait periods are supplied to the framework as arguments or thresholds with defined values. SpiffWorkflow's scripting and customisation options are restricted in this domain, and the timer event would be conducted in real time regardless of the simulation's time format.

5 Simulation

Our goal was to study how an emergency department ideally operates and how unexpected events can occur, using this operating environment and the modelled emergency department with the number of patients arriving, the severity of their cases, the probabilities and rates for each branch from real data. Or in the case of an epidemic, how to optimise turnaround and wait times (since similar studies in many cases have only looked at similar models in terms of staff time or budget): Is it clearly a good idea to increase staff and the number of rooms and equipment needed to perform each activity?

5.1 Scenarios

To be able to create different situations and scenarios to analyze how small changes in patient flow, staffing, or processing time of the different stages affect the simulation throughput and metrics, we first created a baseline scenario based on real data from the ED of Somogyi Kaposi Mór Practicing Hospital [26] to estimate the rates of patient arrival and distribution between the five triage levels (i.e., the urgency of each case) to model. According to their data from 2015 statistics, the ED sees approximately 90 patients per day. In terms of triage levels, 0.67% of patients had triage level 1, 1.24% had triage level 2, 23.35% had triage level 3, 40.17% had triage level 4, and 34.54% had triage level 5. Triage levels 1 and 2 require immediate treatment, level 3 can tolerate waiting times up to 30 minutes, level 4 up to 60 minutes while level 5 even up to 2 hours.

As for the fate of the incoming patients after treatment: 20.9% were hospitalised, 2.7% voluntarily discharged, 1.5% were referred for triage, 0.4% were transferred to another inpatient facility, 0.4% died and 73.5% were discharged to their home.

The baseline scenario was based on the work of Leva and Sulis and was run with 3 doctors, 2 generic nurses, 3 specialist nurses, 2 clinical staff and 4 generic operators, with a 20% chance of a new patient arriving every minute - this resulted in the most even distribution, the element of the simulation to handle increasing or decreasing patient arrival density at given times is currently being tested and will be included in a next pilot phase. The turnaround times at each station, which depend on the triage level, follow the one-to-one model of Leva and Sulis, considering triage level 3 as the dividing line between urgent and less urgent cases. For all other scenarios, these original distributions and proportions were shifted through a type of exacerbated bias. In some cases, we increased the severity of incoming patient cases, in others we reduced the number of emergency department staff, and in still other scenarios, to approximate the impact of COVID, we used estimations of the need for and duration of decontamination to increase wait and turnaround times at each station in the simulation. The type and number of staff in the emergency department was based on the paper by Leva and Sulis. The various scenarios, their specific configurations and expected results are listed below.

- **SC0**: This is the basis of comparison made by merging of the Somogyi Hospital and the Italian sample. Patients are rarely admitted for urgent triage 1 or 2. The time spent at each station follows the original pattern drawn from the papers. **Specification**: 3 doctors, 3 generic nurses, 2 specialized nurses, 2 clinical staff, 4 generic operators; regular processing times; triage distribution: l1-0.00673, l2-0.01241, l3-0.23359, l4-0.40175, l5-0.34549; 20% patient arrival chance. **Expectation**: Patients with lower triage levels have to wait longer at common stations (registration, triage), where congestion and waiting times increase, but the time spent in the system remains within acceptable limits.

- **SC1**: Scenario inspired by red letter holidays. The staff in the Emergency Department has been significantly reduced, the density of incoming patients is lower, but the incoming patients are almost without exception more urgent, with a triage level higher than 3. **Specification**: 1 doctor, 1 generic nurse, 1 specialized nurse, 1 clinical staff, 2 generic operators; regular processing times; triage distribution: l1-0.0873, l2-0.1241, l3-0.23359, l4-0.00417, l5-0.00345; 20% patient arrival chance. **Expectation**: Even for triage level 3 patients, waiting and turnaround times will increase, while for triage level 1 and 2 cases, times will not be dramatically reduced.

- **SC2**: Summer heat, heatwave inspired scenarios. Patient arrival density is doubled compared to SC0, with a significantly higher probability of arriving triage level 1 and 2 patients. **Specification**: 3 doctors, 2 generic nurses, 3 specialized nurses, 2 clinical staff, 4 generic operators; regular processing times; triage distribution: l1-0.1873, l2-0.2241, l3-0.43359, l4-0.00417, l5-0.00345; 20% patient arrival chance but checked at every half minute instead of every minute. **Expectation**: Significant congestion at higher triage levels, with waiting times of several hours for patients at lower triage levels.

- **SC3**: An epidemic-inspired scenario. With only urgent patients coming to the hospital (less urgent cases are not even admitted), the number of staff in the Emergency Department has been increased, but also the minimum waiting and turnaround times due to disinfection procedures. **Specification**: 6 doctors, 4 generic nurses, 6 specialized nurses, 8 clinical staff, 4 generic operators; processing times are increased with a few minutes to simulate disinfection; triage distribution: l1-0.1873, l2-0.2241, l3-0.13359, l4-0.00417, l5-0.00345 ; 20% patient arrival chance. **Expectation**: Barely any patients from lower triage levels, but significantly increased times for higher levels.

- **SC4**: A benchmarking scenario, using the staff number of SC3 to handle the patient load of SC2 **Specification**: 6 doctors, 4 generic nurses, 6 specialized nurses, 8 clinical staff, 4 generic operators; regular processing times; triage distribution: l1-0.1873, l2-0.2241, l3-0.43359, l4-0.00417, l5-0.00345; 20% patient arrival chance but checked at every half minute instead of every minute. **Expectation**: Critical waiting and turnaround times decreased, the churn and trends of the flow will be similar to the ones of SC0

5.2 Results

Each scenario was run with a load of 90 patients (the daily average based on Somogyi Hospital data) and was intended to fill a 7-8 hour shift in the emergency department.

5.2.1 Scenario0

At SC0 we immediately noticed some interesting differences compared to our first hypothesis. As seen in Figure 2, the Diagnosis Establishment phase had the longest combined times (i.e., waiting and execution combined), while other elements of the first two lanes, such as registration and urgency evaluation, were relatively short.

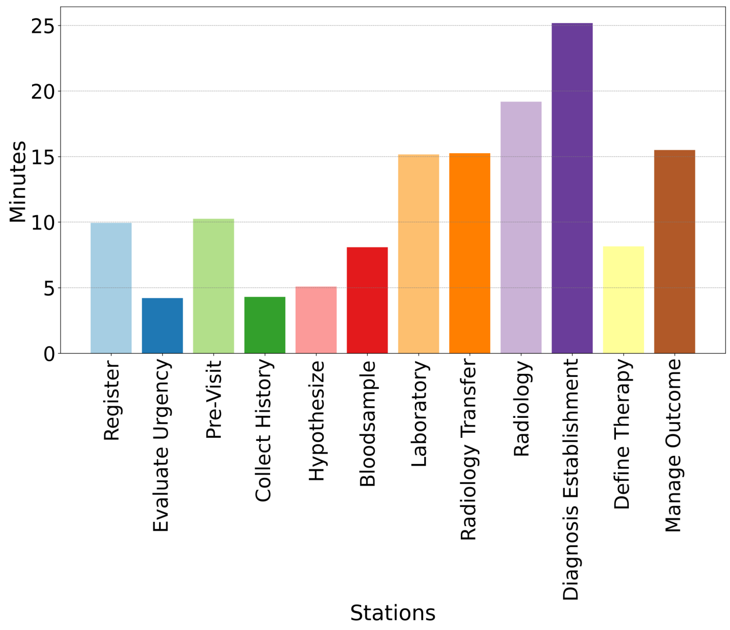


Figure 2: Longest combined times in SC0

Similarly, the maximum number of patients either waited or were studied at the same stage of their simulation. As shown in Figure 3, the longest queues were in

Radiology, Diagnosis Establishment, and Manage Outcomes after the visit.

A more detailed trend chart showcasing the top 3 stations based on maximum number of waiting patients and variability can be seen in Figure 4, which shows how the number of patients at the stations with the strongest bottleneck effect in the given scenario has changed over time in the simulation.

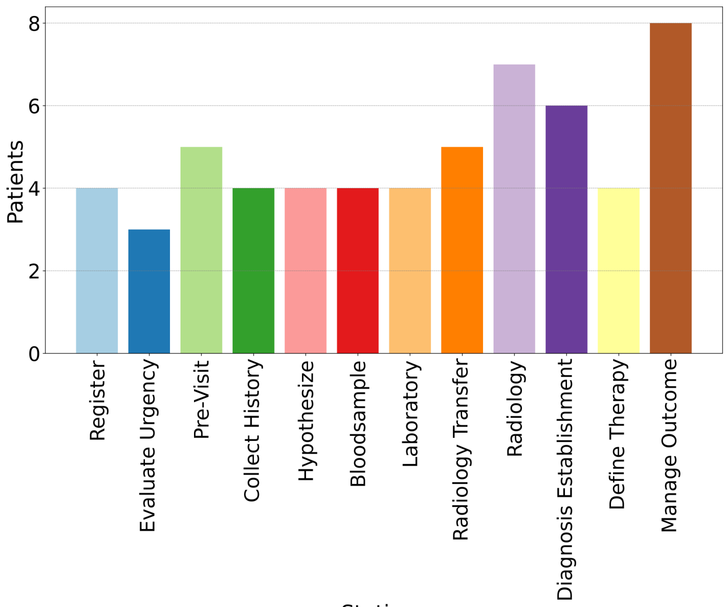


Figure 3: Maximum number of waiting patients in SC0

As for the comparison between the various triage levels, these values can be seen on Figures 5 and 6.

Based on the distributions and probabilities of the Hungarian hospital, not a single triage level 1 patient was admitted to ED during the simulated day. Triage stage 2, of course, had the shortest average time, while others had significantly longer times, with a bottleneck in the outcome management phase after the visit process.

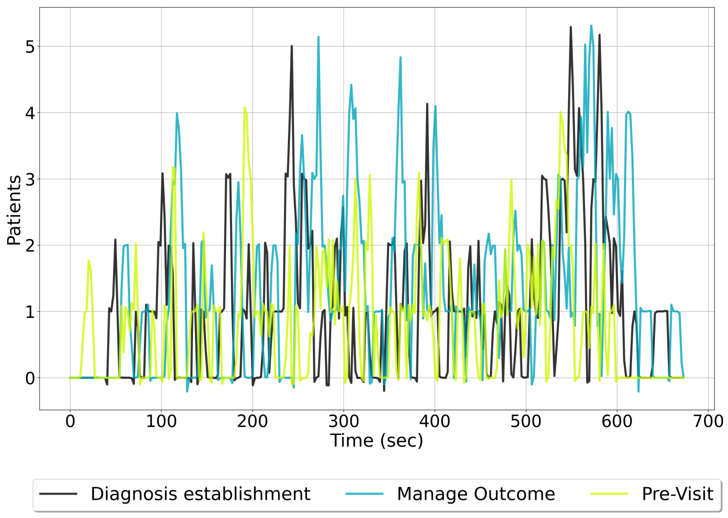


Figure 4: Patient number trends in SC0

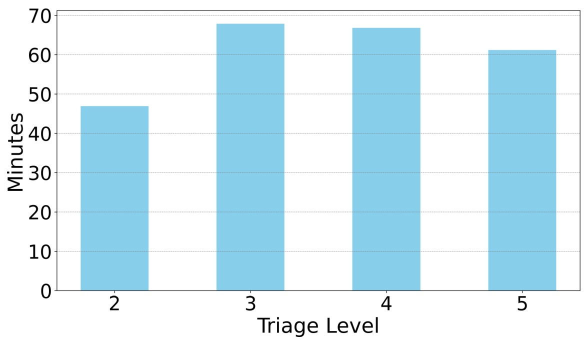


Figure 5: Average times spent in the simulation per triage level in SC0

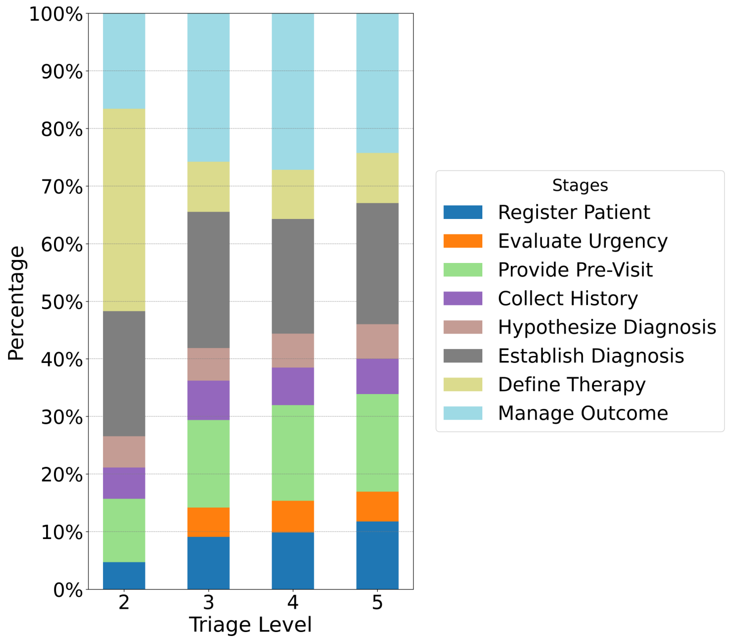


Figure 6: Distribution of times spent at the various stations per triage level in SC0

5.2.2 Scenario1

As with Scenario 1, reducing emergency department staffing and increasing the proportion of urgent patients has led to some interesting results in waiting and turnaround times. As can be clearly seen in Figure 7, the difference from the SC0 results is that the maximum aggregate waiting and turnaround times at each station are almost minimal.

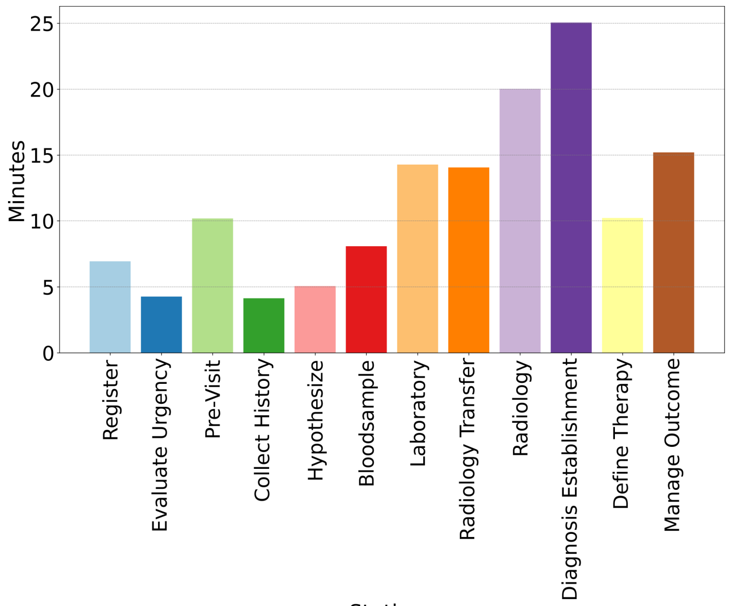


Figure 7: Longest combined times in SC1

The durations hold despite the fact that, as shown in Figure 8, the number of patients waiting at the same time has almost halved compared to the outliers in SC0 due to lower arrival times (with the outliers being the Pre-Visit, Manage Outcome, and Diagnosis Establishment).

Examining the top 3 trends in Figure 9 supports our hypothesis that lower arrival density reduced the number of patients waiting at a site, even at higher triage levels. Here, the peaks occurs when most patients need a clinician (in the simulated case, when preparing or performing radiology examination and during the outcome management), as only one of the clinicians in each role (except the generic operator) is on call during SC1. However, even at the peak, the weighted trend reaches a lower maximum compared to SC0.

The averaged times per triage stage in Figure 10 also confirm one of the key expectations of the simulation. While triage stages 1 and 2 are proportionally faster than the other stages, they are not exceptionally fast-for example, the average duration of Triage Level 5 is not significantly higher than Triage Level 1 (the high

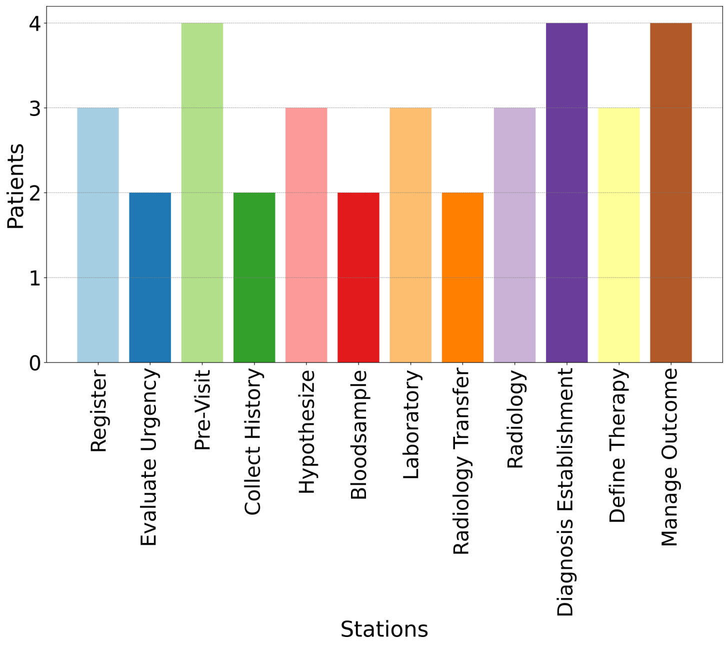


Figure 8: Maximum number of waiting patients in SC1

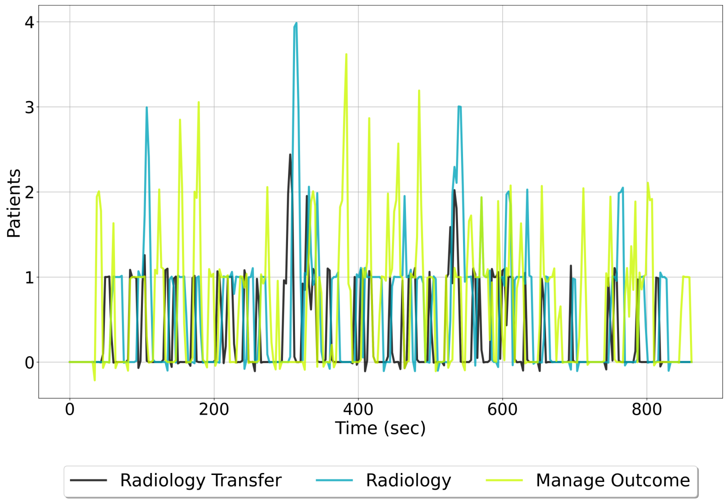


Figure 9: Patient number trends in SC1

values for Triage Levels 3 and 4 may be due to the longer path they follow in the simulation in addition to the arrival time).

They are treated earlier than other patients, they have to wait much less at

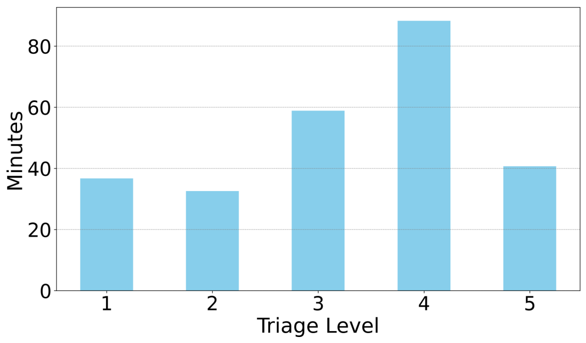


Figure 10: Average times spent in the simulation per triage level in SC1

many bottleneck stations, as can be seen in Figure 11, but their waiting times are not much shorter than those of other patients, and due to congestion and resource constraints, it is not even fully guaranteed that, for example, a level 1 patient will be treated faster than a level 2 patient, because some critical processes cannot be interrupted.

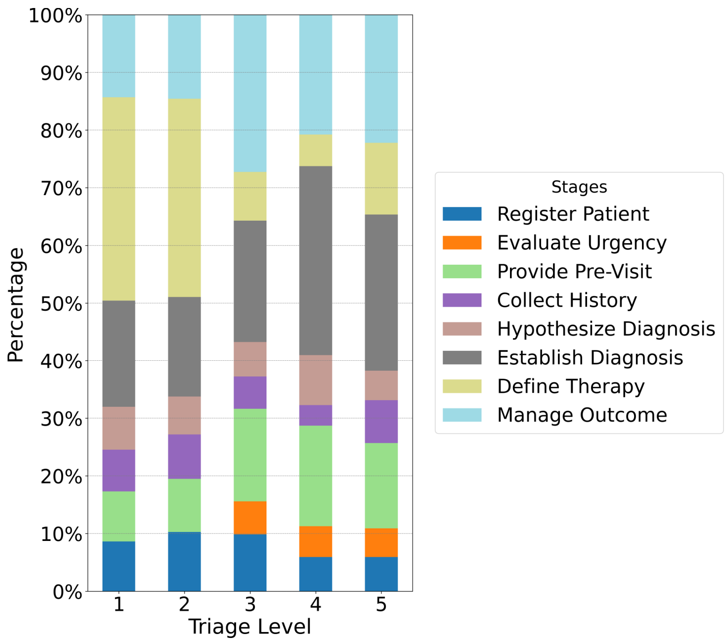


Figure 11: Distribution of times spent at the various stations per triage level in SC1

5.2.3 Scenario2

Scenario 2 represented the next level of workload complexity in patient flow. The goal was to see if the occurrence of higher triage levels and the reduction in hospital staffing would have better highlighted and made visible certain bottlenecks in the model, what the effect would be if more urgent triage levels were more likely than average, and if, despite being fully staffed, the mass of patients arrived in the ED much faster, about twice as fast as normal. And the effect proved to be very interesting. As can be seen immediately in Figure 12, wait times for the stations identified as bottlenecks in the previous scenarios shrink to nearly insignificant amounts compared to registration, where some patients would have to wait up to an almost unrealistic four hours to even be admitted.

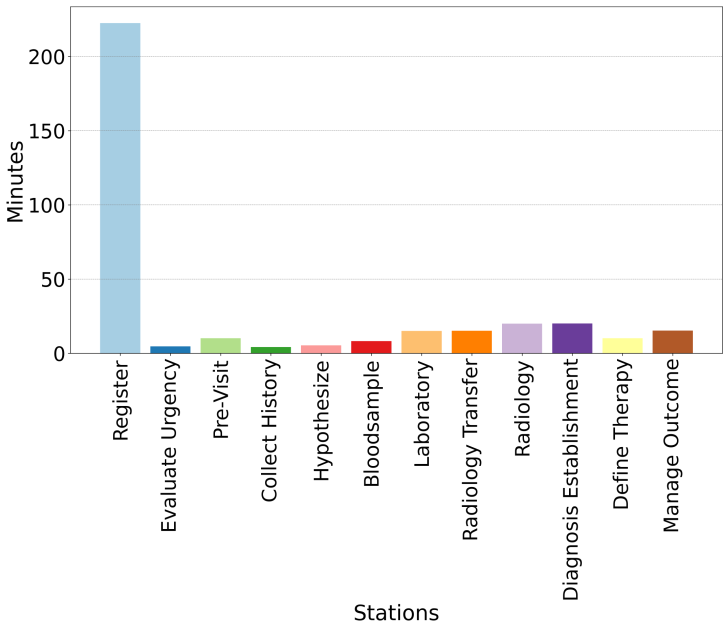


Figure 12: Longest combined times in SC2

This shift is also reflected in the maximum number of patients waiting at the same time, as shown in Figure 13. Diagnosis Establishment, Manage Outcome, and Pre-Visit stations in the simulation can still be considered as outliers, but due to crowding at patient admission, they are significantly exceeded by register_patient and evaluate_urgency.

The exact cause and trajectory of overcrowding are also clearly evident in the weighted trends shown in Figure 14. The number of patients waiting at the same time was slightly higher at the beginning than in the previous scenarios, but around the middle of the simulation there was a huge increase in overcrowding that affected subsequent phases.

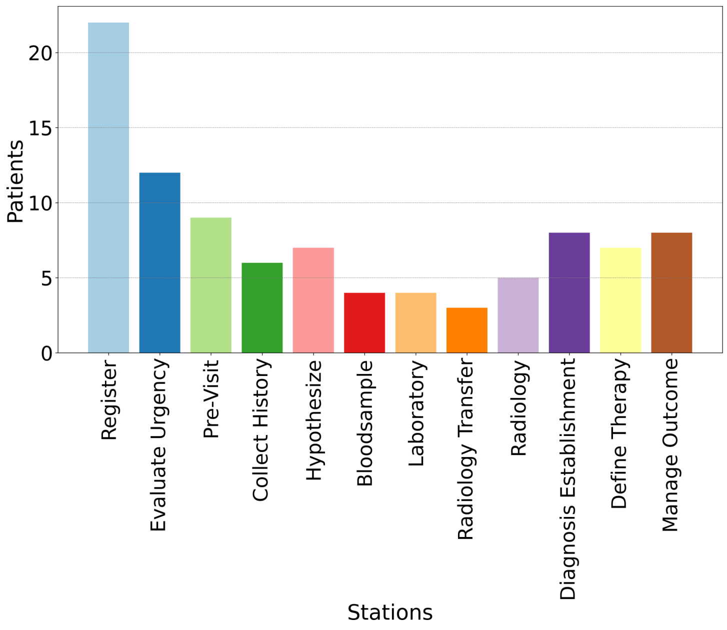


Figure 13: Maximum number of waiting patients in SC2

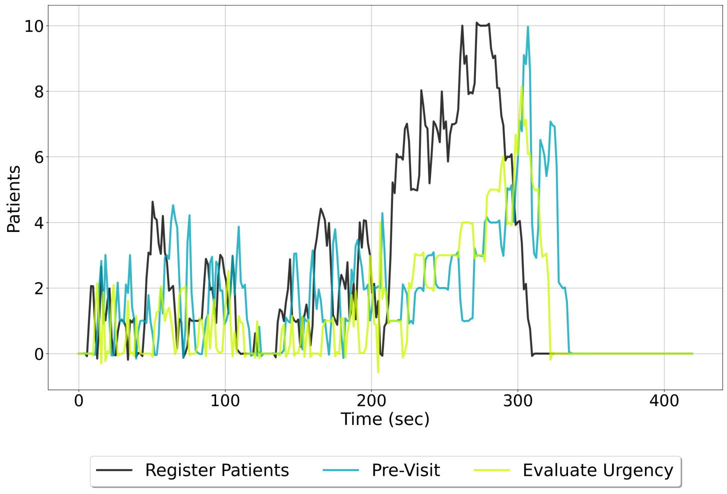


Figure 14: Patient number trends in SC2

However, Figures 15 and 16 also show that prioritisation of triage levels was adhered to despite system congestion. The average turnaround time for patients in triage levels 1 and 2 was remarkably fast in the simulation, spending a relatively large amount of time primarily at the Register and Define Therapy stations, while patients in less urgent cases were in the simulation for up to three to four hours, with a significant amount of waiting time spent at the registration desk.

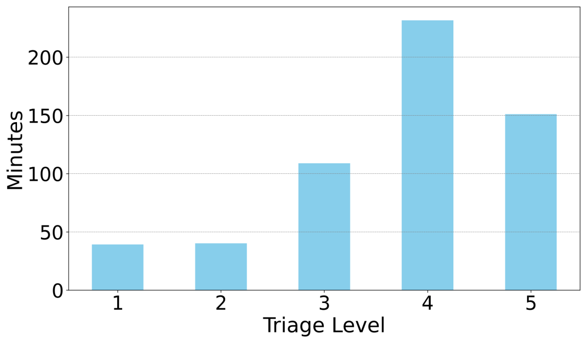


Figure 15: Average times spent in the simulation per triage level in SC2

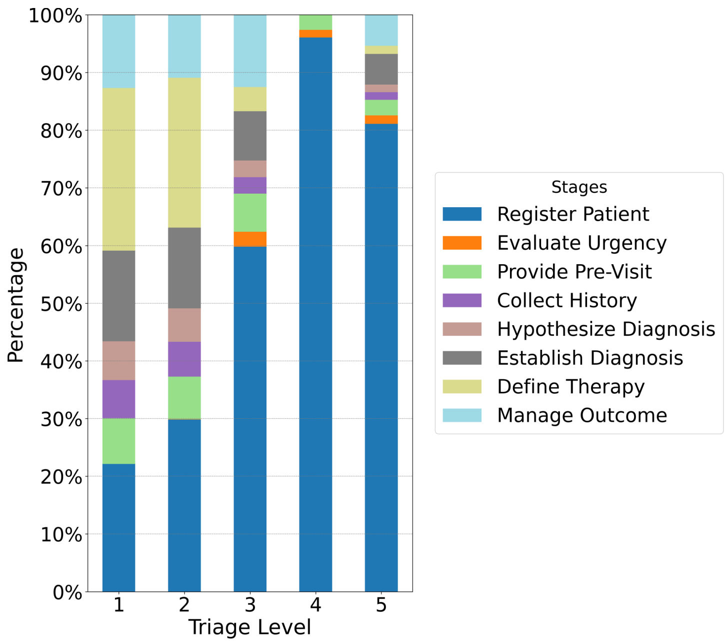


Figure 16: Distribution of times spent at the various stations per triage level in SC2

5.2.4 Scenario3

Scenario 3 was the most critical simulation and the most important for our further research, as we attempted to create a patient flow whose points and biases reflected the characteristics of an actual epidemic compared to the base case. In this case, the emergency department was visited only by urgent patients, typically with a triage level of 3 or higher, and the flow was slowed by the fact that although the number of staff was increased, the passage through the phases was much slower because of mandatory decontamination. Figure 17 shows the primary consequence: average wait and turnaround times per station are significantly longer than for the original cases (especially considering that the majority of cases here required urgent care).

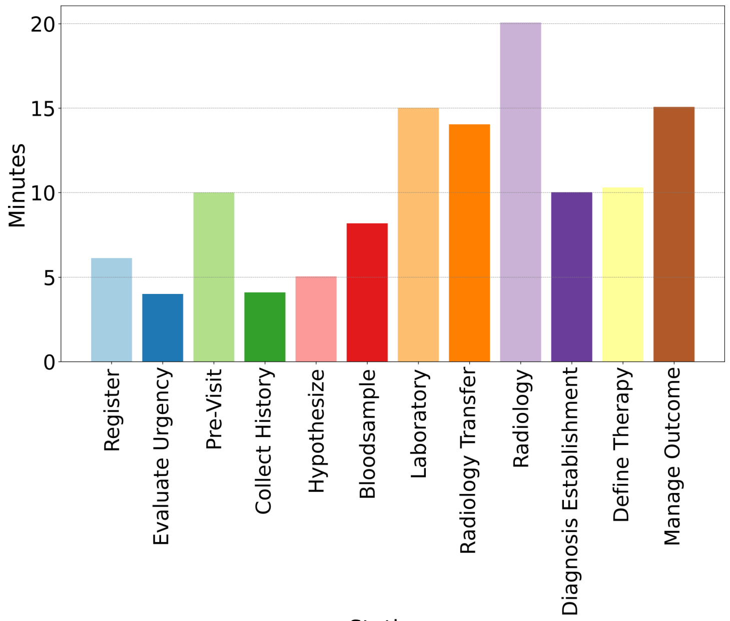


Figure 17: Longest combined times in SC3

The number of patients waiting at the same time has also increased significantly, as can be seen in Figure 18. In addition to the bottlenecks defined so far, the one that stands out is the Hypthosize, the step in the visit process where the patient is first seen by a physician in our model rather than by various nurses and generic operators.

This increase can also be seen in the top 3 trends in Figure 19, which are not only much higher than the results in the previous scenarios, but the peak is not as much of an outlier point as in previous scenarios, but an extended phase that takes up a significant portion of the simulation runtime. In other words, the congestion problem started much earlier, and as the later phases slowed, the number of patients

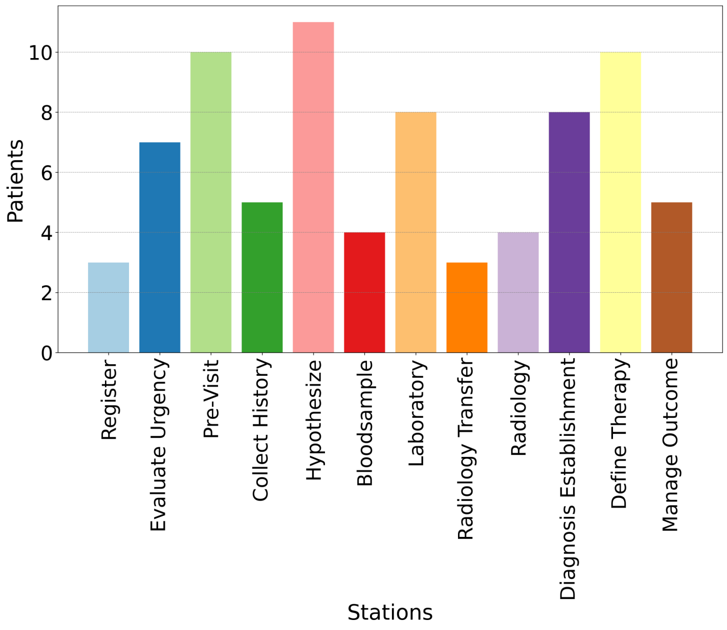


Figure 18: Maximum number of waiting patients in SC3

in the backlog did not start to decline as much as in the earlier cases, even though there should have been more staff available and the patients would have warranted a faster process due to the high triage levels.

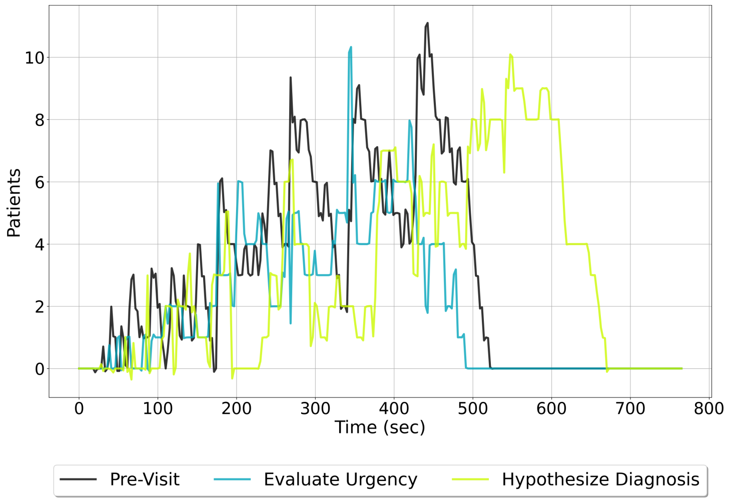


Figure 19: Patient number trends in SC3

Figures 20 and 21 also confirm that, as expected, almost exclusively patients with a triage level of 3 or more were admitted to the emergency department. On average, levels 1 and 2 were completed within an hour. As for the time distribution, it is interesting to note that it is quite similar for the three triage levels, with a significant proportion being spent in Register Patient, Define Therapy and Establish Diagnosis.

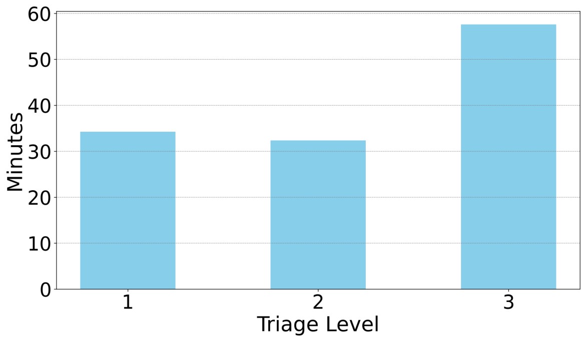


Figure 20: Average times spent in the simulation per triage level in SC3

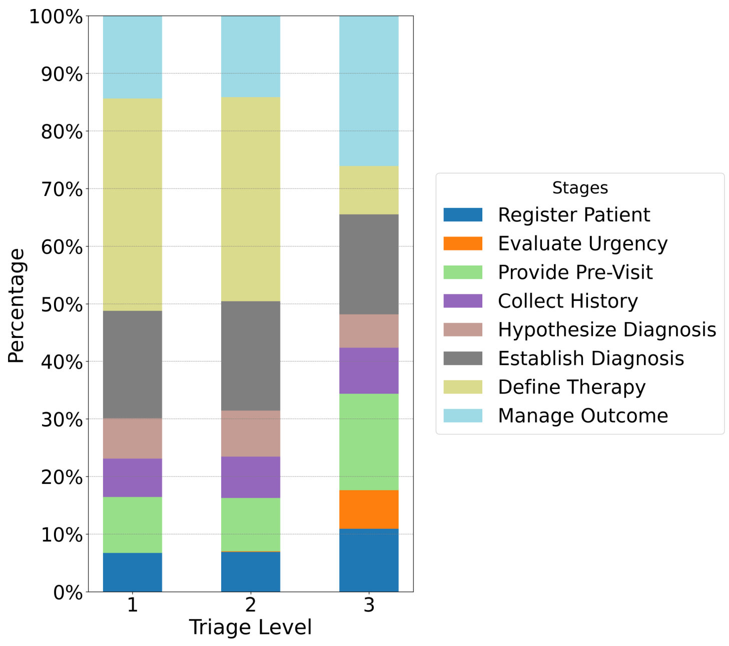


Figure 21: Distribution of times spent at the various stations per triage level in SC3

5.2.5 Scenario4

Based on Scenario3, then, it looked strongly as if simply increasing staffing would not lead to proportionally faster patient flow in the emergency department model. However, in our analyses, we felt that this should not be a generalisation. Therefore, for the fourth scenario, we essentially combined two earlier scenarios by combining the most extreme and in some respects worst Scenario2 with the significantly increased staffing of Scenario3. Figure 22 already shows that the average waiting and turnaround times are much more similar to the baseline Scenario0 than to the original Scenario2.

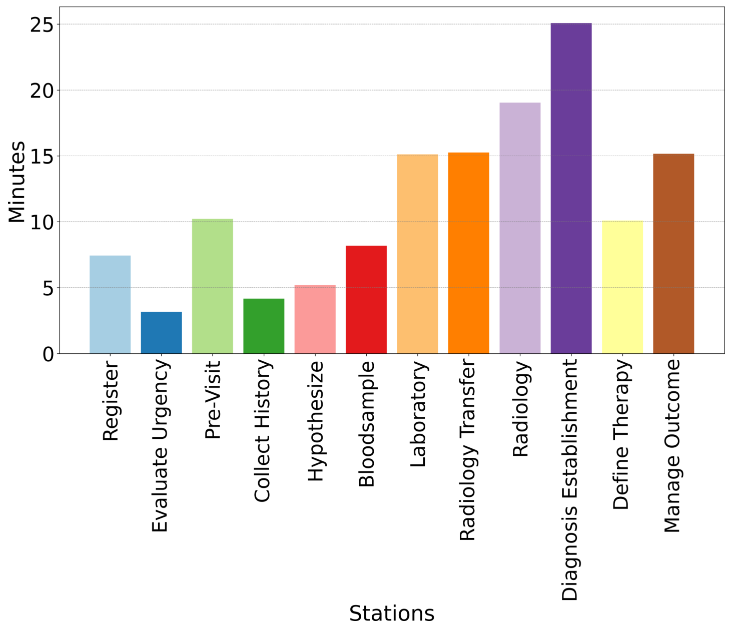


Figure 22: Longest combined times in SC4

As can be seen in Figure 23, the metric of most patients waiting in one place at one time have also changed: Although most stations still have 4-6 patients waiting at the same time, the number is much more balanced because bottleneck stations no longer have as many patients stopped at the same time as they did originally.

This smoothing and apparent reweighting is also reflected in the top 3 trends in Figure 24. The trend almost follows a pattern, with patient peaks at each phase pccuring in a nearly synchronous manner, rather than showing larger and more severe outliers as the simulation nears its end.

Looking at the metrics in Figures 25 and 26, it is noticeable that this time we had no patients in triage level 5, i.e., the least severe patient category, which makes the improved results in the previous figures even more obvious. Triage levels 1 and 2 took almost half as much time as the least severe patients. In addition, the first

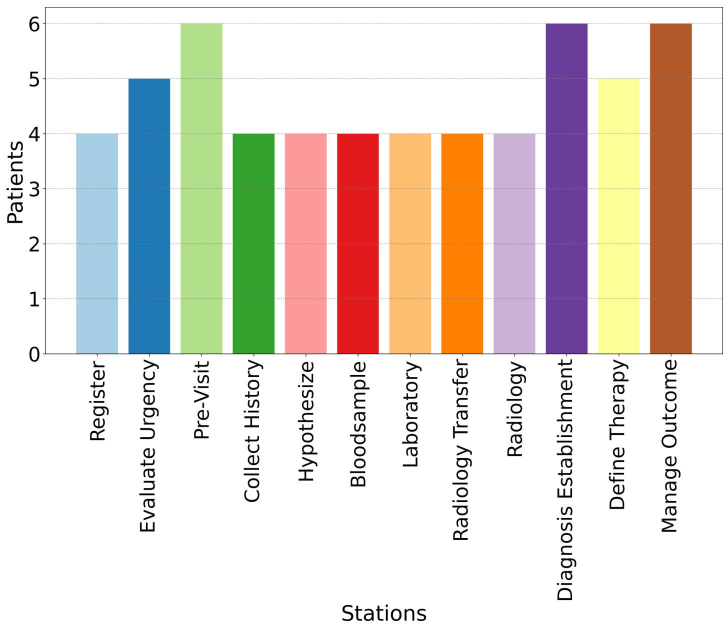


Figure 23: Maximum number of waiting patients in SC4

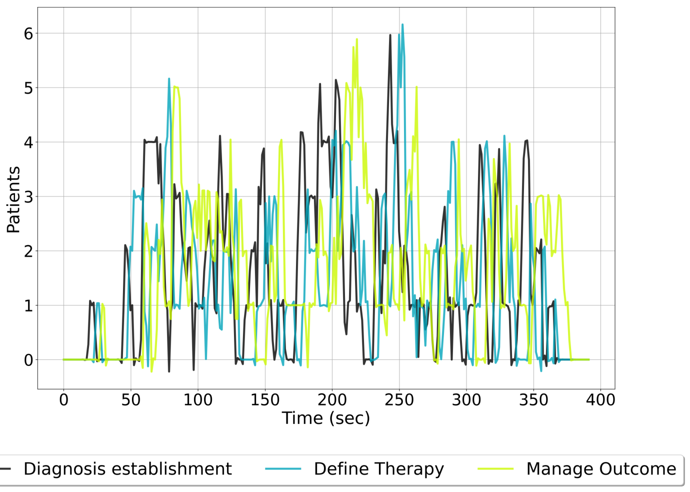


Figure 24: Patient number trends in SC4

two triage levels are similar in their time distribution, compared to levels 3 and 4, where several factors contributed to a longer duration relative to each other. Thus, while in Scenario3 the increase in staffing did not appear to reduce phase wait and turnaround times as much proportionately due to longer durations, here the increase proved to be significantly effective.

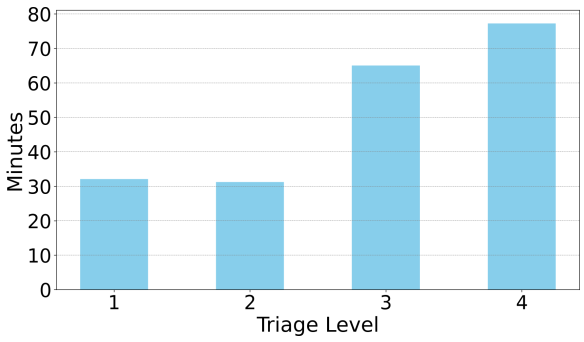


Figure 25: Average times spent in the simulation per triage level in SC4

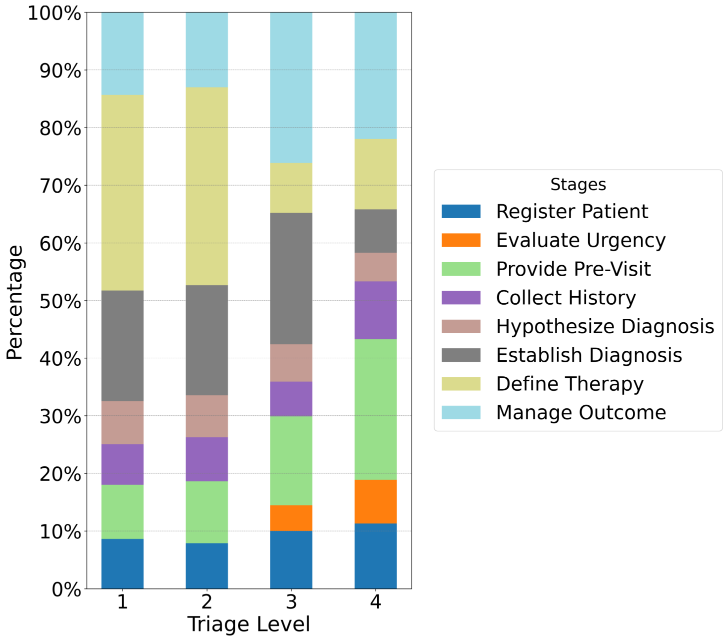


Figure 26: Distribution of times spent at the various stations per triage level in SC4

5.3 Discussion

In reviewing the simulation results, we found the following. First, the expectations for each scenario were either met or deviations occurred that can be interpreted based on the simulation run. Thus, despite the fact that the modelling toolbox itself has been simplified in line with our original objective, the model functions with the same accuracy. For example, with input from Hungarian and Italian hospital sources, the results of Scenario0 meet all specifications, from patient waiting times through triage level prioritisation to the maximum number of patients waiting at any one location. For the additional scenarios, the biases also yielded the expected results, so we can say that ***the reduced modelling toolkit, supported by scripts based on the SpiffWorkflow library, met our expectations and can be used for further simulations.***

The second most important observation from the results is that ***an increase in the number of staff in the emergency department does not necessarily mean an automatic acceleration of patient flow*** (and at this point, the simulation has not even been extended to include elements such as the limited space available for equipment and testing). However, ***there are some points where increased staffing does provide a boost, such as in patient registration and emergency assessment***, where increased staffing not only prevents these stations from becoming bottlenecks, but also helps prevent overcrowding in later stages of patient flow.

Examination of the trend plots for each scenario also shows that overcrowding does not start immediately, with the exception of SC3, where overcrowding was much more continuous due to the time gained from decontamination, with the peak typically occurring in the middle of the Emergency Department simulation, typically at points where throughput was already more critical. So, ***if the goal is to allocate resources more efficiently, it is certainly worthwhile to increase staffing during these periods and decrease it thereafter***.

Finally, in all scenarios, it has been shown that the most problematic phases in the patient flow are determining therapy and waiting for the visit to be evaluated. If the goal is to comply with COVID recommendations and reduce potential infection rates, these are the stages of patient flow where it is worth either reducing the time spent in the waiting room or, ***if this is not possible, providing patients with more separate, well-ventilated waiting rooms where they can wait for results without risking an extended stay that could reduce the effectiveness of infection prevention***.

Moreover, unlike many commercial solutions such as Visual Paradigm[1], Simcad Pro Health Simulation Software[2], or Simul8[3], our solution is a significant improve-

[1]Visual Paradigm — How to Create BPMN Diagram? URL: https://www.visual-paradigm.com/tutorials/how-to-create-bpmn-diagram/. [Accessed 24-Jan-2023]

[2]Simcad Simulation Software — Patient Flow Simulation: Predictive Modeling and Analytics, URL: https://www.createasoft.com/patient-flow-simulation. [Accessed 24-Jan-2023]

[3]Simul8 — Improve patient flow and enhance service quality. URL: https://www.simul8.com/applications/healthcare/improving-emergency-department-processes-with-simulation. [Accessed 24-Jan-2023]

ment in that it provides both free modeling and model execution, the source code of the modules used can be modified freely, as can the simulation's exact elements and output. In addition, the simplified modelling toolset and the BPMN file format do not restrict the usage of the created model, so if a research team has access to alternative simulation systems, the generated model may be utilized as-is or with minor modification. And its usage in its current form, maybe with minor enhancements, enables it to be utilized in conjunction with other simulation and assessment tools or by other processes. For instance, the data may be automatically merged with the hospital's measurements, which can then be run through Jimenez and Peng's tool [11] to create an accurate picture of the possibility of COVID spreading in a particular department or institution. Since the output is customizable, it may be used to study a broad variety of optimisation tasks, answering the demand mentioned by Yousefi et al [28]. The output and Python-based framework will presumably be of great value for reinforcement learning.

6 Conclusions

Our research is therefore currently at a stage where we have a modelling toolkit that can be readily used by researchers in other fields, as well as a simulation environment built from open-source components that can capture metrics and observations according to our needs, and which has already been able to provide valuable observations in its current form. There are, of course, a number of active directions in which we would like to develop this simulation solution further. The first and most obvious would of course be to obtain real Hungarian hospital data, with particular emphasis on data collected during the pandemic, to refine and improve the simulation, which is subject to ethical approval, and the application process is already underway. In addition, we plan to extend the simulation in its current form to include additional agents, such as constraints on equipment and rooms and more complex manipulation of patient arrival density using distribution and probability methods published in literature. We hope that the tools, method, and model resulting from our research will advance to the point where they become a valuable source of information for healthcare professionals to reduce the impact of cases such as COVID and improve hospital efficiency and patient care.

References

[1] About Modeler — Camunda Platform 8 Docs. URL: `https://docs.camunda.io/docs/components/modeler/about-modeler/`. [Accessed 15-Sep-2022].

[2] Arnaud, E., Elbattah, M., Ammirati, C., Dequen, G., and Ghazali, D. Use of artificial intelligence to manage patient flow in emergency department during the COVID-19 pandemic: A prospective, single-center study. *International Journal of Environmental Research and Public Health*, 19(15):9667, 2022. DOI: `10.3390/ijerph19159667`.

[3] Business Process Model And Notation Specification Version 2.0. URL: https://www.omg.org/spec/BPMN/2.0/. [Accessed 10-Sep-2022].

[4] Currie, C., Fowler, J., Kotiadis, K., Monks, T., Onggo, B., Robertson, D., and Tako, A. How simulation modelling can help reduce the impact of COVID-19. *Journal of Simulation*, 14(2):83–97, 2020. DOI: 10.1080/17477778.2020.1751570.

[5] Dieckmann, P., Torgeirsen, K., Qvindesland, S., Thomas, L., Bushell, V., and Langli Ersdal, H. The use of simulation to prepare and improve responses to infectious disease outbreaks like COVID-19: Practical tips and resources from Norway, Denmark, and the UK. *Advances in Simulation*, 5(1):1–10, 2020. DOI: 10.1186/s41077-020-00121-5.

[6] Dinh, M. and Berendsen Russell, S. Overcrowding kills: How COVID-19 could reshape emergency department patient flow in the new normal. *Emergency Medicine Australasia*, 33(1):175–177, 2021. DOI: 10.1111/1742-6723.13700.

[7] El-Bouri, R., Taylor, T., Youssef, A., Zhu, T., and Clifton, D. Machine learning in patient flow: A review. *Progress in Biomedical Engineering*, 3(2):022002, 2021. DOI: 10.1088/2516-1091/abddc5.

[8] He, L., Madathil, S., Oberoi, A., Servis, G., and Khasawneh, M. A systematic review of research design and modeling techniques in inpatient bed management. *Computers & Industrial Engineering*, 127:451–466, 2019. DOI: 10.1016/j.cie.2018.10.033.

[9] Hitzek, J., Fischer-Rosinskỳ, A., Möckel, M., Kuhlmann, S., and Slagman, A. Influence of weekday and seasonal trends on urgency and in-hospital mortality of emergency department patients. *Frontiers in Public Health*, 10, 2022. DOI: 10.3389/fpubh.2022.711235.

[10] Janbabai, G., Razavi, S., and Dabbagh, A. How to manage perioperative patient flow during COVID-19 pandemic: A narrative review. *Journal of Cellular & Molecular Anesthesia*, 5(1):47–56, 2020. DOI: 10.22037/jcma.v5i1.29789.

[11] Jimenez, J. and Peng, Z. Covid-19 airborne transmission tool available. URL: https://cires.colorado.edu/news/covid-19-airborne-transmission-tool-available, 2020.

[12] Kang, S. and Park, H. Emergency department visit volume variability. *Clinical and experimental emergency medicine*, 2(3):150, 2015. DOI: 10.15441/ceem.14.044.

[13] Karakusevic, S. Understanding patient flow in hospitals. URL: https://www.abhi.org.uk/media/1215/understanding_patient_flow_in_hospitals-nuffield-trust.pdf, 2016.

[14] Lee, S. and Lee, Y. Improving emergency department efficiency by patient scheduling using deep reinforcement learning. *Healthcare*, 8(2):77, 2020. DOI: 10.3390/healthcare8020077.

[15] Leva, D. and Sulis, E. A business process methodology to investigate organization management: A hospital case study. *WSEAS Transactions on Business and Economics*, 14:100–109, 2017. URL: https://www.wseas.org/multimedia/journals/economics/2017/a225807-059.php.

[16] Lin, Y.-K. and Chou, Y.-Y. A hybrid genetic algorithm for operating room scheduling. *Health Care Management Science*, 23(2):249–263, 2020. DOI: 10.1007/s10729-019-09481-5.

[17] Litvak, E. and Long, M. Cost and quality under managed care: Irreconcilable differences. *American Journal of Managed Care*, 6(3):305–12, 2000. URL: https://pubmed.ncbi.nlm.nih.gov/10977431/.

[18] NEJM Catalyst. What Is Patient Flow? URL: https://catalyst.nejm.org/doi/full/10.1056/CAT.18.0289. DOI: 10.1056/CAT.18.0289, [Accessed 29-Sep-2022].

[19] Otsuki, H., Murakami, Y., Fujino, K., Matsumura, K., and Eguchi, Y. Analysis of seasonal differences in emergency department attendance in Shiga Prefecture, Japan between 2007 and 2010. *Acute Medicine & Surgery*, 3(2):74–80, 2016. DOI: 10.1002/ams2.140.

[20] Patient flow analysis: An overview of national and international application. URL: https://apps.who.int/iris/handle/10665/59190?locale-attribute=es&. [Accessed 29-Sep-2022].

[21] Salah, H. and Srinivas, S. Predict, then schedule: Prescriptive analytics approach for machine learning-enabled sequential clinical scheduling. *Computers & Industrial Engineering*, page 108270, 2022. DOI: 10.1016/j.cie.2022.108270.

[22] Schramm, P., Vaidyanathan, A., Radhakrishnan, L., Gates, A., Hartnett, K., and Breysse, P. Heat-related emergency department visits during the northwestern heat wave—United States, June 2021. *Morbidity and Mortality Weekly Report*, 70(29):1020–1021, 2021. DOI: 10.15585/mmwr.mm7029e1.

[23] SpiffWorkflow 1.1.6 documentation. URL: https://spiffworkflow.readthedocs.io/en/latest/, 2014. [Accessed 22-Sep-2022].

[24] Tavakoli, M., Tavakkoli-Moghaddam, R., Mesbahi, R., Ghanavati-Nejad, M., and Tajally, A. Simulation of the COVID-19 patient flow and investigation of the future patient arrival using a time-series prediction model: A real-case study. *Medical & Biological Engineering & Computing*, 60(4):969–990, 2022. DOI: 10.1007/s11517-022-02525-z.

[25] Terning, G., Brun, E., and El-Thalji, I. Modeling patient flow in an emergency department under COVID-19 pandemic conditions: A hybrid modeling approach. *Healthcare*, 10(5):840, 2022. DOI: 10.3390/healthcare10050840.

[26] Varga, C., Lelovics, Z., Soós, V., and Oláh, T. Betegforgalmi trendek multidiszciplináris sürgősségi osztályon. *Orvosi Hetilap*, 158(21):811–822, 2017. DOI: 10.1556/650.2017.30749.

[27] Won, Y., Ho Hwang, T., Roh, E., and Chung, E. Seasonal patterns of asthma in children and adolescents presenting at emergency departments in Korea. *Allergy, asthma & immunology research*, 8(3):223–229, 2016. DOI: 10.4168/aair.2016.8.3.223.

[28] Yousefi, M., Yousefi, M., and Fogliatto, F. Simulation-based optimization methods applied in hospital emergency departments: A systematic review. *Simulation*, 96(10):791–806, 2020. DOI: 10.1177/0037549720944483.

Acta Cybernetica **26** (2024) 671–711.

Towards Abstraction-based Probabilistic Program Analysis[*]

Dániel Szekeres[ab] and István Majzik[ac]

Abstract

Probabilistic programs that can represent both probabilistic and non-deterministic choices are useful for creating reliability models of complex safety-critical systems that interact with humans or external systems. Such models are often quite complex, so their analysis can be hindered by state-space explosion. One common approach to deal with this problem is the application of abstraction techniques. We present improvements for an abstraction-refinement scheme for the analysis of probabilistic programs, aiming to improve the scalability of the scheme by adapting modern techniques from qualitative software model checking, and make the analysis result more reliable using better convergence checks. We implemented and evaluated the improvements in our Theta model checking framework.

Keywords: probabilistic systems, stochastic games, abstraction, reliability analysis

1 Introduction

Probabilistic programs are models specified using a software source code syntax, extended with special statements that sample values from probability distributions. Their similarity to program source code makes them easy to use to describe complex probabilistic systems, especially for engineers already familiar with programming.

Probabilistic programs are able to describe both probabilistic and proper non-deterministic behavior, which is useful for creating models for reliability analysis of complex safety-critical systems that interact with humans or external systems with unknown behavior. In such models, the probabilistic behavior described using sampling statements comes from the uncertainty inherent in the physical environment

[*]Project no. 2019-1.3.1-KK-2019-00004 has been implemented with the support provided from the National Research, Development and Innovation Fund of Hungary, financed under the 2019-1.3.1-KK funding scheme.

[a]Department of Measurement and Information Systems, Budapest University of Technology and Economics, Budapest, Hungary

[b]E-mail: szekeres@mit.bme.hu, ORCID: 0000-0002-2912-028X

[c]E-mail: majzik@mit.bme.hu, ORCID: 0000-0002-1184-2882

DOI: 10.14232/actacyb.298287

of the system and the uncertain failures of hardware components of the system. Proper non-determinism comes from the behavior of humans and external systems interacting with the system under analysis, for which we do not have statistical information to base probabilistic modeling on.

There are two main use cases for formulating reliability models as probabilistic programs:

- The expressivity makes it comfortable to use programs for describing the reliability model when the engineers are already familiar with standard programming languages. The advantages of program code compared to other formalisms (like the compositional language of the PRISM tool [23]) are especially apparent when the *process* described by the model is more complex than the *compositional structure* of the model.

- The program-based approach can be advantageous when a modeling formalism for which standard code generation is already available is extended with probabilistic semantics. As an example, our Gamma tool [14] supports modeling by statecharts and offers program code generation from these models. Extending the statecharts with stochastic concepts for dependability modeling leads to the generation of probabilistic programs that fit for analysis.

The analysis of such models is often hindered by *state-space explosion*: even when the program describing the model is relatively short, its actual state space can become intractable large. Abstraction is a widespread approach to counteract this problem: instead of analyzing the original model, an *abstract model* is created by ignoring some information present in the original. As the abstract model is constructed as a conservative approximation of the original, proving that a property is satisfied by the abstract model also proves satisfaction for the original model.

The most successful implementation of the idea of abstraction is a scheme called *Counterexample-Guided Abstraction Refinement (CEGAR)* [10], which automatically finds the appropriate level of abstraction. It starts with a very coarse model, analyzes it, and in case the property is violated, it generates an abstract counterexample. This counterexample is checked to decide whether it appeared only because of ignoring some information (spurious counterexample), or it actually proves violation in the original model (concretizable counterexample). In the spurious case, a refinement of the abstraction is computed based on the counterexample, and the CEGAR loop starts over.

In this work, we focus on the game-based framework proposed in [25] and extended in [18], and present improvements to it. We chose this approach because of its relative success and its ability to provide both upper and lower bounds for the analysed numeric property, giving a metric for measuring how precise the current abstraction is with respect to the property we are trying to check. [13] extended this scheme to domains used in classic abstract interpretation while also switching to an abstract interpretation framework instead of the CEGAR loop. We stayed with the original CEGAR-like refinement loop scheme. Most of the enhancements we implemented are based on our experiences in qualitative software model checking [28, 15].

The game-based abstraction scheme is applicable to any analysis question that can be formulated as computation of expected total rewards on a Markov Decision Process, which is the underlying low-level formalism we use for the semantics probabilistic programs in this work. Our proposed changes do not constrain the generality of this scheme any further, but in this paper, we will only focus on checking probabilistic reachability properties, i.e. computing the probability of reaching a specific set of target states.

Our contributions are the following:

1. We improved the scalability of the scheme by adapting modern techniques from qualitative software model checking:
   a) Adapted a version of *Large Block Encoding* to the probabilistic case.
   b) Adapted the usage of *inexact transition functions* and formally proved that the abstraction scheme remains sound when employing them.
   c) Extended the possible abstract domains with the *explicit value domain* (also known as visible variables domain).
2. We tackled the problem of convergence checking for the abstract model using *bounded value iteration*, and gave an example where using standard value iteration can lead to problems with the refinement loop.
3. We performed *numerical measurements* to answer research questions related to how our improvements and extensions affect performance.

These enhancements make the probabilistic analysis of a larger set of complex safety-critical systems possible. As these techniques are heuristic, they do not promise to enhance performance on *every* reliability model used in practice, but they give new options, which perform better on a set of practical models, some of which are not analysable without them.

The application of bounded value iteration is a semi-exception to this: the overall analysis result will always be more reliable, than with standard value iteration, while the potential performance gain from performing better refinements matters only for some (non-empty, as we show in our measurements) subset of analysis cases, similarly to the other enhancements.

We implemented enhancements these as a probabilistic extension of our open-source Theta model checking framework[1]. The implementation only supports probabilistic reachability properties for now, and for this reason, the benchmarks we performed are also constrained to this analysis task.

Section 2 introduces the necessary mathematical and formal modeling background along with the game-based abstraction refinement scheme. Section 3 describes the main idea of bounded value iteration and shows why a reliable convergence check is especially important in the abstraction refinement scheme. Section

[1]https://github.com/ftsrg/theta

4 describes the technique of Large Block Encoding in general and how we incorporated it for probabilistic program analysis. Section 5 describes the Explicit Value abstract domain and elaborates on our adaptation method. Section 6 introduces the general idea of inexact abstract transition functions, and proposes a way to apply them in the game-based scheme. Section 7 describes our numerical experiments and the related research questions, along with plots of the results and answers to the research questions. Conclusions and proposals for future work can be found at the end of the paper. The appendix contains proofs for the soundness theorems stated in Section 6.

1.1 Related work

Here, we present the general relations between our contributions and previous works. The formalisms used in this work are presented in Section 2. Details regarding the background algorithms can be found in their respective sections (Sections 3, 4, 5 and 6).

Several different abstraction approaches have been proposed for probabilistic systems [12, 21, 27, 17, 9, 25, 29]. Our work builds on the game-based abstraction refinement algorithm presented in [25, 18].

Cartesian abstraction and explicit abstraction have already been applied successfully to non-probabilistic model checking in [4] and [7, 15] respectively. We build on these results and adapt the corresponding algorithms to the analysis of probabilistic systems. We incorporate ideas from [19] to prove the soundness of using these domains (the proofs can be found in the appendix). [13] extended the same abstraction refinement algorithm to numerical abstract domains used in abstract interpretation, similarly to our extensions. While they switched to an abstract interpretation approach, we stayed with the original abstraction refinement loop.

Large block encoding has been used in non-probabilistic model checking for example in [6]. We use a simpler version of it to make sure that the resulting model satisfies the constraints required by the abstraction refinement algorithm of [18].

It has been shown in [3], that employing the classic stopping criterion of value iteration can lead to arbitrarily wrong results. We build on this result to show that this stopping criterion can even lead to performance degradation when the MDP is analysed using abstraction refinement. [3, 24] proposed a bounded version of value iteration to solve the problem of convergence checking, which [20] extended to stochastic games. We incorporated this algorithm in our abstraction refinement implementation, and evaluated its impact on the overall performance of the analysis.

2 General Background and Formalisms

This section introduces the necessary mathematical and formal modeling background and describes the abstraction framework that the paper builds upon.

2.1 Probabilistic Programs

Probabilistic programs are syntactically similar to standard structured program code with basic features, like ANSI C or lisp, but are extended with features making them suitable for the description of random processes and probabilistic models.

The most important such feature is *sampling*, which draws a sample from a given probability distribution - syntactically similar to calling pseudorandom number generators, but with different semantics. When analyzing a probabilistic program, a sampling statement is treated as a random variable with the given distribution, and can be either discrete or continuous with any support, even an infinite one. Probabilistic program interpreters and analyzers can then simulate sample trajectories, compute moments of expressions over program variables, etc.

In our current work, we use a probabilistic extension of ANSI C to specify probabilistic programs. The extension means that there are some predefined functions with special semantics: these functions are interpreted as proper probabilistic sampling when analyzed.

Only discrete distributions with finite support are in the scope of this work. Handling infinite and continuous distributions are possible future extensions planned to be implemented in the probabilistic module of our Theta framework.

Some modern probabilistic programming constructs, like higher-order probabilistic functions and observe statements are out of scope for this paper, our current work focuses on a basic set of features that cover the most important aspects.

Example. The code of a simple running example we will use throughout this paper can be seen in Listing 1. The program has two integer variables, x and y, both initially set to 0. When we show different kinds of state spaces for this program, we will restrict their domain to the set $\{0, 1, 2, 3\}$, instead of the complete set of integers.

```
main() {
    int x = 0, y = 0;
    while(y <= x) {
        //probabilistic assignment with a biased coin-flip
        x = x + coin(0.4);
        if(x > 2) {
            //havoc: assignment to a non-deterministic value
            y = *;
        }
    }
    assert(!(x < 3));
}
```

Listing 1: Probabilistic C code of a simple probabilistic program

These variables are repeatedly changed in a loop while the value of y is less than or equal to the value of x. In this loop, we first “flip a biased coin” by calling a sampling function, and increment x by the result (1 with probability 0.4, 0 with the remaining probability 0.6). If x is greater than 2, we set y to a non-deterministic value from its domain. In software model checking, this action is called *havoc*, and we denote it as $y = *$ here.

After exiting the loop, we *assert* that x is less than 3. Assert calls lead to an error state if the expression they contain evaluates to false. When analysing this probabilistic program, we will be interested in the probability of violating this assertion.

2.2 The Probabilistic Control Flow Automaton formalism

The control flow automaton (CFA) formalism is a widespread formal model used for the qualitative analysis of software source code. Probabilistic programs have similar structure to classical source code, but some functions are equipped with special probabilistic semantics. To enable the analysis of such programs, we use an extension of the CFA formalism: the *Probabilistic Control Flow Automaton (PCFA)*.

The definition we give for this formalism is similar to the definition of probabilistic programs in [18], but we would like to keep the notion of the *program itself*, and the *analysis formalism* used for model checking separate, as is standard in classical software model checking. Treating it as an extension of the classical CFA formalism also makes the relationship between techniques used in this paper and techniques of classical software model checking clearer.

Definition 1 (Probabilistic Control Flow Automaton). *A* Probabilistic control flow automaton (PCFA) *is a tuple* $P = ((L, E), l_{init}, Var, v_{init}, S, Stmt)$*, where:*

- (L, E) *is a finite directed control-flow graph; the nodes of the graph are called the* locations *of the PCFA, and the edges represent successor relationships between them*
- $l_{init} \in L$ *is the initial location*
- Var *is a finite set of* variables
- $v_{init} \in U_{Var}$ *is the* initial valuation *(described below)*
- S *is a set of* statements *(described below)*
- $Stmt : E \to S$ *is a function mapping each edge to an associated statement*

The definition uses a set of *variables* to represent the data state of the program. Each variable v has a type, or equivalently, a countable *domain* denoted by $Dom(v)$, which is the set of values it can take. The set of *valuations* U_{Var} for the variable set Var consists of functions mapping every variable $v \in Var$ to an element of their domain $Dom(v)$.

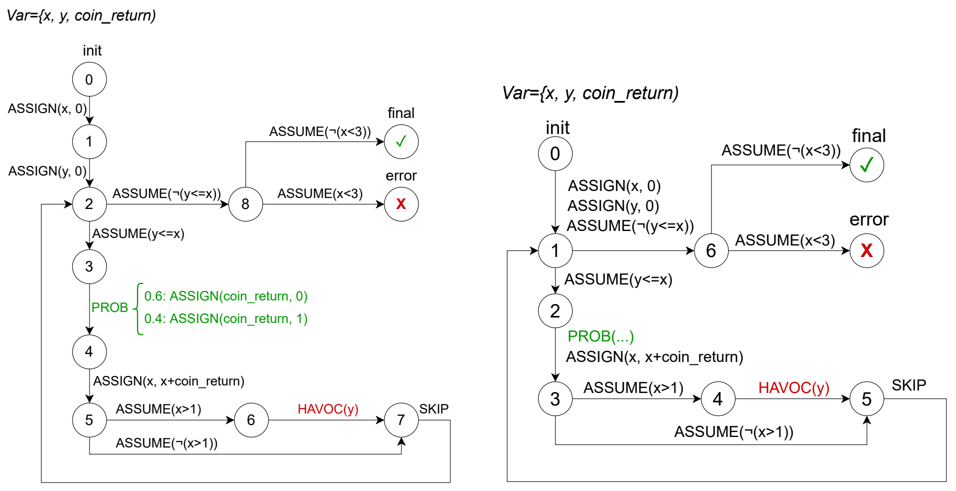
Although this definition allows only a single initial valuation, non-deterministic variable initialization can be modeled by using `HAVOC` statements at the beginning of the program.

A *statement* $s : U_{Var} \rightharpoonup 2^{\mathbb{D}(U_{Var})}$ is a probabilistic valuation transformer: given the current valuation, it returns a set of distributions of resulting valuations after applying the statement. It is a partial function, which is how we handle conditional statements (`ASSUME`s, see below): a statement is *enabled* by a valuation v, if it is defined at v. The result is a *set of valuations*, to make the representation of non-deterministic statements possible. In this paper, this is only used in the case of `HAVOC` statements, which result from calling functions with unknown results (e.g. asking for user input).

We use the following types of statements:

- `ASSIGN`$(v \in Var, e : U_{Var} \rightarrow Dom(v))$: The result is a singleton set of a dirac distribution of a single resulting valuation. This valuation maps the variable v to the result of applying the function e to the original valuation, and maps all other variables to their value in the original valuation. Used for assignment statements, where the right-hand side can be any (type-compatible) expression.
- `ASSUME`$(e : U_{Var} \rightarrow \{\top, \bot\})$: The partial function is defined exactly for those valuations that the function e maps to $\top$. For these valuations, it is basically an identity function, the returned set is a singleton set of a dirac distribution of the original valuation. Used for conditions in the program, like `if` statements and loop condition checks.
- `HAVOC`$(v \in Var)$: The resulting valuation set is $\{\, dirac(U_{orig}^{[v \longrightarrow v_i]}) \mid v_i \in Dom(v) \,\}$, where $U_{orig}^{[v \longrightarrow v_i]}$ is a valuation that maps v to v_i and any other variable to its value in the original valuation. This statement is used for assigning a totally non-deterministic value to a variable from its domain. Combined with an assume statement, it can be used to assign non-deterministic values to a variable from a constrained set of its full domain.
- `PROB`$(d : \mathbb{D}(S_{det}))$: This is the only statement that produces a distribution that is not a dirac distribution. It is parameterized by a distribution over other *deterministic* statements – S_{det} is the set of statements that map only to singleton sets. This is needed as we defined the result of applying a statement as a set of distributions, not distributions of sets. The resulting set has a single distribution in it. This distribution over valuation sets can be derived from the parameter distribution by computing the resulting valuation set from each statement with non-zero probability in the parameter distribution, and assigning the same probability to the resulting valuation as the statement. If multiple substatements lead to the same valuation, their probabilities are summed.
- `SKIP`: Enabled in any valuation, and does not change the valuation. Used for simplifying control flow structures like loops.

In most cases the code under analysis contains `assert` calls, which check the truth of a Boolean expression. If it evaluates to true, the program continues normally. If the expression is false, an error is raised. When transforming the code to a (P)CFA, `assert` calls are transformed to two `ASSUME` edges, one of them leading to the next location in the normal flow of the program, and the other one entering an



(a) Single-block encoding (b) Large-block encoding

Figure 1: PCFA of the running example

error location. Checking the probability of failing an assertion thus can be reduced to computing the probability of entering an error location.

We will describe the formal semantics of PCFA models denotationally by deriving a Markov Decision Process from them later, but for intuitive understanding, we describe the operational semantics of a PCFA can be described semi-formally here:

- The model starts in the location l_{init} with the valuation v_{init}.
- In each step, an outgoing edge e is selected non-deterministically from the outgoing edges of the current location such that $Stmt(e)$ is enabled by the current valuation v_{curr}.
- An element of the set $Stmt(e)(v_{curr})$ is selected non-deterministically, which is a distribution over the possible next valuations.
- A valuation v_{next} is sampled from this distribution. The location of the model in the next step is the location at the end of the selected edge, and the valuation in the next step is v_{next}.
- If the current location has no outgoing edges, the model stops.

Precise formal denotational semantics can be found for example in [5].

Example. The PCFA of our running example can be seen in Figure 1a. Although we did not need it in the code, an auxiliary variable *coin_return* is added to keep track of the value returned from the *coin* function call. This is needed because we need to separate the sampling call and the assignment to two edges.

Deciding whether the body of the loop should be executed or skipped is done using two edges with opposing `ASSUME` statements. The `PROB` statement corresponds to the call to the special sampling function *coin* in the code. This statement sets the coin_return variable to either 1 with probability 0.4 or to 0 with probability 0.6. After this, an `ASSIGN` increments the value of the variable x by the value of coin_return.

The if statement of the code is converted to two edges with opposing `ASSUME`s, similarly to the while loop. Non-deterministic assignment is done through an edge with a `HAVOC` statement. The body of the loop ends with a `SKIP` statement leading back to the head of the loop. The assert call in the code is converted to two `ASSUME`s, one leading to the (non-erroneous) final state, the other leading to the error state.

Figure 1b shows another possible PCFA for the same program, where large block encoding is used to create a more compact model: a single transition here applies multiple statements in succession. This will be explained in detail in Section 4.

2.3 Markov Decision Processes

Definition 2 (Markov Decision Process)**.** *A* Markov Decision Process (MDP) *is a tuple $M = (S, s_{init}, A, Av, \delta)$ where*

- *S is a finite set of* states
- *$s_{init} \in S$ is the initial state*
- *A is a finite set of actions*
- *$Av : S \rightarrow 2^A$ is a function mapping states to the set of* available actions *in the state*
- *$\delta : S \times A \rightharpoonup \mathbb{D}(S)$ is the transition function of the MDP. It is a partial function, which must be defined for every (s, a) where $a \in Av(s)$. For such a pair, it gives the distribution of successor states after taking the action a in state s.*

The transition notation $s \xrightarrow{a} \mu$ is used for $\delta(s, a) = \mu$. We will also use the notation $\delta(s, a, s') = \delta(s, a)(s')$ for the probability of transitioning from s to s' when taking the action a.

A lot of analysis tasks on MDPs use the notion of reward functions on the MDP. A *state reward function* on an MDP is a function $\mathbf{r} : S \rightarrow \mathbb{R}_{\geqslant 0}$ that assigns rewards to each state in the MDP.

A *strategy* on an MDP is a function $S^* \rightarrow \mathbb{D}(A)$ that assigns a distribution over actions to all possible state sequences. A *memoryless* strategy depends only on the last state. A *deterministic* strategy uses only dirac distributions.

In this paper, we will focus on *probabilistic reachability properties*: computing the probability of reaching a given set of *target states*. Such properties can be formulated using rewards: the reward function assigns 1 to the target states, and 0 to all others. The target states must be made absorbing. The expected value of the accumulated reward until absorption with a fixed strategy gives the probability of reaching one of the target states. Computing optimal strategies thus can give the maximal or minimal probability of hitting an error if the targets are the error states of the system.

2.4 MDP semantics of probabilistic programs

The denotational semantics of probabilistic programs can be defined by deriving an MDP from the probabilistic control flow automaton of the program.

Definition 3 (MDP of a PCFA)**.** *The MDP describing the semantics of the PCFA* $P = ((L, E), l_{init}, Var, v_{init}, S, Stmt)$ *is* $M = (S, s_{init}, A, Av, \delta)$*, where*

- $\mathcal{S} = L \times U_{Var}$
- $s_{init} = (l_{init}, v_{init})$
- $\forall e = (l, l') \in E, v \in U_{var} : (v \textit{ enables } Stmt(e)) \iff (\forall d \in Stmt(e)(v) : \exists! a \in A : a \in Av((l, v)) \wedge \delta((l, v), a) = join(l', d))$,
where the distribution $join(l, d) \in \mathbb{D}(L \times U_{var})$ *is defined by* $join(l, d)((l, v)) = d(v)$*, and for any other location, it is 0. This condition defines* A, Av, δ*.*

A state can be identified by selecting the location of the program and the value of each variable. Therefore, the state-space of the derived MDP is the Cartesian product of the set of program locations and valuations of the variable set. The initial state of the MDP is derived from the initial location and initial valuation of the PCFA.

The edges of the PCFA are transformed to actions of the MDP. For each state $s = (l, v_0, ..., v_{|Var|})$ of the MDP, we take the edges whose statement is enabled by the valuation of the state. For each such edge, we compute the possible distributions resulting from applying the statement of the edge. Let l' denote the ending location of the edge. For each computed distribution, we assign a new action to s leading to that distribution, with l' added as the location part of the states.

Example. The MDP obtained from the semantics of our running example can be seen in Figure 2. For the sake of readability, we used the PCFA with large-block encoding (Figure 1b) to reduce the number of states. The value of each variable is unknown at the beginning, so the initial data state can be anything. To make the figure easier to read, the possible initial states are merged and the unknown values are indicated by question marks. As a statement cannot lead to both probabilistic and (proper) non-deterministic choice in the next state, we have either only a single action optionally with multiple states in the next state distribution or multiple actions, each leading to dirac distributions in each state. Green nodes highlight the states where a probabilistic decision is performed, the red node highlights the non-deterministic decision (multiple possible actions).

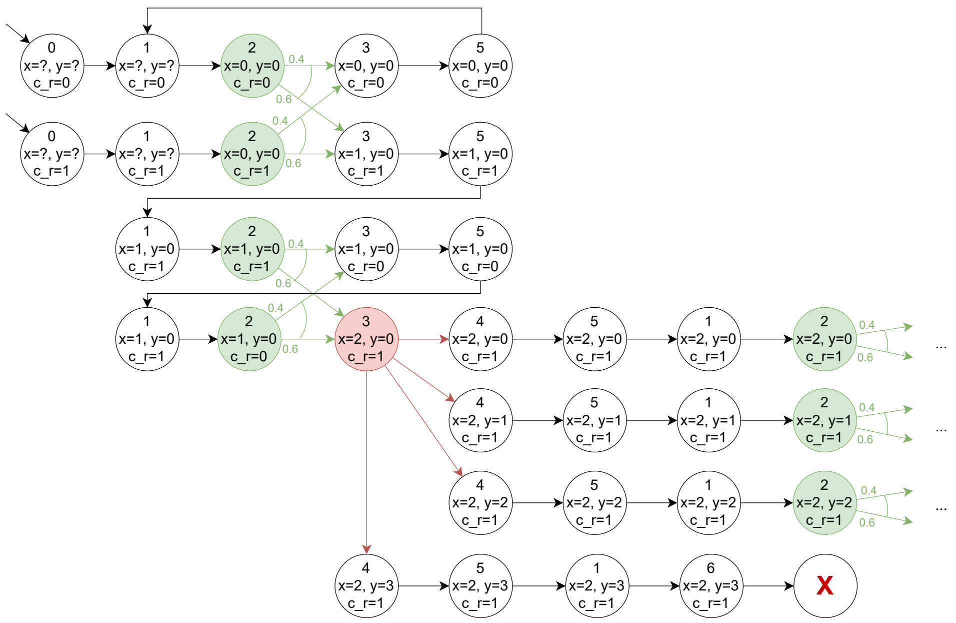


Figure 2: MDP of the running example corresponding to the PCFA in Figure 1b. c_r stands for the auxiliary variable coin_return.

2.5 Stochastic Games

Stochastic games are an extension of MDPs where instead of a single agent, two different players make decisions. This allows representing two separate kinds of non-determinism, which are resolved according to two separate strategies with different objectives. The relevance of Stochastic Games to this work is that they are used as abstract models for abstraction-based analysis of probabilistic programs.

Definition 4. *A* Turn-Based Stochastic Two-Player Game *is a tuple* $G = (S = S_1 \uplus S_2, s_{init}, A, Av, \delta)$*, where:*

- *S is a finite set of* states*, partitioned into the sets of Player 1 (S_1) and Player 2 (S_2) states*
- *$s_{init} \in V$ is the initial state*
- *A is the set of* actions
- *$Av : S \to 2^A$ is a function mapping each state to the set of available actions.*
- *$\delta : S \times A \rightharpoonup \mathbb{D}(S)$ is the transition function of the game. It is a partial function, which must be defined for every (s, a) where $a \in Av(s)$. For such a pair, it gives the distribution of successor states after taking the action a in state s.*

Definition 5 (Play)**.** *A* play ω *in a two-player simple stochastic game* $G = (S = S_1 \uplus S_2, s_{init}, A, Av, \delta)$ *is a (possibly infinite) sequence of state-action pairs* $\omega = (s_1^1, a_1^1)(s_1^2, a_2^1)(s_2^1, a_2^1)(s_2^2, a_2^2)\ldots$ *such that*

- $s_1^1 = s_{init}$
- $\forall s_j^i : a_j^i \in Av(s_j^i)$
- $\forall s_2^i : \delta(s_1^i, a_1^i, s_2^i) > 0$
- $\forall s_1^i : \delta(s_2^{i-1}, a_2^{i-1}, s_1^i) > 0$

In most applications, the goal of one player is to *minimize* the (expected) reward gathered during a play according to a specified reward function, while the other player aims to *maximize* it. A *reward function* assigns rewards to states or transitions. When the game enters a state or takes a transition, the reward associated with that state or transition is collected.

Probabilistic reachability properties can be formulated as a total reward computation problem the same way as for MDPs: the reward function assigns 1 to the target states, and 0 to all others, and the target states must be made absorbing.

Throughout this work, the shortened name *Stochastic Game (SG)* will refer to turn-based stochastic two-player games.

A special case is when the goals of the two players coincide, which is equivalent to an MDP. Here, we will consider only those cases when the players have opposite goals, as the cooperative case can be analysed as an MDP.

Definition 6 (Strategy)**.** *A Player 1 (resp. Player 2) strategy in a stochastic game* $G = (S = S_1 \uplus S_2, s_{init}, A, Av, \delta)$ *is a function* $Strat : S^*S_1 \rightarrow \mathbb{D}(A)$ *(resp.* $S^*S_2 \rightarrow \mathbb{D}(A)$*) from the set of finite plays ending in a Player 1 (resp. Player 2) state to the set of distributions over Player 2 (resp. Player 1) states, such that* $\forall p \in S^*S_1 (resp. \forall p \in S^*S_2) : Strat(p)(a) > 0.0 \implies a \in Av(end(p))$*.*

For probabilistic reachability properties, there always exists a pair of memoryless deterministic Player 1 and 2 strategies that are optimal among all strategies [11]. This means that we can focus only on this finite subset of strategies, making synthesis of optimal strategies much easier than in the general case. We only need to determine a single optimal action to take for each state, not a distribution over actions for every possible state sequence.

If a fixed strategy is considered for each player, we can equip the set of plays on the game with a probability measure derived from the strategies and the probabilistic transition function of the game. Informally, the probability of a play is product of the probability of each transition taken in the play. The probability of a transition is the probability that the player takes an action that can lead to the given transition according to the fixed strategy multiplied by the probability that the action results in the given transition. A more precise formal specification of the probability measure can be found e.g. in [2, Chapter 10].

The reachability probability of a set of target states is the probability measure of the set of all plays that contain at least one target state.

From now on we will assume that the game is played by a maximizer and minimizer (referring to their goals with respect to the reward function), and refer to their state sets as S_{min} and S_{max} respectively.

Definition 7 (Value Function for reachability properties)**.** *The* value function $V : S \to \mathbb{R}$ *for a game* $G = (S = S_{max} \uplus S_{min}, s_{init}, A, Av, \delta)$ *with respect to a set of target states* $T \subset S$ *is the least solution to the Bellman equations:*

$$V(s) = \begin{cases} \max_{a \in Av(s)} V(s,a) & \textit{if } s \in S_{max} \\ \min_{a \in Av(s)} V(s,a) & \textit{if } s \in S_{min} \\ 1 & \textit{if } s \in T \\ 0 & \textit{if } T \textit{ is unreachable from } s \end{cases}$$

The importance of the value function comes from the fact that a strategy is optimal if and only if it always chooses an optimal action with respect to the value function.

The value of a state is equal to the expected accumulated reward if the corresponding optimal strategies are used. If the reward function is derived from a reachability property, the value of a state is equal to the probability that a target state is reached from that state if the players follow the optimal strategies, which is exactly what we aim to calculate for the initial state.

2.6 Abstraction-based analysis

The aim of abstraction is to make the analysis of a model or program tractable by analyzing an abstract model with a smaller state-space than the original model under analysis. This is done by discarding information which is deemed not useful for proving or disproving the target property.

The resulting state-space is called the *abstract state-space*, while the original one is called the *concrete state-space.* As a new state space is created, the transition relation of the system is also lifted to this state space, resulting in the *abstract transition function.*

Discarding information must be performed in a conservative way: the abstract model must be built in such a way that the property being checked is satisfied by the abstract model *only if* it is satisfied by the original system.

The reverse need not be true. If the target property is disproved in the abstraction, we can try to *concretize* the proof of non-satisfaction, mapping the abstract counterexample back to the concrete state-space, thereby proving that the original system does not satisfy the target property either. Obviously, this can only be done if the original system does not satisfy it.

Different approaches for retaining only a fraction of the original information is captured by different *abstract domains.* Abstract domains define what kind of information can be contained in an abstract state and specify the correspondence between concrete and abstract states.

The notion of *abstraction precision* is used to differentiate between retaining different parts of the originally available information.

Abstract domains can have a convenient property called *disjointness*, which means that for a given precision, exactly one abstract state corresponds to each concrete state. In other words, the abstract states considered as sets of concrete states are disjoint for a fixed precision. This makes soundness proofs often much simpler than the general case.

2.7 Predicate Abstraction

Predicate abstraction can be applied to systems described symbolically using a set of state variables. It creates an abstract state-space by defining a set of *predicates*: Boolean expressions over these variables. The precision of the abstraction is the set of predicates.

As the concrete states assign exactly one value to each program variable, each predicate can be evaluated unambiguously for a concrete state. The abstract state corresponding to the concrete state is the Boolean vector resulting from evaluating all of the predicates in the precision against the variable valuation of the concrete state.

Although we can always evaluate the predicates for the concrete states, abstraction is often used exactly because exploration of the concrete state-space is intractable. Instead of exploring the full concrete state-space and mapping the concrete states, abstraction-based analysis techniques explore only the abstract state-space.

Exploration of the abstract state space needs a method for determining which actions are available in a given abstract state, and generating the possible next states after applying one of these actions. This is generally done by solving some form of a SAT (satisfiability) problem. As software involve not only Boolean variables, but other types like integers as well, their operations also have to be encoded into the SAT problem. One option for this is to use bitwise encoding of all types and use standard SAT solvers. This was the approach used in [18].

However, modern model checkers mostly employ *Satisfiability Modulo Theories (SMT)* solvers to handle non-Boolean variables [1]. In this case, the logic is augmented with theories that can handle the operations of variable types. In our implementation, we rely on SMT encoding instead of bitwise SAT.

Example. If in the current state we know that the predicate $x < 3$ is true and $y = 0$ is false, and we apply the statement $ASSIGN(x, x+1)$, the SMT problem for the next state computation is:

$$\underline{x_0 < 3 \land \neg(y_0 = 0)} \land \underline{x_1 = x_0 + 1} \land \underline{(x_1 < 3 \iff a_1) \land (y_0 = 0 \iff a_2)}$$

The first underlined part corresponds to the current state, the second underlined part to the statement applied, and the third part is a representation of the next state.

The Boolean literals a_1 and a_2 are called *activation literals.* All satisfying models of the above expression are computed, and the next abstract states are derived from the values of a_1 and a_2 in each model: the truth value of each predicate is set to the value of the corresponding activation literal.

When the model under analysis is a control flow automaton, abstraction is commonly applied to the data variables, and the location is also added to the abstract states (i.e. it is always precisely tracked).

2.8 Game-based Abstraction Refinement

The game-based abstraction refinement framework for probabilistic systems has been proposed in [25], and its application to probabilistic programs described in [18].

The main idea of this abstraction scheme is to use a stochastic game as the abstract model of an MDP, which allows introducing a second kind of non-deterministic choice which is resolved using a different objective than the one in the original model. This way we can introduce an "abstraction player" (referred to as Player A from now on), whose responsibility will be intuitively to resolve the choice of which concrete state we are in when we know only the abstract state (which represents a set of possible concrete states). The other player will be referred to as Player C, as its responsibility is resolving the original non-determinism already present in the *concrete* model.

By setting the objective of Player A to minimizing the probability of reaching the target state set, we can compute a lower bound for the original probability. By maximizing it instead, we get an upper bound. By computing both bounds, we get not only a conservative approximation of the error probability, but also a measure of how precise the current abstraction is with respect to the property of interest.

Refinement is performed based on the difference between the minimizing and the maximizing strategies of Player A. A state is refinable if the minimizing and the maximizing strategies choose a different action in it. The new precision is computed such that it prevents making this choice.

Two different strategies were proposed in [19] for choosing the state to base the refinement on. We can choose the coarsest refinable state, for which the difference between the upper and the lower abstraction values is the highest (hoping that this will have the largest effect on the precision of values), or choose the refinable state nearest to the initial state (hoping that this will make the abstraction precise enough early in the state space exploration process).

After a refinable state has been chosen, a new predicate is computed to eliminate the choice of the abstraction player in this state using weakest precondition computation.

When analysing a PCFA, it is possible to use a different precision for abstracting the data state in each location. This is called *local* precision, while using a single one for the whole model is called *global* precision. When local precision is used, the new predicate that was computed is added only to the location corresponding to the chosen state to refine. However, propagating the newly chosen predicate to other locations using weakest precondition computation can lead to a much lower number of refinement steps needed. [19] proposed two different strategies for this propagation, one based on the explored abstract state space, the other based directly on the CFA model.

2.8.1 Computing the abstraction game

Here we will briefly describe how the abstraction game is built in the case of predicate abstraction applied to probabilistic programs. An example step of building the game can be seen in Figure 3 for each statement type. A more detailed description and the SMT formulae used can be found in [19].

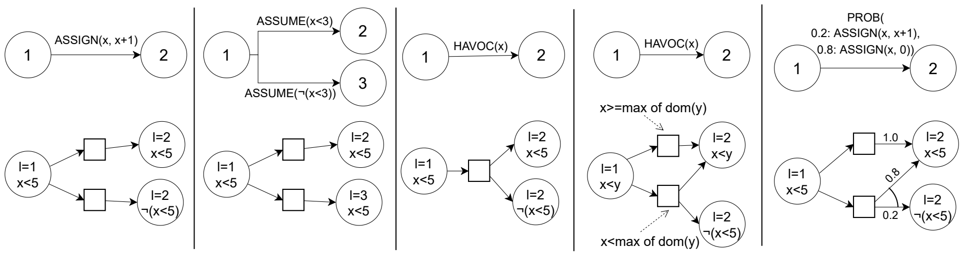


Figure 3: Example PCFA components and the resulting game parts. Circles represent Player A nodes, rectangles represent Player C nodes.

The construction method assumes the following constraints to be true for the model, which are always satisfied by PCFA models created using single-block encoding, and are also satisfied by our large-block encoding implementation:

- If multiple outgoing edges are present in a location, then only one of them can be enabled in any valuation. Because of this, the only source of non-determinism in the model can be `HAVOC` statements.
- If there is a `PROB` or `HAVOC` statement present on an outgoing edge of a location, then there must be no other outgoing edges.

The game used as abstract model alternates between Player A and Player C nodes. Player A nodes correspond to abstract states. For each Player A node (starting with the node corresponding to the initial abstract state), we need to compute groups of next states resulting from applying a statement available in the current state.

A-nodes are labeled by abstract states, while C-nodes get their identity from the set of A-node distributions it can choose from. When building the abstraction game, we take an A-node that has not been expanded yet, take a statement available in it, and compute the result of the statement.

`ASSIGN` and `ASSUME` statements When the statement to apply is an `ASSIGN` or an `ASSUME` statement, only the abstraction player has power over the resulting state.

The result of an assignment and the satisfaction of the assumption depends on the current values of the variables, but we might not know all variables that affect the result exactly with the currently used precision. Thus the abstraction player can select these values arbitrarily as long as they do not contradict what we know from the currently tracked predicates. As there is no concrete non-determinism involved with these statement types, Player C has no power in this state.

`PROB` statements Similarly to non-probabilistic assignments and assumes, only Player A has any power over choosing the result of a `PROB` statement (as we disallowed non-deterministic statements inside probabilistic statements in the definition). In this case however, we have to compute a set of *distributions* over abstract states from which the abstraction player can choose, not stand-alone states. Because of this, the successor states must be computed *in groups*, all elements of a possible distribution at a time. For the example in Figure 3, we need the following query:

$$x_0 < 5 \wedge (x_1^{(1)} = x_0 + 1 \wedge (x_1^{(1)} < 5 \iff a_1^{(1)})) \wedge (x_1^{(2)} = 0 \wedge (x_1^{(2)} < 5 \iff a_1^{(2)}))$$

We are interested in all satisfying models that differ in at least one of $a_1^{(1)}$ and $a_1^{(2)}$, because these models encode different next state distributions. The satisfying models in this example are $\{a_1^{(1)}, a_1^{(2)}\}$, representing the dirac distribution $\{1.0 : (l = 2, x < 5)\}$, and $\{a_1^{(1)}, \neg a_1^{(2)}\}$ representing the distribution $\{0.2 : (l = 2, x < 5), 0.2 : (l = 2, \neg(x < 5))\}$.

We cannot simply evaluate the possible successors for the statements independently: the results of applying the statements depend on each other through the unknown variable values. Computing the possible results of the statements, and then composing them into a set of distributions by taking a Cartesian product would lead to "hallucinating" some successor distributions over the abstract states that cannot result from applying the `PROB` statement to any concrete state. This would be conceptually similar to the inexactness that Cartesian abstraction introduces, described in Section 6.

General `HAVOC` statements Handling `HAVOC`s is the most problematic part, as in the general case, both players can have power over the resulting abstract state. To compute the set of possible next states and group them appropriately, we have to introduce a term in the SMT query for each possible new value of the variable modified by the `HAVOC`, similarly to what we do for each substatement of a `PROB`. This is intractable if the domain of a variable is large.

Simplifiable `HAVOC` statements However, if we can split the information tracked by the current precision so that the predicates one partition depends *only* on the variable modified by the `HAVOC`, and the ones in the other partition *do not* depend on that variable at all, then the computation can be simplified: in this case, only Player C has any actual power over next state selection: it can decide how the first partition changes, while the second partition must stay the same as it was. Thus we know that only one group has to be computed, making a standard flat list of successor states sufficient. For this, a standard SMT query for the successors is enough.

3 Incorporating Bounded Value Iteration

Value iteration (VI) computes an optimal strategy by iteratively approximating the value function of the game. The resulting strategy is derived from the value function using the fact that any strategy that always chooses an optimal action with respect to the value function is optimal.

The algorithm iteratively refines a lower bound approximation to the value function using the Bellman equations. The initial approximation is set to 1 for every target state, and 0 everywhere else.

The approximate value function computed during value iteration converges to the real value function only asymptotically. Because of this, unlike for strategy iteration, it is not guaranteed that the algorithm terminates in a finite number of steps. However, the value function often becomes good enough to compute the optimal strategy in much less time than it would take strategy iteration to converge. Value iteration is often preferred in practice for this reason.

The problem of determining when the value function is good enough still remains. A very basic, but often used approach is to stop value iteration when the difference between two consecutive value function approximations becomes smaller than a predefined threshold. This stopping criterion cannot actually give any guarantees about the optimality of the resulting strategy: as long as the change of the value function is not exactly zero in an iteration, the chosen transition can change in any state, potentially leading to a vastly different mean reward [3].

Bounded Value Iteration (BVI) [24, 3] is a modified version of value iteration originally proposed for Markov Decision Processes. It has been extended to Stochastic Games in [20]. BVI computes both a lower and an upper approximation for the value function. The reason for this is that the difference between the upper and the lower bound can be used as stopping criterion. As the two approximations are constructed in such a way that the real value function is known to be between them, the error is no longer totally unknown, we have an upper bound for it, and can thus be controlled. This leads to a guaranteed ε-optimal strategy as a result. Making the upper approximation converge on stochastic games is non-trivial - for a detailed explanation of the algorithm, see [20].

Having more control over the optimality of strategies computed using the approximate value function has two advantages in the game-based abstraction scheme: apart from the general advantage of making the end result more precise, the refinement step is also enhanced by having a more reliable value function.

We constructed a simple example, where the problem of ineffective refinement can be easily seen to highlight the importance of more reliable convergence checks when computing values in the game-based abstraction refinement scheme.

Based on the model given in [3] as an example for when the standard VI convergence check can lead to a wrong result, we can construct an example for any ε, where a seemingly coarse abstract state is actually perfectly precise: the values of a state with maximizing and minimizing abstraction choices coincide, the difference between lower and upper abstraction comes only from the imprecision of value computation.

Example. An example where the unreliable result of standard value iteration can be problematic for refinement can be seen in Figure 4. The only non-determinism in the abstract state space is an abstraction choice in s_0. As there is no concrete non-determinism, the Player C nodes between the Player A nodes have been omitted from the figure. The target states are c_n and s_1.

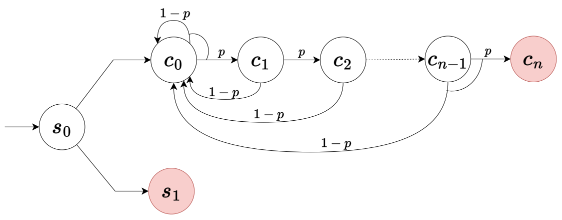


Figure 4: Example model for refinement problem with standard value iteration

The upper part of the model is the chain from [3]. The parameters n and p can be chosen appropriately for any ε such that if standard value iteration is stopped when the latest change was smaller than ε, then the final approximation of the probability of reaching a target from s_0 is less than $\frac{5}{8}$, while the real probability is 1.

The lower part of the model is simply an instant target state, so both the approximate computed probability and the real probability of reaching a target by choosing the lower action are 1.

If the approximate values computed with standard VI are used to compute the refinement, the abstract state s_0 seems to be quite coarse: the computed value difference between the maximizing and the minimizing action is greater than $\frac{3}{8}$. In reality, though, s_0 is perfectly precise with respect to the property of interest, as both actions eventually lead to a target state with probability 1. This can lead to sub-optimal refinement if the refinable state selection strategy is set to *coarsest*: if the given example is only a part of the state space, an actually coarsely abstracted part would be much more important to refine.

If qualitative pre-analysis is used to determine almost sure reachability precisely before running VI, the example model can be modified to "leak" some probability from the chain before reaching the target.

We have also implemented the topological version of VI and BVI [22]: this approach splits the game into the strongly connected components of its edge relation graph, and computes the value function for each component in reverse topological order. This change can often speed up the convergence of value iteration for structurally problematic models, as it enforces propagation of already converged values instead of propagating non-converged values through the whole state-space.

4 Adapting Large-block Encoding

When creating a (P)CFA from the source code of a computer program, the most obvious approach is to create an edge for every atomic statement in the code, as seen in the example in Figure 1a. A disadvantage of this conversion is that it introduces a high number of locations, leading to a wastefully large state-space. This version is called *single-block encoding (SBE).*

Another possibility is *large-block encoding (LBE).* There are several possible versions of this, but the general idea is that an edge is associated with multiple statements. By allowing a single edge to hold multiple statements, we can reduce the number of edges and locations, making the graph of the (P)CFA much smaller. This can both speed up the analysis, and make the results easier to interpret. We can introduce composite statements to be able to do this while staying with our original definition of PCFA.

There are multiple variants of composite statements with different semantics, but our current work uses only *sequence statements (SEQ)* representing a sequential composition of other statements (the statements given in an ordered list are executed one after another according to their order in the list).

A SEQ statement is enabled by a valuation if and only if its first substatement is enabled by the valuation, and each substatement is enabled by the result of transforming the valuation through all substatements before it.

An example PCFA with large-block encoding can be seen in Figure 1b.

The idea of large block encoding can be implemented in several different ways. For example, the authors in [6] used it for standard qualitative software model checking and merged both sequential and parallel (non-deterministic and if-else) edge combinations into single edges.

However, in the probabilistic abstraction case, doing this would prove problematic: using a non-deterministic statement to combine guarded choices would make the edge look like a concrete non-deterministic choice, while it is actually an abstraction choice, or it could merge the choices of the two players into a single edge making building the game much more complex depending on the implementation. Another difference from the qualitative case is that the game construction algorithm of [18] needs all locations of the PCFA to be either choice, non-deterministic or probabilistic, clearly separating ASSUME statements from HAVOCs and PROB.

Accordingly, we decided to implement only sequential LBE for the probabilistic case, and defer the exploration of possibilities for merging parallel edges to future work, and we split the LBE at PROB and HAVOC statements, making those have their own edges separated from the sequences between them. This sequential LBE is implemented as a simple preprocessing step on the PCFA: $l_1 \xrightarrow{stmt_1} l_2 \xrightarrow{stmt_2} l_3$ patterns, where l_2 has no other edges are merged to $l_1 \xrightarrow{SEQ\{stmt_1, stmt_2\}} l_3$. These statements can be converted to SMT expressions by converting each substatement into an expression with the appropriate indexing, and taking the conjunction of these expressions.

5 Adapting Explicit Abstraction

Explicit abstraction (also called explicit-value abstraction and visible variables abstraction) [7] chooses for each variable if it is visible or not. This means that for each variable, its value is either tracked exactly, or it is not tracked at all. The precision of the abstraction is the list of tracked variables. Each abstract state assigns an exact value to each variable tracked according to the precision used.

To use the explicit domain, we needed to adapt the game construction algorithm of [18] to it. Constructing the abstraction game was originally formulated only for predicate abstraction, so we needed to apply the main ideas of the grouped successor computation to the explicit domain. Classical software model checking, which explicit abstraction was previously used for, did not need *grouped* successor computation in the abstract model construction, as the abstraction-induced nondeterminism did not need to be handled separately from the original (concrete) nondeterminism. Because of this, implementing the game-based abstraction scheme with explicit abstraction involved determining how the flat successor computation must be modified for the grouped computation needs.

One advantage of the explicit domain is that instead of always relying on expensive SMT solver calls to compute the successor states, we can often simply evaluate the program statements after all variables currently tracked have been substituted with their values in the current abstract state. SMT calls are needed only if an important variable is missing, and even then, the SMT query often contains much fewer variables than without substituting the known ones.

For deterministic (`ASSIGN` and `ASSUME`) and `HAVOC` statements, explicit abstraction can be used the same way as in the qualitative case, as we compute only a flat list of next states, and assign the choice between them either to Player A (for deterministic statements) or Player C (for `HAVOC` statement). In the case of `HAVOC`, this is correct because the simplification described earlier is always applicable when using explicit abstraction: a `HAVOC` statement always affects only one variable, and no matter what it was, Player C can set it to any value. For `PROB` statements, we simply use the same approach as in predicate abstraction: the state is converted to a Boolean expression as a conjunction of equalities for the known variable values, and then we can solve the same SMT problem as in the predicate case. When LBE is used, `SEQ` statements are handled the same ways as deterministic statements, as they can contain only `ASSIGN`s and `ASSUME`s in our version.

Refinement is also similar to the predicate case, first discovering new predicates using weakest precondition computation for splitting the state selected for refinement and propagating if needed. The states are converted to Boolean expressions for this computation by taking a conjunction of equalities for the known variable values. Then, instead of adding the newly discovered predicates to the precision, we add all variables contained in them - just like in the qualitative version of explicit abstraction.

6 Inexact Abstract Transition Functions for Game-based Abstraction

The exact computation of the abstract transition relation is often computationally very expensive, sometimes infeasible. Because of this, inexact transition functions are often used in standard software model checking that conservatively overapproximate the original relation. Two examples of this are Cartesian abstraction in the case of predicate abstraction and limiting the number of next states enumerated for a single successor state set computation in the case of explicit abstraction.

Here, we will first describe these and their application in the game-based abstraction scheme for probabilistic programs, then state two theorems related to the correctness of using them. The proofs of these theorems are relegated to the appendix.

The approximation introduced by these techniques can be considered a second layer of abstraction. Instead of abstracting the state-space itself, it abstracts the transition function, as it discards some information about the original relation.

Inexact transition functions need more flexible abstract domains than the ones described before, which allow on-demand omission of information during exploration. These domains do not have the disjointness property: for a given precision, multiple abstract states can correspond to the same concrete state, as the precision is not strictly respected by all abstract states. Thus, the correctness of the abstraction scheme must be reevaluated when using inexact transition functions.

We first describe the two inexact abstract transition functions that we implemented, Cartesian abstraction [4] for predicate abstraction and limited enumeration [15] for explicit abstraction, along with how we adapted them to the game-based abstraction refinement scheme. After that, we elaborate on the correctness of their usage.

6.1 Cartesian abstraction

In Section 2, we described the standard version of predicate abstraction, where each predicate in the current precision must be stated to either hold or not in each abstract state. A more general version of predicate abstraction uses three-valued logic for the predicates: true, false and unknown/any.

The generalized version makes it possible to use (conservative) approximations of the exact abstract transition relation which leave predicates unknown instead of branching in both directions during exploration.

One such approximate computation of the abstract transition relation is *Cartesian abstraction*. Instead of evaluating the SMT problem with the whole Boolean vector of the next state, it computes for each predicate whether it or its negation is implied by applying a statement to the current abstract state. If none of these is implied, the predicate is set to unknown in the subsequent abstract state. If the predicate itself is implied, it is set to true, if the negation is implied, the predicate is set to false. Implying both means that the conjunction of the current state expression and the statement expression is unsatisfiable, leading to an empty set of next states (as the statement is not enabled by the current state).

This is an over-approximation of the exact abstract transition function. Consider the case for example when two predicates must be the opposite of each other in the subsequent abstract state, but we do not know which one of them is true. Because of this, both are set to unknown. This abstract state contains all of the possible concrete states, but also contain states where both of the predicates are true or both are false.

Our proposed adaptation of Cartesian abstraction is applying it only partially in the game construction. As Cartesian abstraction modifies the method which we use for the next state computation, we needed to decide how to use that in the game construction.

For deterministic statements and simplified `HAVOC` statement computation, Cartesian abstraction can be used the same way as in the qualitative case, as we compute only a flat list of next states, and assign the choice between them either to Player A (for deterministic statements) or Player C (for `HAVOC` statement when simplification is applicable).

For `PROB` statements and general `HAVOC` statements, we cannot break the grouped SMT problem into atomic implications without introducing a high number of spurious distribution choices (it would still be a conservative abstraction, but a very imprecise one). Because of this, we propose to use Cartesian abstraction in the probabilistic case only partially, and use the precise computation for `PROB` and non-simplifiable `HAVOC` statements.

As most probabilistic programs have only a small amount of `PROB` statements, and `HAVOC` statements are simplifiable in most cases, Cartesian abstraction can still be much more efficient than the precise abstract transition relation.

Another possibility would be to compute an implication for each predicate for all next states in one SMT problem. In this case, instead of computing if the negated or ponated version is implied, we would need to check if any *combination* of ponated and negated versions of the given predicate in each element of the next state group is implied. This is tractable for `PROB`s with a small number of substatements (like `coin()`), but the number of SMT problems for each predicate blows up exponentially with the number of substatements, so we decided against this option.

Investigating other ways to apply the idea of Cartesian abstraction to the next state computations used in the abstraction game construction is part of our future plans, as the grouped computation parts could also benefit from the idea, but care has to be taken to not make the abstraction either unsound or too coarse.

6.2 Explicit abstraction with limited enumeration

The version of explicit abstraction described earlier has the disjointness property. However, there is a more general version of explicit abstraction, which is much more useful: in an abstract state, a variable which is tracked according to the precision can also be set to $\top$, meaning that the value is unknown in that state. This is useful for example for `HAVOC` statements, where the abstract state space could be infinite, or at least intractably large.

In this generalized domain, the abstract state space exploration can set the value of a variable to unknown when there would be too many possible values to evaluate. It is often possible to encounter an `ASSUME` statement or `ASSIGN` statement somewhere after the `HAVOC`, which constrains the possible values of the variable to a set of manageable size again.

The technique of limited enumeration for explicit abstraction means that we set a limit for how many possible successor states we are willing to compute in a single successor set computation. If the number of states would exceed this limit, we merge them into a single state by selecting all the variables that have different values in the possible next states, and set their value to $\top$, even though they should be tracked according to the precision. Unlike in the predicate case, explicit abstraction can be used only for a very small subset of programs without this generalization.

Using explicit abstraction with limited enumeration in the game-based abstraction refinement scheme is quite straightforward.

Similarly to the unlimited explicit case, when computing `ASSIGN`, `ASSUME` and `HAVOC` statements, the successor computation is done the same way as in the qualitative case, no adaptation is needed, as a non-grouped list of successors is sufficient for the construction. In the case of `HAVOC`, although Player C has full power in theory, limited enumeration will lead to Player A having full power over the value of the variable in practice if its domain is larger than the enumeration limit.

`PROB` statements are computed similarly to the unlimited explicit case, but because of the enumeration limit, resulting states might have to be merged. If the states that would be merged are in the same distribution, we can merge them in the distribution by replacing them with a single next state with original probabilities summed. If the enumeration that must be limited is in the different distribution choices, then we can instead merge them into a single distribution, where the variable is set to $\top$ in the next state for the corresponding substatement which caused the enumeration exceeding the limit.

6.3 Correctness

Here, we analyze the correctness of applying these techniques in the game-based scheme.

For an abstraction-based analysis scheme to result in a correct verdict about the analyzed property, the construction used for the abstract model must be *sound*, meaning that whenever the property is provable for the abstract model, it is true in the original model. In the context of game-based abstract analysis of stochastic models, this means that analyzing the game with the abstraction player aiming to minimize/maximize the objective results in valid lower and upper bounds respectively.

The original soundness proof in [25] assumed that the abstract domain used for the abstraction has the disjointness property. However, this does not allow the usage of generalized predicate and explicit abstraction, which Cartesian abstracion and limited enumeration depend on. [13] proved that the approach can be extended to non-disjoint domains. However, the context in that work was abstract interpretation with widening, and they used a slightly different construction for the abstraction game. Because of this, a new proof of soundness is necessary for the adaptation of Cartesian abstraction and explicit abstraction with limited enumeration described above.

Here, we only state the soundness theorems formally, and the proofs themselves are relegated to the appendix.

Theorem 1 (Soundness of Cartesian abstraction)**.** *Given a PCFA P with a set of target locations L_T, optimal strategies on the game abstraction of its MDP semantics using Cartesian abstraction to compute the abstract transition relation gives lower and upper bounds on the reachability probability of L_T. The optimal strategies are understood w.r.t. the reward function that assigns* 1 *to states with a location in L_T and* 0 *to all other states.*

Theorem 2 (Soundness of limited enumeration)**.** *Given a PCFA P with a set of target locations L_T, optimal strategies on the game abstraction of its MDP semantics using explicit abstraction with limited enumeration to compute the abstract transition relation gives lower and upper bounds on the reachability probability of L_T. The optimal strategies are understood w.r.t. the reward function that assigns* 1 *to states with a location in L_T and* 0 *to all other states.*

These correctness proofs make it possible to use Cartesian abstraction and explicit abstraction with limited enumeration in the game-based abstraction refinement scheme, guaranteeing that the maximizing and minimizing strategies lead to upper and lower approximations of the concrete value function respectively, which result in a correct final verdict. Keep in mind, that these results only state *soundness*, not *completeness*: using inexact transition functions without any other modifications to the overall scheme does not lead to a complete decision procedure, the verification might get stuck in the refinement loop, as the information that would be needed for more precise abstraction might be discarded by the transition function itself, which we do not refine.

7 Implementation and Numerical Experiments

We performed numerical experiments to evaluate our modifications: using large block encoding (LBE) in the PCFA model, using the explicit abstract domain, applying inexact abstract transition functions (Cartesian abstraction and limited enumeration), and enhancing the reliability of strategy computation using bounded value iteration (BVI). We used models from [18] which are standard benchmark model in this area. Some of the models are direct implementations of probabilistic protocols, while others are standard stochastic analysis benchmark models converted from PRISM models to probabilistic C code.

These models are scalable using scaling parameters, which makes it possible to analyze how the analysis approach scales with model size. Multiple properties to check are given for each model.

We used only a subset of the benchmark models, because the C frontend of Theta does not support multiple compilation units yet - this feature is work in progress, and further measurements are planned to be performed in the future. Applying the GCC preprocessor to create a single compilation unit was enough for some models originally given as multiple source files, but not all of them.

Our aim with the numerical experiments was to answer the following research questions:

RQ1. How does using large block encoding affect the running time of the analysis?

RQ2. How does using BVI instead of standard VI affect the running time? Is the impact on the running time sufficiently small to be an acceptable cost for higher reliability?

RQ3. How does the running time with the explicit domain differ from the predicate domain? What is the effect of using limited enumeration?

RQ4. How does applying Cartesian abstraction affect the running time?

RQ5. How prevalent is the problem of the refinement loop getting stuck with inexact transition functions on practical models?

The measurements were performed with our prototype implementation in the Theta model-checking framework. The following aspects of the analysis are configurable: abstract domain (including abstract next state computation method), refinable state selection strategy, precision locality (local: each PCFA location has its own precision, global: a single global precision is used), predicate propagation strategy (only applicable with local precision), stochastic game solver (VI, topological VI, BVI, topological BVI), SBE/LBE. Local precision (and because of this, refinement propagation) is not implemented for explicit abstraction yet. For enumeration limits in the explicit domain, we used ∞, 1, and 2.

We measured the performance of all possible configurations on all input models. Analyzing the effect of precision locality, refinement propagation strategies and refinable state selection strategies was not among the goals of these experiments. Therefore, we merged those configurations in the analysis results that differed only in these parameters by taking the smallest running time among them to reduce clutter in the plots. This results in an analysis where these parameters are always assumed to be chosen optimally.

The benchmarks were executed using *BenchExec* [8], the official tool used for the Software Verification Competition (SV-COMP) [2], providing reliable measurements

[2]https://sv-comp.sosy-lab.org/2022/

on resources. The experiment was executed on virtual machines running Ubuntu 20.04.2 LTS with Java OpenJDK 11.0.13. There was a memory limit of 15GB and a CPU core limit of 6 cores set. A timeout of 6 minutes was set for each input-configuration pair.

Figures 5, 6 and 7 show our measurement results with different groupings, aiming to make the effect of different features observable on each plot. The horizontal axis shows the value of the scaling parameter, while the vertical axis shows running time of the analysis. Each subplot has its own y scale in seconds, as the running time can differ greatly depending on inputs and configuration, so having a common scale would make data points with shorter running times incomparable.

Data points with 360s running time are timeouts, as that was the time limit.

The results provided the following answers for the research questions:

RQ1: Effect of LBE In general, LBE increased performance of the analysis. 240 input-configuration pairs succeeded before timing out with LBE, and only 154 input-configuration pairs without it. The difference can be easily seen in Figures 5 and 6 by comparing the blue (SBE) and red (LBE) data points, and in Figure 7 by comparing the number of not timed out data points in the two facet rows. LBE was beneficial regardless of the other configuration parameters. On the herman_P3 input, it even meant a difference between timing out or not.

RQ2: Effect of BVI The running time of BVI was close to the running time of standard VI, when the best refinable state selection strategy was chosen. This can

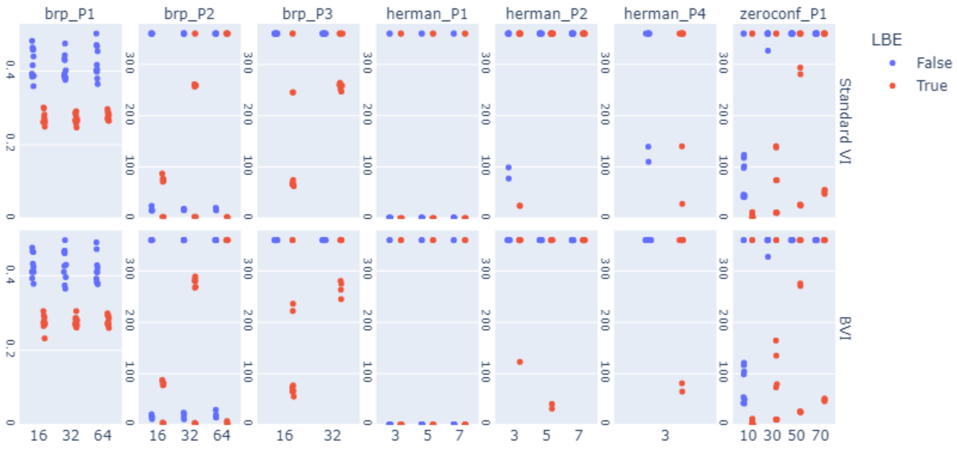


Figure 5: Strip plot showing the measured running times in seconds (note the different scaling in case of brp_P1) grouped by the input (model and analyzed property together) and the application of BVI. Color shows whether LBE was on (red) or off (blue)

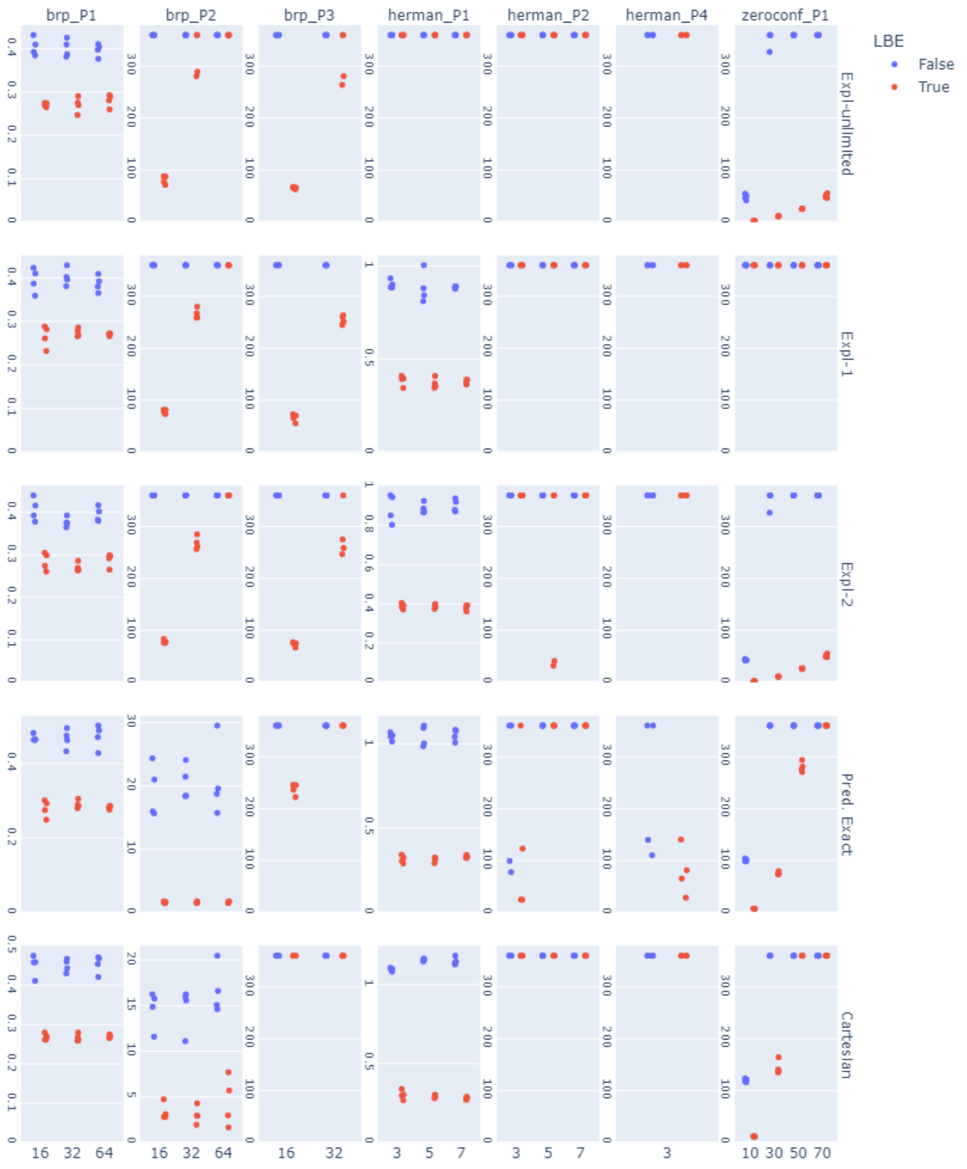


Figure 6: Strip plot showing the measured running times in seconds grouped by the input (model and analyzed property together) and the domain used (including the next state computation method). Color shows whether LBE was on (red) or off (blue)

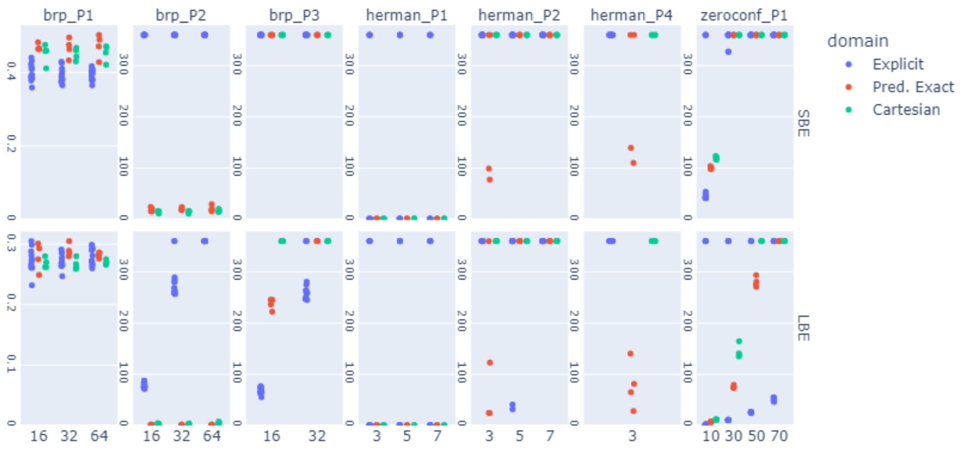


Figure 7: Strip plot showing the measured running times in seconds grouped by the input (model and analyzed property together) and whether LBE was used or not. Color shows which domain was used.

be seen most clearly in Figure 5, by comparing the two facet rows. We observed though that with other refinable state selection choices, BVI can be unacceptably slow compared to standard VI.

RQ3: Efficiency of the explicit domain Explicit abstraction performed better than predicate abstraction on some models. Neither is clearly better than the other, but having both option increases the number of models we can analyze. The unlimited version can often be problematic because of the infinite number of possible next states, and limit 1 often gets stuck, but explicit abstraction with enumeration limit 2 was able to analyze the most models without timing out among all the domain configurations. The different domain configurations can be compared on Figure 6 by comparing the facet rows (but the different y-axis scales might make exact comparison a bit harder), and in Figure 7 by comparing the differently colored data points. In the latter figure, we decided to not distinguish between the limits in the color to not clutter the plot with too many different colors.

RQ4: Effect of Cartesian abstraction Whenever Cartesian abstraction did not get stuck, it was often faster than exact predicate abstraction, but not significantly. In some cases, it was even slower and, unfortunately, the refinement loop did become stuck in a lot of cases. Because of this, these measurements did not lead to any significant answer for this research question. Cartesian abstraction can be compared to the other options in Figures 6 and 7.

RQ5: Refinement getting stuck Inexact next state computations were very prone to getting stuck in the refinement loop, because they dropped the information that would have been needed for refining based on the selected state.

Explicit abstraction with limit 2 is an exception when combined with LBE: it performed well on a lot of models, and it was able to solve the most analysis tasks overall among all domains. More precisely, if we count the number of solved inputs for each domain disregarding all other configuration parameters, Explicit abstraction with limit 2 was able to solve the highest number of tasks without timing out. With this limit, Boolean variables are always tracked exactly, while other variables are abstracted when multiple values are possible. This limit is also able to not merge probabilistic choices with two branches, while limit 1 will merge them, basically getting rid of probabilistic choices in the abstract model.

The problem of getting stuck might be mitigated by different refinable state selection approaches, or by on-demand refinement of the post operator itself. Further research is needed in this area.

8 Conclusions and Future Work

In this work, we set out to (1) improve the scalability of game-based abstraction of probabilistic programs by adapting modern techniques from qualitative software model checking, and to (2) tackle the problem of convergence checking for the abstract model.

Regarding (1) we can conclude the following based on our numerical evaluation:

- Introducing the explicit abstraction and limited enumeration options made the analysis of more models possible.
- Using large block encoding generally resulted in faster analysis.
- Cartesian abstraction and limited enumeration often get stuck in the refinement loop, so more sophisticated techniques are needed to make them usable.

Regarding (2), we can say that applying BVI does not incur a significant increase in running time, making it worth to use the more reliable version instead of standard VI. Performance of BVI was highly dependent on the refinable state selection strategy. We plan to investigate this phenomenon in detail in the future. We also plan to implement optimistic value iteration [16] and sound value iteration [26] adapted to stochastic games as abstract model solvers to compare them with BVI in the game-abstraction context.

Building on the conclusions we have drawn from these experiments, we plan to make inexact post operators more stable by introducing partial switching to exact computation as a refinement option.

References

[1] Armando, A., Mantovani, J., and Platania, L. Bounded model checking of software using SMT solvers instead of SAT solvers. *International Journal on Software Tools for Technology Transfer*, 11(1):69–83, 2009. DOI: 10.1007/11691617_9.

[2] Baier, C. and Katoen, J.-P. *Principles of model checking*. MIT Press, 2008.

[3] Baier, C., Klein, J., Leuschner, L., Parker, D., and Wunderlich, S. Ensuring the reliability of your model checker: Interval iteration for Markov decision processes. In *International Conference on Computer Aided Verification*, pages 160–180. Springer, 2017. DOI: 10.1007/978-3-319-63387-9_8.

[4] Ball, T., Podelski, A., and Rajamani, S. K. Boolean and Cartesian abstraction for model checking C programs. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 268–283. Springer, 2001. DOI: 10.1007/s10009-002-0095-0.

[5] Barthe, G., Katoen, J.-P., and Silva, A. *Foundations of Probabilistic Programming*. Cambridge University Press, 2020. DOI: 10.1017/9781108770750.

[6] Beyer, D., Cimatti, A., Griggio, A., Keremoglu, M. E., University, S. F., and Sebastiani, R. Software model checking via large-block encoding. In *Formal Methods in Computer-Aided Design*, pages 25–32, 2009. DOI: 10.1109/FMCAD.2009.5351147.

[7] Beyer, D. and Löwe, S. Explicit-state software model checking based on CEGAR and interpolation. In *International Conference on Fundamental Approaches to Software Engineering*, pages 146–162. Springer, 2013. DOI: 10.1007/978-3-642-37057-1_11.

[8] Beyer, D., Löwe, S., and Wendler, P. Reliable benchmarking: requirements and solutions. *International Journal on Software Tools for Technology Transfer*, 21(1):1–29, 2019. DOI: 10.1007/s10009-017-0469-y.

[9] Chadha, R. and Viswanathan, M. A counterexample-guided abstraction-refinement framework for Markov decision processes. *ACM Transactions on Computational Logic (TOCL)*, 12(1):1–49, 2010. DOI: 10.1145/1838552.1838553.

[10] Clarke, E., Grumberg, O., Jha, S., Lu, Y., and Veith, H. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM (JACM)*, 50(5):752–794, 2003. DOI: 10.1145/876638.876643.

[11] Condon, A. The complexity of stochastic games. *Information and Computation*, 96(2):203–224, 1992. DOI: 10.1016/0890-5401(92)90048-K.

[12] De Alfaro, L. and Roy, P. Magnifying-lens abstraction for Markov decision processes. In *International Conference on Computer Aided Verification*, pages 325–338. Springer, 2007. DOI: 10.1007/978-3-540-73368-3_38.

[13] Esparza, J. and Gaiser, A. Probabilistic abstractions with arbitrary domains. In *International Static Analysis Symposium*, pages 334–350. Springer, 2011. DOI: 10.1007/978-3-642-23702-7_25.

[14] Graics, B., Molnár, V., Vörös, A., Majzik, I., and Varró, D. Mixed-semantics composition of statecharts for the component-based design of reactive systems. *Software and Systems Modeling*, 19(6):1483–1517, 2020. DOI: 10.1007/s10270-020-00806-5.

[15] Hajdu, Á. and Micskei, Z. Efficient strategies for CEGAR-based model checking. *Journal of Automated Reasoning*, 64(6):1051–1091, 2020. DOI: 10.1007/s10817-019-09535-x.

[16] Hartmanns, A. and Kaminski, B. L. Optimistic Value Iteration. In *International Conference on Computer Aided Verification*, pages 488–511. Springer, 2020. DOI: 10.1007/978-3-030-53291-8_26.

[17] Hermanns, H., Wachter, B., and Zhang, L. Probabilistic CEGAR. In *International Conference on Computer Aided Verification*, pages 162–175. Springer, 2008. DOI: 10.1007/978-3-540-70545-1_16.

[18] Kattenbelt, M., Kwiatkowska, M., Norman, G., and Parker, D. Abstraction refinement for probabilistic software. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 182–197. Springer, 2009. DOI: 10.1007/978-3-540-93900-9_17.

[19] Kattenbelt, M. A. *Automated quantitative software verification.* PhD thesis, University of Oxford, 2010.

[20] Kelmendi, E., Krämer, J., Křetínskỳ, J., and Weininger, M. Value iteration for simple stochastic games: Stopping criterion and learning algorithm. In *International Conference on Computer Aided Verification*, pages 623–642. Springer, 2018. DOI: 10.1007/978-3-319-96145-3_36.

[21] Komuravelli, A., Păsăreanu, C. S., and Clarke, E. M. Assume-guarantee abstraction refinement for probabilistic systems. In *International Conference on Computer Aided Verification*, pages 310–326. Springer, 2012. DOI: 10.1007/978-3-642-31424-7_25.

[22] Křetínskỳ, J., Ramneantu, E., Slivinskiy, A., and Weininger, M. Comparison of algorithms for simple stochastic games. *Information and Computation*, page 104885, 2022. DOI: 10.1016/j.ic.2022.104885.

[23] Kwiatkowska, M., Norman, G., and Parker, D. PRISM 4.0: Verification of probabilistic real-time systems. In Gopalakrishnan, G. and Qadeer, S., editors, *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*, Volume 6806 of *LNCS*, pages 585–591. Springer, 2011. DOI: 10.1007/978-3-642-22110-1_47.

[24] McMahan, H. B., Likhachev, M., and Gordon, G. J. Bounded real-time dynamic programming: RTDP with monotone upper bounds and performance guarantees. In *Proceedings of the 22nd International Conference on Machine Learning*, pages 569–576, 2005. DOI: 10.1145/1102351.1102423.

[25] Parker, D., Norman, G., and Kwiatkowska, M. Game-based abstraction for Markov decision processes. In *Third International Conference on the Quantitative Evaluation of Systems-(QEST'06)*, pages 157–166. IEEE, 2006. DOI: 10.1109/QEST.2006.19.

[26] Quatmann, T. and Katoen, J.-P. Sound value iteration. In Chockler, H. and Weissenbacher, G., editors, *Computer Aided Verification*, pages 643–661, Cham, 2018. Springer International Publishing. DOI: 10.1007/978-3-319-96145-3_37.

[27] Song, L., Zhang, L., Hermanns, H., and Godskesen, J. C. Incremental bisimulation abstraction refinement. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(4s):1–23, 2014. DOI: 10.1145/2627352.

[28] Tóth, T., Hajdu, Á., Vörös, A., Micskei, Z., and Majzik, I. Theta: A framework for abstraction refinement-based model checking. In *Formal Methods in Computer Aided Design (FMCAD)*, pages 176–179. IEEE, 2017. DOI: 10.23919/FMCAD.2017.8102257.

[29] Wachter, B. and Zhang, L. Best probabilistic transformers. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 362–379. Springer, 2010. DOI: 10.1007/978-3-642-11319-2_26.

A Proofs of Soundness

A.1 Notions used in the proofs

Here we show that both Cartesian abstraction with the generalized predicate domain and the usage of the generalized explicit abstraction domain with limited enumeration of values lead to sound abstractions. We will use the abstraction game simulation approach that was proposed in [18] for the proofs.

As abstract states in the same model are no longer disjoint when considered as sets of concrete states, we will use the notion of *covering* when analyzing the resulting abstraction game and the construction algorithm: an abstract state S_1 covers another abstract state S_2, when the set of concrete states corresponding to

S_2 is a subset of the set corresponding to S_1. Covering between A-nodes can be defined using the abstract states of the A-nodes.

By describing abstract states via SMT formulae, cover checking can be formulated using the language of logic: a state A covers the state B if and only if $Expr(B) \implies Expr(A)$, where $Expr(\cdot)$ denotes the SMT formula describing the state.

In the case of (P)CFA analysis, the location is also retained by the abstraction. Covering in this case also requires that the two states have the same location. We will omit mentioning this from now on, but it is always an implicit requirement of covering.

An example game with covering can be seen in Figure 8. It can be seen on this example that whenever an A-node covers another one, the set of C-nodes that can be chosen from it is a superset of the C-nodes that can be chosen from the covered node. This property is ensured by the construction method. This leads to the more abstract A-node having more choices, and thereby its associated abstract state having less strict lower and upper value bounds, as expected.

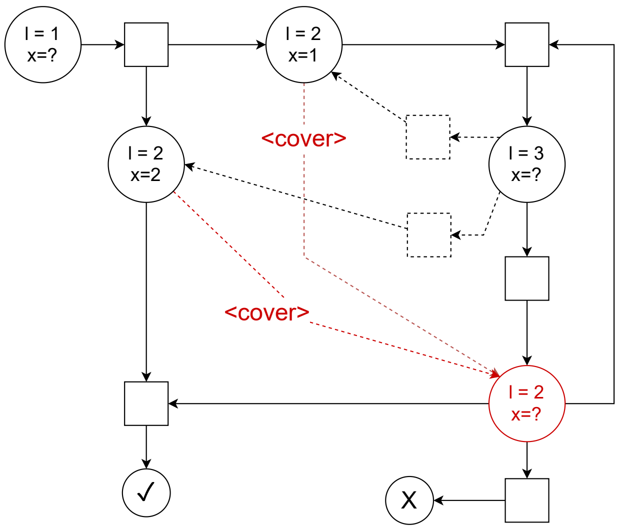


Figure 8: Example of an abstraction game on a domain without the disjointness property (generalized explicit domain).

This intuitively leads to conjecturing that the abstraction is sound, which we will formally prove in the next subsection.

Covering can also be used between states of different abstractions of the same MDP.

We will need to extend the notion of covering to C-nodes as well to simplify the wording of the proofs. In order to do this we first need to define the process of *lifting a relation to distributions*.

Definition 8 (Lifting a relation to distributions)**.** *For a relation $\mathcal{R} \in S_1 \times S_2$, the* lifting of the relation to distributions *is a relation $D(\mathcal{R}) \in \mathbb{D}(S_1) \times \mathbb{D}(S_2)$, defined by:*

$$(d_1, d_2) \in D(\mathcal{R}) \iff \exists \delta : S_1 \times S_2 \to [0,1], \text{ such that:}$$

$$\forall s_1 \in S_1 : d_1(s_1) = \sum_{s_2 \in S_2} \delta(s_1, s_2)$$
$$\forall s_2 \in S_2 : d_2(s_2) = \sum_{s_1 \in S_1} \delta(s_1, s_2)$$
$$(s_1, s_2) \notin \mathcal{R} \implies \delta(s_1, s_2) = 0$$

Unlike for A-nodes which are identified by an associated abstract state, the identity of C-nodes comes from their available choices, which are distributions over A-nodes. Therefore, covering is also determined by the available choices. Let us take the covering relation over A-nodes $R = \{(\hat{s}, s) \mid \hat{s} \text{ covers } s\}$. We say that a C-node $\hat{c}$ covers another C-node c if for all A-node distributions d that can be chosen in c, there is an A-node distribution $\hat{d}$ available in $\hat{c}$ such that $(\hat{d}, d) \in D(R)$.

The covering C-node thus does not have to have a superset of the exact choices of the covered C-node: A-nodes can be replaced by others that cover them. This means that for example, if the covering C-node has a single choice $(0.5 : s_1, 0.5 : s_2)$, and there is an A-node s_3 covering both s_1 and s_2, another C-node with the choice $(1 : s_3)$ covers it.

To prove the soundness of our inexact abstract transition functions, will make use of the notion of *strong probabilistic game simulation* proposed in [19]. Its aim was to define a relation that can be used to decide if a game is *more abstract* than another one.

The definition is asymmetric for the players, as it was defined specifically for game-based abstractions, where Player 1 is the abstraction player, with all of its actions leading to dirac distributions.

When defining the simulation relation for stochastic games, the simulating game is allowed to use *combined actions* to simulate actions of the other game, not only existing pure actions. Intuitively, this is similar to using probabilistic strategies when a player plays the game: instead of choosing a specific action, a distribution over the available actions can be chosen.

Although we adopt the full definition of strong probabilistic game simulation from [19] to be able to directly reference the proofs therein, we do not need the full power of this notion. Specifically, we will not need combined transitions to prove the soundness of Cartesian predicate abstraction and explicit abstraction with limited enumeration.

Combined actions will introduce *"virtual states"* for the players: the result of using a combined action will be treated similarly to existing nodes of the game, although it is not originally in the set.

Definition 9 (Virtual state)**.** *Given a game $G = (S = S_1 \uplus S_2, s_{init}, A, Av, \delta)$, a* virtual state *is a formal sum $\sum_{i=1}^{n} p_i s_i$, where $s_i \in S$, $\sum_{i=1}^{n} p_i = 1$, $0 < p \leqslant 1$ and*

$i \neq j \implies s_i \neq s_j$. All component states must belong to the same player, and the resulting virtual state also belongs to that player.

The set of available actions in a virtual state is given by: $Av(\sum_{i=1}^{n} p_i s_i) = \bigtimes_{i=1}^{n} Av(s_i)$.

The transition function of the game is extended to virtual states as:

$$\delta(\sum_{i=1}^{n} p_i s_i\,,\,(a_1,\ldots,a_n)) = \sum_{i=1}^{n} p_i \delta(s_i, a_i).$$

The resulting sum is not only a formal one: it is a sum over functions and addition is understood pointwise. The constraints on the weights p_i ensure that the result is a probability distribution over states.

Intuitively, a virtual state is a probabilistic superposition of states corresponding to the same player. The available (virtual) actions in the virtual state are combinations of real actions that can be considered one-step strategies: the player does not know which state they are actually in from the superposition, but it can decide which action it would choose in each of the component states.

Definition 10 (Combined actions)**.** *Given a stochastic game $G = (S = S_1 \uplus S_2, s_{init}, A, Av, \delta)$, the set of* combined transitions *available in a state $s \in S$ is $Cmb(s) = \{\sum_i \alpha_i a_i | a_i \in Av(s), 0 \leqslant \alpha_i \leqslant 1, \sum_i \alpha_i = 1\}$. We also extend the transition function δ to accept inputs with combined transitions. For any $s \in S, a = \sum_i \alpha_i a_i \in Cmb(s)$, $\delta(s, a) = \sum_j p_j s_j$.*

Combined actions are also available in virtual states as a similar combination over the virtual actions.

Definition 11 (Strong Probabilistic Game Simulation)**.**
Let $G = (S = S_1 \uplus S_2, s_{init}, A, Av, \delta)$ be a stochastic game where all actions of Player 1 lead to dirac distributions on S_2 and let $\mathcal{R} \subset \mathcal{S}_\infty \times \mathcal{S}_\infty$ be a relation on S_1. $\tilde{\delta} : S_1 \times A \rightharpoonup S_2$ will denote a deterministic version of the transition function constrained to Player 1, meaning that $\tilde{\delta}(s, a) = s' \iff \delta(s, a) = dirac(s')$.

$\mathcal{R}$ is a strong probabilistic game simulation *on G if for all $(\hat{s}, s) \in \mathcal{R}$ all of the following conditions hold:*

- $\forall a \in Av(s) : \exists \hat{a} \in Cmb(\hat{s}) : s_2 = \tilde{\delta}(s, a) \wedge \forall a_2 \in Av(s_2) : \hat{s}_2 = \tilde{\delta}(\hat{s}, \hat{a}) \wedge \exists \hat{a}_2 \in Cmb(\hat{s}_2) : (\delta(\hat{s}_2, \hat{a}_2), \delta(s_2, a_2)) \in D(R)$
- $\forall a \in Av(s) : \exists \hat{a} \in Cmb(\hat{s}) : \hat{s}_2 = \tilde{\delta}(\hat{s}, \hat{a}) \wedge \forall \hat{a}_2 \in Av(\hat{s}_2) : s_2 = \tilde{\delta}(s, a) \wedge \exists a_2 \in Cmb(s_2) : (\delta(\hat{s}_2, \hat{a}_2), \delta(s_2, a_2)) \in D(R)$

A relation between the Player 1 states of two different games is a strong probabilistic game simulation if it is a strong probabilistic game simulation on the disjoint union of the two games, and the pair of the two initial states is in the relation.

The two required properties of the relation intuitively mean that the abstraction has a choice both to limit the choice of the concrete non-determinism at least as much as in the simulated state (underapproximation of the choice set), and to make the choice at least as free as in the simulated state (overapproximation of the choice set).

Being able to convert and MDP to an abstraction game without ignoring any information will also help with the proof. The *game embedding* of an MDP captures this notion.

Definition 12 (Game embedding of an MDP)**.** *The* game embedding *of an MDP $M = (S, s_{init}, A, Av, \delta)$ is the abstraction game $\rho(M) = (\hat{S} = S_a \uplus S_c, \hat{s}_{init}, \hat{A}, \hat{Av}, \hat{\delta})$, where*

- *$S_a = S$ (there is exactly one A-node for each state of the MDP)*
- *There is a single C-node in S_c for each A-node $s \in S$, which will be denoted by $c(s)$*
- *$\forall \hat{s} \in S_a : \exists! a \in \hat{A} : \hat{a} \in \hat{Av}(\hat{s}) \wedge \hat{\delta}(\hat{s}, \hat{a}) = dirac(c(s))$*
- *$\forall s \in S, a \in Av(s), d \in \delta(s, a) : \exists! \hat{a} \in \hat{A} : \hat{a} \in \hat{Av}(c(s)) \wedge \hat{\delta}(c(s), \hat{a}) = \delta(s, a)$ (and $\hat{A}$ is the smallest set satisfying this constraint)*

Game embeddings make it possible to use strong stochastic game simulation to define when a game correctly abstracts an MDP. As Player A has at most one choice in each A-node of a game embedding, there is exactly one Player A strategy on this game. This means that the lower and upper bounds for probabilistic reachability properties coincide when using this game as an abstraction, giving exact results. Figure 9 shows a simple MDP and its game embedding as an example.

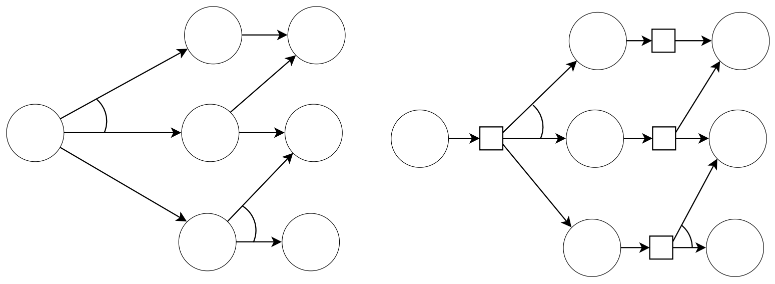


Figure 9: An MDP (left) and its game embedding (right)

Theorem 1. *Given two stochastic games $\hat{G}$ and G such that there is a strong probabilistic game simulation relation R between $\hat{G}$ and G and two sets of target states $\hat{T}$, T on them respectively such that $(\hat{s}, s) \in R \implies (\hat{s} \in \hat{T} \iff s \in T)$, we have that*

$$Prob^{-,-}(\hat{G}, \hat{T}) \leqslant Prob^{-,-}(G, T) \leqslant Prob^{+,-}(G, T) \leqslant Prob^{+,-}(\hat{G}, \hat{T})$$

$$Prob^{-,+}(\hat{G}, \hat{T}) \leqslant Prob^{-,+}(G, T) \leqslant Prob^{+,+}(G, T) \leqslant Prob^{+,+}(\hat{G}, \hat{T}),$$

where $Prob^{a,b}(G,T)$ denotes the probability of eventually reaching a state in the target set T when playing the game G with optimal strategies, (a, b) denoting the optimization objectives of the two players (+: maximize, -: minimize).

Proof. See Theorem 6.18. in [19]. □

This means that computing the reachability probabilities with optimal strategies in a more abstract game provide upper and lower bounds on the reachability probabilities of a less abstract game.

The original version of the theorem in [19] is a bit stronger as it allows the usage of abstract reward functions. For us this version will be enough, as we only care about the analysis of PCFA for now, where the target states of a reachability property are always based on the location, and the location is always exactly known in the abstraction.

A.2 Soundness of Cartesian abstraction

Theorem 1 (Soundness of Cartesian abstraction)**.** *Given a PCFA P with a set of target locations L_T, optimal strategies on the game abstraction of its MDP semantics using Cartesian abstraction to compute the abstract transition relation gives lower and upper bounds on the reachability probability of L_T. The optimal strategies are understood w.r.t. the reward function that assigns 1 to states with a location in L_T and 0 to all other states.*

Proof. It is shown in [19] that strong probabilistic game simulation is transitive, meaning that given a strong probabilistic game simulation $\hat{R}$ between the games $\hat{G}$ and G' and another one R' between G' and G, then $R = R' \circ \hat{R}$ is a strong probabilistic game simulation between $\hat{G}$ and G. It is also proved that there is a strong probabilistic game simulation between an abstraction game of the MDP semantics computed using exact predicate abstraction and the game embedding of the MDP semantics.

These two facts mean that it suffices to show only that there is a strong probabilistic game simulation between the abstraction game computed using Cartesian abstraction and the one using the exact abstract transition function.

Let $\hat{G} = (\hat{S} = \hat{S}_a \uplus \hat{S}_c, s_{init}, \hat{A}, \hat{Av}, \hat{\delta})$ be the abstraction game computed using Cartesian abstraction and let $G = (S = S_1 \uplus S_2, s_{init}, A, Av, \delta)$ be the one computed using the exact transition function. We will show that $R = \{(\hat{s}, s) \mid \hat{s} \in \hat{S}_1, s \in S_1, \hat{s} \text{ covers } s\}$ is a strong probabilistic game simulation relation between $\hat{G}$ and G.

Recall that to prove that R is a strong probabilistic game simulation, we need to show that two properties hold for all $(\hat{s}, s) \in R$:

- $\forall a \in Av(s) : \exists \hat{a} \in Cmb(\hat{s}) : s_2 = \tilde{\delta}(s, a) \wedge \forall a_2 \in Av(s_2) : \hat{s}_2 = \tilde{\delta}(\hat{s}, \hat{a}) \wedge \exists \hat{a}_2 \in Cmb(\hat{s}_2) : (\delta(\hat{s}_2, \hat{a}_2), \delta(s_2, a_2)) \in D(R)$
- $\forall a \in Av(s) : \exists \hat{a} \in Cmb(\hat{s}) : \hat{s}_2 = \tilde{\delta}(\hat{s}, \hat{a}) \wedge \forall \hat{a}_2 \in Av(\hat{s}_2) : s_2 = \tilde{\delta}(s, a) \wedge \exists a_2 \in Cmb(s_2) : (\delta(\hat{s}_2, \hat{a}_2), \delta(s_2, a_2)) \in D(R)$

The difference between Cartesian abstraction and the precise abstract transition function is that a Cartesian abstraction might set a subset of the predicates to unknown. Let s be an abstract state in the stricter predicate domain, where all predicates in the precision are assigned to be either true or false, and let $\mathit{next}(s)$ denote the exact next states available from s. Let $\hat{s}$ denote an abstract state in the generalized predicate abstraction domain with the same precision such that $\hat{s}$ covers s, and let $\mathit{next}(\hat{s})$ denote the set of next states available from $\hat{s}$ using Cartesian abstraction.

From the covering relation we know that predicate assignments in s and $\hat{s}$ never contradict each other: if a predicate is not unknown in $\hat{s}$, it has the same truth value in $\hat{s}$ as in s. This also leads to the fact that $\forall s' \in \mathit{next}(s) : \exists \hat{s}' \in \mathit{next}(\hat{s}) : \hat{s}'$ covers s'.

To see why this is true, let us examine the interaction between next state computation and the covering relation. In the exact case, we compute all satisfying models for the SMT formula $\mathit{expr}(s) \wedge \mathit{expr}(\mathit{stmt}) \wedge \mathit{activation}(s')$, where $\mathit{activation}(\cdot)$ stands for the activation literal representation of the next state. From the logic-based formulation of covering, we know that $\mathit{expr}(s) \implies \mathit{expr}(\hat{s})$, from which

$$(\mathit{expr}(s) \wedge \mathit{expr}(\mathit{stmt}) \wedge \mathit{activation}(s')) \implies (\mathit{expr}(\hat{s}) \wedge \mathit{expr}(\mathit{stmt}) \wedge \mathit{activation}(s'))$$

follows. This means that any model that satisfied the SMT formulation of the step from the covered state also satisfies it from the covering state, so if we used exact computations, at least the same next states are available from the covering state.

Now we have to show that the approximate computation of Cartesian abstraction retains this property, modulo covering, as it suffices if a covering state is available instead of exactly the states available from the covered state. If there exists a satisfying model for $\mathit{expr}(\hat{s}) \wedge \mathit{expr}(\mathit{stmt}) \wedge \mathit{activation}(s')$, where the activation literal of a predicate is set to true, than $\mathit{expr}(\hat{s}) \wedge \mathit{expr}(\mathit{stmt})$ cannot imply the negation of the predicate, as that would lead to a contradiction. A similar statement is true for a false activation literal and the ponated version of the predicate. Because of this, for each resulting state of the exact computation, there is always a state in the result of the Cartesian computation that is consistent with it: each predicate is either the same as in the exact one, or unknown. This state covers the exact state.

Now we have to prove that an extension of this holds for the case when we use Cartesian abstraction with grouped transition functions (when the abstract state is in the generalized domain, but the transition is computed exactly, as explained above), as we want to show that the resulting C-nodes also cover the original ones.

The same reasoning can be used here as above for the flat list of states by simply replacing the single $\mathit{expr}(\mathit{stmt}) \wedge \mathit{activation}(s')$ by $\bigwedge_i (\mathit{expr}(\mathit{stmt}_i) \wedge \mathit{activation}(s'_i))$ with i ranging over the indices of the substatements. By doing this we can see that the same groups (leading to the same distributions) are obtained from the covering state, as well as some additional choices. As these groups are all Player A choices, we only give the abstraction player more choices, but the original choices are still available. Here we do not even have to take the “modulo covering” into account, as the exact computations can result only in states that do not contain unknown predicates.

This means that for all A-nodes s of G, if an A-node $\hat{s}$ of $\hat{G}$ covers it, than for every C-node directly available from s, there is a C-node directly available from $\hat{s}$ covering it. This leads to the satisfaction of both properties required for the simulation: under and overapproximation of the choice sets modulo covering are both satisfied by actually having the same choice sets with some states in the distributions replaced by another state covering the original one. □

A.3 Soundness of limited enumeration

Theorem 2 (Soundness of limited enumeration)**.** *Given a PCFA P with a set of target locations L_T, optimal strategies on the game abstraction of its MDP semantics using explicit abstraction with limited enumeration to compute the abstract transition relation gives lower and upper bounds on the reachability probability of L_T. The optimal strategies are understood w.r.t. the reward function that assigns 1 to states with a location in L_T and 0 to all other states.*

Proof. Although explicit abstraction was not mentioned in [19], it can be regarded as a special case of predicate abstraction where all predicates are equalities, and we have one equality for each possible assignment of a tracked variable. This means that the proof of predicate abstraction being sound also extends to exact explicit abstraction.

Now, similarly to the Cartesian abstraction case, we only have to show that there is a strong probabilistic game simulation between the game constructed using limited enumeration and the one constructed using precise explicit abstraction (without limiting enumeration).

The same reasoning is used here as above: we can show that the covering relation $R = \{(\hat{s}, s) \mid \hat{s} \in \hat{S}_1, s \in S_1, \hat{s} \text{ covers } s\}$ is a strong probabilistic game simulation relation between the game $\hat{G}$ computed with limited enumeration and G compute using the exact abstract transition relation.

For the generalized explicit domain, a state $\hat{s}$ covering another state s means that for all tracked variables if its value is known in $\hat{s}$, it must be the same as it is in s. Now we proceed similarly to the Cartesian abstraction case.

We can see that

$$(expr(s) \wedge expr(stmt) \wedge expr(s')) \implies (expr(\hat{s}) \wedge expr(stmt) \wedge expr(s')).$$

For explicit abstraction, we can use the expression form of the next state instead of a formulation based in activation literals. This shows that if we used the exact next state computation from a covering state, the set of available next states would contain the set of states available from the covered state.

Now we have to show that using limited enumeration instead of this does not cause any problems. With limited enumeration, whenever we encounter a situation where the number of possible next values for a variable grows over a limit, instead of enumerating all possible next states, we merge these into a single one with the variable set to unknown. This merged state will then cover all of the originally available ones where the value of the variable is exactly known.

This means that for each state available from s computed using the exact next state relation, there is always a state available from s' computed using limited enumeration that covers it.

From this, the theorem can be proven using the same reasoning as in the previous proof.

$\square$

Acta Cybernetica **26** (2024) 713–747.

Uncovering Hidden Dependencies: Constructing Intelligible Path Witnesses using Dataflow Analyses*

Kristóf Umann[ab], Gábor Horváth[ac], and Zoltán Porkoláb[ad]

Abstract

The lack of sound, concise and comprehensive error reports emitted by a static analysis tool can cause increased fixing cost, bottleneck at the availability of experts and even may undermine the trust in static analysis as a method. This paper presents novel techniques to improve the quality of bug reports for static analysis tools that employ symbolic execution. With the combination of data and control dependency analysis, we can identify the relevance of particular code snippets that were previously missing from the report. We demonstrated the benefits of our approach by implementing an improved bug report generator algorithm for the Clang Static Analyzer. After being tested by the open source community our solution became enabled by default in the tool.

Keywords: static analysis, symbolic execution, control dependency analysis, reaching definitions analysis, Clang Static Analyzer, report generation, code comprehension

1 Introduction

Maintenance costs take a larger part of the price of the software systems. Most of these expenses are spent fixing bugs. The earlier a bug is detected, the lower the cost of the fix [12]; therefore, various efforts are applied to speed up the *development–bug detection–bug fixing* cycle. The classical test-based approach – although still important – is insufficient on its own. Writing meaningful tests requires high code coverage and takes substantial development workload and time. Another approach,

*This work is supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.3-VEKOP-16-2017-00002).

[a]Department of Programming Languages and Compilers, Faculty of Informatics, Eötvös Loránd University, Budapest, Hungary

[b]E-mail: szelethus@caesar.elte.hu, ORCID: 0000-0002-6679-5614

[c]E-mail: xazax@caesar.elte.hu, ORCID: 0000-0002-0834-0996

[d]E-mail: gsd@caesar.elte.hu, ORCID: 0000-0001-6819-0224

DOI: 10.14232/actacyb.299805

the *dynamic analysis method* using tools like *Valgrind* [38], or *Google Address sanitizer* [47] which work runtime, evaluates the correctness only those parts of the system which have been executed. Although such *dynamic analysis methods* are precise and could catch real errors with a very low rate of false reports, they require carefully selected input data, and can also easily miss certain corner cases. Dynamic analysis *trades coverage for precision*, and reaching even a close to full coverage is usually infeasible.

Contrary to testing and dynamic analysis methods, *static analysis* techniques do not require the concrete execution of the program, and are often based only on the program's source code, and do not require any input data. It is a popular method for finding bugs and code smells [10, 50, 42, 17]. They do not depend on the selection of input data while they can (at least theoretically) provide full coverage of the code. Compiler warnings are almost exclusively based on various static analysis methods. Many of the applied techniques are fast enough to be integrated into the Continuous Integration (CI) loop, therefore, they have a positive impact on speeding up the development–bug detection–bug fixing cycle. Another advantage of the static analysis method is that it is in many cases applicable for parts of the code. This is useful when we have no full control over the system, e.g. we use third party libraries, not all source is available, or we just have no resources to check the whole system.

Most static analysis methods apply heuristics, which means that often they may *underestimate* or *overestimate* the program behavior [3]; in other words, static analysis *trades precision for coverage*. In practice, this means static analysis tools sometimes do not report existing issues which is called a *false negative*, and sometimes they report correct code erroneously as a problem, which is called as *false positive*. There is a continuous struggle to improve tools and methods, but there is a theoretical limitation: paraphrasing Rice's theorem [44] from '53: all non-trivial properties of a program are undecidable at compile time. Therefore, at the end all reports need to be reviewed by a professional who has to decide manually whether the report stands, and if so, fix it. This, however, creates a serious bottleneck in the otherwise automated process as humans who are experts both in the problem domain and in the implementation techniques are usually the most expensive and the least available resources. It has the uttermost importance to maximize the effectiveness of the step where humans involved [28]. Considering the mentioned theoretical limitations, the best possible way to do it is to improve the communication between the automated analysis tool and the human actor: i.e., to teach the analysis tool to provide sound, concise and comprehensive reports.

While more complex static analyses can detect even deep-rooted programming errors, the construction of intelligible bug reports also gets much more difficult. In this paper, we present the report generation challenges faced by a technique called symbolic execution. Symbolic execution explores a high number of execution paths within the program and can constrain the values of runtime variables, allowing it to gain a considerable understanding of the program's runtime behavior. However, after finding a bug, it usually struggles to relate back to the source code and many time is unable to consider the proper context broader than the actual path of execution leading to the bug.

We discuss possible new techniques to allow a symbolic execution tool to better understand of code contexts outside a given path of execution. We also demonstrate one of these techniques implemented as an extension to the open-source analyzer tool Clang Static Analyzer. As one of the more mature and popular static analyzer tools that implement symbolic execution for C, C++ and Objective C languages, it is considered stable and reliable to be used on large code-bases for both academic and industrial purposes. Our report generation improvement was tested by the open source community and accepted to merge into the tool since version 10.0.0. As this improvement has been enabled in the releases since[1], we feel there is a real world benefits to our results. We documented our research, implementation, and some of the evalution processes leading up to this paper in [54].

This paper is structured as follows. In Section 2 we overview the technical background related to symbolic execution and its implementation in the Clang Static Analyzer tool. In Section 3, we discuss our expectations for an intelligible bug report, and present techniques to generate them and their shortcomings. Section 4 details our proposals and implementations. We evaluate our solution implemented for the Clang Static Analyzer in Section 5. Related work is surveyed in Section 7. Future areas of research and implementation are discussed in Section 8. Finally, we conclude our paper in Section 9.

2 Technical background

An often celebrated advantage of static analysis is its greater code coverage compared to most dynamic analyses. However, this does not come without a cost; arguing about runtime values is often difficult or impossible with only static information. More complex analyses also tend to be expensive in terms of computing resources, and are often several times slower than compilation and consume more memory.

Various techniques approach these challenges from different angles – abstract syntax tree analysis (*AST analysis*) [16] and control flow analysis trade understanding of runtime behavior for faster analysis speed. Dataflow analyses [43] are able to argue about the flow of information within the bounds of a given function, and most variants strike a middle ground in terms of space and time complexity and the effectiveness of the analysis. Symbolic execution [30] takes a rather radical approach, by essentially interpreting the source code, and analyze a large number of execution paths in the program. This leads to a combinatorial explosion according to the number of possible program states, which makes the analysis rather expensive, but provides more information about runtime behavior.

This section discusses symbolic execution and its implementation in the Clang Static Analyzer [15].

[1]As of the writing of this paper, the latest Clang release is 12.0.0.

2.1 Symbolic execution

Concrete execution, what we would consider the "normal" execution of the program or the simulation of such, is done by a specific input set to exercise a single path of execution. In contrast, most forms of symbolic execution need no input values and explore multiple paths of execution, covering entire classes of inputs [33]. These input values, and runtime variables of the program are assigned *symbolic* rather than concrete values. An analysis engine models the program behavior with a store, which is a mapping of variables to symbolic values, and a constraint solver, which contains constraints on symbolic values [7, 59]. The store is updated when a memory location is written, and the constraint solver is updated after each evaluation of a conditioned branch.

The Clang Static Analyzer [15, 14, 20, 60] is an open-source tool that implements symbolic execution on the C family of languages. Over the course of a decade, it grew to be regarded as a stable and reliable tool for academic and industrial purposes. It enjoys a variety of advantages of being built directly into the Clang compiler, such as a thoroughly tested and up-to-date abstract syntax tree (*AST*) and control flow graph (*CFG*, pictured in Figure 1a). Clang [34] is a compiler frontend for LLVM, the umbrella project that offers a wide selection of algorithms that are useful for optimization. For the remainder of the paper, we refer to the Clang Static Analyzer under the term *analyzer*.

Symbolic execution in Clang starts after the conclusion of syntactic and semantic analysis, and the construction of the AST and the CFG. The analyzer then creates a call graph for the input file's translation unit. The call graph's nodes are functions, and directed edges describe function invocations from one function to another. If possible, symbolic execution starts from functions with no ingoing edges in the call graph, otherwise in some other function. From this initial function, called the *top-level function*, the analyzer will explore paths of execution using the CFG. For the code snippet in Figure 1a, `f` will be the top-level function, and a possible path of execution would be $B5$ $B4$ $B3$ $B2$ $B0$. The CFG describes one specific function at a time yet both $B4$ and $B2$ contain function calls. This obstacle is resolved by the analyzer "jumping" from the invocation site to the entry blocks of the callee function's CFG. We call this process *inlining*. Should we denote the symbol Bi_{foo} as the ith block of `foo`'s CFG, the path of execution mentioned above is as follows:

$$B5_f\ B4_f\ B3_g\ B2_g\ B1_g\ B4_f\ B3_f\ B2_f\ B3_g\ B2_g\ B1_g\ B2_f\ B0_f$$

2.2 The ExplodedGraph

During analysis, the analyzer builds a data structure to keep track of the program state (most notably the store and the constraint manager) at each point of symbolic execution. This data structure is called the `ExplodedGraph`, which is pictured in Figure 2. The ExplodedGraph is a different data structure to the CFG because it contains far more information (assumptions on values, memory regions) [52]. It is not even isomorphic with it; a path of execution on which the body of a loop is visited four times, each visit would be represented with a linear path, instead of a directed loop. Also, while a CFG is built for each function, only a single ExplodedGraph is built for the entire analysis. With that said, it is possible to map each ExplodedNode (a node of the ExplodedGraph) to a specific CFGBlock (or simply block, a node of the CFG), or CFGEdge (a directed edge of the CFG).

Mind that the analyzer does not view the path of execution from a "human" perspective. It processes these nodes from the ExplodedGraph unaware of contextual information in the source code, or even nodes on other paths of execution. For instance, the path in red in Figure 2 does not include any information about user's intent to assign `x` a non-null object, and will not be considered.

3 Report generation

A frequently researched problem of static analysis is to discover as many real bugs as possible while keeping their false positive rate within a margin of error [41, 28]. However, making the generated reports intelligible and easily digestible is rarely discussed. However, many researches [28, 29, 36, 45] point to the fact, that understanding the error report and converting it to an executable action by the developer is crucial for the acceptance and the effective use of static analysis tools.

In this section, we define a non-comprehensive set of guidelines on an ideal bug report, and overview how the analyzers approach this issue, and struggle relating to the limitations of the ExplodedGraph.

Bug report generation is done after the entire analysis is concluded by the inspecting nodes of the ExplodedGraph. To avoid confusion, we define the following terms:

- An *(explored) path of execution* is a directed path in the ExplodedGraph starting from the root terminating in one of its leaves.
- A *bug path* is the shortest path of execution, which terminates in an error node. Error nodes in the ExplodedGraph are the program points where a bug was discovered (the red path seen in Figure 2).
- A *bug report* is a user-readable set of messages and notes that explains the control flow leading to the bug, and the values of related variables (see Figure 1b-1c).

3.1 Goals

The bug path is a collection of all events on a given path of execution and is not a user-readable set of events. Some nodes describe relatively low-level actions, like an lvalue-to-rvalue cast, the cleanup of local variables, or other events that may not be relevant to the actual bug. Hence, we define the ideal bug report to be:

- *Minimal*, so that it is void of events that are irrelevant to the comprehension of the bug report,
- *Complete*, so that it highlights every relevant event.

In a sense, an ideal bug report tells how to reproduce the bug. Unfortunately, these goals are rather vague and leave room for subjectivity. For instance, control flow through a constexpr-conditioned branch is obvious from the compiler's perspective but *may or may not be* obvious to a reader. To keep our study free of personal preference, we precisely define the vaguest word of these goals: *relevance.*

Definition. *We say that statement a is relevant to statement b, if for a given (b, {set of variables}) pair, a is a control dependency of b, or contains an expression that is a data dependency of any of the variables in the pair's set. We call a (statement, {set of variables}) pair a slicing criterion.*

3.2 Report generation techniques prior to our research

To construct a bug report, the analyzer, starting from the error node, inspects the bug path's nodes individually all the way to the root of the ExplodedGraph, looking for noteworthy events to the slicing criterion (error location, {bug causing variables}). Two techniques are employed for each of the goals mentioned above. *Bug path visitors* add user-readable messages and notes to the final bug report, and *interestingness propagation* helps to discard of a portion of these.

3.2.1 Bug path visitors

Contrary to their name, bug path visitors function as callbacks. As the analyzer visits a new node in the bug path, it notifies each visitor to inspect it. If a visitor finds a noteworthy event, they may construct a diagnostic message. For instance, `ConditionBRVisitor` is responsible for constructing a message for each evaluated condition. When a condition is seen by this visitor in an ExplodedNode, for instance, `if (coinflip())`, the message "Assuming the condition is true" may be constructed.

Visitors greatly expand the number of variables and values to consider for explaining. Data dependence is a great example; if variable `x` caused a bug, we could register `FindLastStoreBRVisitor` to find x's last write preceding the error node (e.g., "Value assigned to 'x' "). Visitors can themselves create new visitors, if warranted. Suppose that that last write is in the form of an assignment (`x = y;`), `FindLastStoreBRVisitor` would register a new instance of itself to explain y.

3.2.2 Interestingness propagation

By design, visitors cannot always be aware of whether the constructed message is relevant to the bug report. In anticipation, the analyzer marks some entities (such as the denominator for a division-by-zero bugs) interesting. Just as visitors may themselves create new visitors during bug report construction, they may also mark new entities interesting, or propage interestingness from one entitity to another. In a later stage, after all diagnostics have been constructed, messages in function calls not describing any interesting entity are pruned.

At last, these two techniques are combined in what we call *expression tracking*. A common desire for bug report generation is to explain all events relating to a variable: why it holds a specific value, control flow around the usages of said variable, and other properties. This is achieved by registering a set of visitors relating to that variable, and mark it interesting. We call this process the *tracking* of said variable.

3.3 Deficiencies

Even when describing multiple iterations of a loop, bug paths contain no directed cycles. They are a sequence of program states, leading to a node where the program state is erroneous. This linearity and the lack of information on code not explored by the analyzer on that bug path can make it challenging to understand the intent of the programmer.

Figure 1a shows a code snippet where the global variable `flag` controls whether `x` will be initialized, and whether `x` will be dereferenced. Function `g` sets `flag` to some unknown value, and the lack of parameter passing makes this a non-trivial realization from a user's perspective. Figure 1b shows a report from the analyzer displayed by CodeChecker [21] before our research: the analyzer failed to understand that `x`'s value, and its dereference depends on `flag` and is worth explaining. As a result, it pruned diagnostic messages relating to function calls to `g`. In a later section, we will discuss our results to improve this bug report as shown in Figure 1c.

For each example in Figure 3, the analyzer can discover a null dereference bug on line 21, but will also fail to find all relevant statements to it during bug report construction. These examples correspond to four classes of problems[2], which we discuss in further detail as follows:

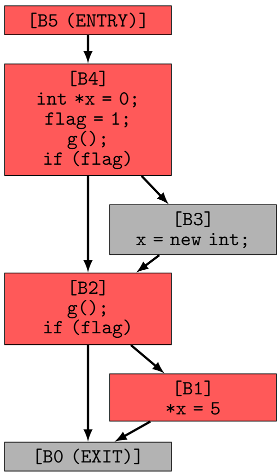
3.3.1 Figure 3a: Control dependency is not recognized

Analysis starts at line 14, noting variable `x` to be a null pointer, and the global variable `flag` to be 1. Then, the function call to `g()` will appropriately set `flag`'s value to unknown. On line 20, the analyzer will explore a path of execution on which `flag`'s new value is 0, and one where it is not. On the former, a dereference of `x` is found on line 21, which is known to be null. The analyzer will cut a bug path out of the ExplodedGraph which terminates in this erronous program state, and configure its bug report generation facilities to start tracking `x`.

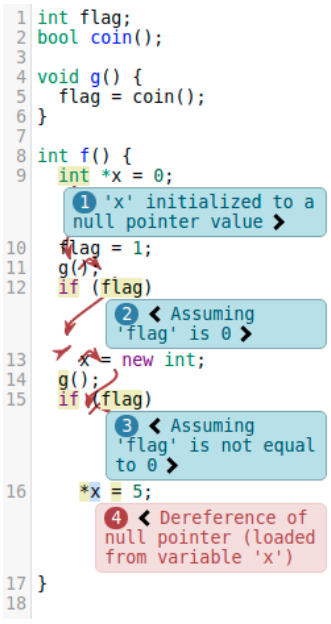
During bug report generation, the analyzer can find the relevant statement to `x` regarding data dependencies, which is its initialization on line 14. It will, however, fail to recognize a relevant statement to the bug – namely, had `flag` not been 0 on line 20, the bug would not have occurred. We will define it more precisely in later

[2]Figure 1a combines the classes of problems displayed in Figure 3a and Figure 3b

```
int flag;
bool coin();

void g() {
  flag = coin();
}

int f() {
  int *x = 0;
  flag = 1;
  g();
  if (flag)
    x = new int;
  g();
  if (flag)
    *x = 5;
}
```

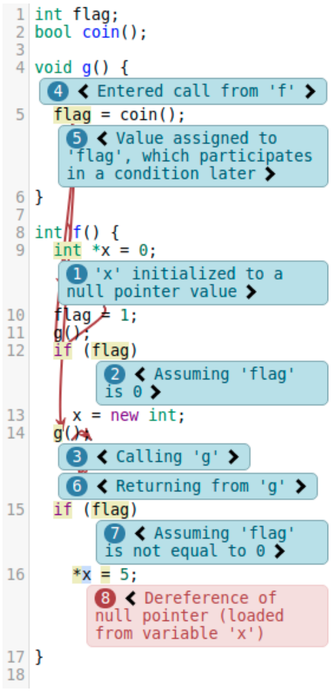


(a) A code snippet and function f's control flow graph.



```
int flag;
bool coin();

void g() {
  flag = coin();
}

int f() {
  int *x = 0;
    1 'x' initialized to a null pointer value
  flag = 1;
  g();
  if (flag)
    2 Assuming 'flag' is 0
    x = new int;
  g();
  if (flag)
    3 Assuming 'flag' is not equal to 0
    *x = 5;
    4 Dereference of null pointer (loaded from variable 'x')
}
```

(b) Before



```
int flag;
bool coin();

void g() {
  4 Entered call from 'f'
  flag = coin();
  5 Value assigned to 'flag', which participates in a condition later
}

int f() {
  int *x = 0;
  1 'x' initialized to a null pointer value
  flag = 1;
  g();
  if (flag)
    2 Assuming 'flag' is 0
    x = new int;
  g();
  3 Calling 'g'
  6 Returning from 'g'
  if (flag)
    7 Assuming 'flag' is not equal to 0
    *x = 5;
    8 Dereference of null pointer (loaded from variable 'x')
}
```

(c) After

Figure 1: A code snippet and analysis results demonstrating how the analyzer struggles to realize that x's value and derefence depends on flag, and as a result, will not construct diagnostic messages to explain relevant events to it.

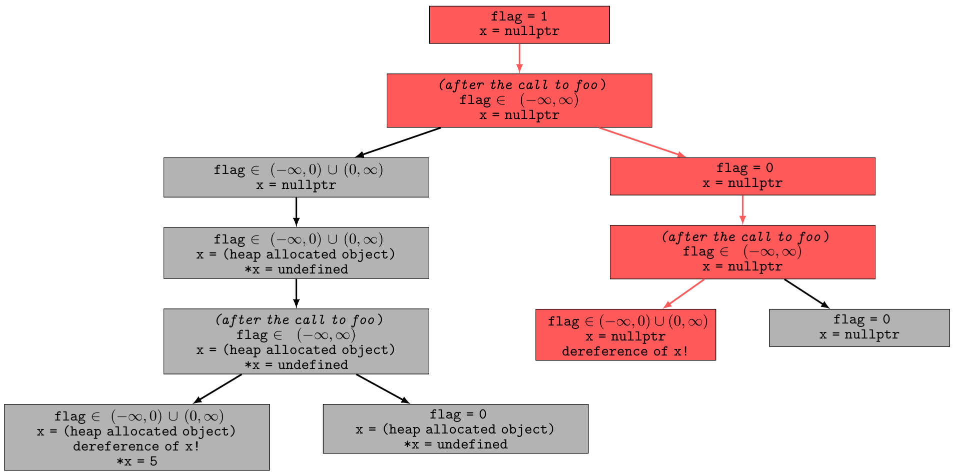


Figure 2: A simplified `ExplodedGraph` after analyzing Figure 1a.

section, but this property makes line 20 a *control dependency* of line 21. Control dependency is defined on the CFG, not the bug path, hence the analyzer being oblivious to it at this phase. Note that `flag`'s value was set to 1 a few lines earlier, and should `g()`'s definition be unavailable or obscured, it would not obvious why `flag`'s value is assumed to be 0 on line 20.

3.3.2 Figure 3b: Reaching definition is not in the bug path

The analysis, and the eventual costruction of the bug path is done similarly to Figure 3a. The analyzer can again find `x`'s initialization as important, but as line 18 is not on the bug path, the analyzer fails to recognize that the user likely intended to set `x`'s value properly. This assignment and `x`'s initialization are so-called *reaching definitions* to `x` on line 21 – loosely, there exists a path in the CFG from them to line 21 without any interleaving assignments to `x`. `flag`'s value on line 17 is no longer a control dependency to the CFG block in which the bug is found, yet it is clear that should `flag`'s value be non-zero on line 17, the bug would not have occurred. Reaching definitions is also a property of the CFG, so line 18 is not recognized as important, leading the analyzer to believe that its control dependency, line 17, which *is* on the bug path is not worth explaining in further detail either.

3.3.3 Figure 3c: Reaching definition is in a different, but inlined stack frame

This example presents another layer of difficulty to Figure 3b – the statement on which `x` could have obtained a non-null value is not only outside the bug path, but is in another function call. In the previous cases, control dependency and reaching

```
int flag;
bool coin();

void g() {
  flag = coin();
}






int f() {
  int *x = 0;
  flag = 1;
  g();



  if (!flag)
    *x = 5;
}
```

(a)

```
int flag;
bool coin();

void g() {
  flag = coin();
}






int f() {
  int *x = 0;
  flag = 1;
  g();
  if (flag)
    x = new int;


  *x = 5;
}
```

(b)

```
int flag;
bool coin();

void g() {
  flag = coin();
}

void h(int **x) {
  if (flag)
    *x = new int;
}

int f() {
  int *x = 0;
  flag = 1;
  g();

  h(&x);
  g();
  if (flag)
    *x = 5;
}
```

(c)

```
int flag;
bool coin();

void g() {
  flag = coin();
}

void h(int **x) {

  *x = new int;
}

int f() {
  int *x = 0;
  flag = 1;
  g();
  if (flag)
    h(&x);
  g();
  if (flag)
    *x = 5;
}
```

(d)

Figure 3: Code snippets that highlight deficiencies in the analyzer's understanding of bugs when generating reports.

definitions could have been recognized within `f` itself; here, an interprocedural technique is required, whereas a CFG is constructed for only a single function at a time. While we can say that the assignment to `x` in `h` is a reaching definition to `x` on line 21, this information needs to be carried from one CFG to another. Although symbolic execution is interprocedural, control dependency analysis and reaching definitions analysis (and many similar lightweight techniques) are not.

3.3.4 Figure 3d: Reaching definition is in a different and not inlined stack frame

A relevant statement, line 10, is not only outside the bug path, but the containing function `g` was not inlined either (the analyzer has not entered this function on the path where the bug is discovered). Inlining functions can demand non-trivial modeling, such as lifetime extension, moves, and the evaluation of arguments. This makes bridging the gap in between CFGs all the more difficult. Generally speaking, the “further” the analyzer has to stray from the bug path, the more challenging bug report construction becomes.

4 Proposed solution

Program slicing is a field of study about creating a *program slice*, which is a subset of the program's statements, relevant to a point of interest, usually defined by a (statement, {set of variables}) pair, called a *slicing criterion*. Relevance in this context is defined by whether a statement could influence the value of one of the variables in the slicing criterion. Program slicing combines data and control dependency analysis in a fix-point algorithm to slice irrelevant statements away from the program, converging to a minimal, but complete slice.

The original program slicing algorithm [57] was intraprocedural, and aimed at monolithic, single-procedure programs [40]. Interprocedural variants are explored in numerous studies [26, 11, 56], but they demand the existence of a data structure that describes data and control dependencies across function calls, most commonly a system dependence graph, which at the time of writing was absent from Clang, and its implementation would be a challenging task with the current Clang's AST and CFG design.

As feasible implementations of slicing algorithms are confined to the bounds of a single function, and most bug paths span multiple functions in the source code, adjustments would be required to make program slicing a valuable part of Clang's bug report generation facilities. A great candidate to bridge this gap might be bug reporter visitors, as they can reason about data dependencies with rather great precision across function calls. However, they would be partially redundant with the data dependency analysis built into program slicing. For these reasons, we approached slicing in terms of its core components, not in its entirety.

In Section 3.3, we have shown four classes of problems the analyzer could not tackle prior to our research. We propose two techniques as a potential solution, as well as how they can be incorporated into the existing bug report generation infrastructure: control dependency analysis and reaching definitions analysis. While we are cautious about unforeseen challenges, we feel confident that these would solve three of the four cases, and pave the way to approach the fourth. As a demonstration, we implemented control dependency analysis and observed measurable improvements for the first case.

4.1 Control Dependence Analysis

On most occasions, control dependence center around conditional statements (e.g., `if`, `for`, `switch`), where the value of the condition dictates which part of the code (e.g., branches of `if` statements) will be executed next. For instance, cases of a switch-case statement are control dependent on the expression in the switch statement.

We defined relevance in part such that slice $a := (stmt_a, vars_a)$ is relevant to slice $b := (stmt_b, vars_b)$, if $stmt_a$ is a control dependency of $stmt_b$. The following exercise demonstrates why this is reasonable: Suppose that we constructed b after discovering a null pointer dereference error (for instance, $(line16, \{x\})$ for Figure 1a), and have already marked $stmt_b$ as interesting. We argue that regarding its control dependency, $stmt_a$, as relevant enables the analyzer to better understand the context of this bug; had control not flown from $stmt_a$ to $stmt_b$, the bug would not have occurred.

In this section, we overview our proposal and implementation of adding control dependency analysis to bug report construction.

4.1.1 Defining Control Dependence

We define control dependence on the CFG with the help of *dominance* and *postdominance*. We say that block A *dominates* block B ($A \operatorname{dom} B$), if every path from the entry block to B must go through A. We say A *strictly dominates* B ($A \operatorname{sdom} B$), if A dominates B and $A \neq B$. We say B *postdominates* A ($B \operatorname{pdom} A$), if every path from A to the exit block must go through B, and similarly, B *strictly postdominates* A ($B \operatorname{spdom} A$) if B postdominates A and $B \neq A$ [1]. An example can be seen regarding dominance in Figure 4.

We say that block B is *control dependent* on block A ($B \operatorname{cd} A$) if there exists an edge from A to C such that B postdominates C, and if B is not equal to C, B doesn't postdominate A. In looser terms, this expresses that B is control dependent on A if B doesn't postdominate A, but post dominates all blocks “in between them”. As an example, in Figure 1a, $B1$ is control dependent on $B2$, but $B0$ is not control dependent on $B2$, as $B0$ post dominates $B2$. We extend control dependency to statements as follows: If block B is control dependent on block A, we also say that all statements in B are *control dependent* on the condition expression in A, if such exists.

Control dependencies can be calculated with *post dominance frontier* sets [19]. The post dominance frontier set of block A ($PDF(A)$) is the set of B nodes from

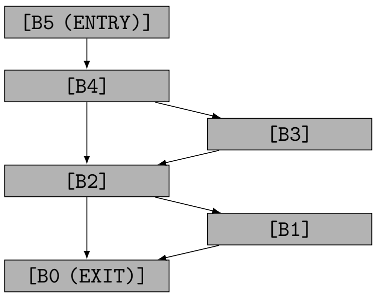


Figure 4: A simple control flow graph. The entry block (strictly) dominates every (other) block, and the exit block (strictly) postdominates every (other) block. B4 dominates B3, but B3 does not dominate B2, since the path B5 $\rightarrow$ B4 $\rightarrow$ B2 excludes B3. B2 postdominates B4, but B3 does not postdominate B4.

the inverse CFG[3] such that A dominates a predecessor of B but does not strictly dominate B:

$$\begin{aligned} PDF(A) := \{B | (\exists P \in \mathit{Pred}(B)) \wedge \\ \wedge\, (A \,\mathrm{pdom}\, P \wedge \neg A \,\mathrm{spdom}\, B)\} \end{aligned}$$

Calculating PDF sets quickly yields control dependence:

$$A \,\mathrm{cd}\, B \Leftrightarrow B \in PDF(A)$$

We implemented dominance frontier sets with the algorithm described in [48].

For a CFG with E edges and N CFGBlocks, calculating PDF sets has a worst-case complexity of $\mathcal{O}(E + N^2)$, but is often linear in practice [19].

4.1.2 Integration of control dependence

As bug path visitors continously expand the code contexts (values, variables) to explain, we chose to weave our control dependency calculator into a new visitor. A new instance of our visitor is registered *for each* new tracked expression value. As a new node in the bug path is visited, the visitor checks whether the statement described in the node is a control dependency of the location where the tracking started. If so, it will instruct the analyzer to track the condition of that statement. Essentially, each visitor instance holds a (statement where tracking starts, {tracked variable}) slicing criterion.

Figure 3a demonstrates a code snippet where the analyzer can detect a null pointer dereference of `x`, but fails to realize that had the value of flag may have been a guard of this error. With our improvement, the bug report generation would work as follows: The analyzer would start tracking `x`, registering several

[3]An inverse of a CFG is constructed by reversing all of its edges in the graph.

visitors, including our own. As it ascends the bug path and finds the ExplodedNode describing the evaluation of `!flag` on line 20, our visitor checks whether line 20 is a control dependency of where it started tracking from (line 21). As a result, it will instruct the analyzer to track `flag`. `FindLastStoreBRVisitor` would find `flag`'s last store on line 5, and a diagnostic message will be constructed to describe it. Since `flag` is tracked, it is also an interesting variable and the analyzer will prevent the pruning of messages inside `g()`.

While this information was indeed found on the bug path, inspection of its nodes alone did not reveal the relevance in between `flag` and `x`, and led to the analyzer discarding information about `g()` to keep the bug report minimal. Control dependence analysis unearthed and preserved the importance of this function call.

In application, we employ a number of heuristics to limit the impact of our solution to only display diagnostics when they provide a meaningful addition to the bug report. When displaying a bug report to the user, the source code is decorated with diagnostic messages and notes, and often relevant information about condition values is readily available in the same function as the condition itself. We found that additional notes in the same function did not add much value the user experience, and on occasion needlessly polluted the report. We chose to display new diagnostic messages only when information relating to a condition was found in a function call that would otherwise be disregarded.

4.2 Data Dependence Analysis

Data dependency analysis on the bug path (done by `FindLastStoreBRVisitor`) benefits from all the information the analyzer gathered during symbolic execution. Common obstacles in this realm might already be resolved: the analyzer's memory model keeps track of pointers and their pointees, and if possible, function calls are inlined and evaluated. The linearity of the bug path makes this kind of analysis also relatively efficient. Data dependence analysis on the CFG, which is done with a *dataflow algorithm*, is not in such a privileged position. However, as we have demonstrated before, analyses on the CFG could yield information on code outside of what the bug path that can be valuable.

In this section, we discuss a dataflow algorithm called reaching definitions. When inquiring about which parts of the program was meant to affect the value of a bug causing variable, reaching definitions is an elegant solution to find relevant statements.

4.2.1 Dataflow analyses

As the name suggests, control flow analyses describe the flow of control within a program by inspecting the *structure* of the CFG; dataflow analyses complements this by analyzing how information (e.g., values of variables, state of mutexes, whether a value will be read from in a later basic block) flows, changes, and is accessed from one node to another by inspecting the *contents* of the CFG. Many notable dataflow analyses are defined by calculating an initial set of properties for each basic block, and propagating these properties along the edges of the CFG, so that properties “flow” from one block to another. Propagations may be described with dataflow equations; these are then repeatedly solved until these property sets change no more, reaching a fixpoint. Common initial property sets include GEN and KILL. Though might be defined somewhat differently from algorithm to algorithm, they are usually similar to the following: for basic block B, $GEN[B]$ is the set of variables read in B, and $KILL[B]$ is the set of variables written in B.

As an example, live variable analysis [18] calculates the set of live variables in a given basic block. A variable is live if its value *may* be read in subsequent blocks. Formally, a variable x is live in block i, if block j uses the value of x, and there exists a path from i to j without any interleaving assignments to x. $LIVE_{in}[B]$ is the set of variables live at the beginning of block B, and $LIVE_{out}[B]$ is the set of variables live at the end of B. [4] In Figure 1a, `x` is live in blocks B4, B3 and B2, but not in B1 and B0. It is indeed possible to express liveness with dataflow equations:

$$LIVE_{in}[B] = GEN[B] \cup (LIVE_{out}[B] \setminus KILL[B])$$
$$LIVE_{out}[B] = \bigcup_{S \in succ[B]} LIVE_{in}[S]$$

This definition *overapproximates* the actual set of live variables. Suppose in Figure 1a `g()` is known to always set `flag`’s value to false. Although x would be a dead variable throughout the entire function, liveness analysis, and dataflow analyses in general are incapable of telling whether a path of execution in the CFG is feasible.

C/C++ presents several challenges to overcome in calculating GEN/KILL sets. Due to the aliasing problem presented by pointers, it can be difficult or impossible to tell which variables are read or written through aliasing. Another significant obstacle is posed by function calls, as dataflow analyses are defined to reason about a single CFG at a time. These limiting factors force the analysis to over- or underestimate its results even further. Clang in particular faces a number of additional problems; its AST, to which the CFG links back to, was designed for diagnostics construction, not for such an analysis [53]. While lacking an intermediate representation higher than LLVM IR but lower then Clang AST makes it rather difficult to implement in Clang for the purpose of finding programming errors, there are a few, such as Clang’s thread safety analysis [27] and lifetime analysis [49, 24, 25, 32].

4.2.2 Reaching definitions analysis

We call the write of variable x a *definition* of x. Any statement that *may* write x (e.g. through aliasing) is also regarded as a definition of x. When describing analyses concerning definitions, we define $GEN[B]$ sets such that they contain

[4] As basic blocks are sequences of operations executed sequentially, they might not be granular enough, as the same variable may be written multiple times in a given block. In such a case, valuable liveness information is lost *inside* the block. This problem can be solved by splitting up basic block to only contain a single statement.

the set of definitions present in basic block B („B writes x"), and $KILL[B]$ sets such that they contain every other definitions in the CFG that generate the same variables as B („B overwrites the value x may have gotten in other blocks").

The ingoing reaching definitions [18] set of B ($REACH_{in}[B]$) is the set of definitions reaching B. The outgoing reaching definitions set of B ($REACH_{out}[B]$) is the incoming set minus the definitions killed by B, as well as the definitions generated by B. We can define reaching definitions with the following dataflow equations:

$$REACH_{in}[B] = \bigcup_{P \in pred[B]} REACH_{out}[P]$$

$$REACH_{out}[B] = GEN[B] \cup (REACH_{in}[B] \setminus KILL[B])$$

If the set of definitions to x at block B (a subset of $REACH_{in}[B]$) contains no elements, that means x is first defined in B, or not yet defined. If the set contains a single element, that means we can precisely tell which statement defines `x`'s value in B. If it has two or more elements, then `x`'s value might be different if different execution paths are chosen to reach B. In Figure 1a, if we denote definitions by a (variable, line number) pair, B1's reaching definitions set would be the following:

$$REACH_{in}[B1] = \{(flag, 15), (x, 9), (x, 13)\}$$

4.2.3 Integration of reaching definitions

The reaching definition set of B1 is very telling: it highlights that the definition of x in B3 reaches the block where x was a cause of a bug, which could be a hint that the developer intention to prevent the bug from occuring. Although the analyzer did not visit B3 on this path of execution and is absent from the bug path, one can tell that a control dependency of B3, namely the evaluation of the condition in B4, is. This realization could trigger the analyzer to start tracking flag on line 12. This would force the creation of diagnostic messages for the last store to flag, which is in a function called on line 11.

With reaching definitions, we would be able to find a point of interest to the bug but outside the bug path, and could instruct the analyzer to explain control flow around these points better. This would theoretically solve the class of problems demonstrated in Figure 3b. Reaching definitions analysis overapproximates the set of statements that are considered definitions, meaning that the analyzer should keep track of whether a definition was found as a result of overapproximation. With that said, the analyzer might be able to fill the gaps of information dataflow algorithms usually struggle with; suppose a pointer is written right before a division-by-zero error is discovered. Reaching definitions might be forced to conservatively assume that said pointer points to the denominator; however, it could ask the analyzer whether this aliasing is possible, and might be able to disregard the pointer assignment.

The class of problems displayed in Figure 3c is more difficult to detect accurately. Reaching definitions is confined to the bounds of f's CFG, and will not detect `x`'s potential write on line 10. One aspect that makes this case approachable is the fact that the function call to `h` is present on the bug path, and the analyzer will resolve that the parameter of `h` will alias with `x` in `f`. This won't make reaching definitions interprocedural but would grant a limited toolset to reason across a limited set of functions present on the bug path. Nonetheless, we would need to enhance our reaching definitions algorithm with some pointer aliasing capabilities.

For the last class of problems demonstrated in Figure 3d, we lose the ability to ask the analyzer to resolve parameter passing. While reaching definitions would find that `x` might be written on line 18, it will be a result of overapproximation, so the analyzer might not trust is enough to explain control flow around it. To reason about `h`, we are forced the reimplement some of the analyzers inlining technology to make reaching definitions to understand more than one function on its own. Should such a technology exist, we would need to survey how deep of a function call chain should we investigate to look for points of interest. This highlights how much more difficult it is to discover relevant information from the program the further we stray from the bug path.

The concept behind the interaction of reaching definitions with control dependency analysis displays the many of the characteristics of static backward program slicing.

5 Results

We evaluated our work from two perspectives. First, we gathered data on open source projects by running the Clang Static Analyzer on their source code before and after our improvements. We inspected almost all reports individually and tried to subjectively argue for or against whether the reports' readability improved. We also tried to find certain objective metrics to measure the impact of our work.

Second, we sent out surveys to participants with varying degree of expertice in C/C++ and static analysis to learn whether other developers would find our improvements beneficial.

5.1 Measurements on open source projects

We tested our solution on, as seen in Table 1, the following open-source C and C++ projects: Bitcoin [51], CppCheck [37], Gravity [8], gRPC [23], LLVM and Clang [35], OpenSSL [39], Protobuf [22], Rtags [6], S2N [2], TinyVM [31], Xerces [4] and XGBoost [58]. Combined, these projects cover a wide variety of coding techniques, codebase sizes, and different versions of the languages' standards.

In Table 1 we show how many reports did our contribution affect. Out of the 12 open source projects, reports remained unchanged in 7. Out of 1096 bug reports, 2.4% received additional notes. We intentionally fine tuned our solution to limit its impact, and have observed that preserved information from previously disregarded function calls always meaningfully added to the intelligility of the analyzed path of execution.

Table 1: Evaluation of control dependency tracking in terms of how many reports received additional notes. The last row shows findings every other project other than the first five remained unchanged.

	Total reports	Changed	% of changed
CppCheck	44	7	15.9 %
Gravity	16	1	6.3 %
gRPC	229	15	6.6 %
LLVM + Clang	249	2	0.8 %
Xerces	106	1	0.9 %
Others combined	451	0	0 %

In the case for Xerces, Gravity, some of the CppCheck reports, we were especially pleased on how the extra information on conditions provided further high-level information. We found that conditions closer to the bug point are more likely to be directly data dependent on the bug causing variable. Upon learning more about the condition, we also learned of more high-level properties on the bug causing variable.

In the case for LLVM, gRPC, and the other half of the CppCheck reports, when the pivotal point (like assigning null to a pointer) was very close to the bug point, the extra information did not add much to the already decent report. Even in these cases however, the rest of the report (altough not important to understand why the bug occured) were easier to understand.

In the context of how memory and runtime intensive static analysis is, the costs of bug report construction are usually assumed to be negligable. We expect our contribution in particular to very little impact even in the context of bug report construction, as all of the control flow analyses are calculated at a prior step in the compilation process, and we simply reuse it.

5.2 Survey

We sent out surveys 11 people to measure whether a developer not taking part in our research would find our improvements beneficial. Out of them, 9 participated All of them were male, their avarage age was 30 at the time of the survey, ranging from 24 to 56. Participation was free and voluntary.

While all of our participants were software developers, 3 of them mainly wrote code in Python, 2 of them were teaching C++ at our university but wrote little C/C++ code outside the classroom. The remaining 4 were full-time C++ developers.

All of the participants were familiar with static analysis, with 4 of them being active contributors to Clang itself (but to our research). The remaining 5 worked on visualizer tools for static analyzers, but not the analyzers themselves.

We selected 11 bug reports from those that we collected on analyzing open source projects. After our contribution, all of these reports contained additional information than prior to it. We will call these versions of the same bug report the "after" and the "before" versions.

All surveys contained all of the bug reports, but each report was only presented in either "before" or "after" state. Each survey way unique in terms of which reports were shown in which state, but all survey contained roughly the same number of "before" and "after" reports.

In total, we received 99 bug report evaluations. On the following question: "Sometimes, I was unsure how the analyzer analyzed this path of execution, and wished for more explanation.", answers could be given on a range from 1 to 5, with 1 strongly disagreeing and 5 strongly agreeing. As seen in Figure 5, on avarage, before our contribution users answered with 2.901/5, but this desire was somewhat lower after our contribution, a 2.804/5, while users rated "Some notes were annoying and made it more difficult to understand the bug." with the same score before and after our improvement.

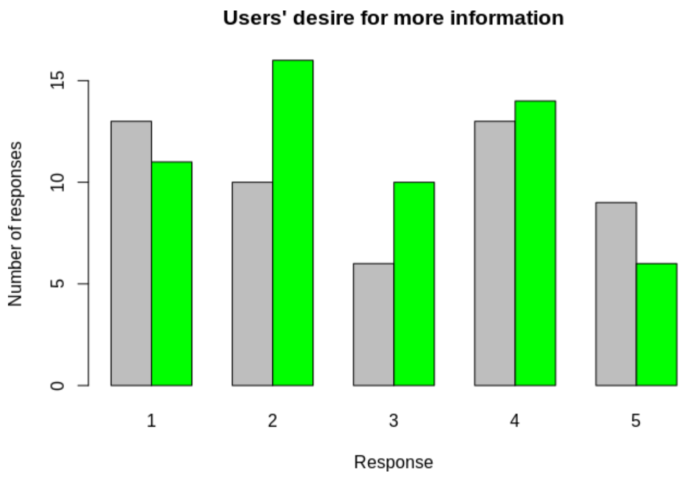


Figure 5: Responses to the question "Sometimes, I was unsure how the analyzer analyzed this path of execution, and wished for more explanation.". 1 strongly disagrees, 5 strongly agrees. The columns in grey display the score on bug reports prior to, and the columns in green display the score after our improvement.

5.3 Threats to validity

As for the evaluations on open source projects, our selection lacks meaningful amount of modern C++ code, specifically, C++14 or newer.

As for our survey, while in terms of expertise in C/C++, our participants varied in range, they were are rather knowledgable about the Clang Static Analyzer, with 7 of the 9 having made at least one contribution to it. Our survey could have benefitted from a greater range on familiarity with static analyzers. All of the participants that responded to our survey were male, a fact that could also use some diversifying.

Most importantly, our sample size of 9 participants and 11 bug reports is small. It is our view that a participant should have at least an intermediate C/C++ knowledge, and at least some familiarity with the concept of static analysis in real world applications, and it proved difficult to find people who met these criteria.

6 Notable examples

In this section, we highlight a few bug reports where control dependency tracking made a poor bug report significantly more readable.

While evaluating a large number of reports during static analysis, it is a good idea to read from the bottom up, as the root cause of the bug, for instance the last assignment to a variable before it participates in a division by zero error, may be close to the error point. This means essential part of the bug report might be shorter than the full report. Starting from the bottom allows the user to disregard the first few of the report as non-consequential.

In the following examples, we advise to read the reports from the top down, unless stated otherwise.

6.1 Example 1

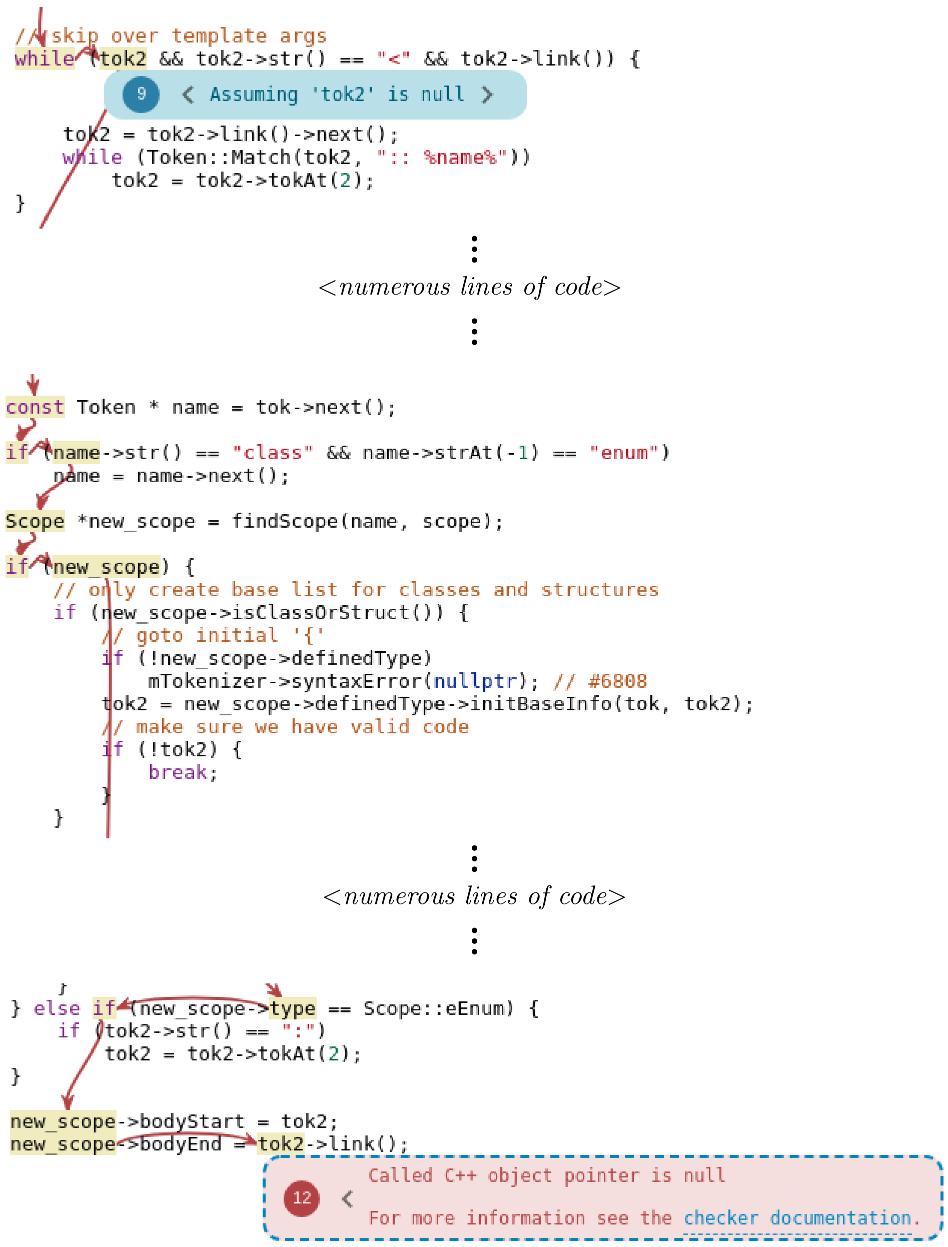
This example is from the project CppCheck, in the file `lib/symboldatabase.cpp`. In Figure 6, the bug report is shown which was generated without our improvement. At one point, variable `tok2` is assumed to be null, which eventually leads to a null dereference bug. To decide whether this report is a true positive, and if so, how to fix it, a good question to ask is *"Are there any interleaving condition points that should've prevented the flow of control from reaching a derefence of `tok2` while its null?"*.

This is rather challenging in this bug report: at one point, the local variable `new_scope` is defined, and is already known to be null in the next condition (if the analyzer would have assumed its value on the condition point, it would have placed a note there, implying that the analyzer learned of its value earlier). Is this because `findScope` unconditionally returns a null pointer, and its effect is only observable on its parameters? If not, why are there no explanations?

In Figure 7, we show the relevant part of the bug report, but after our improvements. A pair of new notes appeared on the function call to `findScope`, and leads to its definition. It forwards us to a call to a non-cost member function with the same name. There, we learn that this function can indeed return non-null values, but the analyzer managed to find a path of execution where this function returns null.

Our improvement recognized that `new_scope` is a control dependency to the bug point, and information about it should be presented.

BEFORE:



```
// skip over template args
while (tok2 && tok2->str() == "<" && tok2->link()) {
    9  < Assuming 'tok2' is null >
    tok2 = tok2->link()->next();
    while (Token::Match(tok2, ":: %name%"))
        tok2 = tok2->tokAt(2);
}
```

⋮
<numerous lines of code>
⋮

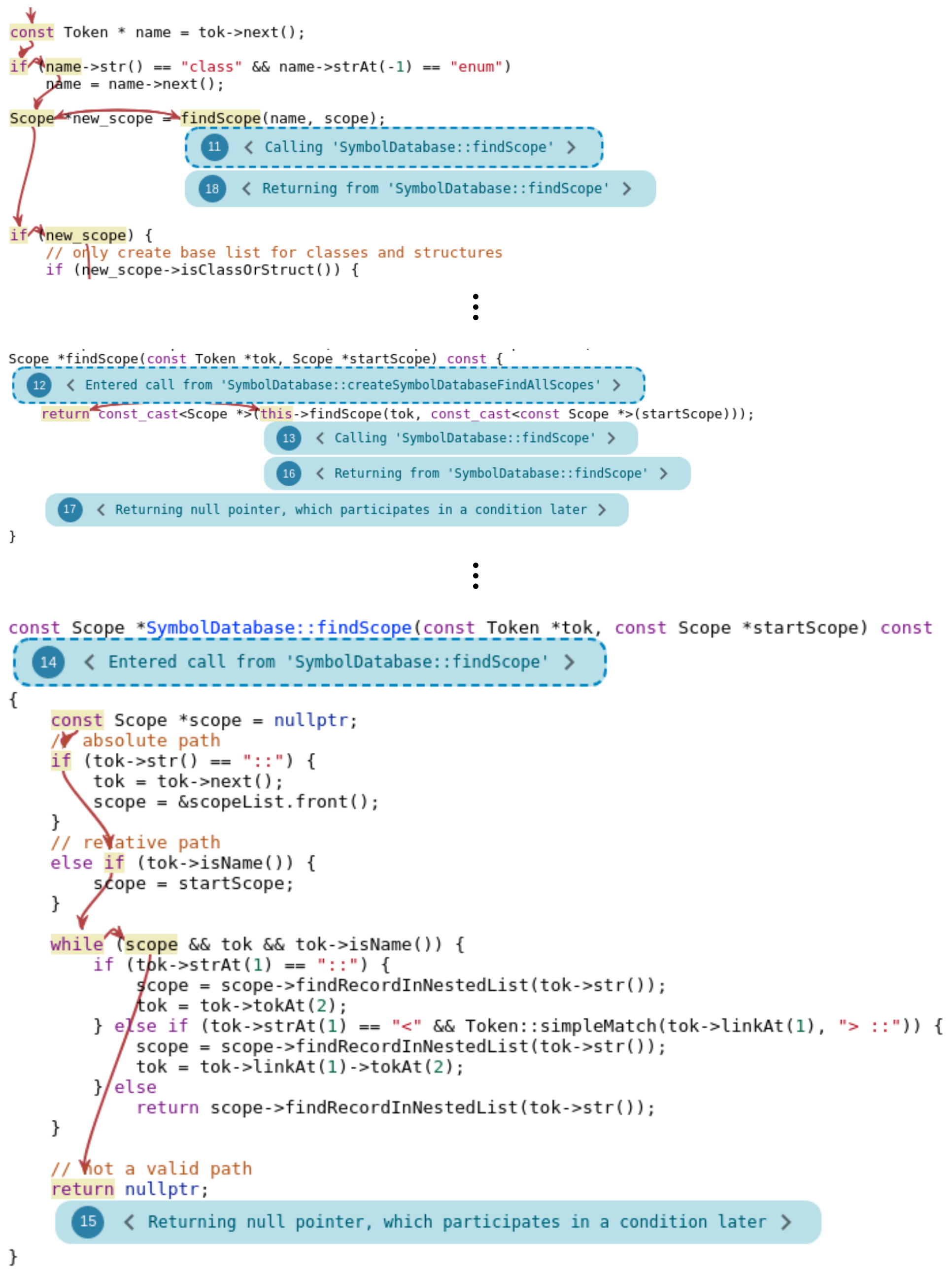
```
const Token * name = tok->next();

if (name->str() == "class" && name->strAt(-1) == "enum")
    name = name->next();

Scope *new_scope = findScope(name, scope);

if (new_scope) {
    // only create base list for classes and structures
    if (new_scope->isClassOrStruct()) {
        // goto initial '{'
        if (!new_scope->definedType)
            mTokenizer->syntaxError(nullptr); // #6808
        tok2 = new_scope->definedType->initBaseInfo(tok, tok2);
        // make sure we have valid code
        if (!tok2) {
            break;
        }
    }
```

⋮
<numerous lines of code>
⋮

```
} else if (new_scope->type == Scope::eEnum) {
    if (tok2->str() == ":")
        tok2 = tok2->tokAt(2);
}

new_scope->bodyStart = tok2;
new_scope->bodyEnd = tok2->link();
    12  <  Called C++ object pointer is null

           For more information see the checker documentation.
```

Figure 6: Bug report before our improvement from CppCheck

AFTER:



```
const Token * name = tok->next();
if (name->str() == "class" && name->strAt(-1) == "enum")
    name = name->next();
Scope *new_scope = findScope(name, scope);
                    11  < Calling 'SymbolDatabase::findScope' >
                    18  < Returning from 'SymbolDatabase::findScope' >
if (new_scope) {
    // only create base list for classes and structures
    if (new_scope->isClassOrStruct()) {
```

⋮

```
Scope *findScope(const Token *tok, Scope *startScope) const {
  12  < Entered call from 'SymbolDatabase::createSymbolDatabaseFindAllScopes' >
    return const_cast<Scope *>(this->findScope(tok, const_cast<const Scope *>(startScope)));
                      13  < Calling 'SymbolDatabase::findScope' >
                      16  < Returning from 'SymbolDatabase::findScope' >
      17  < Returning null pointer, which participates in a condition later >
}
```

⋮

```
const Scope *SymbolDatabase::findScope(const Token *tok, const Scope *startScope) const
  14  < Entered call from 'SymbolDatabase::findScope' >
{
    const Scope *scope = nullptr;
    // absolute path
    if (tok->str() == "::") {
        tok = tok->next();
        scope = &scopeList.front();
    }
    // relative path
    else if (tok->isName()) {
        scope = startScope;
    }

    while (scope && tok && tok->isName()) {
        if (tok->strAt(1) == "::") {
            scope = scope->findRecordInNestedList(tok->str());
            tok = tok->tokAt(2);
        } else if (tok->strAt(1) == "<" && Token::simpleMatch(tok->linkAt(1), "> ::")) {
            scope = scope->findRecordInNestedList(tok->str());
            tok = tok->linkAt(1)->tokAt(2);
        } else
            return scope->findRecordInNestedList(tok->str());
    }

    // not a valid path
    return nullptr;
      15  < Returning null pointer, which participates in a condition later >
}
```

Figure 7: New notes after our improvement in the report from CppCheck

6.2 Example 2

This example is from the project Xerces in the file `TraverseSchema.cpp`. Reading the report in Figure 8 from the bottom up shows that the first parameter on `reportSchemaError` is dereferenced as a null pointer. Moving up, we can see that the null pointer originated from the caller function's local variable, `content`. We entered this code block because `simpleTypeRequired` is true, which was set because a chain of conditions, among them the fact that `baseTypeInfo` is non-null, lead to that assignment. Earlier, we can see that the execution was not halted by an exception, in part because `baseValidator` is null (shown by the flow of control, which is visualized by the arrows). `content`'s last store is present in the full the bug report, but we omitted it on this figure.

It is clear why `simpleTypeRequired` is known to be true at the condition point, but as to why were both `baseTypeInfo` and `baseValidator` known to be non-null and null respectively, is not explained by the report. Their definition gives a clue, it is related to `typeInfo`, but how does the analyzer *know* the values returned by those getter functions so precisely?

In Figure 9, which displays parts of the bug report after our improvement, we are greeted by new notes explaining what happens inside `processBaseTypeInfo`. Inside the function, we see two variables with the same types that `baseTypeInfo` and `baseValidator` had being initialized to null. Later, `baseComplexTypeInfo`'s value changes, and is assumed to be non-null, while `baseDTValidator`'s value remains unchanged. The last two statements of the function uses setter functions on `typeInfo` with these variables.

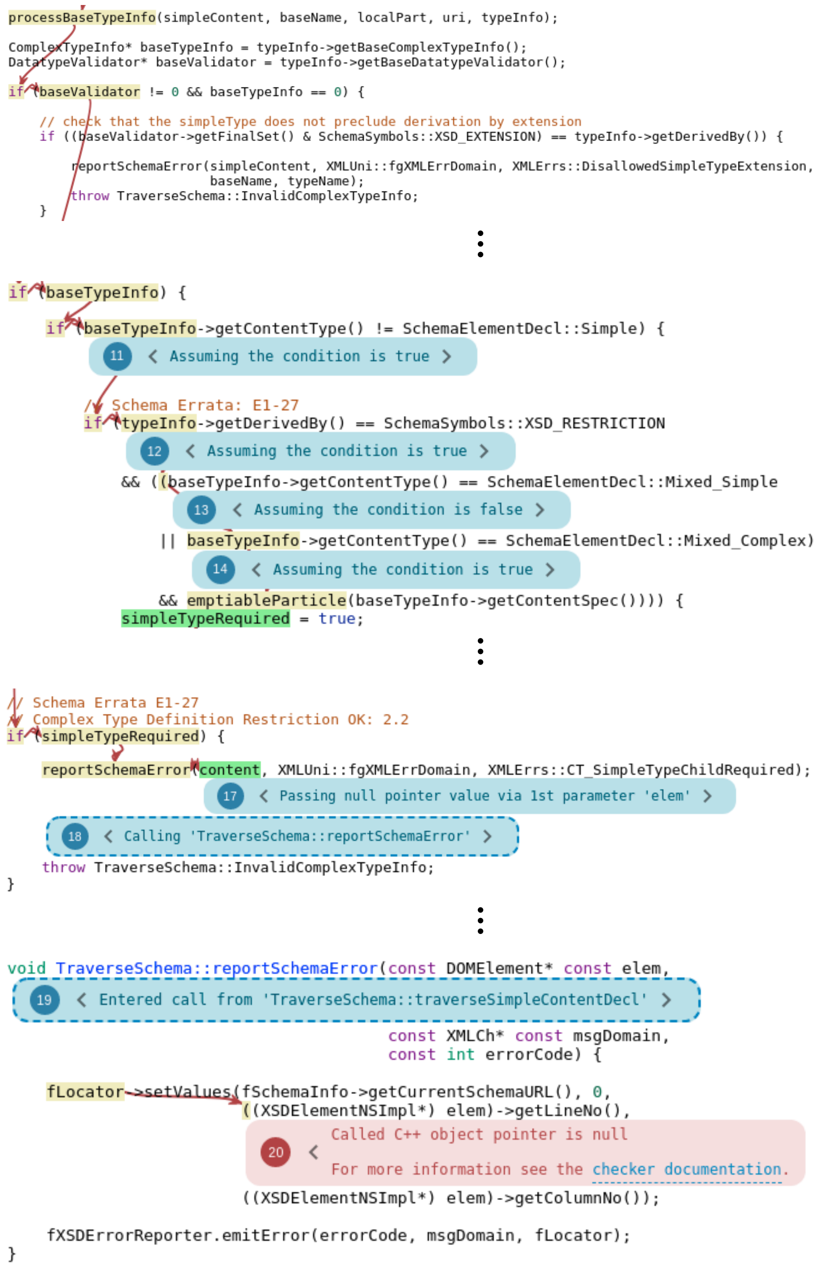
Upon reviewing the definition of `baseTypeInfo` and `baseValidator`, we can see that the getter functions they are initialized with pair with these setter functions, explaining how the analyzer knew their precise value.

Our improvement saw that `simpleTypeRequired` is a control dependency to the bug-causing function call, and started tracking it. Its last store was control dependent (in part) by `baseTypeInfo`, which initiated its tracking. `baseValidator` is tracked as it played a part in preventing the program from throwing an exception, but this could have been omitted, as the value of `baseTypeInfo` would have prevented that anyways.

6.3 Example 3

This example is from LLVM, in the file `clang/lib/CodeGen/CGObjCGNU.cpp`. In Figure 11, we see a bug report before our solution. Reading from the bottom up, we see that `OID` is dereferenced as a nullpointer. In a branch inside a range-based for loop, we see the only statement that could have written this variable before its definition. `II` seems to play an important role in the retrieved range, which is initialized based on the parameter of the function, `Name`. Following the flow on control up, we see that `isWeak` is known to be false. Notably, we see `GetClassVar` being initialized by a call to `SymbolForClassRef`, which takes both `Name` and `isWeak` as parameter.

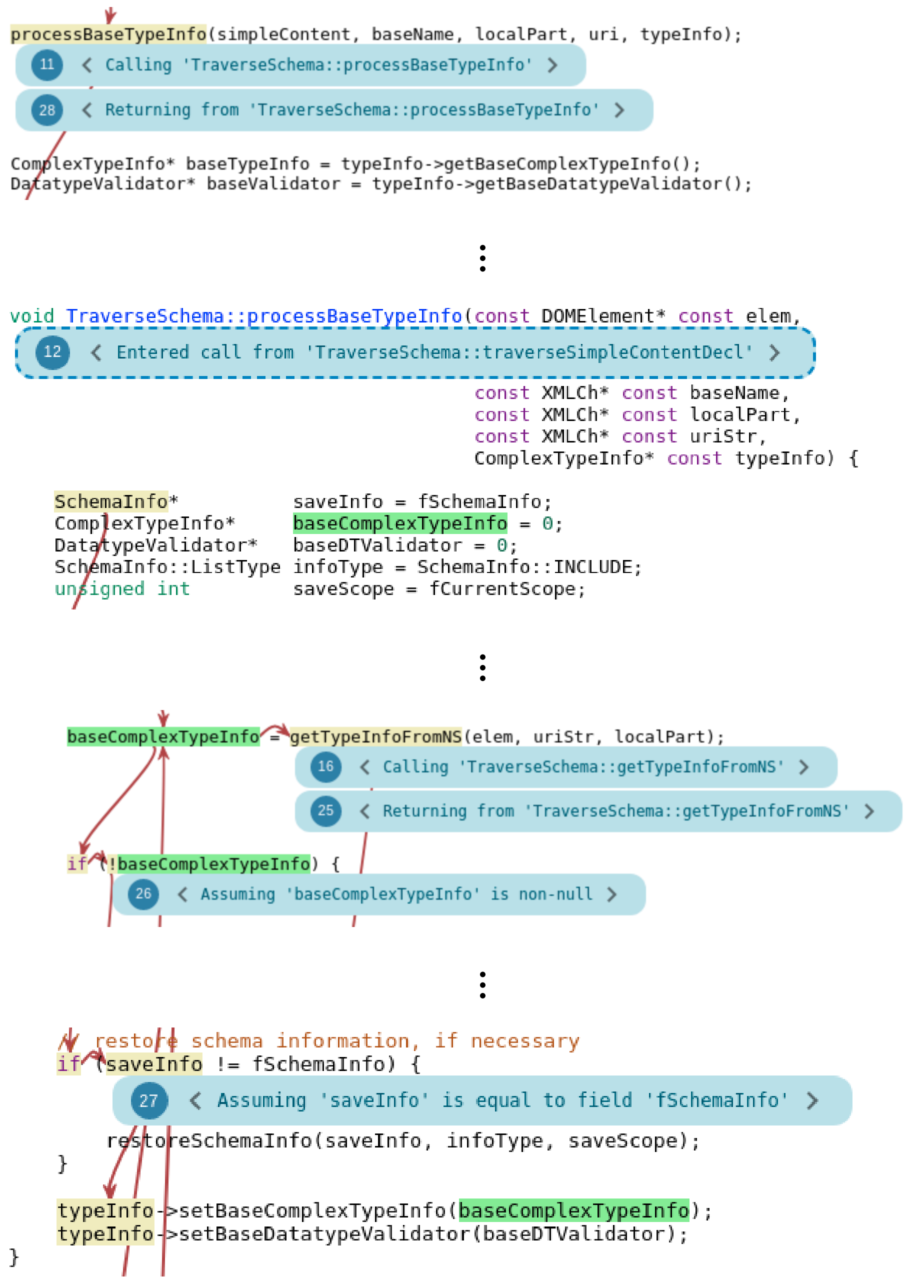
BEFORE:



```
processBaseTypeInfo(simpleContent, baseName, localPart, uri, typeInfo);

ComplexTypeInfo* baseTypeInfo = typeInfo->getBaseComplexTypeInfo();
DatatypeValidator* baseValidator = typeInfo->getBaseDatatypeValidator();

if (baseValidator != 0 && baseTypeInfo == 0) {

    // check that the simpleType does not preclude derivation by extension
    if ((baseValidator->getFinalSet() & SchemaSymbols::XSD_EXTENSION) == typeInfo->getDerivedBy()) {

        reportSchemaError(simpleContent, XMLUni::fgXMLErrDomain, XMLErrs::DisallowedSimpleTypeExtension,
                          baseName, typeName);
        throw TraverseSchema::InvalidComplexTypeInfo;
    }
                                    ⋮
if (baseTypeInfo) {

    if (baseTypeInfo->getContentType() != SchemaElementDecl::Simple) {
      11  < Assuming the condition is true >

        // Schema Errata: E1-27
        if (typeInfo->getDerivedBy() == SchemaSymbols::XSD_RESTRICTION
          12  < Assuming the condition is true >
            && ((baseTypeInfo->getContentType() == SchemaElementDecl::Mixed_Simple
              13  < Assuming the condition is false >
                || baseTypeInfo->getContentType() == SchemaElementDecl::Mixed_Complex)
                14  < Assuming the condition is true >
                && emptiableParticle(baseTypeInfo->getContentSpec()))) {
            simpleTypeRequired = true;
                                    ⋮
// Schema Errata E1-27
// Complex Type Definition Restriction OK: 2.2
if (simpleTypeRequired) {

    reportSchemaError(content, XMLUni::fgXMLErrDomain, XMLErrs::CT_SimpleTypeChildRequired);
                      17  < Passing null pointer value via 1st parameter 'elem' >
      18  < Calling 'TraverseSchema::reportSchemaError' >
    throw TraverseSchema::InvalidComplexTypeInfo;
}
                                    ⋮
void TraverseSchema::reportSchemaError(const DOMElement* const elem,
  19  < Entered call from 'TraverseSchema::traverseSimpleContentDecl' >
                                       const XMLCh* const msgDomain,
                                       const int errorCode) {

    fLocator->setValues(fSchemaInfo->getCurrentSchemaURL(), 0,
                        ((XSDElementNSImpl*) elem)->getLineNo(),
                          Called C++ object pointer is null
                   20  <
                          For more information see the checker documentation.
                        ((XSDElementNSImpl*) elem)->getColumnNo());

    fXSDErrorReporter.emitError(errorCode, msgDomain, fLocator);
}
```

Figure 8: Bug report before our improvement from Xerces

AFTER:



```
processBaseTypeInfo(simpleContent, baseName, localPart, uri, typeInfo);
  11  < Calling 'TraverseSchema::processBaseTypeInfo' >
  28  < Returning from 'TraverseSchema::processBaseTypeInfo' >

ComplexTypeInfo* baseTypeInfo = typeInfo->getBaseComplexTypeInfo();
DatatypeValidator* baseValidator = typeInfo->getBaseDatatypeValidator();
```

⋮

```
void TraverseSchema::processBaseTypeInfo(const DOMElement* const elem,
  12  < Entered call from 'TraverseSchema::traverseSimpleContentDecl' >
                                         const XMLCh* const baseName,
                                         const XMLCh* const localPart,
                                         const XMLCh* const uriStr,
                                         ComplexTypeInfo* const typeInfo) {

    SchemaInfo*           saveInfo = fSchemaInfo;
    ComplexTypeInfo*      baseComplexTypeInfo = 0;
    DatatypeValidator*    baseDTValidator = 0;
    SchemaInfo::ListType  infoType = SchemaInfo::INCLUDE;
    unsigned int          saveScope = fCurrentScope;
```

⋮

```
     baseComplexTypeInfo = getTypeInfoFromNS(elem, uriStr, localPart);
                             16  < Calling 'TraverseSchema::getTypeInfoFromNS' >
                             25  < Returning from 'TraverseSchema::getTypeInfoFromNS' >

     if (!baseComplexTypeInfo) {
        26  < Assuming 'baseComplexTypeInfo' is non-null >
```

⋮

```
    // restore schema information, if necessary
    if (saveInfo != fSchemaInfo) {
        27  < Assuming 'saveInfo' is equal to field 'fSchemaInfo' >
        restoreSchemaInfo(saveInfo, infoType, saveScope);
    }

    typeInfo->setBaseComplexTypeInfo(baseComplexTypeInfo);
    typeInfo->setBaseDatatypeValidator(baseDTValidator);
}
```

Figure 9: New notes after our improvement in the report from Xerces.

To decide whether this bug report is a true positive or not, we must, in part, show that `Name` can hold values that allows the flow of control to reach the range-based for loop, but never reach the assignment to `OID` inside it. Since the return value of `SymbolForClassRef` seems to influence whether the function returns early, the intention could have been that problematic values of `Name` should result in this early exit. It is also unclear why `isWeak` is known to be false, the intention of the programmer could have been to prevent the flow on control reaching the error point with its value as well.

In Figure 11, we see the relevant parts of the bug report after our improvement. A pair of notes around the call to `SymbolForClassRef` link to the function's definition. There, we learn where `isWeak`'s value was assumed, and we get a better picture of the return value that was later provided for the initialization of `ClassSymbol`. Also, the correlevance of `Name`, `isWeak` and the early return is proven, so an expert on this domain can likely judge the validity of the report.

7 Related work

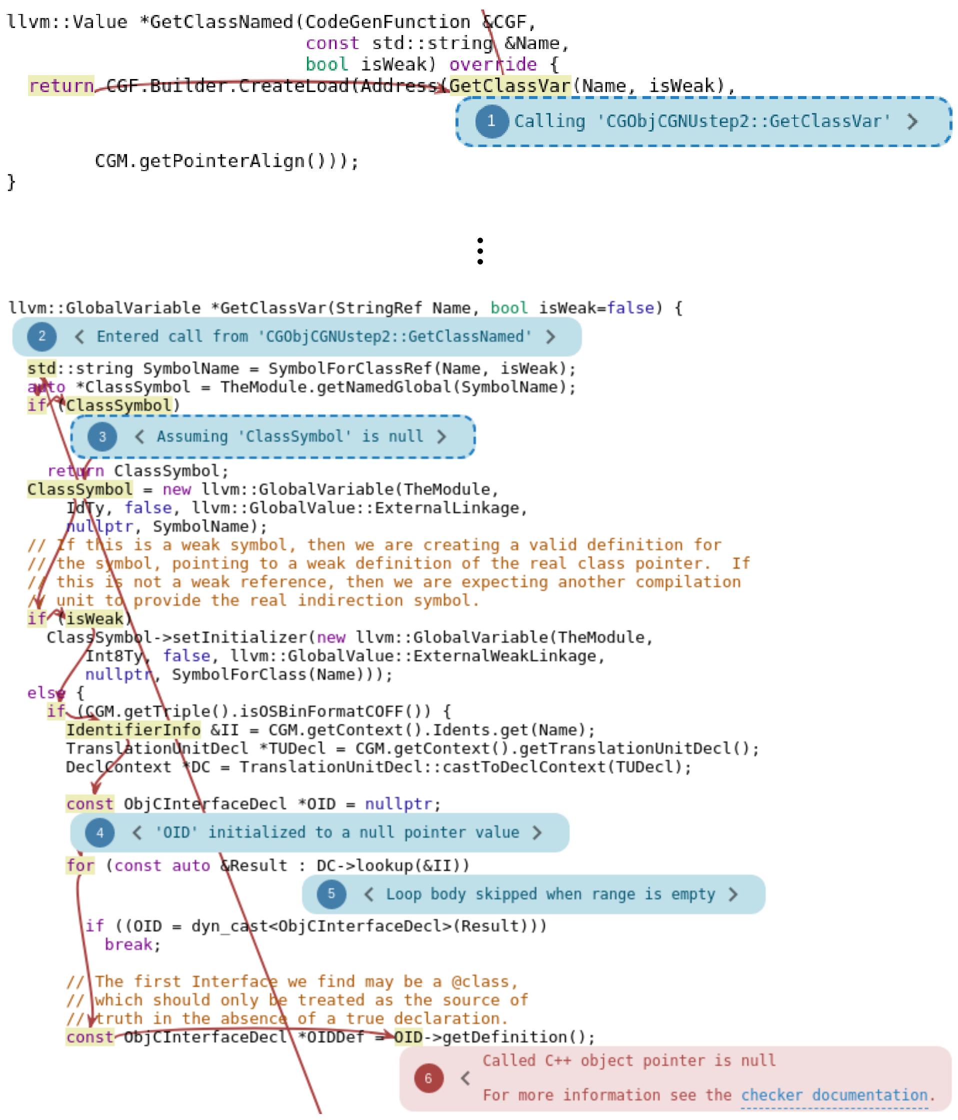
In 2008, the authors of paper [5] reported that while the static analysis methods are frequent research areas in the academy, there are no many usage examples in the industry. Current trends show a growing industrial interest of the static analysis tools [9, 61].

Industry leader software companies show the most positive approach towards static analysis and its application in every day development. Google, Apple, Microsoft, Facebook, and others also participate in the development of such tools. Paper [45] reports about the lessons learned while developing static analysis tools at Google. The authors list the most frequent problems resulting in the developers not using static analysis tools or ignoring their warnings. These are the lack of tool integration into the developer's workflow; the fact that *many warnings are not actionable*; the high number of false positives; situations where the bug is theoretically possible but in practice it does not manifest; the possibly high cost of the fix; and that *the users do not understand the warnings.*

The authors emphasize the importance of *actionable* messages: the warnings should include a suggestion to the (possible) fix, which in the best case could be applied mechanically. However, the authors state that many serious issues cannot be detected correctly or automatically fixed. In that case of the latter, the fix depends on the correct understanding of the report. They also claim that the developer's happiness is a crucial factor for the successful introduction of static analysis on an organizational level. Non-understandable reports cause frustration among engineers and work against trust in static analysis tools.

The authors in [28] investigate why the use of static analysis tools is not as widespread as it would be possible. Unlike earlier studies, they focused on the developer's perception on using the tools, including the interaction with the user interface. The research was conducted via 40-60 minutes long semi-structured interviews with 20 developers with experience ranging from 3 to 25 years. Among

BEFORE:

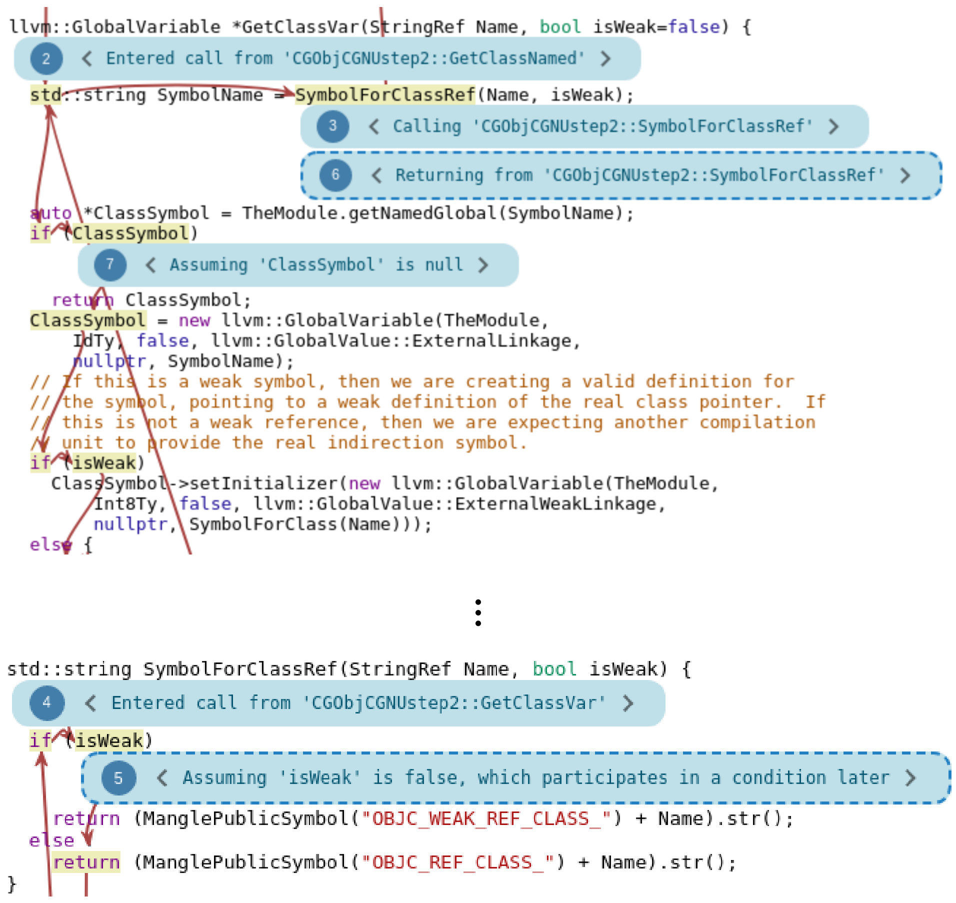


```
llvm::Value *GetClassNamed(CodeGenFunction &CGF,
                           const std::string &Name,
                           bool isWeak) override {
  return CGF.Builder.CreateLoad(Address(GetClassVar(Name, isWeak),
                                       1 Calling 'CGObjCGNUstep2::GetClassVar' >
        CGM.getPointerAlign()));
}

                                ⋮

llvm::GlobalVariable *GetClassVar(StringRef Name, bool isWeak=false) {
  2 < Entered call from 'CGObjCGNUstep2::GetClassNamed' >
  std::string SymbolName = SymbolForClassRef(Name, isWeak);
  auto *ClassSymbol = TheModule.getNamedGlobal(SymbolName);
  if (ClassSymbol)
      3 < Assuming 'ClassSymbol' is null >
    return ClassSymbol;
  ClassSymbol = new llvm::GlobalVariable(TheModule,
      IdTy, false, llvm::GlobalValue::ExternalLinkage,
      nullptr, SymbolName);
  // If this is a weak symbol, then we are creating a valid definition for
  // the symbol, pointing to a weak definition of the real class pointer.  If
  // this is not a weak reference, then we are expecting another compilation
  // unit to provide the real indirection symbol.
  if (isWeak)
    ClassSymbol->setInitializer(new llvm::GlobalVariable(TheModule,
        Int8Ty, false, llvm::GlobalValue::ExternalWeakLinkage,
        nullptr, SymbolForClass(Name)));
  else {
    if (CGM.getTriple().isOSBinFormatCOFF()) {
      IdentifierInfo &II = CGM.getContext().Idents.get(Name);
      TranslationUnitDecl *TUDecl = CGM.getContext().getTranslationUnitDecl();
      DeclContext *DC = TranslationUnitDecl::castToDeclContext(TUDecl);

      const ObjCInterfaceDecl *OID = nullptr;
      4 < 'OID' initialized to a null pointer value >
      for (const auto &Result : DC->lookup(&II))
                                5 < Loop body skipped when range is empty >
        if ((OID = dyn_cast<ObjCInterfaceDecl>(Result)))
          break;

      // The first Interface we find may be a @class,
      // which should only be treated as the source of
      // truth in the absence of a true declaration.
      const ObjCInterfaceDecl *OIDDef = OID->getDefinition();
                    6 < Called C++ object pointer is null
                        For more information see the checker documentation.
```

Figure 10: Bug report before our improvement from LLVM.

these 20 developers, 14 people expressed negative impact on *the way in which the warnings are presented*. Apart from the possibility of overwhelming false positive warnings they mentioned that the reports are non-intuitive.

AFTER:



```
llvm::GlobalVariable *GetClassVar(StringRef Name, bool isWeak=false) {
  2 < Entered call from 'CGObjCGNUstep2::GetClassNamed' >
  std::string SymbolName = SymbolForClassRef(Name, isWeak);
                  3 < Calling 'CGObjCGNUstep2::SymbolForClassRef' >
                  6 < Returning from 'CGObjCGNUstep2::SymbolForClassRef' >
  auto *ClassSymbol = TheModule.getNamedGlobal(SymbolName);
  if (ClassSymbol)
        7 < Assuming 'ClassSymbol' is null >
    return ClassSymbol;
  ClassSymbol = new llvm::GlobalVariable(TheModule,
      IdTy, false, llvm::GlobalValue::ExternalLinkage,
      nullptr, SymbolName);
  // If this is a weak symbol, then we are creating a valid definition for
  // the symbol, pointing to a weak definition of the real class pointer.  If
  // this is not a weak reference, then we are expecting another compilation
  // unit to provide the real indirection symbol.
  if (isWeak)
    ClassSymbol->setInitializer(new llvm::GlobalVariable(TheModule,
        Int8Ty, false, llvm::GlobalValue::ExternalWeakLinkage,
        nullptr, SymbolForClass(Name)));
  else {
```

⋮

```
std::string SymbolForClassRef(StringRef Name, bool isWeak) {
  4 < Entered call from 'CGObjCGNUstep2::GetClassVar' >
  if (isWeak)
        5 < Assuming 'isWeak' is false, which participates in a condition later >
    return (ManglePublicSymbol("OBJC_WEAK_REF_CLASS_") + Name).str();
  else
    return (ManglePublicSymbol("OBJC_REF_CLASS_") + Name).str();
}
```

Figure 11: New notes after our improvement in the report from LLVM.

As the result of their research, they conclude that the developers are not able to understand what the tool is telling her, and it is a definite barrier to use static analysis tools. Nineteen of our 20 participants, felt that many static analysis tools do not present their results in a way that gives enough information for them to assess what the problem is, why it is a problem and what they should be doing differently.

We discussed that in the general case, the more complex a static analysis system is, the harder it is to construct intelligible reports for them. The authors in [46] discuss a static analysis technique with the usage of the preprocessor – a historically difficult concept to write good diagnostics around.

A similar methodology was conducted in the research published in paper [55]. The authors surveyed 40+ developers and interviewed 11 industrial experts to understand the possibilities of better prioritization of static analysis tool reports. Among other interesting results, they found that *... warnings hard to integrate in case they do not have teammates having enough expertise for fixing them. However, those warnings can be easily understood if the tools provide exhaustive descriptions.*

8 Future work

We feel cautiously optimistic about our proposal regarding reaching definitions analysis, though we are yet to implement it and gather real world-results. This links back to the problems posed by Clang's infrastructure: its AST was constructed to make the construction of user-readable diagnostics easy, not so much for dataflow analysis. We made considerable progress in implementing a reaching definition analysis but paused to reflect on whether changes to the current repertoire intermediate representations are in order. Creating a new IR is a large undertaking, so we are researching the best course of action to take on this front.

The analyzer is aware that it is limited in terms of the information it can harness. For instance, calling functions with unavailable definitions often force a clear of its constraints on a subset of variables. Similar events are often large contributors to the appearance of false positive reports. After the analysis is concluded, the analyzer will inspect each bug find whether they are likely false positives, and regularly suppresses a portion of them. The more the analyzer understands what parts of the program are relevant to a bug, the more precisely it can suppress such reports; we are currently researching how to integrate our results and proposals into this library.

Reaching definitions analysis could be a valuable component for new checkers to find even more intricate bugs by complementing symbolic execution with dataflow information.

The authors in [13] discuss combining the Clang Static Analyzer with the dynamic symblic analyzer KLEE to refine the analysis. They highlight that traces provided by Clang are not that useful, and that Clang struggles to find non-trivial true positive. Maybe if the communication in between these tools improves (with the help of Clang itself better understanding the intetion of the programmer), research in this area could show new results as well.

9 Conclusion

Static analysis and symbolic execution specifically is a powerful technique to find deeply rooted programming errors. As an interprocedural path sensitive analysis, it gains a sophisticated understanding of how values would behave in a runtime environment without actually executing the program. However, it often struggles to turn these discoveries to easily comprehensible bug reports, demanding even more of the most expensive resource and least available in a software development project: human experts.

In this paper, we demonstrated that the root cause of poor bug reports to otherwise valuable discoveries are caused by the fact symbolic execution can only reason about a single path of execution at a time. After the analysis is concluded, tools such as the Clang Static Analyzer inspect the sequence of program states leading to the error, and construct a set of diagnostic messages and notes to explain the flow control and change of values. However, these program states are oblivious to what could have happened on alternative paths of execution, as well as control dependence.

We propose two techniques to complement bug report generation. Control dependency analysis can tell that a condition point may have played a large part in the bug's occurrence. Reaching definitions analysis finds parts of the code could have changed the value of a bug causing variable had control flown there. We project the interaction of these techniques to replicate a program slicing-like behavior, significantly increasing an analyzer tools' understanding of the causes behind a bug. Our improved bug report generation facilities, which has been a part of the Clang Static Analyzer stable releases since version 10.0.0., demonstrates how the discovery of such information allows a tool to construct more comprehensive bug reports.

Acknowledgment

We would like to extend our heartfelt gratitude towards Artem Dergachev, who actively participated in every part of this research, including planning, implemeneting, measuring and evaluating.

References

[1] Aho, A., Sethi, R., and Ullman, J. *Compilers principles, techniques, and tools*. Addison-Wesley, Reading, MA, 1986.

[2] Amazon Web Services. S2n, 2022. URL: https://github.com/awslabs/s2n/.

[3] Anders, M. and Michael, I. Static program analysis, 2012. URL: https://users-cs.au.dk/amoeller/spa/spa.pdf.

[4] Apache Software Foundation. Apache Xerces, 2022. URL: https://xerces.apache.org/.

[5] Ayewah, N., Pugh, W., Hovemeyer, D., Morgenthaler, J., and Penix, J. Using static analysis to find bugs. *IEEE Software*, 25(5):22–29, 2008. DOI: 10.1109/ms.2008.130.

[6] Bakken, A. Rtags, 2022. URL: http://www.rtags.net.

[7] Baldoni, R., Coppa, E., D'Elia, D., Demetrescu, C., and Finocchi, I. A survey of symbolic execution techniques. *ACM Computing Surveys*, 51(3):1–39, 2018. DOI: 10.1145/3182657.

[8] Bambini, M. Gravity, 2022. URL: https://github.com/marcobambini/gravity.

[9] Beller, M., Bholanath, R., McIntosh, S., and Zaidman, A. Analyzing the state of static analysis: A large-scale evaluation in open source software. In *Proceedings of the IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering*. IEEE, 2016. DOI: 10.1109/saner.2016.105.

[10] Bessey, A., Block, K., Chelf, B., Chou, A., Fulton, B., Hallem, S., Henri-Gros, C., Kamsky, A., McPeak, S., and Engler, D. A few billion lines of code later: Using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, 2010. DOI: 10.1145/1646353.1646374.

[11] Binkley, D. and Harman, M. A large-scale empirical study of forward and backward static slice size and context sensitivity. In *Proceedings of the International Conference on Software Maintenance*, pages 44–53. IEEE, 2003. DOI: 10.1109/ICSM.2003.1235405.

[12] Boehm, B. and Basili, V. Software defect reduction top 10 list. *Computer*, 34(1):135–137, 2001. DOI: 10.1109/2.962984.

[13] Busse, F., Gharat, P., Cadar, C., and Donaldson, A. Combining static analysis error traces with dynamic symbolic execution (experience paper). In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 568–579. ACM, 2022. DOI: 10.1145/3533767.3534384.

[14] Checker Developer Manual. Clang Static Analyzer: Checker Developer Manual, 2019. URL: https://clang-analyzer.llvm.org/checker_dev_manual.html (last accessed: 24-04-2023).

[15] Clang Static Analyzer, 2019. URL: https://clang-analyzer.llvm.org/.

[16] Clang-Tidy, 2019. URL: https://clang.llvm.org/extra/clang-tidy/ (last accessed: 24-04-2023).

[17] CodeSecure. CodeSonar, 2019. URL: https://codesecure.com/our-products/codesonar/ (last accessed: 15-02-2024).

[18] Cooper, K. and Torczon, L. *Engineering a compiler*. Elsevier, 2011. ISBN: 9780120884780.

[19] Cytron, R., Ferrante, J., Rosen, B., Wegman, M., and Zadeck, F. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991. DOI: 10.1145/115372.115320.

[20] Dergachev, A. Clang Static Analyzer: A checker developer's guide, 2016. URL: https://github.com/haoNoQ/clang-analyzer-guide (last accessed: 24-04-2023).

[21] Ericsson. CodeChecker, 2022. URL: https://github.com/Ericsson/codechecker.

[22] Google. Protobuf, 2022. URL: https://github.com/protocolbuffers/protobuf.

[23] gRPC Authors. grpc, 2022. URL: https://grpc.io/.

[24] Horváth, G. and Gehre, M. Implementing the C++ Core Guidelines' lifetime safety profile in Clang. European LLVM Developers Meeting, Brussels, 2019. URL: https://llvm.org/devmtg/2019-04/talks.html#Talk_18.

[25] Horváth, G. and Pataki, N. Categorization of C++ classes for static lifetime analysis. In *Proceedings of the 9th Balkan Conference on Informatics*, pages 1–7. ACM, 2019. DOI: 10.1145/3351556.3351559.

[26] Horwitz, S., Reps, T., and Binkley, D. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, 1990. DOI: 10.1145/77606.77608.

[27] Hutchins, D., Ballman, A., and Sutherland, D. C/C++ thread safety analysis. In *Proceedings of the IEEE 14th International Working Conference on Source Code Analysis and Manipulation*. IEEE, 2014. DOI: 10.1109/scam.2014.34.

[28] Johnson, B., Song, Y., Murphy-Hill, E., and Bowdidge, R. Why don't software developers use static analysis tools to find bugs? In *Proceedings of the 35th International Conference on Software Engineering*, pages 672–681. IEEE, 2013. DOI: 10.1109/ICSE.2013.6606613.

[29] Khoo, Y., Foster, J., Hicks, M., and Sazawal, V. Path projection for user-centered static analysis tools. In *Proceedings of the 8th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '08, pages 57—-63, New York, NY, USA, 2008. Association for Computing Machinery. DOI: 10.1145/1512475.1512488.

[30] King, J. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976. DOI: 10.1145/360248.360252.

[31] Kogut, J. TinyVM, 2022. URL: https://github.com/jakogut/tinyvm.

[32] Kovács, R., Horváth, G., and Porkoláb, Z. Detecting C++ lifetime errors with symbolic execution. In *Proceedings of the 9th Balkan Conference on Informatics*, pages 1–6, 2019. DOI: 10.1145/3351556.3351585.

[33] Kremenek, T. Finding software bugs with the Clang Static Analyzer. Apple Inc., 2008. URL: https://llvm.org/devmtg/2008-08/Kremenek_StaticAnalyzer.pdf.

[34] Lattner, C. LLVM and Clang: Next generation compiler technology, 2008. Lecture at BSD Conference. URL: https://llvm.org/pubs/2008-05-17-BSDCan-LLVMIntro.html.

[35] Lattner, C. and Adve, V. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, pages 75–86. IEEE Computer Society, 2004. DOI: 10.1109/CGO.2004.1281665.

[36] Layman, L., Williams, L., and Amant, R. Toward reducing fault fix time: Understanding developer behavior for the design of automated fault detection tools. In *Proceedings of the First International Symposium on Empirical Software Engineering and Measurement*, pages 176–185. IEEE Computer Society, 2007. DOI: 10.1109/ESEM.2007.82.

[37] Marjamäki, D. CppCheck: A tool for static C/C++ code analysis, 2013. URL: http://cppcheck.sourceforge.net/.

[38] Nethercote, N. and Seward, J. Valgrind: A framework for heavyweight dynamic binary instrumentation. *ACM SIGPLAN Notices*, 42(6):89–100, 2007. DOI: 10.1145/1273442.1250746.

[39] OpenSSL Software Foundation. OpenSSL, 2022. URL: https://openssl.org/.

[40] Ottenstein, K. and Ottenstein, L. The program dependence graph in a software development environment. In *Proceedings of the first ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 177–184. ACM Press, 1984. DOI: 10.1145/800020.808263.

[41] Park, J., Lim, I., and Ryu, S. Battles with false positives in static analysis of Javascript web applications in the wild. In *Proceedings of the IEEE/ACM 38th International Conference on Software Engineering Companion*, pages 61–70. IEEE, 2016. URL: https://ieeexplore.ieee.org/document/7883289.

[42] Perforce. Klocwork, 2024. URL: https://www.perforce.com/products/klocwork (last accessed: 15-02-2024).

[43] Reps, T., Horwitz, S., and Sagiv, M. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 49–61. ACM, 1995. DOI: 10.1145/199448.199462.

[44] Rice, H. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74:358–366, 1953. DOI: 10.2307/1990888.

[45] Sadowski, C., Aftandilian, E., Eagle, A., Miller-Cushon, L., and Jaspan, C. Lessons from building static analysis tools at Google. *Communications of the ACM*, 61(4):58–66, 2018. DOI: 10.1145/3188720.

[46] Schubert, P., Gazzillo, P., Patterson, Z., Braha, J., Schiebel, F., Hermann, B., Wei, S., and Bodden, E. Static data-flow analysis for software product lines in C. *Automated Software Engineering*, 29(1), 2022. DOI: 10.1007/s10515-022-00333-1.

[47] Serebryany, K., Bruening, D., Potapenko, A., and Vyukov, D. AddressSanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, Berkeley, CA, USA, 2012. USENIX Association. URL: http://dl.acm.org/citation.cfm?id=2342821.2342849.

[48] Sreedhar, V. and Gao, G. A linear time algorithm for placing φ-nodes. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 62–73. ACM Press, 1995. DOI: 10.1145/199448.199464.

[49] Sutter, H. Lifetime safety: Preventing common dangling. Technical report, Microsoft Corporation, 2018. URL: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1179r1.pdf.

[50] Synopsys. Coverity, 2019. URL: https://scan.coverity.com/ (last accessed: 24-04-2023).

[51] The Bitcoin Core. Bitcoin Core, 2022. URL: https://bitcoincore.org/.

[52] Umann, K. The penultimate challange: Constructing bug reports in the Clang Static Analyzer. LLVM Developers' Meeting, San Jose, CA, 2019. URL: https://llvm.org/devmtg/2019-10/talk-abstracts.html#tech17.

[53] Umann, K. Enhancing bug reports in the Clang Static Analyzer, 2019. URL: https://szelethus.github.io/gsoc2019/ (last accessed: 24-04-2023).

[54] Umann, K. A survey of dataflow analyses in Clang, 2020. URL: https://lists.llvm.org/pipermail/cfe-dev/2020-October/066937.html (last accessed: 24-04-2023).

[55] Vassallo, C., Panichella, S., Palomba, F., Proksch, S., Zaidman, A., and Gall, H. Context is king: The developer perspective on the usage of static analysis tools. In *Proceedings of the IEEE 25th International Conference on Software Analysis, Evolution and Reengineering*, pages 38–49. IEEE, 2018. DOI: 10.1109/SANER.2018.8330195.

[56] Vidács, L., Beszédes, ., and Gyimóthy, T. Combining preprocessor slicing with C/C++ language slicing. *Science of Computer Programming*, 74(7):399–413, 2009. DOI: 10.1016/j.scico.2009.02.003.

[57] Weiser, M. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, 1984. DOI: 10.1109/tse.1984.5010248.

[58] XGBoost Contributors. XGBoost, 2022. URL: https://xgboost.ai/.

[59] Xu, Z., Kremenek, T., and Zhang, J. A memory model for static analysis of C programs. In *Proceedings of the 4th International Conference on Leveraging Applications of Formal Methods, Verification, and Validation — Volume Part I*, ISoLA'10, pages 535–548, Berlin, Heidelberg, 2010. Springer-Verlag. URL: http://dl.acm.org/citation.cfm?id=1939281.1939332.

[60] Zaks, A. and Rose, J. Building a checker in 24 hours, 2012. URL: https://www.youtube.com/watch?v=kdxlsP5QVPw.

[61] Zampetti, F., Scalabrino, S., Oliveto, R., Canfora, G., and Di Penta, M. How open source projects use static code analysis tools in continuous integration pipelines. In *Proceedings of the IEEE/ACM 14th International Conference on Mining Software Repositories*. IEEE, 2017. DOI: 10.1109/msr.2017.2.

Contents

ISSN 0324—721 X (Print)
ISSN 2676—993 X (Online)

Editor-in-Chief: Tibor Csendes