Volume 27 Number 2

ACTA CYBERNETICA

Szeged, 2025

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/.

Gabriel Istrate
Faculty of Mathematics and Computer Science, University of Bucharest, Romania
gabriel.istrate@unibuc.ro

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

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 2024

Guest Editor:

Judit Jász

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

Preface

The *14th 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 Jul 3 – July 5, 2024.

The members of the *Scientific Committee* were the following representatives of the Hungarian doctoral schools in Computer Science: János Csirik (SZTE), Lajos Rónyai (SZTAKI, BME), András Benczúr (ELTE), András Erik Csallner (SZTE) Erzsébet Csuhaj-Varjú (ELTE), József Dombi (SZTE), József Dániel Dombi (SZTE), Richárd Farkas (SZTE), István Fazekas (DE) Zoltán Fülöp (SZTE), Katalin Hangos (MTA) Ferenc Hartung (PE) Zoltán Horváth (ELTE), Márk Jelasity (SZTE), Tibor Jordán (ELTE), Attila Kertész (SZTE), Ákos Kiss (SZTE), László Kóczy (SZE), Andrea Kő (Corvinus), János Levendovszki (BME), Gyöngyvér Márton (Sapientia EMTE), Valerie Novitzka (TUKE), László Nyúl (SZTE), Kálmán Palágyi (SZTE), Attila Pethő (DE), Tamás Pflanzer (SZTE), Gábor Szederkényi (PPKE), János Végh (ME).

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

There were more than 36 participants and 20 talks in several fields of computer science and its applications (7 sessions). The talks were going in sections in Image Processing, Security, Blockchain, Testing, Computation, and Development 1-2.

The talks of the students were completed by 2 plenary talks of leading scientists: Gábor Péter Nagy (SZTE, Hungary) and Márk Jelasity (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 9 manuscripts were submitted for review, out of which 7 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 **27** (2025) 117–140.

Optimizing SAP Machine Learning-based Solutions through Custom API Integration

Georgina Asuah[ab], Arafat Md Easin[ac], and Tamás Orosz[ad]

Abstract

Rapid changes, dynamic consumer preferences, and evolving market trends are the hallmarks of the business environment. SAP HANA has emerged as a potent platform to meet this demand due to its resilient foundation for real-time data analytics and processing and in-memory processing architecture. This research aims to improve anomaly detection capabilities by integrating machine learning (ML) models into the SAP HANA Fiori web application. This will be achieved by developing a custom Application Programming Interface (API). The proposed solution integrates ML models with the SAP system using FastAPI, providing real-time insights and decision-making capabilities, by employing Local Outlier Factor (LOF) for anomaly detection. Multiple ML estimators were evaluated and the results indicate that LOF consistently outperforms other models, offering higher detection accuracy and computational efficiency. This research provides a practical framework for integrating machine learning-based anomaly detection into enterprise applications, addressing the limitations of SAP's built-in Predictive Analysis Library (PAL). To guarantee seamless performance and scalability, the API is deployed on Azure using Docker containers. This paper presents the capability of custom APIs to integrate ML models into enterprise systems, enhance operational efficiency, and establish a reliable framework for real-time anomaly detection as a practical solution. The article addresses challenges associated with API integration, scalability, and system configuration, providing valuable insights for enhancing the deployment of machine learning in enterprise applications. These findings offer valuable insights for organizations seeking to enhance their predictive analytics capabilities using modern AI-driven approaches.

Keywords: SAP HANA Fiori, machine learning, API integration, anomaly detection, Local Outlier Factor (LOF)

[a]Department of Data Science and Engineering, Faculty of Informatics, Eötvös Loránd University, Budapest, Hungary
[b]E-mail: asuahgeorgina@inf.elte.hu, ORCID: 0009-0004-4390-7494
[c]E-mail: arafatmdeasin@inf.elte.hu, ORCID: 0000-0003-4014-9144
[d]E-mail: orosztamas@inf.elte.hu, ORCID: 0000-0003-0595-6522

DOI: 10.14232/actacyb.312225

1 Introduction

Organizations nowadays strive to derive actionable insights from their massive data sets, owing to prevalent digital technologies and big data [31]. The business world is characterized by rapid changes, evolving tastes of customers, and unpredictable market trends [35]. Companies in diverse sectors now consider real-time decision assistance a need rather than a luxury. SAP HANA emerged as a strong platform to address this need, due to its in-memory computing design and solid basis for processing and analytics of data in real-time [10].

However, a smooth integration of machine learning and artificial intelligence (AI) capabilities is necessary to fully explore SAP HANA's potential [9, 15]. Tasks like anomaly detection benefit greatly from this connection since ML models can detect variations in data trends, which can help spot vulnerabilities like fraud, system failures, or inventory shortages early. This link allows companies to quickly and easily conclude their data which will help them make better decisions. Unsupervised anomaly detection methods can be easily built using SAP HANA's Predictive Analysis Library (PAL) [22]. However, this approach has limitations, such as not letting the user modify the algorithm's settings or apply domain expertise for anomaly identification. This necessitates using custom APIs to enhance the precision and efficiency of anomaly detection.

Nowadays, ML algorithms utilize data analysis techniques to identify patterns and correlations in historical data, enabling the extraction of valuable information and the creation of algorithms [33]. Application Programming Interfaces are vital for connecting machine learning models to enterprise systems such as SAP HANA [27]. Organizations can efficiently tackle unique business difficulties by adapting ML models to their specific needs and requirements through these APIs [7]. Custom APIs offer flexibility in integrating specialized machine learning models tailored to the unique needs of a business, providing a means to optimize these models' deployment and scaling [26]. APIs act as connectors, enabling seamless communication between machine learning algorithms and SAP applications [16]. They allow data to flow efficiently between these systems, ensuring that ML models can be integrated without significantly disrupting existing workflows.

The capacity to promptly identify deviations from typical behavior is essential for the preservation of operational efficiency, security, and system performance in anomaly detection. The anomaly detection process can be automated and enhanced by the integration of machine learning models into SAP HANA through custom APIs, which can provide real-time insights that traditional rule-based systems may overlook [13]. This research investigates the potential of custom API integration to optimize SAP machine learning-based solutions, with a particular emphasis on the improvement of anomaly detection capabilities in enterprise environments.

The main motivation is derived from the constraints of the current SAP HANA PAL capabilities, which restrict customization and domain-specific tailoring. Anomaly detection is essential for identifying unusual patterns in data, including fraud, system malfunctions, and inventory discrepancies. Although SAP HANA's PAL provides fundamental anomaly detection capabilities, it is unable to integrate so-

phisticated ML models. In this study, we developed a custom API that incorporates sci-kit-learn's Local Outlier Factor (LOF) anomaly detection model to overcome these constraints and offer a more flexible anomaly detection solution. The aim of this research is the development of a resilient custom API using FastAPI to integrate machine learning models, specifically the LOF, with SAP Fiori Web, and the deployment of this solution on Azure Kubernetes Service (AKS) to guarantee scalability, security, and high availability. Improve the pace and precision of decision-making in real-time enterprise settings.

The main contributions of this study are summarized as follows:

- Developing a FastAPI-based model for the integration of machine learning anomaly detection with the SAP HANA Fiori application.
- Optimizing anomaly detection where various machine learning models were analyzed, revealing that the LOF exhibits enhanced accuracy, recall, and ROC AUC.
- Employing containerized cloud-based deployment through Docker and AKS to enhance scalability, security, and high availability.
- Designing an interactive interface for seamless integration of SAP HANA Fiori, enabling real-time anomaly detection within enterprise SAP applications.

2 Literature Review

Applying machine learning models for anomaly detection within SAP systems is becoming more popular as organizations strive to become more operationally efficient, reduce risks, and improve data-oriented decision-making [4, 12]. It has been noted that anomaly detection is important for any enterprise system as it involves detecting suspicious activity.

Anomaly detection methods are numerous, with some basic techniques extending to the use of artificial intelligence in application. Traditional statistical techniques such as Z-score and boxplot-based outlier detection are commonly used [18]. Techniques such as these establish a cutoff point based on mean-variance or other statistical moments such as quantiles. A Z-score, for example, tells how many standard deviations a given observation is away from the average, with higher scores meaning that they are closely related to anomaly activity [8]. However, these methods do not work satisfactorily with multi-dimensional or other complex data distributions. These methods are quite simple to adopt but the gutter lies on the prerequisite of a certain type of distribution which effectively dismisses them on dynamic or complex spheres.

Machine learning methodologies have become increasingly popular owing to their capacity to represent intricate patterns. For instance, Support Vector Machines (SVM) have proven effective in anomaly detection by identifying a hyperplane that separates typical instances from atypical ones [28]. Similarly, neural

networks, especially autoencoders, have found widespread application, as these models reconstruct input data, with significant reconstruction errors signaling the presence of anomalies [25]. Although these models are adaptable, they typically necessitate substantial datasets for training and may struggle with limited interpretability. Ensemble techniques, such as Isolation Forest and Random Forest, have also been employed to improve anomaly detection. Isolation Forest isolates anomalies through recursive partitioning, making it particularly effective for high-dimensional datasets [17]. While ensemble methods provide robustness and enhanced generalization, they also introduce increased computational complexity and often necessitate careful hyperparameter tuning to achieve the best performance.

Custom API integration is essential for embedding machine learning-based anomaly detection models within SAP systems. APIs are the interface between ML models and enterprise systems, allowing for smooth data transfer and enabling real-time predictions. According to [11], RESTful APIs are frequently used to expose ML models, providing a standardized method for communication between SAP systems and ML services. These APIs should accommodate various data formats (e.g., JSON, XML) and feature clear endpoints for model training, inference, and monitoring. Creating custom APIs for machine learning integration necessitates adherence to some fundamental principles. Best practices for API integration emphasize several key considerations as described by [5, 34]. Firstly, versioning is critical to guarantee backward compatibility as APIs progress. Secondly, adhering to RESTful principles and using JSON-based communication can make APIs more adaptable and easier to integrate with different applications. Lastly, security is essential, especially when integrating APIs into enterprise systems. Implementing encryption, authentication, and role-based access control mechanisms is crucial to protect sensitive information.

As machine learning models evolve, continuous deployment pipelines should be implemented to automate model updates in production environments [3]. In [21], the authors proposed a distributed and unified API service for machine learning models that helps ensemble multiple models. This results in better predictions and benefits such as wider availability, greater usability, and lesser resource constraints. [24] tackled the issue of developing user-friendly ML APIs, particularly for beginners. Their research centered on examining how the Kaggle community utilizes scikit-learn, a popular ML API. The work of [23], discussed a case study showing how they integrated an SAP ERP system with an external web service through API access, illustrating the use of algorithms and transactions within SAP ERP.

Enhancing SAP machine learning solutions for anomaly detection involves utilizing various methods, including custom API integration. Creating custom APIs is essential for linking machine learning models with enterprise systems, simplifying complicated processes, and improving user experience. Custom APIs will continue to be essential for achieving smooth and scalable ML integration in enterprise systems as organizations delve into AI-powered solutions.

3 Methodology

The proposed solution is intended to capitalize on the machine learning models available in the esteemed scikit-learn (sklearn) library. The SAP HANA Fiori solution's proposed implementation employs machine learning models from the sklearn library to improve predictive modeling and data analysis. The FastAPI framework is implemented to incorporate predictive capabilities within a scalable and accessible application programming interface. The Azure cloud service provider is selected for its seamless integration with FastAPI and robust infrastructure, which is where the API deployment is orchestrated. The API that has been finalized, functions as a connection between the SAP HANA Fiori web application and the machine learning model. It improves the web application's functionality by integrating intelligent decision-making capabilities that are based on the predictions of the ML model. A critical component of the operational strategy is integrating an anomaly detection API into the SAP HANA Fiori web application.

3.1 System Architecture

The system architecture integrates five key components: SAP HANA, SAP Fiori, FastAPI, Azure Kubernetes Service (AKS), and Azure Container Registry (ACR). This design ensures flawless interaction between enterprise data management, anomaly detection, and cloud-based deployment. Figure 1 provides a detailed view of the system's flow, showcasing how user interactions in SAP HANA Fiori trigger anomaly detection through the custom API. The entire process starts with SAP HANA, which stores and preprocesses the dataset before sending it via OData services to the FastAPI backend. A Local Outlier Factor (LOF) model is hosted by FastAPI to evaluate incoming data and produce anomaly predictions in real time that are returned in JSON format. Azure Container Registry manages containerized instances of the FastAPI application, guaranteeing version-control, and secure image storage. AKS coordinates scalable deployment, integrating HTTPS encryption and token-based authentication for secure operations and dynamically adjusting resources to meet demand.

Afterwards, the processed data are sent to SAP Fiori, which offers an easy-to-use, role-based interface for interactive anomaly analysis and prediction visualization. Using SAP platforms (HANA, Fiori) for data handling and user interaction and Azure Cloud Services (ACR, AKS) for robust infrastructure management, the architecture prioritizes modularity. It is secured by Role-based Access Controls (RBAC) and built for smooth scalability in enterprise settings.

3.1.1 SAP HANA

SAP HANA (High-performance Analytic Appliance) is an advanced in-memory database and application development platform designed for processing large volumes of real-time data [20]. In-memory processing stores data directly in the main memory of a system rather than on traditional disk storage, and this significantly

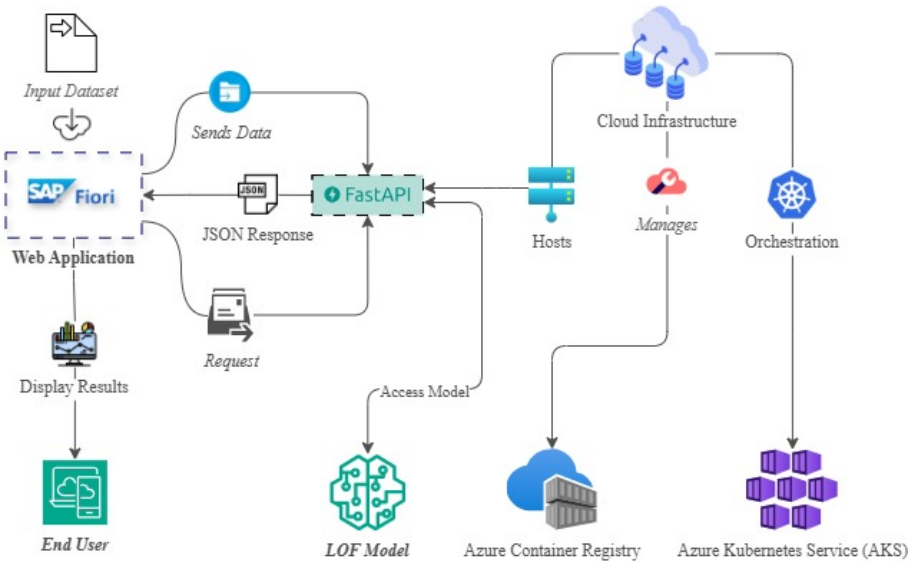


Figure 1: Overall architecture of the proposed system

enhances the processing speed for analytics and transactional workloads. It reduces the time taken to fetch data and accelerates computations by avoiding the latency related to disk I/O operations. Its Integrated Development Environment facilitates the creation of applications with the SAP HANA Deployment Infrastructure (HDI) containers, enabling smooth integration and data administration. Through its Predictive Analysis Library (PAL) [30], it facilitates predictive analytics, allowing developers to apply a variety of machine learning algorithms directly. The data source and preparation engine for this study is SAP HANA, which also prepares datasets and stores them in HDI containers for convenient access. It makes effective use of OData services to move data to external systems, such as the custom API.

3.1.2 SAP Fiori

SAP Fiori is a user experience (UX) platform for communicating intuitively and easily with enterprise systems through role-based interfaces [29]. It provides a responsive workflow for users on different devices through user-centered design, hence making laborious jobs less straining and hard business processes much easier. This work integrates it with the anomaly detection system using FastAPI, where the user can trigger the analysis in real time and in several directions with dynamic dashboards. Using SAPUI5 in the development, this platform provides much-needed customization possibilities according to organizational needs. The presented research extends the default functionality of SAP Fiori by incorporating new components, which interface directly with the anomaly detection API and showcase its adaptability in an advanced analytics context.

3.1.3 FastAPI

This study uses a scikit-learn LOF model to detect anomalies in real-time, and the real emotional core is a FastAPI-based backend. Incoming queries from SAP Fiori are processed by the API. It then uses the trained LOF algorithm to check the data for abnormalities and delivers predictions in standardized JSON format via RESTful endpoints. The system uses asynchronous request processing to maximize speed, guaranteeing low latency responsiveness and good scalability. This architecture allows anomaly scores to be dynamically shown on business dashboards by bridging the gap between machine learning algorithms and SAP Fiori's user interface.

3.1.4 Azure Container Registry

The centralized location for managing and storing Docker container images related to the FastAPI application is the Azure Container Registry. Immutable image tags provide strong version control, while integrated vulnerability assessment and role-based access restrictions guarantee safe deployment. ACR provides automated continuous integration and deployment (CI/CD) pipelines. It also and enables smooth connection with Azure Kubernetes Service by simplifying image distribution and authentication. This preserves adherence to company security rules while guaranteeing regular, auditable upgrades to the anomaly detection system.

3.1.5 Azure Kubernetes Service

The containerized FastAPI application is deployed and managed using Azure Kubernetes Service, which facilitates high availability and smooth scaling to satisfy business needs. Through auto-scaling capabilities, it automatically adjusts workloads in response to traffic changes, guaranteeing optimal resource use. To protect API endpoints and user interactions, AKS incorporates strong security mechanisms. These include token-based authentication using Azure Active Directory (AAD) and HTTPS encryption via ingress controllers. The platform supports zero-downtime upgrades and maintains a secure, auditable pipeline by utilizing Azure Container Registry for image retrieval and deployment. AKS is positioned as the foundation for production-ready anomaly detection processes because of its fault tolerance, scalability, and enterprise-grade security.

3.2 Data Preparation

The dataset[1] was the transactional sales data from SAP HANA (shows in Figure 2), preprocessed to remove null values and categorical anomalies. The sales transaction data from SAP HANA was deployed into a NativeDevelopment HDI container. This container functions as a repository for structured data, which is indispensable for developing and training machine learning models. The Multi-Target Application (MTA) paradigm was employed to import table definition files and construct

[1] https://webide.h08z.ucc.ovgu.de/watt/index.html

the MTA project, which initiates the configuration process. A `node.js` module with XSJS support was used to establish OData services, guaranteeing connectivity between the database and auxiliary services. The SAP HANA Fiori application enables data transfer to the API to facilitate real-time decision-making. Categorical variables, including Currency and Product, were encoded through Label Encoding, whereas numerical variables underwent standardisation via StandardScaler to maintain scale-invariance in model performance.

This study employed Stratified 10-Fold Cross-Validation to evaluate model performance in a robust and generalisable way. This approach guarantees that each fold maintains the same anomaly distribution as the complete dataset, which is essential in tasks involving imbalanced anomaly detection. Training and testing were conducted iteratively for each fold, with performance metrics averaged across the folds. A rigorous cross-validation process was employed on various anomaly detection models, such as Local Outlier Factor, One-Class SVM, Isolation Forest, and Robust Covariance, to determine the most effective estimator.

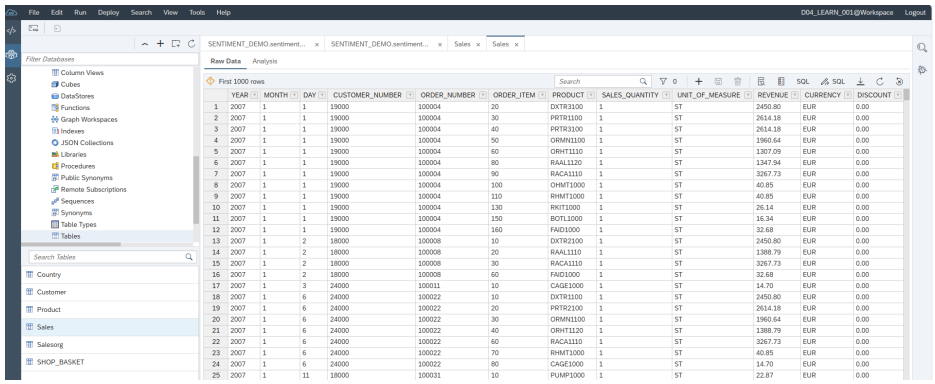


Figure 2: Snapshot of the sales dataset

3.3 Anomaly Detection Algorithms

In this paper, we have applied four machine-learning algorithms: Robust Covariance, Isolation Forest, One-Class Support Vector Machine, and Local Outlier Factor using the scikit-learn toolkit.

3.3.1 Robust Covariance

This method detects outliers by fitting data distribution through a robust estimation of covariance (e.g., Minimum Covariance Determinant) [1]. It calculates Mahalanobis distances to identify deviations from normality, making elliptical assumptions regarding data distributions. While performing well on low-dimensional Gaussian-like data, in high-dimensional data spaces, performance degrades since covariance estimation becomes unstable. The contamination parameter was set

to contaminatiset=0.05 which determines the predicted fraction of outliers and is important to its success. A mismatch in this parameter results in over- or under-detection. This scikit-learn method is intended to find outliers in datasets with a Gaussian distribution. It successfully detects outliers that depart from the center distribution by modeling the data and fitting an ellipse to it. When the data follows elliptical assumptions, this approach works well.

3.3.2 Isolation Forest

Isolation Forest is a tree-based ensemble method that effectively isolates anomalies by recursive partitioning. A process utilizing the fact that anomalies require fewer splits to be separated due to sparsity in the feature space [18, 32]. Though effective on high-dimensional data, it scales poorly with dataset size. The predicted percentage of outliers is indicated by the contamination parameter, set to contamination=0.05 in this implementation. The number of tren_estimators controls the number of trees which was set to 100 in this experiment to balance speed and accuracy.

3.3.3 One-Class SVM

This technique, which assumes anomalies are few and unique, finds anomalies by learning a decision border around normal data [19]. The choice of the kernel (radial basis function, for example) and hyperparameter tuning, specifically, nu (contamination estimate), kernel, and gamma, determine how successful it is. The implemented One-Class SVM algorithm uses the hyperparameters nu=0.05, kernel='rbf', and gamma='auto'. This is well suited for cases when the anomalies are well separated and kernel parameters match the inherent structure of the data.

3.3.4 Local Outlier Factor

This algorithm estimates the density of every point as a function depending on its k nearest neighbors. If the point's local density is lower compared to its k nearest neighbors, then it can be labeled as an anomaly. This reachability Distance to smoothen out this density notion can be formalized as the maximum actual distance between the two points, or the k^{th} nearest neighbor distance. [14]. Local Reachability Density (LRD) [2] is the inverse of the average reachability distance of a point's neighbors. LOF creates a score ratio between LRD for each point and an average of that same neighborhood around it. Scores greater than 1 would mark possible anomalies. The usefulness of LOF resides in the fact that it is a non-parametric technique with no assumptions of particular data distributions. This algorithm handles datasets with different densities more efficiently than global methods like the Z-score by evaluating the local density deviation of a data point about its neighbors. The n_neighbors (denoted as k) option adds versatility by enabling customization for various anomaly features. To find novel abnormalities in new data points The following hyperparameters were used: n_neighbors=80, contamination=0.05, metric='manhattan', and novelty=True.

The base algorithm, the LOF, was selected because of its versatility and performance. The ability of the LOF model to detect local density deviations and identify outliers in complex datasets led to its selection for deployment. Because it is non-parametric, it can adapt to different data distributions. When anomalies differ in degree from the norm, LOF performs exceptionally well. The pseudocode for the LOF algorithm is presented in Algorithm 1.

Algorithm 1: Local Outlier Factor (LOF) Algorithm

Input : Dataset $D = \{x_1, \ldots, x_N\}$; number of neighbors k; contamination level τ; distance metric d; novelty detection flag

Output: Anomaly labels $L = \{l_1, \ldots, l_N\}$, where $l_i \in \{\text{normal}, \text{outlier}\}$

foreach *point* $p \in D$ **do**
  Compute distance matrix M where $M_{ij} = d(p_i, p_j)$
end
foreach *point* $p \in D$ **do**
  Find k^{th} nearest neighbors: $\text{kNN}(p) \leftarrow \text{sort}(M_p)[1{:}k+1]$
end
foreach *point* $p \in D$ **do**
  foreach $q \in kNN(p)$ **do**
    Compute reachability distance:
    $\text{reach-dist}(p, q) \leftarrow \max(d(p, q), \text{distance}(q, q_k))$
  end
end
foreach *point* $p \in D$ **do**
  Compute local reachability density:
  $\text{LRD}(p) \leftarrow \frac{1}{\text{avg}(\{\text{reach-dist}(p,q) \mid q \in \text{kNN}(p)\})}$
end
foreach *point* $p \in D$ **do**
  Compute LOF score:
  $\text{LOF}(p) \leftarrow \frac{\text{avg}(\{\text{LRD}(q) \mid q \in \text{kNN}(p)\})}{\text{LRD}(p)}$
end
Determine threshold θ using contamination level τ
foreach *point* $p \in D$ **do**
  if $LOF(p) > \theta$ **then**
    $l_p \leftarrow$ outlier
  else
    $l_p \leftarrow$ normal
  end
end
return L

3.4 Performance Evaluation

The performance metrics and evaluation protocols used in this study to precisely assess the efficacy of the anomaly detection models are as follows:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{1}$$

The percentage of correctly categorized cases (both normal and abnormal) is quantified.

$$\text{Precision} = \frac{TP}{TP + FP} \tag{2}$$

Evaluate the percentage of actual anomalies among all predicted anomalies to determine how well the model prevents detection errors.

$$\text{Recall} = \frac{TP}{TP + FN} \tag{3}$$

Assesses the model's ability to prevent false detections while identifying the majority of true anomalies.

$$\text{F1} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{4}$$

Represents the balance between false positives and false negatives by taking the harmonic mean of precision and recall.

Execution Time: The Total amount of time (in seconds) needed for both model inference and training is essential for evaluating scalability.

Area Under the ROC Curve (AUC-ROC):

$$\text{AUC} = \int_0^1 \text{TPR}(x)\,dx \quad \text{or equivalently,} \quad \text{AUC} = \sum(\text{TPR} \times \Delta\text{FPR}) \tag{5}$$

The model's capability to distinguish anomalies from normal instances at various thresholds of classification.

$$\text{TPR} = \frac{TP}{TP + FN}; \quad \text{FPR} = \frac{FP}{FP + TN} \tag{6}$$

The assessment of anomaly detection models is predicated on several crucial terms: False Negatives (FN) are anomalies that the model failed to identify, False Positives (FP) are normal cases that were mistakenly identified as anomalies, True Negatives (TN) are normal instances that were correctly classified, and True Positives (TP) are anomalies that were accurately identified. Overall, anomaly classification based on threshold values determined by contamination levels. The evaluation employed a stratified 10-fold cross-validation approach to enhance robustness and generalizability. This method ensures that each fold preserves the distribution

of anomalies across the dataset, thereby mitigating bias in imbalanced learning tasks. Performance metrics, including accuracy, precision, recall, F1 score, and ROC AUC, were calculated using Scikit-learn's classification tools and averaged in all folds to obtain a reliable estimate of the effectiveness of each model.

4 Results and Discussion

This work involves the creation of a REST API using Python to enhance SAP ML-based solutions. The experimental results are shown and discussed below. This application consumes OData API/Service and loads data from the system. The experiment evaluated various machine learning models for anomaly detection, including Robust Covariance (RC), Isolation Forest (IF), One-Class SVM (OCSVM), and Local Outlier Factor (LOF) from the scikit-learn. To test the models for anomaly detection, the sales transaction data was used to evaluate the estimators. Table 1 compares the performance metrics of evaluated models, highlighting LOF's superior results. The hyperparameter specification for LOF includes k (number of neighbors): 80, Contamination: 0.05, metric: 'manhattan', Novelty: True. The Python pickle library saves the trained model for reuse via API.

4.1 Model Selection for Anomaly Detection

Our primary goal was to deliver robust outlier detection within SAP's inherently time-critical transaction environment, where prediction latency must remain minimal. Classical methods such as RC, IF, OCSVM, and LOF offer sub-second inference times and have demonstrated very strong detection performance in our benchmarks. We therefore prioritized these algorithms to ensure both speed and accuracy under production constraints.

Table 1: Comparison of different anomaly detection models evaluated on the full feature set using 10-fold cross-validation

Model	Acc	Precision	Recall	F1-Score	ROC AUC	Execution Time
RC	0.989	0.898	0.900	0.898	0.941	4.3s
IF	0.983	0.834	0.828	0.829	0.910	4.73s
OCSVM	0.969	0.674	0.756	0.711	0.868	**1.48s**
LOF	**0.991**	**0.911**	**0.920**	**0.914**	**0.958**	9.37s

Finding the optimum anomaly detection approach necessitates balancing sensitivity to class imbalance, computational efficiency, and accuracy [6]. Table 1 compares four popular anomaly detection models (RC, IF, OCSVM, and LOF) for all the features using 10-fold cross-validation, across accuracy, precision, recall, F1-score, ROC AUC, and execution time. Whereas, LOF achieves the best overall detection quality with the highest accuracy (0.991), precision (0.911), and recall

(0.920), yielding the top F1-score (0.914) and ROC AUC (0.958). However, this superior performance comes at the expense of speed: LOF is the slowest, requiring 9.37s to execute.

In contrast, OCSVM is the fastest (1.48s) but delivers the weakest detection metrics (accuracy 0.969, precision 0.674, recall 0.756, F1-score 0.711, AUC 0.868), making it less reliable for high-stakes anomaly identification. Both RC and IF offer more balanced trade-offs: RC provides strong recall (0.900) and accuracy (0.989) with moderate precision (0.898) and AUC (0.941) in 4.30s, while IF yields respectable accuracy (0.983) and precision (0.834) in 4.73s but slightly lower recall (0.828) and AUC (0.910).

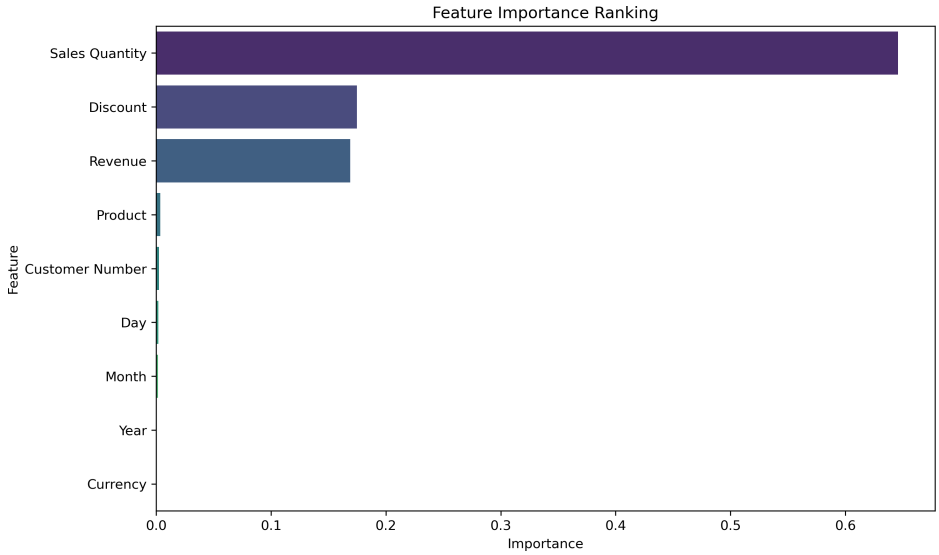


Figure 3: Feature importance from the complete feature set

After experimenting with the complete feature set on our anomaly-detection models, we utilized a RandomForestClassifier for feature-importance ranking to choose importance features. During the feature selection, three features were dropped including Order Number, Order Item, and Unit of Measure. As Figure 3 illustrates, Sales Quantity, Discount, and Revenue together contribute over 90% of the overall importance, whereas Product, Customer Number, Day, Month, Year, and Currency have only a little effect. Thus, we re-trained all our outlier detector models based on 9 features out of 12 features.

Table 2 demonstrates the superiority of the LOF even when using only significant features with 10-fold cross-validation. LOF achieves the highest accuracy (0.993) and outperforms all other models in precision (0.927), recall (0.932), F1-score (0.928), and ROC AUC (0.964). These gains translate into a more reliable detection of anomalies with fewer false positives and false negatives, making LOF the strongest choice for scenarios where detection quality is paramount.

Table 2: Comparison of different anomaly detection models evaluated on the significant features using 10-fold cross-validation

Model	Acc	Precision	Recall	F1-Score	ROC AUC	Execution Time
RC	0.987	0.868	0.876	0.871	0.934	10.66s
IF	0.986	0.869	0.864	0.865	0.928	4.11s
OCSVM	0.972	0.699	0.784	0.737	0.882	**1.34s**
LOF	**0.993**	**0.927**	**0.932**	**0.928**	**0.964**	7.45s

IF and RC both hover around 98.6–98.7% accuracy which is closer to LOF but fall behind LOF in other key metrics. IF delivers a slightly faster execution time (4.11s vs. LOF's 7.45s) yet its precision (0.869), recall (0.864) and ROC AUC (0.928) are notably lower, indicating less consistent anomaly coverage. RC, while yielding solid accuracy (0.987), requires the longest runtime (10.66s) and offers lower recall (0.876), precision (0.868) and AUC (0.934) compared to LOF. This makes it less attractive for both speed-critical and high-performance use cases. OCSVM exhibits the fastest inference (1.34s) but at the cost of significantly reduced detection quality (accuracy 0.972, F1-score 0.737, AUC 0.882). Its poor balance between precision (0.699) and recall (0.784) underlines why it is unsuitable for applications demanding both reliability and robustness.

LOF strikes the optimal balance and achieves peak anomaly-detection performance across all major metrics while maintaining acceptable latency. Overall, if detection quality is paramount and latency is less critical, LOF is recommended. However, the LOF method is particularly well-suited for mission-critical applications where minimizing undetected anomalies is a top priority, such as fraud detection and system monitoring, given its improved recall and processing efficiency. Therefore, it was chosen as the base method for this study due to its balanced performance metrics, especially in scenarios that need both scalability and accuracy. For time-sensitive scenarios where some performance can be sacrificed, RC or IF may be preferable, with OCSVM reserved only for cases demanding the fastest inference despite lower accuracy.

Figures 4 and 5 present ROC-curve comparisons for our four anomaly detectors, including RC, IF, OCSVM, and LOF; first on the complete full-feature set and then on the reduced significant-feature set. In Figure 4 (all features, 10-fold CV), LOF's ROC curve consistently lies above the others, achieving an AUC of 0.9576. This steep rise toward the top-left corner reflects LOF's excellent true-positive rate at very low false-positive rates, confirming its superior recall and precision trade-off (highest F1-score). RC follows closely with AUC = 0.941, indicating strong overall discrimination but slightly less sensitivity at low FPR. IF attains AUC = 0.9096, demonstrating good but not top-tier performance, while OCSVM trails behind (AUC = 0.8682), consistent with its lower precision and recall.

After pruning the less important features, Figure 5 shows that all models' ROC curves tighten and AUCs improve except for RC: LOF increases to 0.9640, IF to

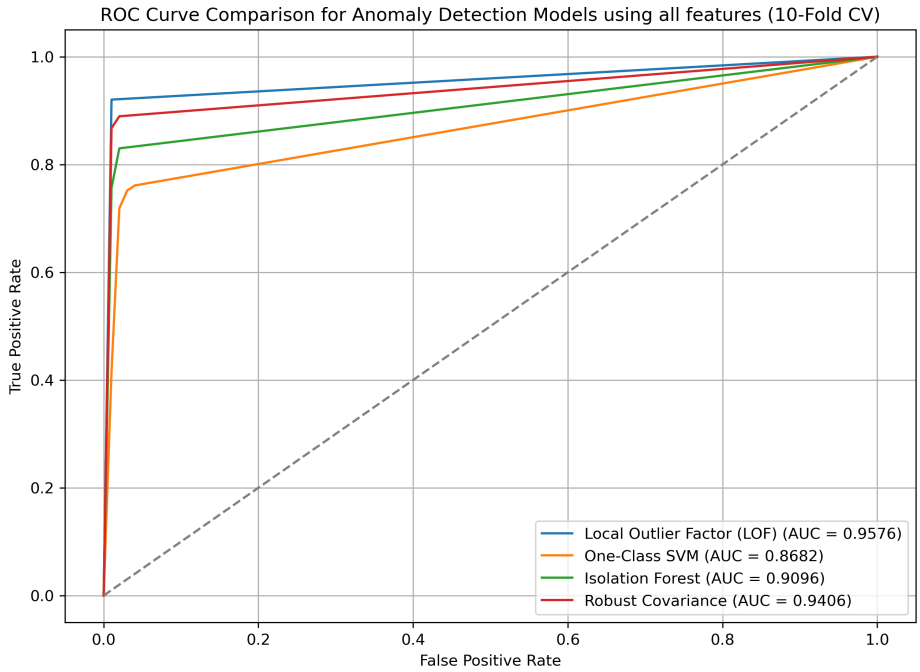


Figure 4: Comparison of ROC curves for different anomaly detection models with complete feature set

0.9284, and OCSVM to 0.8828 while RC decreases to 0.9346. Notably, LOF's curve becomes even sharper, underscoring that dimensionality reduction enhances its anomaly-separation power. IF benefits substantially as well, losing much of its gap to LOF while retaining fast inference. OCSVM also show modest gains, though OCSVM remains the weakest overall. However, RC shows a slight reduction in all performance metrics. These plots demonstrate that LOF is the top performer under both feature sets. Eliminating the three insignificant predictors further boosts all models' ability to distinguish anomalies from normal data except RC; especially improving LOF and IF in practical, real-time settings.

Figure 6 illustrates how a significant, relevant feature affects the model's functionality. Plotting the data demonstrates how changes in this characteristic correlate to changes in anomaly scores, hence improving the ability to distinguish between normal and anomaly cases. Notably, the graphic highlights the model's crucial role in reducing false negatives by showing that as the feature value rises, the model achieves increased detection accuracy and improved recall. Having all factors considered, the figure emphasizes how important the feature is for maximizing the anomaly detection procedure.

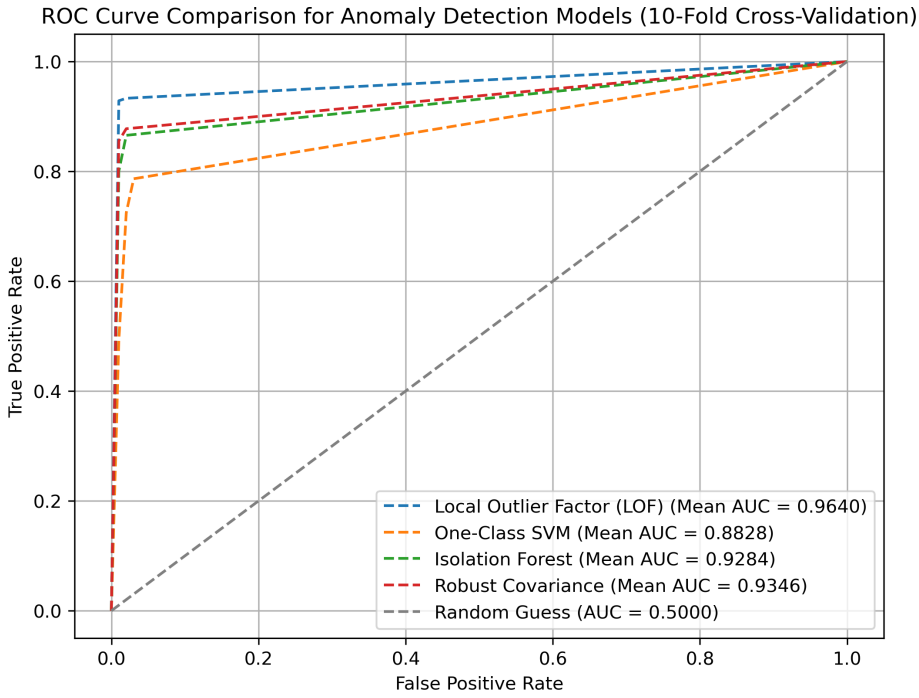


Figure 5: Comparison of ROC Curves for Different Anomaly Detection Models with significant features

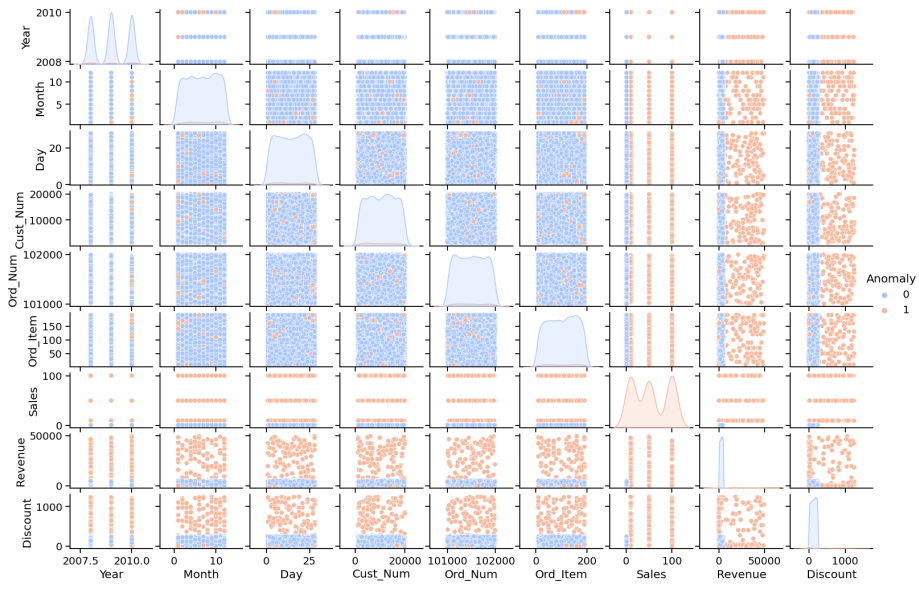


Figure 6: Visualizing sales patterns and anomalies

Figure 7 is a heatmap that shows a sales dataset's feature correlation matrix. It displays how strongly and in which direction linear correlations exist between two variables. The heatmap aids in uncovering potential correlations for feature engineering and further study. While execution time stays efficient, higher feature values are linked to increases in accuracy, recall, and the F1 score. This indicates that the function is essential for improving the model's detection power without compromising speed.

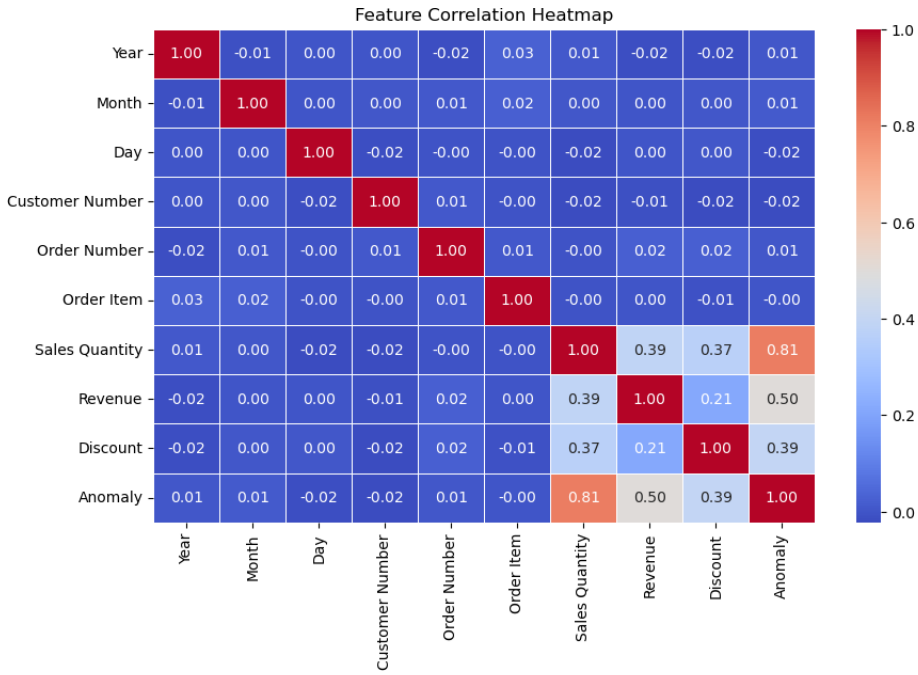


Figure 7: Sales feature correlation heatmap

4.2 API Development and Deployment

The FastAPI application encapsulates the predictive capabilities of the trained anomaly detection model. FastAPI facilitates effortless interaction with the anomaly detection model by providing automatic OpenAPI documentation and asynchronous support. A real-time API endpoint is established to expose the model's anomaly detection capabilities, accept input data, and return predictions.

To assure consistent deployment across varying environments, the FastAPI application is containerized using Docker. Azure Kubernetes Service is employed to deploy the containerized API on Azure, guaranteeing efficient and scalable administration. Security was guaranteed via HTTPS encryption and token-based authentication, while Azure's scalability accommodated high-throughput demand.

The LOF model's predictive capabilities were integrated into a custom API utilizing the FastAPI framework. The API enabled external systems, including the SAP HANA Fiori web application, to transmit real-time transaction data and obtain anomaly detection results in return. The API was deployed on Azure Cloud utilizing Docker containers, guaranteeing scalability and uniform performance across environments. FastAPI's automatic documentation and capacity to manage high-volume API requests rendered it an optimal framework for showcasing the machine learning model's functionalities. Additionally, the use of Azure's scalable cloud infrastructure ensured that the API could handle varying workloads without sacrificing performance. This was critical for ensuring that the system could meet real-time processing requirements in enterprise settings, providing a scalable and accessible platform for real-time anomaly detection.

4.3 SAP HANA Fiori Integration

The SAP HANA Fiori web application is seamlessly integrated with the API, which allows for the real-time detection of anomalies within the user interface. The web application is connected to the API by configuring a service destination in the SAP HANA Cloud Platform Cockpit. To facilitate communication between the web application and the API, an extension to the OData service is developed that specifies input parameters and response structures.

The web application was configured to interact with the anomaly detection (shown in Figure 8). This application consumes the OData service and loads data from the system. The app is configured to show the sample data and make the external API call. The system facilitated smooth communication between the SAP HANA database, which held the transactional data, and the deployed ML model through the API. The configuration involved setting up OData services, defining input parameters, and establishing routes for API calls within the Fiori interface. This integration enabled end-users to interact with the model predictions seamlessly. The SAP HANA Fiori web application was successfully integrated with the anomaly detection API, allowing real-time interaction and decision-making based on the model's predictions Figure 9.

A user-provided service is created on SAP HANA XS Advanced Cockpit and the custom API endpoint credentials are assigned. Afterwards, the OAfterwardcation is assigned to this user-provided service in *mta.yaml* file. The route of service to OData has been used in *outbound.controller.js*. This is defined in the *xs-app.json* file.

The proposed environment offers significant advantages by incorporating FastAPI for API development and scikit-learn's ML models to improve anomaly detection in the SAP HANA Fiori web application. This integrated system utilizes powerful algorithms, offering immediate insights for well-informed decision-making. Moreover, the adaptability of customized APIs effectively overcomes the constraints of SAP HANA's PAL, guaranteeing the streamlined identification of irregularities. This methodology could be applied to other enterprise systems requiring anomaly detection, such as network security or operational monitoring. Despite its bene-

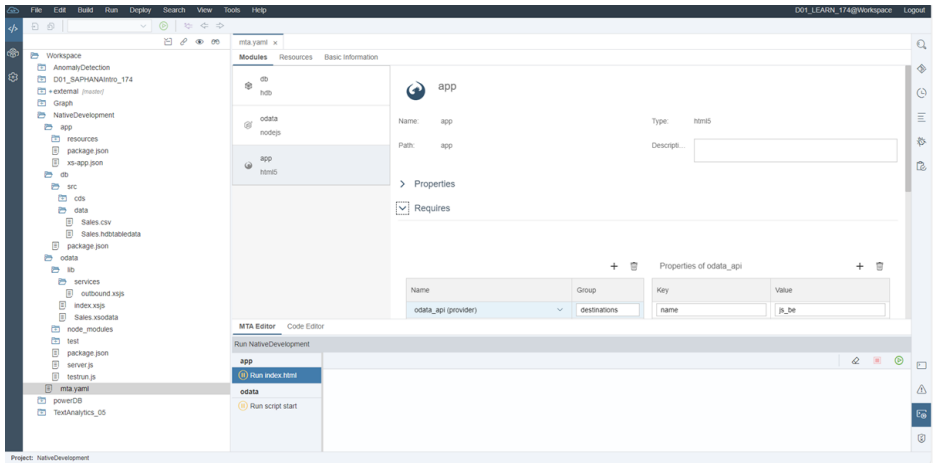


Figure 8: Overview of SAP HANA Fiori web configuration

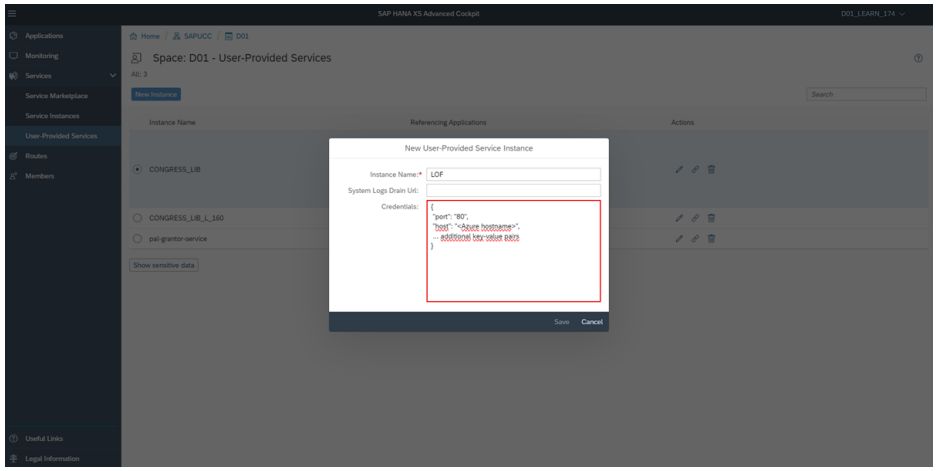


Figure 9: Creating a user-provided service on SAP HANA XS advanced cockpit and assigning the custom API endpoint credentials

fits, the challenges included ensuring seamless integration of the machine learning model, the API, and the SAP HANA Fiori web application. This incurs additional costs and effort for maintenance and the possibility of scaling issues as the volume of data grows. Future research could concentrate on optimizing the model for even larger datasets and automating updating the model as new transaction data becomes available.

5 Conclusions

In this study, an anomaly detection system inside the SAP HANA Fiori Web Tool was successfully implemented. The experimental results indicate that LOF surpasses other anomaly detection methods, including Isolation Forest and One-Class SVM, in terms of both accuracy and processing speed. The model utilized transactional sales data from SAP HANA, confirming its relevance to practical enterprise contexts. The constraints of SAP HANA's PAL were addressed by integrating a custom API-based solution powered by scikit-learn's LOF model, thus creating a more reliable anomaly detection method. Moreover, FastAPI allowed a high-performance API interface, boosting the LOF model's usability. Azure deployment guaranteed scalability and dependability by employing Docker and Kubernetes. This solution easily links scikit-learn, FastAPI, Azure, and SAP HANA Fiori to create a robust and unified predictive analytics system that tackles technical issues and improves functionality, adaptability, and real-time anomaly detection in business environments. Therefore, many technical challenges are addressed using this solution. This approach proved to be effective, but there are still many areas that need improvement. Maintaining the performance of the anomaly detection system as data quantities becomes a challenge. Future studies might look into using anomaly detection algorithms based on evaluating some lightweight neural network alternatives (e.g., shallow autoencoders or one-layer graph networks). Once we have validated that their computational overhead remains compatible with SAP's real-time requirements, which could offer even more accuracy and flexibility. This study particularly contributes to the field of enterprise AI integration by presenting a scalable and efficient solution for real-time anomaly detection in SAP HANA environments. The findings highlight the significance of integrating machine learning, cloud technologies, and API-driven architectures to improve enterprise analytics capabilities.

Data Availability Statement

The dataset utilized in this study was obtained from the SAP HANA platform. While it is not publicly available, access may be granted upon formal request through the appropriate official channels. The source code and supplementary materials can be accessed at the following URL: https://github.com/Georgina-asuah/SAPML.

References

[1] Agyemang, E. F. Anomaly detection using unsupervised machine learning algorithms: A simulation study. *Scientific African*, 26:e02386, 2024. DOI: 10.1016/j.sciaf.2024.e02386.

[2] Albtsoh, L. and Omar, M. Textguard: Identifying and neutralizing adversarial threats in textual data. *International Journal of Informatics, Information System and Computer Engineering (INJIISCOM)*, 6(2):212–224, 2025. URL: https://ojs.unikom.ac.id/index.php/injiiscom/article/view/15232.

[3] Amershi, S., Begel, A., Bird, C., DeLine, R., Gall, H., Kamar, E., Nagappan, N., Nushi, B., and Zimmermann, T. Software engineering for machine learning: A case study. In *Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice*, pages 291–300. IEEE, 2019. DOI: 10.1109/ICSE-SEIP.2019.00042.

[4] Baur, B. *Machine Learning mit SAP HANA*. Espresso Tutorials GmbH, 2022. URL: https://www.vitalsource.com/products/machine-learning-mit-sap-hana-benedict-baur-v9783960121688.

[5] Baylor, D. et al. Tfx: A TensorFlow-based production-scale machine learning platform. In *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 1387–1395, 2017. DOI: 10.1145/3097983.3098021.

[6] Birihanu, E. and Lendák, I. Optimal sensor data resampling for anomaly detection in industrial control systems. In *The International Conference on Recent Innovations in Computing*, pages 697–710. Springer, 2023. DOI: 10.1007/978-981-97-3442-9_49.

[7] Buitinck, L. et al. API design for machine learning software: Experiences from the scikit-learn project. arXiv preprint, 2013. DOI: 10.48550/arXiv.1309.0238.

[8] Chandola, V., Banerjee, A., and Kumar, V. Anomaly detection: A survey. *ACM Computing Surveys*, 41(3):1–58, 2009. DOI: 10.1145/1541880.1541882.

[9] Easin, A. M. and Orosz, T. Enhancing SAP ecosystem: Harmonizing open-source technologies for integration and innovation. In *Proceedings of the 14th Conference of PhD Students in Computer Science*, page 7, 2024. URL: https://www.inf.u-szeged.hu/~cscs/pdf/cscs2024.pdf.

[10] Easin Arafat, M., Asuah, G., Saha, S., and Orosz, T. Empowering real-time insights through LLM, LangChain, and SAP HANA integration. In *Proceedings of the International Conference on Recent Innovations in Computing*, pages 483–495. Springer, 2023. DOI: 10.1007/978-981-97-3442-9_33.

[11] García, Á. L. et al. A cloud-based framework for machine learning workloads and applications. *IEEE Access*, 8:18681–18692, 2020. DOI: 10.1109/ACCESS.2020.2964386.

[12] Gole, V. and Shiralkar, S. *Empower decision makers with SAP analytics cloud*. Springer, 2020. DOI: 10.1007/978-1-4842-6097-5.

[13] Jin, J. Anomaly detection and exploratory causal analysis for SAP HANA. Master's thesis, Karlsruher Institut für Technologie, 2019. DOI: 10.5445/IR/1000089289.

[14] Khasawneh, M. A., Daraghmeh, M., Awasthi, A., and Agarwal, A. Multilevel learning for enhanced traffic congestion prediction using anomaly detection and ensemble learning. *Cluster Computing*, 28(3):160, 2025. DOI: 10.1007/s10586-024-04871-z.

[15] Kohli, M. Using machine learning algorithms on data residing in SAP ERP application to predict equipment failures. *International Journal of Engineering & Technology*, 7(2.28):312–319, 2017. DOI: 10.14419/ijet.v7i2.28.12952.

[16] Lavin, A. et al. Technology readiness levels for machine learning systems. *Nature Communications*, 13(1):6039, 2022. DOI: 10.1038/s41467-022-33128-9.

[17] Liu, F. T., Ting, K. M., and Zhou, Z.-H. Isolation forest. In *Proceedings of the 2008 Eighth IEEE International Conference on Data Mining*, pages 413–422. IEEE, 2008. DOI: 10.1109/ICDM.2008.17.

[18] Mahmud, J. S., Birihanu, E., and Lendak, I. A semi-supervised framework for anomaly detection and data labeling for industrial control systems. In *Proceedings of the Conference on Information Technology and its Applications*, Volume 872 of *Lecture Notes in Networks and Systems*, pages 149–160. Springer, 2023. DOI: 10.1007/978-3-031-50755-7_15.

[19] Mahmud, J. S. and Lendak, I. Enhancing one-class anomaly detection in unlabeled datasets through unsupervised data refinement. In *Proceedings of the 2024 IEEE 22nd Jubilee International Symposium on Intelligent Systems and Informatics*, pages 000497–000502. IEEE, 2024. DOI: 10.1109/SISY62279.2024.10737577.

[20] Mogîldea, A. SAP HANA – SAP high-performance analytic appliance. Tehnica UTM, 2016. URL: http://repository.utm.md/handle/5014/808.

[21] Nandigramwar, H., Mittal, A., Bhatnagar, A., and Rashid, M. A distributed and unified API service for machine learning models. In *Proceedings of the 2021 2nd International Conference on Intelligent Engineering and Management*, pages 480–485. IEEE, 2021. DOI: 10.1109/ICIEM51511.2021.9445348.

[22] Oliveira, J. P. and Sousa, R. D. Unsupervised anomaly detection of retail stores using predictive analysis library on SAP HANA XS Advanced. *Procedia Computer Science*, 181:882–889, 2021. DOI: 10.1016/j.procs.2021.01.243.

[23] Peksa, J. Autonomous data-driven integration into ERP systems. In *Design, Simulation, Manufacturing: The Innovation Exchange*, pages 223–232. Springer, 2021. DOI: 10.1007/978-3-030-77719-7_23.

[24] Reimann, L. and Kniesel-Wünsche, G. Improving the learnability of machine learning APIs by semi-automated API wrapping. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, pages 46–50, 2022. DOI: 10.1145/3510455.3512789.

[25] Sakurada, M. and Yairi, T. Anomaly detection using autoencoders with nonlinear dimensionality reduction. In *Proceedings of the 2014 2nd Workshop on Machine Learning for Sensory Data Analysis*, pages 4–11, 2014. DOI: 10.1145/2689746.2689747.

[26] SAP HANA Machine Learning Overview, 2021. URL: https://help.sap.com/doc/ae101ea615324a41addb8b9552805f20/2.0.05/en-US/SAP_HANA_Machine_Learning_Overview_Guide_en.pdf.

[27] Sarferaz, S. *Embedding Artificial Intelligence into ERP Software*. Springer, 2024. DOI: 10.1007/978-3-031-54249-7.

[28] Schölkopf, B., Platt, J. C., Shawe-Taylor, J., Smola, A. J., and Williamson, R. C. Estimating the support of a high-dimensional distribution. *Neural Computation*, 13(7):1443–1471, 2001. DOI: 10.1162/089976601750264965.

[29] Settibathini, V. S. Enhancing user experience in SAP Fiori for finance: A usability and efficiency study. *International Journal of Machine Learning for Sustainable Development*, 5(3):1–13, 2023. URL: https://ijsdcs.com/index.php/IJMLSD/article/view/467.

[30] Shaik, M. and Siddque, K. Q. Predictive analytics in supply chain management using SAP and AI. *Journal of Computer Sciences and Applications*, 11(1):1–6, 2023. DOI: 10.12691/jcsa-11-1-1.

[31] Shi, Z. and Wang, G. Integration of big-data ERP and business analytics (BA). *The Journal of High Technology Management Research*, 29(2):141–150, 2018. DOI: 10.1016/j.hitech.2018.09.004.

[32] Sivakumar, V., Prasad, M. R., Vadivel, M., Prasad, S. T., Aranganathan, A., and Murugan, S. Isolation forests integration for proactive anomaly detection in Augmented Reality-enhanced Tele-ICU systems. In *Proceedings of the 2024 6th International Conference on Energy, Power and Environment*, pages 1–6. IEEE, 2024. DOI: 10.1109/ICEPE63236.2024.10668946.

[33] Smith, D., Khorsandroo, S., and Roy, K. Machine learning algorithms and frameworks in ransomware detection. *IEEE Access*, 10:117597–117610, 2022. DOI: 10.1109/ACCESS.2022.3218779.

[34] Subramanian, H. and Raj, P. *Hands-On RESTful API Design Patterns and Best Practices: Design, develop, and deploy highly adaptable, scalable, and secure RESTful web APIs*. Packt Publishing Ltd, 2019. ISBN: 9781788998581.

[35] Yang, K. *Quality in the Era of Industry 4.0: Integrating Tradition and Innovation in the Age of Data and AI.* John Wiley & Sons, 2024. ISBN: 9781119932468.

Acta Cybernetica **27** (2025) 141–153.

Selecting Execution Path for Replaying Errors*

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

Abstract

The identification of the sources of a runtime error is a common task for Erlang developers. Dynamic and static tools can assist in this task. Our work aims to help Erlang developers in debugging processes to reproduce a runtime error. We would like to use and extend the static analyzer framework of RefactorErl with new algorithms to support this fault localization process. In our previous paper, we presented a symbolic execution-based analysis method to find the source of runtime errors. This paper extends that work with path selection heuristics to improve the efficiency of the algorithm in the RefactorErl framework.

Keywords: static analysis, Erlang, symbolic execution, fault localization, path selection

1 Introduction

Debugging Erlang programs, particularly in large-scale, distributed systems, presents significant challenges due to the complexity of tracing and reproducing runtime errors. Although bugs in the software are usually discovered due to faulty behaviour (e.g. a runtime error occurs), finding the origin of the fault is a non-trivial task. Traditional debugging methods often require developers to manually trace through code, which can be time-consuming and error-prone, especially when dealing with complex control flows and multiple execution paths. Traditional static analysis tools, while helpful, often lack the precision to pinpoint the exact source of a runtime error. Program analysis techniques with symbolic execution can help to solve this task.

In a concrete execution, a program is evaluated on a specific input, and a single control-flow path is explored. Symbolic execution [1, 9] uses unknown symbolic variables in evaluation, allowing to simultaneously explore multiple paths that a

*Project no. TKP2021-NVA-29 has been implemented with the support provided by the Ministry of Culture and Innovation of Hungary from the National Research, Development and Innovation Fund, financed under the TKP2021-NVA funding scheme.

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

DOI: 10.14232/actacyb.312432

program could take under different inputs. The use of symbolic execution can help us in fault localization.

We have previously implemented our prototype algorithm using backtracking and demonstrated how it finds an execution path to a given expression containing an error [7]. The algorithm uses a combination of the control-flow graph and the RefactorErl[1] [13] frameworks graph representation of the analyzed code to determine an appropriate execution path that may lead to a given runtime error in Erlang software.

Because of the path-explosion problem, it is infeasible for symbolic execution tools to explore all execution paths of any nontrivial programs. Therefore, search heuristics are required elements of symbolic execution. Using a good search heuristic can maximize code coverage and improve the effectiveness of the analysis in practice.

In this paper, we examine several path selection heuristics that can be used to improve the efficiency of our algorithm and make our method feasible for error detection on larger software bases. In Section 2 we introduce the Erlang language and the RefactorErl tool. Section 3 discusses related work in the field of symbolic execution and fault localization, highlighting the contributions of our approach. Section 4 introduces various path selection heuristics that can be employed to improve the efficiency of symbolic execution algorithms. Section 5 provides an overview of our proposed algorithm and Sections 6 and 7 we present our solution for managing the problem of path explosion. Section 7 contains a short evaluation on an example found in an open source project.

2 Background

Erlang [4] is a versatile, dynamically typed, concurrent functional programming language that allows developers to build highly scalable, soft real-time systems. Initially developed for telecommunication software, Erlang has since found applications in banking, chat services, and database management systems. Its robustness and fault-tolerant nature make it an excellent choice for large-scale distributed system development. Erlang programs run on a virtual machine (Erlang VM or node), which ensures platform independence. The standard library, known as OTP (Open Telecom Platform), along with the Erlang runtime environment, is collectively referred to as Erlang/OTP.

Erlang's module system enables programs to be broken down into smaller units called modules. Each Erlang program consists of multiple modules, each stored in a file with the extension `.erl`. A module starts with a declaration at the beginning, followed by an export declaration that lists functions intended for external use. This is then followed by the function definitions within the module.

RefactorErl [2, 13] is a static analysis and transformation tool for Erlang, developed at Eötvös Loránd University. It employs static code analysis techniques and offers a wide range of features, including data flow analysis, detection of dynamic

[1] Refactorerl's website. https://plc.inf.elte.hu/erlang/

function calls, side-effect analysis, and a user level query language for querying semantic information or structural complexity metrics of Erlang programs. Other functionalities include examining dependencies among functions or modules and generating function call graphs that include information on dynamic calls. RefactorErl provides multiple user interfaces, such as a web-based interface, an interactive console, and plugins for Emacs or Vim.

During the initial analysis, RefactorErl constructs an abstract syntax tree from the source code and enhances it with additional semantic information to form a Semantic Program Graph (SPG) [8]. After analyzing the source code, this graph is stored in a database. The tool is capable of transforming the graph back into source code at any time. The process of refactoring essentially consists of graph transformation steps, using the SPG to collect the necessary information for the transformation.

3 Related work

Symbolic execution [1, 9] is a technique used by many program analysis and transformation techniques, such as partial evaluation, test-case generation or model checking. It can be used for fault detection by exploring different execution paths of a program with symbolic values instead of concrete values. Symbolic values represent a range of possible values that can satisfy certain constraints. Tools based on such techniques can find errors that are hard to detect with conventional testing methods, such as buffer overflows, division by zero errors, etc.

Symbolic execution maintains a symbolic state and a path condition for each execution path. The symbolic state contains the symbolic values of variables. The path condition contains the constraints on the symbolic values that are derived from branch conditions along the path. Symbolic execution uses a constraint solver to check the feasibility of each path and to generate concrete inputs that can trigger faults.

KLEE [3] uses two main search strategies: Random Path Selection and State-Based Search. Random Path Selection maintains a binary tree recording the program path followed for all active states, where the internal nodes are the ones where the execution has forked and the leaves represent the current states. The states are selected by traversing this tree from the root and randomly selecting the path to follow at the branch points. During the symbolic execution when an internal node is reached, all child nodes of the given node have an equal probability to be selected by the algorithm regardless of the size of the subtrees. The biggest advantage of this strategy is that it avoids starvation occurring in loops containing symbolic conditions and resulting in quick new state creation.

While symbolic execution is not a new topic in the Erlang ecosystem, previously published papers mostly focus on formal [14, 15] and informal [6] definitions with the aim of program verification. In a previous paper [7], we present a symbolic execution technique for Erlang that can support debugging processes of Erlang developers through the RefactorErl framework. Our goal was not to verify Erlang programs

but to support their debugging processes through the RefactorErl framework.

In Erlang programming, fault localization is a critical yet complex task, especially in large-scale software systems. Finding the source of runtime errors can be time-consuming and costly, necessitating the use of automatic methods to assist developers. Traditional debugging involves reproducing faulty executions, but this becomes challenging when dealing with multiple paths a program might take under various inputs. This work addresses the problem of static fault localization in Erlang programs using a targeted approach to explore the control-flow graph of the software.

Our previously proposed method builds on the RefactorErl static analysis framework to identify execution paths that lead to specific runtime errors. The aim is to reproduce faulty behavior by selecting an appropriate execution path in the program's control-flow graph that may lead to the identified error. The approach employs symbolic execution, where unknown symbolic variables are used to explore multiple paths within the program. The analysis targets a specific line or expression in the program, referred to as the "error path," which potentially leads to runtime errors.

The approach begins with symbolic execution to find a realizable path in the program from the entry point to the specified error line. As the program is explored, a set of conditions is gathered, which are then analyzed using an SMT (Satisfiability Modulo Theories) solver, Z3 [5], to verify the feasibility of the identified path. By collecting symbolic constraints along the paths in the control-flow graph, the method can identify the conditions and input values that lead to the runtime error, aiding developers in reproducing and understanding the fault.

The algorithm operates in a two-step manner: First, it traverses the program's control-flow graph in a breadth-first manner to find a potential path to the target error expression. During this traversal, conditions from branch statements, such as if expressions, are gathered to build a set of constraints. Variables are tracked using a map data structure to ensure each unique instance is accounted for in the constraints. If the conditions derived from the selected path are unsatisfiable, the algorithm backtracks to find an alternative route. The second step involves repeating this process recursively through function calls, allowing the exploration of execution paths that span multiple functions within the program.

This targeted symbolic execution is integrated with the RefactorErl tools extensive code analysis capabilities. RefactorErl constructs a Semantic Program Graph (SPG) that contains lexical, syntactic, and semantic information about the source code, as well as control-flow and control dependence information. The SPG aids in tracking variables and expressions as the program is analyzed. By employing static backward symbolic execution, the method identifies relevant input parameters and conditions, supporting fault localization in a static analysis context. The constraints gathered are then passed to the Z3 solver to check for satisfiability and determine potential inputs that would result in the observed faulty behavior.

Unlike other tools like CutEr[2] [11], which uses concolic testing [12] based on dy-

[2] https://github.com/cuter-testing/cuter

namic symbolic execution, this method is fully static, working with the control-flow graph to analyze execution paths backward from the error point. It distinguishes itself by focusing on debugging support rather than program verification, providing a practical tool for developers dealing with runtime errors in Erlang programs.

4 Path selection algorithms

Path selection heuristics in symbolic execution algorithms are crucial for efficiently exploring the numerous execution paths in a program. These heuristics aim to guide the symbolic execution engine to explore the most promising paths, helping to detect bugs or vulnerabilities while minimizing computational resources.

One common strategy, employed by tools like KLEE [3], is random path selection. This heuristic builds a binary tree of the program paths being explored. Each node in this tree represents a decision point (a branch), and the leaves represent the active execution paths. KLEE traverses this tree randomly, selecting branches in a way that ensures each path has an equal probability of being chosen, regardless of the number of processes under it. This approach helps KLEE in two significant ways, it prioritizes paths that are higher in the tree, which are less constrained and therefore more likely to lead to new parts of the code, and it prevents KLEE from being trapped in regions where new branches are generated rapidly, which could lead to "fork bombing" or an excessive creation of paths. This method is effective in providing broad coverage of the execution space, which increases the likelihood of uncovering unexpected bugs. However, random path selection can be inefficient when a goal is to discover a specific path in the program graph, as it often leads to exploring irrelevant paths and may fail to prioritize paths that are more likely to lead to the target expression.

A more targeted heuristic is concolic execution [12], where symbolic execution is combined with concrete execution to guide path exploration. Concolic testing uses concrete inputs to steer symbolic execution towards different branches, ensuring that the tool avoids paths that have already been explored with similar concrete inputs. One of the advantages of this method is that it can avoid the path explosion problem seen in pure symbolic execution by using concrete inputs to prune the space of possible paths. This heuristic is effective for finding bugs that are triggered by specific input patterns.

Error-guided path selection is another approach, which focuses on paths that are likely to lead to known or suspected errors. This heuristic can be highly efficient when the goal is to locate a particular fault or error condition within a program. By directing the exploration towards paths where errors are most likely to occur, it reduces the number of irrelevant paths examined, leading to faster bug detection.

5 Overview of the algorithm

The algorithm uses a kind of symbolic backward execution called call-chain backward symbolic execution [10]. This is a type of symbolic execution that mixes

forward and backward symbolic execution. Inside each function, it explores the execution paths forward but it follows the call-chain backwards from the target point to the program's entry point. Starting at the target expression, we search for a path from the entry point of the function containing the target expression itself. This intraprocedural part of the algorithm uses the control-flow graph of the function to look for possible paths to the target node.

Once a valid intraprocedural path is found, the next step is to determine the callers of the function. Using RefactorErl we can collect all expressions that contain such a function call. Now the expression containing the function call will be our target, and the new starting point will be the new function containing that expression.

We can see that our algorithm has two points when path selection is needed, once in the intraprocedural part and once in the interprocedural part. Using different strategies would make sense in each of these cases.

6 Intraprocedural strategy

The intraprocedural part of our fault localization algorithm focuses on analyzing execution paths within a single function or procedure to find paths that lead to an error. At this stage, the algorithm works by exploring the control-flow graph (CFG) of the function to examine all possible execution paths that may reach a specific target expression, such as a line of code responsible for a runtime error. The intraprocedural analysis builds upon symbolic execution, where variables are treated as symbolic values, and conditions at branching points (such as if expressions, variable assignments or pattern matching) generate constraints that must be satisfied for a path to be feasible. These constraints are gathered as the algorithm traverses each possible path within the function.

The algorithm starts from the function's entry point and follows each control-flow path, collecting symbolic constraints and tracking the flow of execution until it either reaches the target expression (such as an error). Starting at the root of the control-flow graph of the selected function, we explore as far as possible along each branch before backtracking. If a path to the target node is found, we check the conditions along the path with the help of a constraint solver for feasibility. Depending on the result we either return the path or reject it and continue the backtracking to find another one.

Even though there are no loops at the intraprocedural level, making path explosion less significant than at the interprocedural level, exploring all paths within a function can still be computationally expensive if the function contains multiple branching points. This challenge is compounded by the fact that, if a contradiction arises in the set of conditions during interprocedural exploration, a new intraprocedural path must be identified. This new iteration may involve revisiting previously explored branches with updated constraints or exploring alternative paths that were not considered in the previous iteration. The algorithm continues to iterate until a feasible path to the target expression is identified or all possible paths have

been exhausted. This approach ensures that the algorithm thoroughly explores the function's control-flow graph, increasing the likelihood of finding a valid path to the error.

To make our algorithm more efficient, we can use estimations in the intraprocedural part based on the depth of the target expression within the function's semantic program graph to reduce the problem space. The SPG is a representation that contains not only the syntactic structure of the function but also semantic information about variables, expressions, and dependencies between them. By determining how deeply nested the target expression is within the SPG, we can establish a depth limit for path exploration. We can use this metric to reduce the size of the tree by removing sections of the tree that are deeper than our target.

This depth-based heuristic improves the algorithm's efficiency by restricting the exploration to paths that are likely to lead to the error without examining irrelevant branches or deeply nested conditions that cannot feasibly reach the target. For example, if the target expression is located within a nested conditional block, the algorithm sets a maximum depth for the search, focusing only on paths that descend to the same level of depth as the target expression in the SPG. Paths that exceed this depth are deprioritized or discarded from the exploration process, as they cannot feasibly reach the target within the given structure.

Consider the simple example in Figure 1. This code snippet contains divisions, and if the denominator `C` is zero, a division by zero error occurs. Suppose that the error occurred in line 12. We can use the algorithm to find a realizable path to the target expression from the entry point of the program, and also determine a set of input values that may trigger the error. We need to traverse the control-flow graph to find the target expression, but to enumerate all paths might be very expensive in larger functions. To reduce our searchspace we can cut branches that are deeper in the tree then our target expression. The tree next to the code snippet shows the path the algorithm traverses on the simple example function.

7 Interprocedural strategy

We propose a new heuristic for the interprocedural phase of our algorithm, which leverages the stack trace to optimize the search for execution paths leading to runtime errors. By using the stack trace to trace the chain of function calls leading to an error, we can significantly reduce the search space within the control-flow graph (CFG), improving both the efficiency and precision of the algorithm.

The stack trace is a valuable tool for identifying the chain of functions involved in an error. Traditionally, developers use stack traces to pinpoint the function where the error occurred, but this information alone often lacks the accuracy necessary to determine the specific path through the program leading to the fault. Since the algorithm is designed to precisely locate the source of a known runtime error, the stack trace can be provided as part of the initial problem setup. Erlang's stack trace is a structured and informative data format that provides a detailed account of the sequence of function calls leading up to a runtime error. When an exception

```
1  -module(example1).
2  -export([foo/2]).
3  foo(A, B) ->
4    C = A - B,
5    if
6      C == A -> 0;
7      C < A ->
8        if
9          B > C -> 1;
10         true -> 2
11       end
12     C > A -> A / C
13   end.
```

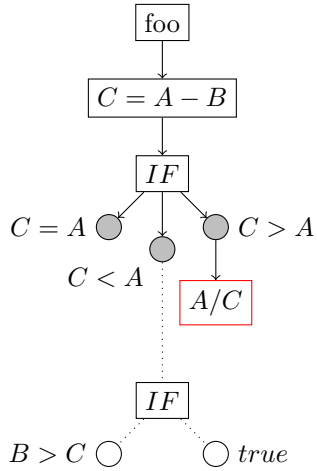


Figure 1: Example module and corresponding path selection

occurs, Erlang generates a stack trace that includes the module name, function name, arity, and the line number where the error was encountered. This trace also captures the hierarchical chain of function calls, showing how execution flowed from one function to another until the error was triggered. For example, a typical stack trace might look like this:

```
[{module_name, function_name, [arguments], [{file, line_number}]}, ...],
```

where each tuple represents a function call in the call stack.

While the stack trace provides a high-level view of the call-chain, it does not provide the precise sequence of conditions and decisions that led to the error. Our heuristic takes advantage of the stack trace by using it as a guide to narrow down the relevant sections of the CFG that need to be explored, avoiding unnecessary traversal of unrelated branches. This targeted approach enhances the algorithm's ability to find a concrete execution path from the program entry point to the error.

The integration of this heuristic into the algorithm is straightforward. Once an error occurs and the stack trace is available, the algorithm uses it to follow the function call sequence in reverse, starting from the point where the error occurred. For each function in the stack trace, the algorithm identifies the relevant control-flow path by focusing only on the functions listed in the trace and using the intraprocedural algorithm to generate the necessary conditions within the function. As the stack trace provides a natural ordering of function calls, the search is restricted to a narrower subset of the program, reducing the number of potential execution paths to explore. This is particularly effective in large codebases where the number of possible paths can be overwhelming.

In example in Figure 2, when the function `f(A)` is called with a negative number, a runtime error occurs in the arithmetic expression within `f2(A)`, specifically a division by zero, as indicated by the stack trace. The interprocedural part of our

```
1  -module(multi_fun_example).
2  -export([f/1,l/1,r/1,f2/1]).
3
4  f(A) ->
5      if
6          A >= 0 ->
7              l(A+1);
8          true ->
9              r(A)
10     end.
11 l(A) ->
12     {ok, f2(A)}.
13
14 r(0) ->
15     {ok, f2(0)};
16 r(A) ->
17     r(A+1).
18
19 f2(A) ->
20     1/A.
```

Figure 2: Interprocedural example

algorithm uses this stack trace to trace the chain of function calls leading to the error. Starting with the function `f2(A)`, the algorithm traces back to the caller, `r(A)`, which recursively calls itself until A becomes zero, triggering the call to `f2(0)`. The algorithm then follows the call-chain back to `f(A)`, which, when `A` is negative, directs execution to `r(A)`. By reconstructing this path from `f2/1` through `r/1` and `f/1`, our algorithm gathers the conditions along the way, such as the fact that `A` is initially negative and `r(A)` will recursively increment it until it reaches zero. These symbolic constraints are then used to generate inputs, confirming that any negative value of A leads to the runtime error in `f2/1`. This targeted path exploration, guided by the stack trace, allows our algorithm to efficiently pinpoint the error and the specific input conditions that cause it.

8 Short evaluation

In this section, we evaluate the interprocedural part of our fault localization algorithm using an example from an open-source Erlang codebase shown in Figure 3[3]. The selected code snippet handles arithmetic expression parsing in a simple interpreter and was chosen because it exhibits a runtime error when processing an incorrect expression that cannot be parsed shown in Figure 4. This error shows the evaluation of how our algorithm traces function calls and identifies the root cause of errors across multiple functions.

The code consists of functions for parsing and evaluating expressions (`parse/1`, `parser/1`, `expression/1`, `bin/1`), with the error manifesting in the `expression/1` function due to an invalid argument passed through the function chain. Specifically, the stack trace generated by the program shows the sequence of calls illustrated in Figure 5.

The error occurs because `expression/1` expects a valid token sequence to parse, but it receives an invalid list starting with the operator `'+'` that leads to a pattern matching failure. This is a typical scenario, where we must trace the error through

[3] https://github.com/pichi/epexercises

```
1  parse(L) -> parser(lexer(L)).
2
3  parser(L) when is_list(L) ->
4    {T, []} = expression(L),
5    T.
6
7  expression(['let',{id,I},'='|T]) ->
8    {V, ['in'|R1]} = expression(T),
9    {E, R2} = expression(R1),
10   {{'let', I, V, E}, R2};
11 expression(['if'|T]) ->
12   {C, ['then'|R1]} = expression(T),
13   {X, ['else'|R2]} = expression(R1),
14   {Y, R3} = expression(R2),
15   {{'if', C, X, Y}, R3};
16 expression(['~'|T]) -> {X, R} = expression(T), {{'~', X}, R};
17 expression(['('|T]) -> {X, [')'|R]} = bin(T), {X, R};
18 expression([{id, _}=X|T]) -> {X, T};
19 expression([{num, _}=X|T]) -> {X, T}.
20
21 bin(L) -> {X, [Op|T]} = expression(L),
22   true = lists:member(Op, ['+','-','*', '/']),
23   {Y, R} = expression(T),
24   {{Op, X, Y}, R}.
```

Figure 3: Evaluation

```
1 ** exception error: no function clause matching
2                 e38:expression(['+',{num,1},')']) (e38.erl, line 11)
3      in function  e38:bin/1 (e38.erl, line 27)
4      in call from e38:expression/1 (e38.erl, line 21)
5      in call from e38:parser/1 (e38.erl, line 8)
```

Figure 4: Runtime error evaluating an expression

multiple function calls, following the chain of execution from the initial function call in `parse/1` to the final error point in `expression/1`.

The interprocedural algorithm effectively traces the error through multiple function calls by following the stack trace. While the depth-limiting heuristic is not utilized in this example due to the lack of branching in the code, the stack trace guided approach proves highly efficient in following the exact path to the error without unnecessary exploration. Unlike random path selection or other heuristics that might explore the control-flow graph exhaustively, our method provides a direct and efficient route to identifying the path to the error and also generates possible input values that can lead to the fault.

```
{'EXIT',{function_clause,[{e38,expression,
           [['+',{num,1},')']],
           [{file,"e38.erl"},{line,11}]},
      {e38,bin,1,[{file,"e38.erl"},{line,25}]},
      {e38,expression,1,[{file,"e38.erl"},{line,21}]},
      {e38,parser,1,[{file,"e38.erl"},{line,8}]},
      {erl_eval,do_apply,7,[{file,"erl_eval.erl"},{line,904}]},
      {erl_eval,expr,6,[{file,"erl_eval.erl"},{line,636}]},
      {shell,exprs,7,[{file,"shell.erl"},{line,893}]},
      {shell,eval_exprs,7,[{file,"shell.erl"},{line,849}]}]}}
```

Figure 5: Stack trace of the example code

9 Conclusion

Our proposed method builds upon the RefactorErl framework, a static code analysis tool designed for analyzing and refactoring existing Erlang codebases. Our prototype algorithm utilizes call-chain backward symbolic execution, a combination of forward and backward symbolic exploration. Within each function, it analyzes execution paths forward, while tracing the call-chain backwards from the target point to the program's entry point. Starting at the target expression, the algorithm seeks a path from the entry point of the function containing that expression. The intraprocedural phase uses the function's control-flow graph to identify potential paths to the target node.

Given the branching structure of the program graph, checking every possible path would not be feasible. To make our prototype algorithm more efficient we use various path selection strategies. We use backtracking within the functions, supplemented with improvements that take advantage of the information that can be extracted from the graph of RefactorErl, reducing the size of the graph to be traversed. In the case of the interprocedural part, we use random path selection to prevent starvation when some part of the program rapidly creates new states. Combining these strategies we can effectively identify execution paths that might lead to runtime errors.

References

[1] Baldoni, R., Coppa, E., D'elia, D. C., Demetrescu, C., and Finocchi, I. A survey of symbolic execution techniques. *ACM Computing Surveys*, 51(3), 2018. DOI: 10.1145/3182657.

[2] Bozó, I., Horpácsi, D., Horváth, Z., Kitlei, R., Köszegi, J., M., T., and Tóth, M. RefactorErl — Source code analysis and refactoring in Erlang. In *Proceedings of the 12th Symposium on Programming Languages and Software Tools*, pages 138–148, Tallin, Estonia, 2011. ISBN: 978-9949-23-178-2.

[3] Cadar, C., Dunbar, D., and Engler, D. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, USA, 2008. USENIX Association. URL: https://dl.acm.org/doi/10.5555/1855741.1855756.

[4] Cesarini, F. and Thompson, S. *Erlang programming*. O'Reilly, 2009. ISBN: 978-0-596-51818-9.

[5] de Moura, L. and Bjørner, N. Z3: An efficient SMT solver. In Ramakrishnan, C. R. and Rehof, J., editors, *Tools and Algorithms for the Construction and Analysis of Systems*, Volume 4963 of *Lecture Notes in Computer Science*, pages 337–340, Berlin, Heidelberg, 2008. Springer. DOI: 10.1007/978-3-540-78800-3_24.

[6] Earle, C. B. Symbolic program execution using the Erlang verification tool. In Alpuente, M., editor, *9th International Workshop on Functional and Logic Programming*, pages 42–55, 2000. URL: http://elp.webs.upv.es/workshops/wflp2000/WFLP2000Proceedings.zip.

[7] Erdei, Z., Tóth, M., and Bozó, I. Supporting the debugging of erlang programs by symbolic execution. *Acta Universitatis Sapientiae, Informatica*, 16:44–61, 2024. DOI: 10.47745/ausi-2024-0004.

[8] Horváth, Z., Lövei, L., Kozsik, T., Kitlei, R., Víg, A. N., 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(2009) Special Issue of *Studia Universitatis Babeş-Bolyai, Series Informatica*, pages 7–16. Cluj-Napoca, Romania, 2009. URL: https://www.cs.ubbcluj.ro/~studia-i/contents/2009-kept/Studia-2009-Kept-1-KCL.pdf.

[9] King, J. C. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976. DOI: 10.1145/360248.360252.

[10] Ma, K.-K., Yit Phang, K., Foster, J. S., and Hicks, M. Directed symbolic execution. In Yahav, E., editor, *Static Analysis*, Volume 6887 of *Lecture Notes in Computer Science*, pages 95–111. Springer, Berlin, Heidelberg, 2011. DOI: 10.1007/978-3-642-23702-7_11.

[11] Sagonas, K. A CutEr tool. Talk at Erlang Factory, 2016. URL: http://www.erlang-factory.com/static/upload/media/1457739488660923kostissagonasacutertool.pdf. Accessed: Feb, 2023.

[12] Sen, K., Marinov, D., and Agha, G. CUTE: A concolic unit testing engine for C. *ACM SIGSOFT Software Engineering Notes*, 30(5):263–272, 2005. DOI: 10.1145/1095430.1081750.

[13] Tóth, M. and Bozó, I. Static analysis of complex software systems implemented in erlang. In Zsók, V., Horváth, Z., and Plasmeijer, R., editors, *Central European Functional Programming School: 4th Summer School, Revised Selected Papers*, Volume 7241 of *Lecture Notes in Computer Science*, pages 440–498. Springer, Berlin, Heidelberg, 2012. DOI: 10.1007/978-3-642-32096-5_9.

[14] Vidal, G. Towards Erlang verification by term rewriting. In Gupta, G. and Peña, R., editors, *Logic-Based Program Synthesis and Transformation*, Volume 8901 of *Lecture Notes in Computer Science*, pages 109–126. Springer International Publishing, Cham, 2014. DOI: 10.1007/978-3-319-14125-1.

[15] Vidal, G. Towards symbolic execution in Erlang. In Voronkov, A. and Virbitskaite, I., editors, *Perspectives of System Informatics*, Volume 8974 of *Lecture Notes in Computer Science*, pages 351–360. Springer, Berlin, Heidelberg, 2015. DOI: 10.1007/978-3-662-46823-4_28.

Acta Cybernetica **27** (2025) 155–174.

Towards Correct Dependency Orders in Erlang Upgrades*

Daniel Ferenczi[ab] and Melinda Tóth[ac]

Abstract

Erlang tooling offers rich options to control the exact tasks to perform during an upgrade. This control aims to allow for zero-downtime upgrades. Upgrades affecting multiple dependent modules must reflect the dependency order in the upgrade's configuration, as an erroneous configuration results in unintended behavior, possibly even downtime. This paper presents two static analysis-based checkers for verifying module-related aspects of upgrades. In our first analysis, we compare the actual dependency order derived from the source code with that expressed in the upgrade configuration. We also analyze the code to find circular dependencies among its modules. These pose a problem during upgrades and are generally good practices to avoid. Both checkers present an argument in favor of using static analysis methods to define upgrade specifications.

Keywords: Erlang, upgrades, static analysis, upgrade safety, dependency order, RefactorErl

1 Introduction

Ensuring continuous operation of IT services is considered the norm in today's software environment. While this was typically a feature of safety-critical systems decades ago, today we can encounter it in many customer-facing applications: retail, banking, and entertainment. The need for this is reasonable considering the global nature of these services. These applications operate around the clock and are changed while running without a noticeable impact on the end user. The state-of-the-art tooling based on containers, serverless services, and other features offered

*Project no. TKP2021-NVA-29 has been implemented with the support provided by the Ministry of Culture and Innovation of Hungary from the National Research, Development and Innovation Fund, financed under the TKP2021-NVA funding scheme. Supported by the EKÖP-KDP-24 University Excellence Scholarship Program Cooperative Doctoral Program of the Ministry for Culture and Innovation from the source of the National Research, Development and Innovation Fund.

[a]Faculty of Informatics, Eötvös Loránd University, Budapest, Hungary
[b]E-mail: danielf@inf.elte.hu, ORCID: 0009-0002-0611-006X
[c]E-mail: tothmelinda@elte.hu, ORCID: 0000-0001-6300-7945

DOI: 10.14232/actacyb.312294

by hyperscalers standardizes some of the associated tasks. However, these solutions require additional effort from operational specialists, for example, load balancers and draining periods have to be configured, and the application state has to be preserved. These update schemes also work on high granularity: even small code changes require the replacement of a unit typically composed of the whole application binary. As larger applications contain multiple such units, upgrades with a broader scope will require care when determining which unit to change at each upgrade step.

Some languages and runtimes allow runtime changes on a small granularity. This allows for preserving the application state during software changes. Erlang is a language that allows for state-preserving code changes. In our work, we research the challenge of changing dependent code units in Erlang-based software stacks [2]. Erlang was developed with built-in features for concurrency, fault tolerance and continuous operation. This thins the software stack Erlang applications require. Consequently, developers maintaining them can create zero-downtime upgrades by solely using the features of the language and its runtime. With regard to upgrades, Erlang allows for live replacement of application modules and has upgrade-related tooling built-in into the language as well. The tooling, the small upgradeable units, and the runtime together allow the developer to reason about changes on a small granularity and declare disruption-free upgrades for her application.

Although upgradeable units are small, the problem of identifying how they depend on each other inside a given application is still applicable. Module dependency structure is also a good candidate for analysis: circular dependencies are best to avoid in general, and the Erlang release handling guidelines even advise against using them, as they might make safe upgrades impossible. We also support the detection of these with our checker, which will aid developers in structuring their code.

As the application modules' code meant to be upgraded and the upgrade specification are both expressed in Erlang, we based our work on existing static analysis tools to inspect the dependencies in the code.

This paper is structured as follows: in the next Section 2 we briefly present the Erlang language and the RefactorErl static analyzer on which we base our work. In Section 3 we introduce how upgrades work in Erlang applications. Section 4 exposes the specific problems we investigated and our developed checkers. We have dedicated a specific subsection for each analyzed problem: dependency discrepancies 4.1 and upgrades of dependency cycles 4.2. We evaluate our work in Section 5. Related work is presented in Section 6. Finally, concluding remarks and future work are described in Section 7.

2 Erlang and RefactorErl

Erlang is a dynamically typed functional programming language. It was developed at Ericsson for use in the telecommunications domain. It contains in-language features for developing highly scalable, fault-tolerant distributed software. These fea-

tures are provided by the runtime and the standard libraries included with Erlang distributions. This contrasts with other languages that require the introduction of other components into the software stack to allow for fault-tolerance or disruption-free code changes. Bundled tooling includes software for defining and managing upgrades on a fine-grained level to ensure disruption-free upgrades. These tools are used to define applications, releases (entities composed of multiple Erlang applications), and respective upgrade files, `appup` and `relup`. These upgrade files are interpreted by the Erlang runtime's Release Handler [10]. The requirements for reliable, upgradeable software have become more general since Erlang's first release. By now, the language has been adopted across several other domains: banking, instant messaging, cloud services [3, 4].

RefactorErl [1] is a static source code analyzer for Erlang that also supports code comprehension and refactoring. It is available for Linux, macOS and Windows and can be used through IDE integration, the command line, or a web interface. The tool analyzes loaded code, generates its abstract syntax tree, and enhances it with output from different analyzers: function, data-flow, etc. The resulting Semantic Program Graph (SPG) [8] allows easy analysis of the loaded program through a query language [17]. It is distributed with several built-in checkers for inspecting OWASP vulnerabilities, dependency structure, and dynamic function calls. It offers a rich framework for semantic analysis, including a rich query language. These features allow the user to develop their own static analyzers. As RefactorErl is also open-source, even more elaborate checkers can be developed and integrated into the tool. Given its features and extensibility, we chose it as our tool to implement our code checkers.

3 Upgrading dependent Modules in Erlang

Erlang source files (modules) may contain references to functions exported in other modules. We say that module `a` depends on module `b` if there is a call in `a` to a function in `b`. We call a module that has dependencies a dependent module. We represent this relationship with an arrow pointing from `a` to `b`: $\mathtt{a} \rightarrow \mathtt{b}$. In this context `a` is a dependent module. Dependency relationships can consequently be represented using directed graphs, and we can analyze the dependencies of application by inspecting such graphs.

Figure 1 shows a simple dependency relationship between modules. Module `a` depends on module `b`, which in turn depends on modules `c` and `d`.

The order upon which modules depend on each other is important during a release's upgrade cycle: as complex upgrades involve changes in multiple modules, if these depend on each other, their dependency has to be reflected in the upgrade steps as well. This is required as the application runs and function calls can happen during the upgrade process. To this result, a developer has to ensure that the version of the dependent module is aligned with that of the dependency in periods when calls can be made from the dependent to the dependency. A call from a different version could result in the dependent assuming a different interface for the

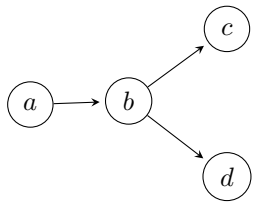


Figure 1: A simple dependency relationship between 4 modules represented as a graph

function in the dependency than what is actually implemented. To solve this, a safe upgrade procedure must ensure that running dependent modules are compatible with their dependencies' interfaces as these are changed.

When using the standard Erlang tooling for managing upgrades, the steps for performing the upgrade are declared in an `appup`, *application upgrade* file by the developer and are specific to the application that is updated. These files contain high-level instructions for declaring the module-specific actions that are to be performed during the application's upgrade. These actions offer control over whether modules are suspended while changed, added, or removed when upgrading the application to a new version or downgrading to a previous one. As a release may consist of multiple applications, `appup` files are combined into a `relup`, *release upgrade* file that must contain lower-level instructions on how to perform the upgrade. These files contain upgrade and downgrade instructions to support changes to different versions. As instructions are executed sequentially, their order must be aligned with the dependency relation of the modules and the interoperability between release versions. The structure of these files is illustrated in Figure 2. In the tuple describing the upgrade `Vsn` refers to the version to which we want to upgrade or downgrade. The second element of the tuple lists the versions we can upgrade from `Vsn` and the required set of instructions for the given upgrade. The last element of the tuple lists similarly downgrades paths and instructions.

```
{Vsn,
  [{UpFromVsn, Instructions}, ...],
  [{DownToVsn, Instructions}, ...]}.
```

Figure 2: Structure of `appup` and `relup` files

`relup` files are interpreted by the Erlang Release Handler and must only contain low-level instructions for the upgrade's definition. Low-level instructions differ from higher-level ones in that they offer control of the lifecycle of running processes, including their suspension, transformation of their state or synchronization of Erlang nodes. `relup` files are typically generated from `appup` files with the help of the release-related tooling offered by Erlang. This automatic conversion to `relup` files assumes however backward compatibility of new modules when determining

the order of module changes. As the interoperability of modules between versions might differ from that assumed by the Release Handler, `relup` files may also be written manually.

To account for both manual and automated workflows, we compare the dependencies derived from the source files with those implicitly expressed in the `relup` files. An example of an `appup` and its derived `relup` file can be seen in Figure 3. The sample's first section, from lines 2 to 11, shows high-level instructions for defining a release. This block includes the identifier of the released version in line 2, and two lists, from lines 3 to 6, and from lines 7 to 10. These lists allow for listing upgrade and downgrade paths respectively, following the structure presented in Figure 2. In our example, we declare the the rules for upgrading from version 1.0 to 1.1, and for downgrading from version 1.1 to version 1.0. Both blocks support the declaration of multiple paths, so, for example we could define the instructions to upgrade from version 0.9 as well.

The second section, from lines 14 to 28, shows a corresponding `relup` file. Although the structure is the same as that of the `appup` file, the commands defined must be low-level instructions which are executed by the Erlang Release Handler. Details of the example are described in Section 4.1.1.

Defining the steps necessary for an upgrade is a manual, error-prone process that requires a thorough understanding of the application source code and the dependency relations within. It is also unsafe, as `appup` or `relup` instructions inconsistent with the actual dependency relationship can result in errors or even temporary failures which are hard to debug. A developer declaring the upgrade instructions would need to review the dependency relationship of the affected modules, and in case of an inconsistency either redefine the upgrade instructions or change the source of the new release. As an incorrect manual analysis can consequently lead to either a failed upgrade or unnecessary changes, the developer would benefit of static checkers that contrast upgrade steps with the dependency structure.

In order to support the otherwise unsafe task of declaring upgrade definitions, we extended RefactorErl to analyze `relup` files and contrast the specified upgrade steps with the actual dependencies of the application. Although existing tools, such as erlup[1], relflow[2] or the appup plugin for the rebar3 build tool[3] support the generation of `appup` or `relup` files, they work by assuming specific code structures and backward compatibility and do not support validation of custom `relup` files against the actual dependency relationship. Our work is novel in the approach to verifying custom release definitions using effective module relationships. Our main contributions are as follows:

- The development of new features for RefactorErl, to retrieve upgrade-related information from `relup` files
- The development of two checkers for RefactorErl that use existing dependency-related analysis along with the one developed for upgrade specifications

[1] https://github.com/soranoba/erlup
[2] https://github.com/RJ/relflow
[3] https://github.com/lrascao/rebar3_appup_plugin

```
%% appup file for application release_tst
{"1.1",
  [{"1.0", [
    {load_module, depmod},
    {update, servermod, [depmod]}
  ]}],
  [{"1.0", [
    {load_module, depmod},
    {update, servermod, [depmod]}
  ]}]
}.

%% relup file for release consisting of app release_tst
{"1.1",
 [{"1.0",[],
   [{load_object_code,{release_tst,"1.1",[servermod,depmod]}},
    point_of_no_return,
    {suspend,[servermod]},
    {load,{depmod,brutal_purge,brutal_purge}},
    {load,{servermod,brutal_purge,brutal_purge}},
    {resume,[servermod]}]}],
 [{"1.0",[],
   [{load_object_code,{release_tst,"1.0",[servermod,depmod]}},
    point_of_no_return,
    {suspend,[servermod]},
    {load,{servermod,brutal_purge,brutal_purge}},
    {load,{depmod,brutal_purge,brutal_purge}},
    {resume,[servermod]}]}]}.
```

Figure 3: Example of `appup` and `relup` files

4 Supporting Correct Release Definitions

In the following subsections, we present the checkers that we have developed for detecting instruction order-related problems in `relup` files and recognizing updates of circular dependencies. They present the details of their respective domains that define the goals of our analyzer.

4.1 Discrepancy Detection in Upgrade Definitions

Our research aims to verify whether the dependency order expressed in `relup` files is consistent with the actual dependency of the modules. We begin by discussing the details of `relup` files relevant to our checker and how RefactorErl can support

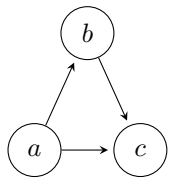
our analysis. We continue with the objectives, implementation, and limitations of our checker algorithm.

4.1.1 Problem Description

To understand how dependency order can be taken into account during upgrades, we can look at the example in Figure 3. In the example's `appup` file, we declared an upgrade from version 1.0 to 1.1 and a downgrade from version 1.1 to 1.0 respectively at lines 3 and 7. Specifically, we tell the Release Handler to load the newer version of `depmod` and update module `servermod`, which depends on `depmod`. The dependency relation is declared in the lists in lines 5 and 9. If we look at the list of instructions, both `load_module` and `update` atoms declare code changes. The difference lies in that `update` takes care of temporarily suspending processes running the target module, and transforming the internal state of the running process if the new version requires it. These additional operations allow for zero-downtime upgrades. `servermod` being a server implementation requires `update` for its code upgrade.

The generated `relup` file is of a similar structure: it contains first the upgrade and then the downgrade instructions. The set of instructions can only contain however lower-level operations executed by Erlang's Release Handler. Without going into detail, we can observe how the dependency relation results in `depmod` being changed before the dependent `servermod` module in lines 19 and 20 for the upgrade, and 26 and 27 for the downgrade. In these files, we are looking for `suspend`, `load` and `resume` instructions to ensure that these dependencies are updated before dependent modules.

The actual dependencies of an application are expressed as function calls in the Erlang source files. For analyzing and retrieving them, we rely on RefactorErl's features to inspect module dependencies. These features allow us to list the set of modules each dependent module depends on. Figure 4 shows an example of how RefactorErl generates the text representation of dependency relationships within an application.




```
c -> []
b -> [c]
a -> [c, b]
```

Figure 4: An example for dependent modules (Left). RefactorErl's textual representation of the dependencies (Right).

4.1.2 Detection Methodology

Our task is to determine if the upgrade steps declared in `relup` files are aligned with the actual dependencies of the application's modules. A *fitting* upgrade defini-

tion ensures that dependent modules use dependencies of the corresponding release version. If the versions between the dependencies are not aligned, modules might attempt to use non-existing functions from their dependencies. Using implementations from other versions can also be dangerous if they contain side effects. An example of `relup` instructions that can result in a runtime problem is shown in Figure 5.

```
...
    {load,{a,brutal_purge,brutal_purge}},
    {load,{b,brutal_purge,brutal_purge}},
...
```

Figure 5: A simple sequence of loaded modules

Here, assuming that module `a` depends on module `b`, we load the new version of the dependent before that of its dependency. As the load instruction simply replaces the running code without suspending processes, for a brief time window, between the two steps, code in module `a` can call functions in module `b` that do not yet exist. If we load the dependency before the dependent, the period between the two steps will allow `a` to attempt to use functions in `b` that were present in the previous version. If `b`'s new release is backward compatible with the old one, this will not result in a runtime error.

If the dependency's new version is not backward compatible, we need to ensure that calls only happen between modules of the same release version. This can be achieved by suspending the dependent module and replacing it and its dependencies during the suspension. Once all affected modules are replaced, the dependent module can be resumed. This process ensures that all affected modules are of compatible versions during active periods of the dependent module. Although execution of code halts during suspension, this still does not cause a disturbance in the application's availability, as the Erlang runtime will take care of processing any requests on the dependent once it is running again. An example of loading changing code during the dependent's suspension can be seen in Figure 6.

```
...
    {suspend,[a]},
    {load,{b,brutal_purge,brutal_purge}},
    {load,{a,brutal_purge,brutal_purge}},
    {resume,[a]}
...
```

Figure 6: Loading a module during suspension

In our research, we look at how dependencies are updated concerning the suspension of their dependents. In terms of the instructions in a `relup` file, a module's suspension period is the set of instructions between the module's `suspend` and `resume` instructions. An upgrade can be either a `load` instruction by itself or surrounded by a pair of `suspend` and `resume` instructions. Suspension periods can be

nested, as dependencies might also require suspension when updating them or their own set of dependencies.

Our work does not consider the different versions of a module across releases and hence we do not attempt to reason about interface compatibility of dependencies and the flexibility this offers. However, if a dependent is updated while suspended along with its dependencies, we can argue about the correctness of the order of instructions inside the `relup` file describing the update. Therefore, we analyze upgrade sequences where dependent modules are suspended and identify the update of a dependency with its `load` instruction.

We can summarize our goals with the following rules:

- A dependency does not have to be loaded if the dependent is suspended
- A dependency must only be loaded during the suspension of its dependent

A release does not have to include changes for all source modules, and the lack of change in a module does not impact the reliability of the software. With regard to the second observation, if a dependency were to be upgraded outside the suspension of the dependent, there could be room for discrepancies between what the dependent expects and the source of the dependency. Consequently, we assume that in releases where a dependent is suspended, changed dependencies are to be modified during the dependent's suspension period. To do so, we iterate through the list of dependency relationships generated by RefactorErl (see the example in Figure 4), identifying suspension periods of dependents and verifying whether dependencies are updated exclusively in these segments.

4.1.3 Algorithm

Our algorithm for detecting issues in `relup` files is presented in Algorithm 1.

The algorithm receives as input the `relup` file and the dependency relationships generated by RefactorErl and presented in Figure 4. In line 1 we retrieve the set of release definitions from the `Relup` file. This includes the set of instructions for both upgrade and downgrade releases. Recall from Figure 3 that a single file may contain multiple instruction lists, one for each upgrade and downgrade path. To group our results on a release definition basis, we iterate through these sets in line 2, and through the dependents in a nested loop in line 3. Next, we iterate through the individual instructions of the current release definition. For each instruction, we are interested in whether it affects the suspension state of the actual dependent, or if it relates to a dependency of the dependent. Throughout the loop, in line 6 we observe whether the dependent is currently suspended. In line 7 we verify if the current instruction relates to an update of a dependency. If so, the dependency is upgraded outside the suspension period of the dependent, and we add the instruction along with the name of the dependent module and the version identifier to a list in line 8. Storing the name and version is important so that the developer can find the instruction in the `relup` file more easily. The loops will perform the same analysis through the different sets of instructions described in the `relup` file. Finally, we return the list of unsafe instructions.

Algorithm 1 Finding discrepancies between `relup` instructions and actual dependencies

Funct FindUpdateDiscrepancy(*Relup*, *dependents*)

```
1:  Release Definitions ← Relup
2:  for all Release Definition ∈ Release Definitions do
3:    for all dependent ∈ dependents do
4:      Dependencies are determined along with dependent
5:      for all Instruction ∈ Release Definition do
6:        IsdependentSuspended is determined based on dependent and processed
          Instruction
7:        if not IsdependentSuspended and Instruction updates a Dependency of
          dependent then
8:          Store Instruction and dependent pair in UnsafeInstructionList
9:        end if
10:     end for
11:   end for
12: end for
13: return UnsafeInstructionList
```

A developer should treat this list as a warning, as in practice she knows best if a module can be changed while the ones using it are still running.

4.1.4 Limitations

As mentioned, our checker does not take into account the interoperability of modules across releases. Backward compatibility allows flexibility in organizing upgrade instructions. Therefore, our approach overapproximates and we could produce a more exact analysis by taking into account actual changes in the source code.

Another limitation is present when analyzing modules that have unrelated dependents. Figure 7 presents such an example.

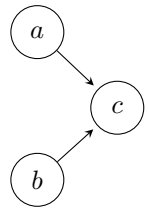


Figure 7: Multiple independent modules (`a` and `b`) depending on a single dependency (`c`)

In such scenarios, it is difficult to argue in which dependent's suspension period should the dependency be upgraded. Again, the correct way to upgrade such an application depends on the actual details in the source code.

4.2 Circular Dependency Detection

Circular dependencies amongst modules may also put upgrades at risk. We look at the information present in `relup` files and RefactorErl reports and identify the exact patterns that we wish to detect. We follow by presenting an algorithm for this purpose and discuss opportunities for improvement.

4.2.1 Problem Description

As established in Section 3, two modules depend on each other if one module's code calls functions implemented in other modules. Module `a` and `b` depend on each other if both `a` → `b` and `b` → `a` hold true. A circular dependency between two modules might be either direct, when the modules call each other's functions explicitly, or indirect if there are additional modules in the dependency circle. Examples of these are presented in Figure 8. A code sample with two circularly dependent modules is shown in Figure 9. In the examples modules `a` and `b` directly depend on each other.

Figure 8: Direct (left) and indirect (right) circular dependencies

The difficulties in determining the correct order for defining an upgrade are noted in the Release Handling Section of the Erlang manual. In most cases, it is best to avoid them altogether for code meant to be upgraded. For this checker, we also rely on the graph analysis feature of RefactorErl. For the analysis, we use a graph model of the dependency structure, where the modules will be the nodes, and the dependency relations the edges. Our task will be to determine if such a graph, excluding modules not being subject to an upgrade, has a topological ordering. We base our work on RefactorErl's features for analyzing dependency graphs. To retrieve the list of modules changed in a release from a `relup` file, as in our work presented in Section 4.1.

```
%% code of module a
-module(a).
-export([sum/0, number/0]).

sum() ->
  b:number().
number() ->
  42.

%% code of module b
-module(b).
-export([number/0]).

number() ->
  a:number().
```

Figure 9: Sample code with two modules depending on each other

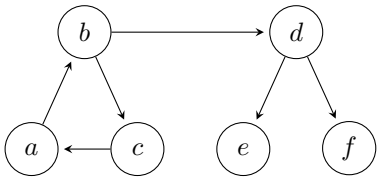
4.2.2 Detection Methodology

Our checker aims to warn developers if their `relup` contains `load` instructions for modules that are part of a dependency cycle. We do not wish to raise warnings because of the general presence of a cycle, as in such an application it is still possible to define upgrades of modules that are not part of any cycle.

However, we do want to raise a warning if a module is part of a cycle, as reasoning about the safe way to release a change would depend on the code. We present these scenarios in Figure 10. The application consists of six modules, three of which, `a`, `b` and `c` depend on each other. The top instructions in the `relup` file relate to upgrading modules `a` and `b` and could as a consequence be unsafe, as there is no clear dependency order by which the upgrade should be performed. In contrast, the `load` instructions for modules `f` and `d` (assuming that `f` is backward compatible) are safe as a dependency order can be determined from the graph. An application might also contain multiple cycles. As a result, the set of all cycles should be an input of our checker. As output, we expect those *load* instructions that relate to modules present in circular dependencies.

RefactorErl already provides a method for retrieving all dependency cycles present in an application. This functionality also detects dynamic function calls and includes them in the dependency graph. In dynamic function calls, called modules are not explicitly invoked, but rather the target module is referred to using a variable. The variable can be defined in other parts of the source code, making manual dependency analysis more difficult. An example of a module using a dynamic function call is presented in Figure 11. This code snippet shows a call retrieving the name of module `b` in line 5, and the invocation of `b`'s `my_function` in line 6. RefactorErl's dependency checker can detect dynamic dependencies through data-flow analysis [16, 7]. Detecting dependencies stemming from dynamic function

calls increases our checkers precision, as we can find dependency relationships typically only detectable during runtime. We base our work on RefactorErls data-flow analysis-based dependency checker.




```
...
%% possibly unsafe instruction sequence
{load,{a,brutal_purge,brutal_purge}},
{load,{b,brutal_purge,brutal_purge}},
...
%% safe instruction sequence
{load,{f,brutal_purge,brutal_purge}},
{load,{d,brutal_purge,brutal_purge}},
...
```

Figure 10: The dependency relationship of six modules contains a cycle (left). The release instructions include changes to the modules not present in the circular relationship (right)

```
1  -module(a).
2  -export([fun/0]).
3
4  fun() ->
5    Mod = get_mod(),
6    Mod:my_function().
7
8  get_mod() ->
9    b.
```

Figure 11: A dynamic function call

4.2.3 Algorithm

Our algorithm for detecting changes in dependency cycles is presented in Algorithm 2.

The algorithm receives the `relup` contents and set of dependencies as input. In line 1 we retrieve all dependency cycles from the dependencies. Next, in line 2 we gather the release definitions from the Relup file. We start iterating through these definitions in line 3, and their individual instructions in line 4. We inspect each instruction to see if it `load`s a module present in any of the cycles identified previously. If the instruction relates to such a module, we store it in a list of unsafe instructions in line 7, together with an identifier of the release definition so that the developer can place the problematic instruction more easily. Finally, we return the List of Unsafe Instructions.

Algorithm 2 Finding changes in dependency cycles expressed in *relup* instructions

Funct FindChangeInCycle(*Relup*, *Dependencies*)

```
1: Dependency Cycles ← dependency cycles from Dependencies
2: Release Definitions ← Relup
3: for all Release Definition ∈ Release Definitions do
4:   for all Instruction ∈ Release Definition do
5:     Changed Module ← Instruction
6:     if Changed Module ∈ Dependency Cycles then
7:       Store Instruction in UnsafeInstructionList
8:     end if
9:   end for
10: end for
11: return UnsafeInstructionList
```

4.2.4 Limitations

RefactorErl provides a solid basis for retrieving all dependency cycles present in the application. There exists the possibility of safe upgrades of dependency cycles if the affected modules are backward compatible. Therefore, our method might mark instructions as unsafe that are safe in practice, but to be more precise we would have to be aware of implementation details of the application being analyzed.

5 Evaluation

`appup` and `relup` files are not typically put under version control and published. Regardless, we aimed to assess the value of our checkers by inspecting the dependency structure of publicly available sources, made available on GitHub[4]. We analyzed 5 popular Erlang applications from several domains: instant messaging (MongooseIM[5]), MQTT (emqx[6]), web servers (Cowboy[7], Yaws[8]) and databases (couchdb[9]).

For our evaluation, we used RefactorErl's same dependency analysis features applied in the checkers we developed. They identified dependency relations are include dynamic dependencies in separate rows. These are determined with data-flow analysis and include dynamic function calls and invocation of the `apply` function that allows calling functions set as its arguments. The details and limitations of combining dependency and static analysis was presented by the authors formerly [7].

[4] https://github.com
[5] https://github.com/esl/MongooseIM
[6] https://github.com/emqx/emqx
[7] https://github.com/ninenines/cowboy
[8] https://erlyaws.github.io
[9] https://couchdb.apache.org

All analyzed applications are designed for implementing highly scalable services, and fault-tolerant services, where the operator can expect to perform disruption-free upgrades. For this task, they would have to create the `appup` or `relup` files necessary for their release tooling. In Table 5 we show the number of dependent modules these projects have, the number modules taking part in a dependency cycle, the highest number of dependencies a module has and whether this module and its dependencies has changed in the latest minor release. For contrast, we have also made these measurements counting dynamic dependencies as well.

Table 1: Dependency complexity in six popular projects: couchdb, MongooseIM, Cowboy, emqx and Yaws

Metric	couchdb	MongooseIM	Cowboy	emqx	Yaws
# of Dependents	378	496	18	115	35
# of Dependents (including Dynamic Dependencies)	383	501	20	117	37
# of Dependents in Dependency Cycles	167	83	2	61	15
# of Dependents in Dependency Cycles (including Dynamic Dependencies)	224	257	14	68	19
Highest Number of Dependencies	31	50	12	35	32
Latest Update Affects Most dependent Modules	Yes	Yes	Yes	Yes	Yes

As the table shows, all releases contain dependency cycles that make reasoning about release correctness more difficult. Additionally, about half of the dependent modules are also present in a dependency cycle if we take into account dynamic dependencies. Thus, including dynamic dependencies leads to a significant difference in the number of modules present in cycles. This shows the value of applying a broader set of static analysis techniques not only for planning upgrades, but for analyzing and improving code structure as well. Changing code of such complexity manually would be unsafe to manage, consequently we find that our tools would help with these tasks, especially with upgrades where a large number of dependencies is changed. Finally, we have found that all projects had their most dependent module have its dependencies changed since the last minor release.

6 Related work

Circular dependencies. Li and Thompson in their previous research [9] have analyzed the issue of circular dependencies in Erlang as part of their work on the refactoring tool, Wrangler. The authors' analyzer focuses on refactoring problematic patterns into clean code. However, it does not analyze upgrade specifications and can not reason about upgrade safety. Also, Wrangler does not include dynamic dependencies in its cycle analysis and does not feature analysis of `relup` files.

Upgrade safety. Naseer, Noccolini, Pain, Frindell, Dasineni and Benson have researched upgrade safety emphasizing runtime facets typically unrelated to an application's implementation language, like connection migration [13] between application versions. Being able to migrate connection is important, as reestablishing them would not only impact the user, but perhaps even require an unavailable amount of resources. Erlang is singular in the regard, that its runtime provides facilities for handling state preservation during code changes, without the need for introducing new tools or bespoke solutions.

Static analysis. Tools to support schema changes in backing databases have been researched by Maule, Andy and Emmerich [11]. The authors developed an approach for verifying whether a database schema change is consistent with the application's source code, improving on existing string-based checkers with static analysis. They argue for improving the accuracy of impact analysis by introducing further methods from static analysis. Meurice, Nagy and Cleve in their work [12] used static analysis as well to locate source code affected by database schema changes. They also assess whether a given change affects the developed application. It would be worth to investigate whether other upgrade-related static properties can be defined for Erlang using RefactorErl.

Microservice changes. Sampiao, Kadiyala, Hu, Steinbacher, Erwin, Rosa, Beschastnikh and Rubin have investigated challenges with regards to support upgrades in running microservice systems [15]. The research proposes modelling the software's evolution by analyzing static and dynamic information obtained from the system. Such models would help developers design their upgrade schemes and plan the evolution of their software in a consistent manner. Erlang developers design the scale, distribution and upgrades of their application in the same codebase. Our work could be extended by researching a broader set of changes between application releases.

RefactorErl. Tóth and Bozó have used RefactorErl for analyzing other static properties of source code, including the presence of common vulnerabilities [18]. These checkers can be used to assess the security of new releases. RefactorErl's database and queries can be also used to verify further upgrade-related properties and data-flow analysis allows inspecting behavior that could typically be only observed during execution.

Code upgrades and downtime. Neamtiu and Dumitra analyzed the relationship between upgrades and downtime [14]. The authors look at challenges across the stack: database schema migrations, infrastructure changes, mixed-version race conditions and protocol changes. They highlight the value of upgrade schemes that provide more control than rolling upgrades and of expressing the details of an upgrade explicitly. Erlang allows for expressing the details of an upgrade and the application's operation in it's language. This also allows for using existing static analysis methods for inspecting the details of an upgrade and is the motivation of our current and future work.

Safe Upgrades for Erlang Software. In our previous work, we have also analyzed other conditions for upgrading Erlang software without disruptions. For example, code meant to be upgraded must consistently call functions that are still present in the runtime. State transitions constitute another example: Erlang allows for changing a running application's state during its upgrade, but it is the developer's responsibility to change and use the state consistently. In previous research, we looked into these two problems and identified several coding patterns that would disrupt safe upgrades. We have also implemented code checkers based on RefactorErl to aid developers write upgradable software.

We began by researching [5] if applications contain any references to functions that would expire as the code is upgraded. The Erlang runtime only holds two versions of a given runtime at the same time. As consequence, local function references that remain unchanged during upgrades are unsafe as they become obsolete. By using fully qualified references, the functions used will be of the module version loaded last. Our checker helps the developer identify places where fully qualified references should be used.

In our second work [6] we investigated if state uses in a new application version are consistent with the state transformations performed during code changes. Erlang allows the developer to modify their application's state during an upgrade. To this effect they must implement `code_change` functions that specify a state transformation logic for the different upgrade paths. Researching additional unsafe patterns and a generic approach to support upgrade safety are further areas worth exploring.

7 Conclusions and Future Work

Erlang offers the tools necessary to create application releases with fine-grained instructions to ensure disruption-free upgrades of modules. To achieve this, code has to be structured in an upgradeable manner, and the release's descriptor file should also reflect this structure correctly. We researched how specific upgrade-related instructions should be ordered to be in line with the actual code structure, identified two problem categories, and developed checkers for them using the RefactorErl framework. Our first checker identifies if dependencies are upgraded during their dependent's suspension period. This checker does not cover the flexibility

that interfaces compatible between releases would allow. For example, assuming that the new release of a dependency is capable of receiving new calls, its new version can be `load`ed before we `load` the new version of the dependent, without requiring any sort of suspension. Our research can be extended in two steps: analyzing if modules are `load`ed in the correct order assuming that their interface changes are backward compatible; and analyzing actual backward compatibility as well between code releases. Of course, interface compatibility would not guarantee a well-working application. Upgrades can introduce domain-specific discrepancies into application code that retain interface-compatibility but will result in runtime problems. Such issues are hard to detect with static code analysis, and thus remain outside the scope of our work.

Our second checker investigates if changed modules are part of dependency cycles. Upgrading such structures is not recommended, as it is difficult to reason about the correct instruction order to implement an upgrade. Our analysis' precision can again be improved by taking into account the interoperability between code releases.

In conclusion, our research offers two checkers for developers to evaluate the correctness of their upgrade definitions and covers the directions to improve this analysis.

References

[1] Bozó, I., Horpácsi, D., Horváth, Z., Kitlei, R., Köszegi, J., M., T., and Tóth, M. RefactorErl — Source code analysis and refactoring in Erlang. In *Proceedings of the 12th Symposium on Programming Languages and Software Tools*, pages 138–148, Tallin, Estonia, 2011. ISBN: 978-9949-23-178-2.

[2] Cesarini, F. and Thompson, S. *Erlang Programming: A Concurrent Approach to Software Development*. O'Reilly Media, 2009. ISBN: 9780596555856.

[3] Cesarini, F. *Which companies are using Erlang, and why?* Erlang Solutions, 2019. URL: https://www.erlang-solutions.com/blog/which-companies-are-using-erlang-and-why-mytopdogstatus/.

[4] Erlang Solutions. *20 Years of Open Source Erlang: OpenErlang Interview with Anton Lavrik from WhatsApp*, 2018. URL: https://www.erlang-solutions.com/blog/20-years-of-open-source-erlang-openerlang-interview-with-anton-lavrik-from-whatsapp/.

[5] Ferenczi, D. and Tóth, M. Static analysis for safe software upgrade. In *Annales Mathematicae et Informaticae*, Volume 58, pages 9–19, 2023. DOI: 10.33039/ami.2023.08.010.

[6] Ferenczi, D. and Tóth, M. Safe process state upgrades through static analysis. In *Proceedings of the 2024 IEEE 18th International Symposium on Applied Computational Intelligence and Informatics*, pages 000351–000356. IEEE, 2024. DOI: 10.1109/SACI60582.2024.10619854.

[7] Horpácsi, D. and Koszegi, J. Static analysis of function calls in erlang. refining the static function call graph with dynamic call information by using data-flow analysis. *e-Informatica Software Engineering Journal*, 7(1), 2013. DOI: 0.5277/e-Inf130107.

[8] Horváth, Z., Lövei, L., Kozsik, T., Kitlei, R., Víg, A. N., 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(2009) Special Issue of *Studia Universitatis Babeş-Bolyai, Series Informatica*, pages 7–16, Cluj-Napoca, Romania, 2009. URL: http://www.studia.ubbcluj.ro/arhiva/abstract_en.php?%20editie=INFORMATICA&nr=Sp.Issue%201&an=2009&id_art=6521.

[9] Li, H. and Thompson, S. Refactoring support for modularity maintenance in Erlang. In *Proceedings of the 2010 10th IEEE Working Conference on Source Code Analysis and Manipulation*, pages 157–166, 2010. DOI: 10.1109/SCAM.2010.17.

[10] Logan, M., Merritt, E., and Carlsson, R. *Erlang and OTP in Action*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2010. ISBN: 9781933988788.

[11] Maule, A., Emmerich, W., and Rosenblum, D. S. Impact analysis of database schema changes. In *Proceedings of the 30th International Conference on Software Engineering*, pages 451–460, New York, NY, USA, 2008. Association for Computing Machinery. DOI: 10.1145/1368088.1368150.

[12] Meurice, L., Nagy, C., and Cleve, A. Detecting and preventing program inconsistencies under database schema evolution. In *Proceedings of the 2016 IEEE International Conference on Software Quality, Reliability and Security*, pages 262–273, 2016. DOI: 10.1109/QRS.2016.38.

[13] Naseer, U., Niccolini, L., Pant, U., Frindell, A., Dasineni, R., and Benson, T. A. Zero downtime release: Disruption-free load balancing of a multi-billion user website. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 529–541, New York, NY, USA, 2020. Association for Computing Machinery. DOI: 10.1145/3387514.3405885.

[14] Neamtiu, I. and Dumitraş, T. Cloud software upgrades: Challenges and opportunities. In *Proceedings of the 2011 International Workshop on the Maintenance and Evolution of Service-Oriented and Cloud-Based Systems*, pages 1–10. IEEE, 2011. DOI: 10.1109/MESOCA.2011.6049037.

[15] Sampaio, A. R., Kadiyala, H., Hu, B., Steinbacher, J., Erwin, T., Rosa, N., Beschastnikh, I., and Rubin, J. Supporting microservice evolution. In *Proceedings of the 2017 IEEE International Conference on Software Maintenance and Evolution*, pages 539–543. IEEE, 2017. DOI: 10.1109/ICSME.2017.63.

[16] Tóth, M. and Bozó, I. Static analysis of complex software systems implemented in Erlang. In *Proceedings of the Fourth Central European Functional Programming School*, Volume 7241 of *Lecture Notes in Computer Science*, pages 440–498. Springer-Verlag, 2012. DOI: 10.1007/978-3-642-32096-5_9.

[17] Tóth, M., Bozó, I., Kőszegi, J., and Horváth, Z. Static analysis based support for program comprehension in Erlang. *Acta Electrotechnica et Informatica*, 11(3):3–10, 2011. DOI: 10.2478/v10198-011-0022-y.

[18] Tóth, M. and Bozó, I. Supporting secure coding for Erlang. In *Proceedings of the 39th ACM/SIGAPP Symposium on Applied Computing*, page 1307–1311, New York, NY, USA, 2024. Association for Computing Machinery. DOI: 10.1145/3605098.3636185.

Acta Cybernetica **27** (2025) 175–196.

Radial Harmonic Fourier Moments for CT-based Quantitative Radiomics*

A. H. M. Sajedul Hoque[ab], Gergő Bognár[ac], and Sándor Fridli[ad]

Abstract

Radiomics is an emerging field of CT image processing, that offers non-invasive quantification of tumour phenotypes using quantitative image features. Radiomics analysis has promising applications in cancer treatment and personalized medicine, like treatment planning and the prediction of clinical factors. However, the optimal feature selection is not established in the literature, and the applications usually involve data mining of a large pool of features. In this paper, we propose to extract higher-level radiomic features using Radial Harmonic Fourier moments (RHFM). Image moments, and specially orthogonal Fourier moments are widely used in image processing, providing efficient and invariant shape descriptors. In particular, RHFMs are known to perform well on small noisy images, making them a promising candidate for CT tumour analysis. Motivated by these advantages, we developed a feature extraction scheme based on RHFM, and we performed radiomics analysis on lung CT images of non-small cell lung cancer patients. The proposed method is validated on multiple annotated datasets following the literature guidelines, evaluating the accuracy, stability, reliability, and prognostic value of the proposed features. The results show better reliability and otherwise comparable performance compared to the state-of-the-art wavelet descriptors. Furthermore, Fourier moments provide higher level of flexibility and possible adaptivity compared to wavelets, and unlike wavelet features, RHFM features are invariant of position, size and orientation in the tumor region.

Keywords: radiomics, lung CT, quantitative imaging, radial harmonics, orthogonal moments

*Project no. K146721 and TKP2021-NVA-29 have been implemented with the support provided by the Ministry of Culture and Innovation of Hungary from the National Research, Development and Innovation Fund, financed under the K_23 "OTKA" and the TKP2021-NVA funding schemes.

[a]Department of Numerical Analysis, ELTE Eötvös Loránd University, Budapest, Hungary
[b]E-mail: sajed@inf.elte.hu, ORCID: 0000-0001-8291-9990
[c]E-mail: bognargergo@staff.elte.hu, ORCID: 0000-0001-7818-5760
[d]E-mail: fridli@inf.elte.hu, ORCID: 0000-0003-3531-2576

DOI: 10.14232/actacyb.312363

1 Introduction

Medical imaging, especially X-ray computed tomography (CT), is a primary diagnostic tool of clinical oncology. CT, as an imaging modality, noninvasively quantifies the internal tissue density, that might help the localization and characterization of the tumour. CT imaging is routinely used in many areas of clinical oncology not only for diagnosis, but also for therapy planning and monitoring. In therapy planning, CT provides precise visualization of the geometric shape of the tumour and the normal tissue, which helps to determine the optimum radiation dose distribution in the tumour [13].

In this paper, we research quantitative imaging for lung CT motivated by personalized medicine. Personalized medicine is an emerging field that promises better patient care by taking the genetic differences of the tumour into account. In this personalized medicine, predictive and prognostic data factors coming from multimodal information including clinical, imaging, and molecular data are merged to forecast treatment outcomes [10]. However, the molecular characterization of cancer is challenging, and usually requires invasive approaches (biopsies and surgeries), which themselves may be limited if the tumour is heterogeneous. CT imaging is a promising supplementary tool to quantify tumour phenotypes [11]. As a noninvasive tool, imaging is feasible not only to support oncological diagnosis and treatment planning, but also for the long term monitor of the therapy outcomes over time. Radiomics [20] is a quantitative imaging approach that aims to extract robust image features to quantify the tumour phenotype. These radiomics features employ mathematical algorithms describing the intensity, shape, statistical, and textural properties of the tumour, usually involving a large number of features. The heterogeneity of tumour region due to its molecular characteristic expresses the texture, which holds the information about the structural arrangement of its surface and the relationship with the surrounding environment. It is already shown that radiomics correlate to tumour phenotypes [2], and can also be utilized to predict distant metastasis [4].

This research discusses radiomics features characterized by first, second and higher order statistics, with the main focus on higher order features. Briefly, first-order statistical features aims to describe the overall gray-level distribution of the tumour by quantifying the voxel intensity histogram in the tumour region. Second-order statistics are used to characterize the tumour texture using local histograms in the voxel neighborhoods. Meanwhile, higher order statistical features aims to quantify potential hidden patterns inside the tumour region, usually involving an image transformation method, like wavelets or Laplacian-of-Gaussian (LoG) pyramids [21]. Here, the desired transformation method provides a compact scale-space or frequency-space decomposition, is spatially localized, and is invariant of position, size and orientation in the tumour region. The most widely used wavelet transform (see e.g., [2]) are favored compared to other transformations (e.g., Fourier transform) because of its efficiency, its ability to provide both space and frequency (i.e., scale) representation, and its spatial localization property. However, wavelets are shift-variant and lack explainability, which might be undesired in medical applica-

tions, and which serves as a motivation of our research.

Image moments are widely used transformation-invariant feature descriptors [12], popular for pattern recognition, object representation, and feature extraction. In particular, orthogonal moments provide efficient and stable time-frequency decompositions, with the support for adaptivity and interpretability. In this paper, we focus on radial harmonic Fourier moments (RHFMs) [16], since they provide better numerical properties compared to several other moments, and even perform well on small images in a noisy environment. Compared to wavelets they also offer shift-invariance. Therefore, they seem promising for higher order analysis of CT tumour. We propose RHFM reconstructions for higher order radiomics feature extraction. The proposed model is developed and validated using multiple annotated lung CT datasets of non-small cell lung cancer (NSCLC) patients. The feature selection, and the optimization of the decomposition parameters are performed according to the analysis of the reconstruction accuracy. Finally, we evaluated the reliability, stability and prognostic power of the proposed features, that are crucial requirements in a clinical environment. Here, we followed the literature guidelines, and compared the proposed method to wavelet-based radiomics features [2].

The key contributions of this paper are highlighted as follows:

1. We designed higher order radiomics features based on RHFM reconstruction of CT images.

2. We performed an accuracy analysis to select features and to optimize the decomposition parameters (order and repetitions) of RHFM decomposition.

3. We explored and analyzed a total number of 456 RHFM-based radiomics features of the reconstructed CT image, and compared them with the same number of wavelet-based features.

The rest of this paper is organized into the five sections. "Related Works" overview the most relevant related literature results. In the "Materials and Methods" section, a summary of the utilized radiomics features, a short description of the lung CT datasets, and the basics radial harmonic Fourier moments are provided. Here, we also introduce proposed method and optimize our feature selection. The "Result and Discussion" section presents the outcomes of this work and analyzes the results in terms of reliability, stability and prognosis value. Finally, in the "Conclusion and Future Work" section, we give the concluding remarks with some future works.

2 Related Works

In this section, we briefly review the related literature of quantitative radiomics and orthogonal moments.

2.1 Quantitative radiomics

Quantitative radiomics became popular recently, since it has been shown that quantitative image features are related to tumour phenotypes, offering noninvasive approach to support cancer treatment. In [2], the authors proposed 440 high-order radiomics features based on the intensity, shape, texture and wavelet transform and analyzed those features in terms reliability, stability and prognostic power on the RIDER dataset. The author claimed that he found many radiomics features having prognostic power which were not addressed before. Another claim was that the selected features through stability and reliability analysis was more informative. Finally, the authors proposed a radiomics signature for making association with gene profiles and showed the signature represents the general prognostic tumour phenotype.

In [4], the authors worked on the extraction of 635 radiomics features based on intensity, shape, texture, LoG and wavelet-based features to predict distant metastasis (DM) for lung adenocarcinoma patients. The authors did the prognostic analysis on these feature over 182 patients. They showed that only 35 features are strongly prognostics for DM and twelve features are prognostics for survival. A standardized mathematical model for extracting radiomics features is felt in the clinical oncology. In [21], the authors developed a flexible open-source framework including a set of well-defined and tested mathematical models for easing the extraction of features from 3D or 2D medical image.

Tang et al. extracted 688 radiomics features based on the first-order statistics, shapes, texture and wavelet filters to build a classification model for hepatocellular carcinoma (HCC) patients [18]. In their work, they showed that the combined features extracted from original CT image and wavelet-filtered image increase the classification performance significantly to classify HCC and non-HCC patients. However, these high-order features are not shift-invariant which motivates us to move orthogonal moment-based features. These orthogonal moments are explained briefly in the next section.

2.2 Orthogonal moments

The characterization, evaluation and manipulation of visual information inside the CT image is a general problem in clinical oncology. The preferable representations extract features that are invariant of size, position and orientation of the CT image. Image moments (see e.g., [15]) provide potentially transformation-invariant descriptors that are desired properties for CT image quantization in order to achieve reliability and stability. Here, we overview some historical developments in this field.

In 1962, Hu introduced a non-orthogonal moment known as geometric moment for image description, and derived moment invariants based on algebraic invariance in rectangular coordinates for visual pattern and character recognition [7]. One disadvantage of this classic moment is that its invariants are restricted to second- and third- order moments only. In addition, low-order geometric moments provide

less information about image details, while high-order moments are sensitive to noise. These problems can be resolved by using circular and orthogonal moments.

In 1980, Teague proposed Zernike Moments (ZMs) as image descriptors which were constructed from a set of orthogonal Zernike circular polynomials over the unit circle [19]. The zero points of the ZMs are located in long radial distance from the origin. Thus, although the ZMs are rotation-invariant, its application for scale-invariant pattern recognition is challenging for small images. Then, Sheng and Shen explored orthogonal Fourier–Mellin moments (OFFMs) including generalized ZMs and orthogonalized complex moments [1] in 1994. The main advantage of OFFMs is that the zero points are uniformly distributed over the radial interval. For this reason, OFFMs have better performance than ZMs, providing better description of small images [17].

However, OFFMs have difficulties to describe the center of the image for higher order moments because they tend to be infinite in the origin. Ping and Sheng in 2002 solved the problem by proposing Chebyshev–Fourier moments (CHFM) which used various orders of Chebyshev polynomials over radial interval [14]. The aforementioned orthogonal moments including ZMs, OFFMs and CHFMs are based on the radial polynomials which cause numerical instability at high order of moments and high time complexity to compute the corresponding moments. In order to address these problems, Ping et al. developed new orthogonal moments known as Radial Harmonic Fourier Moments (RHFMs) where triangular function is used as radial function [16]. RHFMs are shifting, scaling, rotation, and intensity invariant, and performs better compared to CHFMs in multiple aspects, like representation near the origin, the description of small images, and noise sensitivity. These aspect are also desired for feature extraction of CT tumours that motivated our choice.

3 Materials and Methods

In this section, we provide details of the theoretical and computational background, the validation datasets, and the proposed method.

3.1 Radiomics Features

In clinical oncology, radiomics features can used to monitor the development, progression of the cancer and the response to therapy. Those features are constructed employing advanced hand-coded algorithms, that provide a large set of quantitative imaging features. Aerts et al. [2] decomposed the CT image using wavelet decomposition and analyzed 440 radiomics features in order to build radiogenomics signature. These are experimental features, without detailed explanation. In [21], the authors developed a flexible open-source PyRadiomics platform for extracting features from medical image. This platform provides a set of very well-defined, tested and standardized mathematical models for radiomics features. Thus, we used these verified features to analyze in our study which are grouped under the following categories.

- **Shape and size related features** illustrates the three-dimensional size and shape of tumour region, using elementary geometric descriptors. The common features include elongation, flatness, least axis length, major axis length, maximum 2D diameter column, maximum 2D diameter row, maximum 2D diameter, mesh volume, minor axis length, sphericity, surface area, surface volume ratio, voxel volume. We note that these features are independent of the high-order decompositions discussed in this paper, so they are excluded from the analysis, and they are mentioned only for the sake of completeness.
- **First order statistical (FOS) features** describes the gray distribution in the tumour area by means of statistical properties of the image histogram. We considered 18 descriptors: 10th percentile, 90th percentile, energy, entropy, inter quartile range, kurtosis, maximum, mean absolute deviation, mean, median, minimum, range, robust mean absolute deviation, root mean squared, skewness, total energy, uniformity and variance.
- **Second order statistical features** illustrate the statistical correlation between a voxel and its neighboring voxels, addressing texture information. That is, second order features describe the heterogeneity of the tumour. In order to extract the texture features, matrices like Gray-Level Co-Occurrence Matrix (GLCM) and Gray-Level Run-Length Matrix (GLRLM) are formed from the CT image. GLCM provides the probability of combined occurrence of two intensity values. From that matrix, 24 features including autocorrelation, cluster prominence, cluster shade, cluster tendency, contrast, correlation, difference average, difference entropy, difference variance, inverse difference (ID), inverse difference moment (IDM), inverse difference moment normalized (IDMN), inverse difference normalized (IDN), informational measure of correlation 1 (IMC1), informational measure of correlation 2 (IMC2),inverse variance, joint average, joint energy, joint entropy, maximal correlation coefficient (MCC), maximum probability, sum average, sum entropy and sum square can be extracted. GLRLM represents the run-length of gray level in the CT image and the 16 associated GLRLM features are gray level non uniformity, gray level non uniformity normalized, gray level variance, high gray level run emphasize, long run emphasize, long run high gray level run emphasize, long run low gray level run emphasize, low gray level run emphasize, run entropy, run length non uniformity, run length non uniformity normalized, run percentage, run variance, short run emphasize, short run high gray level run emphasize and short run low gray level run emphasize.
- **High-order statistical features** aim to characterize repeated or nonrepetitive potential patterns inside the tumour region. For this purpose, images are decomposed using different scale-space transformations, like wavelets and Laplacian-of-Gaussian (LoG) pyramids. After the transformation, first and second-order statistical features are extracted from the decomposed images.

We also refer the reader to [20] for further information. In our work, as the the GLCM matrix is symmetrical, sum average feature under the GLCM category

is 2 times of joint average feature. So, the total number of first and second order statistical features is 57 (18+23+16). Higher-order statistical features are extracted from 8 decomposed images using wavelet decomposition and 8 order reconstructed images employing RHFMs. Therefore, we analyzed a total of 456 wavelet and the same number of RHFMs features in terms of reliability, stability and prognostic value.

3.2 Datasets and Data Analysis

In this study, three lung CT datasets are considered, involving non-small cell lung cancer (NSCLC) patients:

- **The RIDER test/retest dataset** [2] provides blind delineations to 32 patients of the RIDER Lung CT dataset where the delineations of three patients are not perfectly provided [22]. RIDER Lung CT consists of same day repeat scans, where two CT scans were acquired from each patient within 15 minutes. In this study, this dataset is used to assess the reliability of the features.
- **The multiple delineation dataset** [2] consists of lung CT scans of 21 patients which were manually delineated by five oncologists independently. As one patient has no manual delineation, twenty CT images of patients are used to assess the feature stability.
- **The Lung1 dataset** [2] consists of lung CT scans of 422 patients, together with manual delineations, clinical, survival data and gene profiles. It has been observed that three out of 422 patients have absent of proper masks. Thus, the radiomics features are extracted from 419 patients. This dataset is used to assess the prognostic value of the radiomic features.

3.3 Image Reconstruction using RHFM

In this study, we propose to extract higher order radiomic features based on RHFM reconstruction. The theoretical background, and the RHFM decomposition and reconstruction method are described below.

3.3.1 Radial Harmonic Fourier Moments

Following [16], consider the radial harmonic basis function H_{pq} ($p \in \mathbb{N},\ q \in \mathbb{Z}$), defined in polar coordinates as

$$H_{pq}(r,\varphi) := R_p(r)e^{iq\varphi} \qquad (r \in [0,1],\ \varphi \in [0,2\pi))\,, \tag{1}$$

where

$$R_p(r) := \begin{cases} 1/\sqrt{r}, & \text{if } p = 0, \\ \sqrt{2/r}\cos(\pi p r), & \text{if } p \text{ is even}, \\ \sqrt{2/r}\sin\left(\pi(p+1)r\right), & \text{if } p \text{ is odd}. \end{cases}$$

Radial harmonic basis functions form a complete orthonormal system in the space of the square integrable functions over the unit disk (i.e., in $L^2(\mathbb{D})$) with respect to the usual scalar product

$$\langle F, G\rangle := \frac{1}{2\pi}\int_0^{2\pi}\int_0^1 F(r,\varphi)G^*(r,\varphi)r dr d\varphi \qquad \left(F, G \in L^2(\mathbb{D})\right). \tag{2}$$

A grayscale image, represented as a function $f \in L^2(\mathbb{D})$ over the unit disk, can be decomposed into a series expansion as

$$f(r,\varphi) = \sum_{p=0}^{+\infty}\sum_{q=-\infty}^{+\infty} M_{pq}H_{pq}(r,\varphi) \qquad (r \in [0,1],\ \varphi \in [0,2\pi)),$$

where

$$M_{pq} := \langle f, H_{pq}\rangle = \frac{1}{2\pi}\int_0^{2\pi}\int_0^1 f(r,\varphi)H^*_{pq}(r,\varphi)r dr d\varphi$$

is the radial harmonic Fourier moment (RHFM) of order $p \in \mathbb{N}$ and repetition $q \in \mathbb{Z}$.

3.3.2 Image Decomposition

The real application of RHFMs involves discretization and the restriction of the input image to the unit disk. Here, we followed the discretization and numerical integral approximation proposed in [3]. Normally, planar images are organized as the matrix of pixels over rectangular coordinate system, where the coordinate of the left top corner pixel is $(0, 0)$. Consider a grayscale image $f(x, y)$, and assume that it is square of size $N \times N$. As the RHFM is based on the polar coordinate system, an inner unit circle is inscribed over the grayscale image. Then, the origin of the image is moved to the center of the inscribed circle, and the central coordinate of all pixels of the input image $f(x_i, y_j)$ are computed as

$$x_i = \frac{2j - N + 1}{N}, \quad y_j = \frac{N - 1 - 2i}{N} \qquad (i, j = 0, 1, 2, ..., N - 1). \tag{3}$$

After mapping, the new Cartesian coordinates (x_i, y_j) over the inscribed inner circle are transformed into the polar coordinate (r_{ij}, θ_{ij}) as

$$r_{ij} = \sqrt{x_i^2 + y_j^2}, \quad \theta_{ij} = \operatorname{atan2}(y_j, x_i) \qquad (i, j = 0, 1, 2, ..., N - 1), \tag{4}$$

where $\operatorname{atan2}(y, x) = \arg(x+\imath y)$ denotes the two-argument arctangent. The RHFMs M_{pq} of the image $f(r_{ij}, \theta_{ij})$ with order $p \in \mathbb{N}$ and repetition $q \in \mathbb{Z}$ over a unit inner circle are computed employing the discrete representation of (2) as of

$$M_{pq} = \frac{2}{\pi N^2}\sum_{i=0}^{N-1}\sum_{j=0}^{N-1} f(r_{ij}, \theta_{ij})H^*_{pq}(r_{ij}, \theta_{ij}). \tag{5}$$

Note that the radial factor in (1) depends on the radius of the pixel located in (x_i, y_j) coordinate and the order p of that radius, and the exponential factor relies on the angular distance from x-axis of the pixel and its repetition q. Assuming that the maximal order and repetition of the computed moments are $p_{max} \in \mathbb{N}$ and $q_{max} \in \mathbb{Z}$, respectively, the total number of RHFMs is $(1 + p_{max}) \times (1 + 2q_{max})$.

3.3.3 Image Reconstruction

In this paper, we extract radiomics features using low-level reconstructions using RHFMs. To this order, consider the partial reconstruction of order $p_{max} \in \mathbb{N}$ and repetition $q_{max} \in \mathbb{Z}$ as

$$f(r,\varphi) \approx \widehat{f}(r,\varphi) := \sum_{p=0}^{p_{max}} \sum_{q=-q_{max}}^{q_{max}} M_{pq} H_{pq}(r,\varphi) \qquad (r \in [0,1],\ \varphi \in [0,2\pi)) \,. \quad (6)$$

The reconstructions represent low-dimensional approximations of the input image based on the transformation-invariant moment decomposition.

3.4 Proposed Method

Based on the above motivations and background information, we propose RHFM reconstructions for high-order radiomic feature extraction. Orthogonal moments provide efficient frequency-space representations that can capture high-level patterns on the image, and have further beneficial properties like transformation-invariance. These make them promising replacements of the wavelet-based high-order features suggested in [2] and [21]. In particular, RHFMs show better numerical stability compared to other constructions, and perform well for small noisy images, desired for CT tumour images. In the following, we summarize the proposed algorithms, and provide further insight and justification.

3.4.1 Image Representation

The key part of the proposed method is the representation of the CT tumour image using RHFM. The overall steps are summarized in the Algorithm 1. The algorithm takes four inputs: the 3D CT image (I), its 3D mask (M), and the maximal order (p) and repetition (q) for RHFM decomposition and reconstruction; and returns the reconstructed image based on the RHFM representation.

We note that the representation is performed on the raw CT images without any filtering or resampling. The only preprocessing step is the correction, cropping, and squaring the CT image and tumour mask in order to handle the inconsistencies of the target databases regarding pixel size, spacing, and origin. A sample of cropped CT tumour image is shown in Fig. 1.

3.4.2 Feature Selection

In the next phase, we optimized feature selection, where we investigated the optimal parameters of RHFM decomposition and reconstruction. To this order, we

Algorithm 1 Proposed Algorithm for Image Representation using RHFM.

Funct RECON(I, M, p, q)

1: Remove inconsistency by correcting, cropping and squaring the image I and the mask M with $N \times N$ size of each slice
2: Combine the 3D image I and the mask M into a new 3D image C
3: Shift the intensity range of the image C to [0, range of intensity]
4: Map each Cartesian coordinate (x, y) into the central coordinate (x_i, y_j) over inscribed circle as of (3) and compute the polar coordinate (r_{ij},θ_{ij})
5: Select the list of orders $P = [0, 1, ..., p]$ and the list of repetitions $Q = [-q, -q + 1, ..., q - 1, q]$
6: Compute the radial harmonic basis functions $H_{pq}(r_{ij}, \theta_{ij})$ being a matrix of size $((p+1) \times (2q+1) \times N \times N)$, where p and q is an order and repetition from P and Q, respectively
7: Setup the output array $\widehat{C}$ as shape of image C
8: **for** each slice $f(x, y)$ in C **do**
9: Compute matrix of moments M_{pq} of size $((p + 1) \times (2q + 1))$ by performing the inner product of the image $f(x, y)$ and the radial harmonic basis function $H_{pq}(r_{ij}, \theta_{ij})$ as of (5)
10: Compute the reconstructed slice $\widehat{f}(x, y)$ by the partial sum of radial harmonic basis functions $H_{pq}(r_{ij}, \theta_{ij})$ weighted by moments M_{pq} as of (6)
11: Store the reconstructed slice $\widehat{f}(x, y)$ in $\widehat{C}$
12: **end for**
13: Clip and shift back the intensity of image $\widehat{C}$ over the range of image C
14: **return** the reconstructed image $\widehat{C}$

evaluated the error in terms of mean squared reconstruction error of the form

$$MSRE = \frac{\sum_{x=0}^{N-1} \sum_{y=0}^{N-1} \left| f(x, y) - \widehat{f}(x, y) \right|^2}{\sum_{x=0}^{N-1} \sum_{y=0}^{N-1} |f(x, y)|^2}. \tag{7}$$

Fig. 2, 3, and 4 present the average MSRE of the three datasets. In this evaluation, we set the order and repetition of RHFM to be the same, i.e., $p = q$, as of [16]. Furthermore, we investigated even orders only in order to include both the sine and cosine radial functions of the same magnitude. The analysis show that the representation is optimal around order 10, which behaviour is following the theoretical expectations: when the order is too low, then the error is high due to the loss of detail information on the image, and when the level is too high, then numerical errors arise due to noise and discretization artifacts. In conclusion, we propose to select the eight orders around 10 for CT image representation, namely we suggest the list of $[2, 4, 6, 8, 10, 12, 14, 16]$.

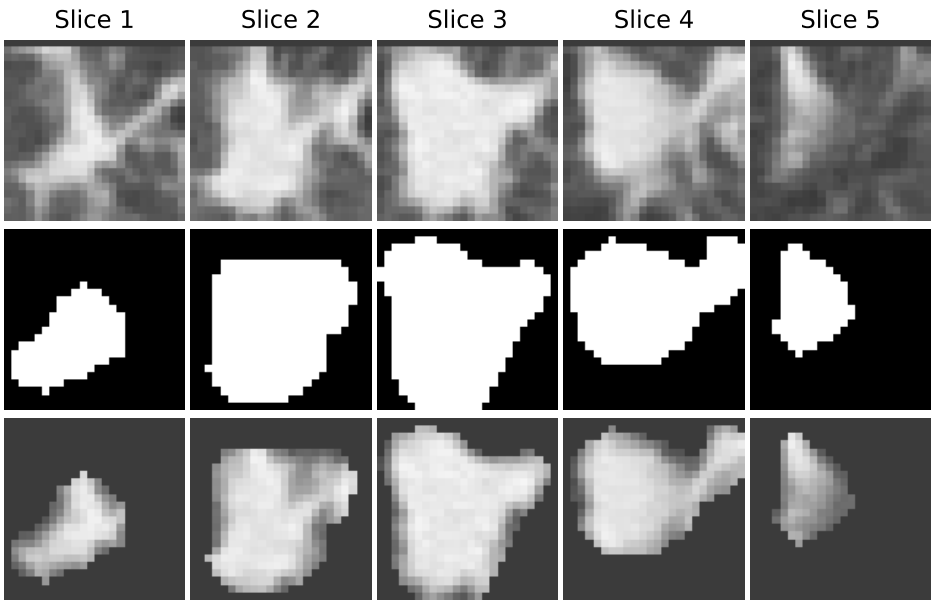


Figure 1: A sample of preprocessed CT Image of Lung1 Dataset. Rows: original CT slices (cropped, squared), tumour masks, and masked tumour images.

3.4.3 Feature Extraction

We propose radiomics feature extraction following the methodology of [2], and utilizing the lower-order features of [21]. There, one step of 3D stationary wavelet decomposition was utilized using Coiflet 1, resulting eight image representations for high-order feature extraction. Instead, we propose eight RHFM reconstructions here in the above mannner. Then, 18 first-order statistical, 23 GLCM and 6 GLRLM-related features are extracted from those reconstructed images, as introduced above. Totally, we investigated 456 RHFM-based features, which we compared to the same amount of wavelet-based features as of [2]. The low-order features were extracted using the pyradiomics package [21].

As a comparison, the eight wavelet decompositions of the CT image shown in Fig. 1 are illustrated in Fig. 5 and 6, and eight RHFM reconstructions are shown in Fig. 7 and 8.

4 Results and Discussion

In this study, the extracted 456 radiomics features of lung cancer are analyzed to evaluate their reliability, stability, and prognostic power, following the workflow proposed in [2]. For this reason, statistical tests are applied to determine scores

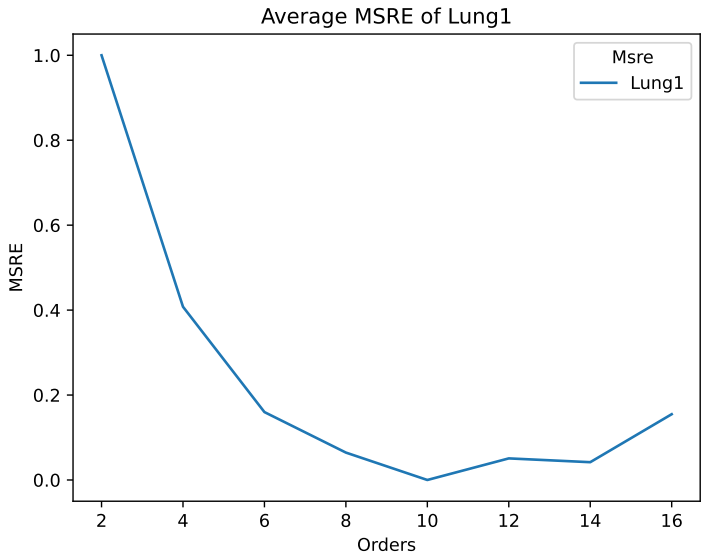


Figure 2: Average MSRE of Lung1 Dataset

for the reliability, stability and prognostic value of those extracted features. We explain and evaluate the score below, and compare the proposed method to the wavelet-based approach in [2].

4.1 Reliability Analysis

In conservative medicine, the reliability of numerical measurements of patients are always intended to evaluate for making decisions. Reliability implies the assessment of the reproducibility of the numerical measurements over the same set of patients. Thus, it emphasizes not only the correlation but also the agreement between measurements. Mathematically, reliability is the ratio of true variance over the true variance plus the error variance of measurements. The score of the reliability ranging from 0 to 1 can be evaluated by Pearson correlation coefficient, paired t-test and Bland-Altman plot. Those approaches are not ideal for reliability analysis, as Pearson correlation coefficient focus only on the correlation while paired t-test and Bland-Altman plot emphasize only on the agreement. Then, intraclass correlation coefficient (ICC) first introduced by Fisher in 1954 is widely used for reliability analysis. The ICC score representing both correlation and agreement between measurements is the index to quantify the reliability. The conservative care practitioners usually perform three types of reliability: interrater, intrarater and test-retest reliability. In interrater reliability, ICC score is based on the measurements taken by different raters over same patients, whereas same raters take measurements on same patients through one or more trial in intrarater reliability.

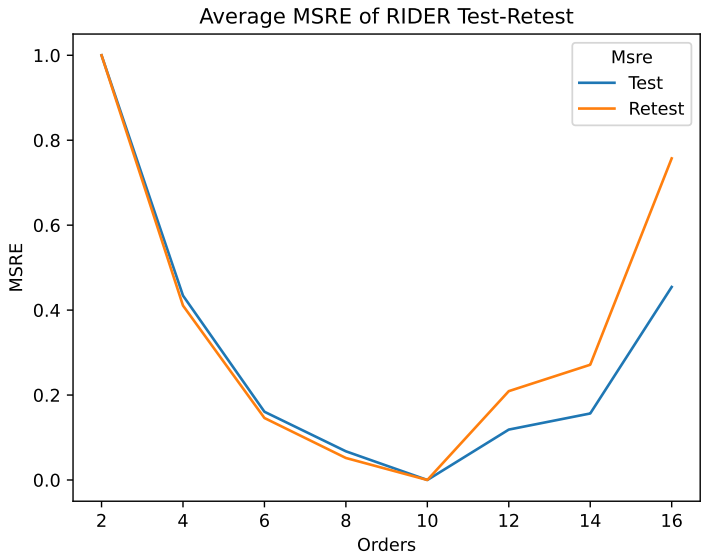


Figure 3: Average MSRE of Test-Retest Dataset

In the test-retest reliability, the measurements of the same subjects are taken by same instruments under same conditions at different time. In this study, the retest CT images of 31 subjects are taken after a 15 minutes tea break of taking the test CT image of the same patients in RIDER test-retest dataset. Therefore, the test-retest reliability is suitable for our study. There is no standard acceptable ICC value for reliability and the only expected value of ICC is the true ICC estimate. For this reason, the level of reliability is applied to determine for testing if the obtained ICC exceeds in statistical inference. Koo and Li suggested four levels of reliability in his guidelines for ICC reporting: Poor Reliability($ICC < 0.5$), Moderate Reliability($ICC \geq 0.5$ and $ICC < 0.75$), Good Reliability ($ICC \geq 0.75$ and $ICC < 0.9$) and Excellent Reliability($ICC \geq 0.9$ and $ICC \leq 1$) [9]. This guideline for level of reliability has been used in this research where the ICC score of each feature is determined by employing intraclass correlation coefficient approach over two samples of each feature coming from test and retest CT scans. Table 1 shows the distribution of the number of RHFM and wavelet features among those groups (poor, moderate, good and excellent) to the usual guidelines. The table implies that the RHFM features show better reliability than wavelet features.

4.2 Stability Analysis

In clinical oncology, the tumor region of CT image of patients is delineated by multiple radiation oncologists for stability analysis. Multiple samples including desired extracted radiomics features from delineated CT images are compared to test if

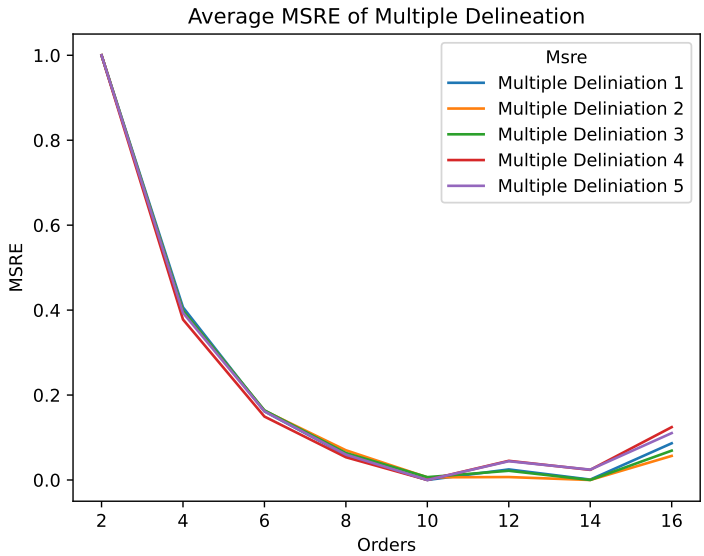
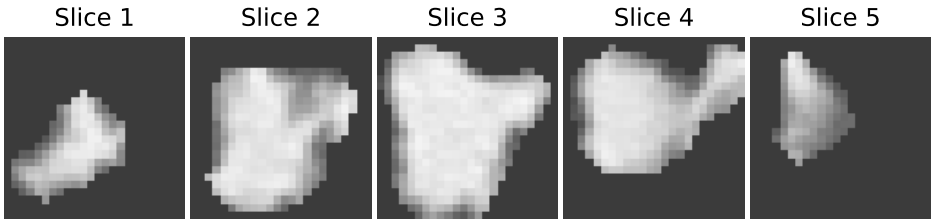


Figure 4: Average MSRE of Multiple Delineation Dataset

Table 1: Reliability: number of features based on ICC

Type of Features	**Poor**	**Moderate**	**Good**	**Excellent**
RHFM	22	105	175	154
Wavelet	38	89	157	172

there are any significant difference among the means of those samples. As those multiple samples are dependent, Repeated-Measures ANOVA and Friedman Test are used for the statistical test. As Repeated-Measures ANOVA test is parametric test, the assumptions of samples are to be normally distributed. If the assumptions of normality are not met, the Friedman test is used [5]. In this study, RIDER Multiple delineation data consisting of the CT scans of 21 patients delineated by five radiation oncologists are used to test stability. The Friedman test is applied over five samples of each feature avoiding the distribution of those sample. The test provides the p-value which is a probability that measures the evidence against the null hypothesis. That is, if the p-value of a feature is greater than the usual significance level of 5%, the feature is considered to be stable. Otherwise, the feature is not stable. It has revealed that there are 74 and 135 stable features for RHFMs reconstruction and wavelet decomposition, respectively, at a 5% significance level. The test shows that the RHFMs features and wavelet features are comparable in terms of stability.



(a) Original CT Image

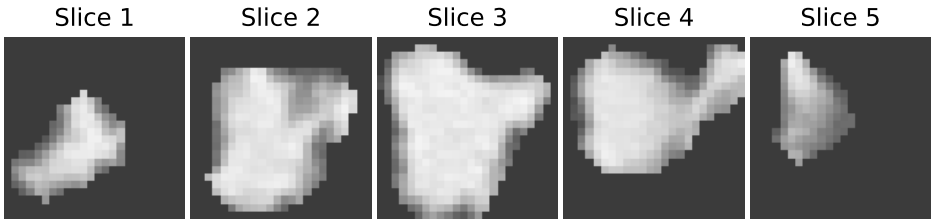
(b) LLL Wavelet Image

(c) LLH Wavelet Image

(d) LHL Wavelet Image

(e) LHH Wavelet Image

Figure 5: Wavelet Decomposed Images (LLL, LLH,LHL,LHH)

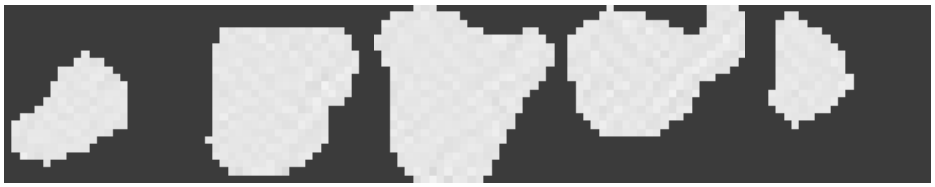


(a) Original CT Image

(b) HLL Wavelet Image

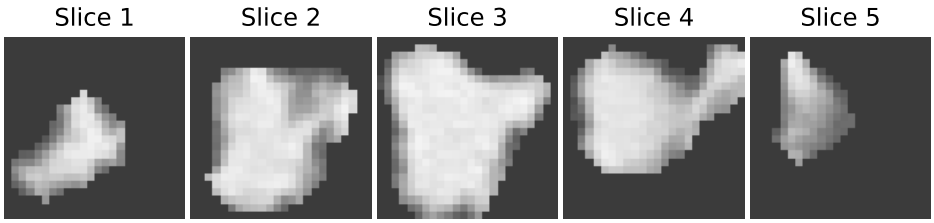
(c) HLH Wavelet Image

(d) HHL Wavelet Image

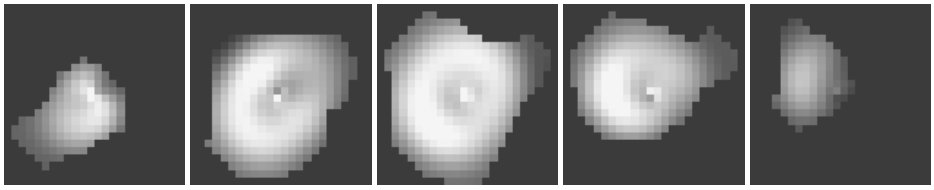


(e) HHH Wavelet Image

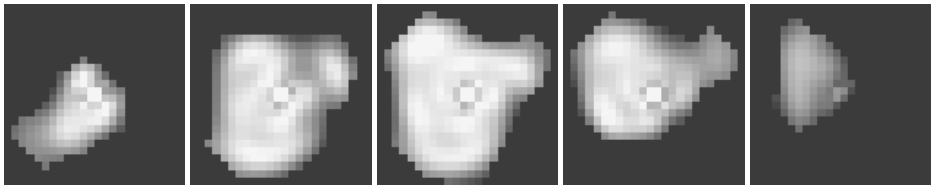
Figure 6: Wavelet Decomposed Images (HLL,HLH,HHL,HHH)



(a) Original CT Image



(b) Order 2 (MSRE: 0.163)

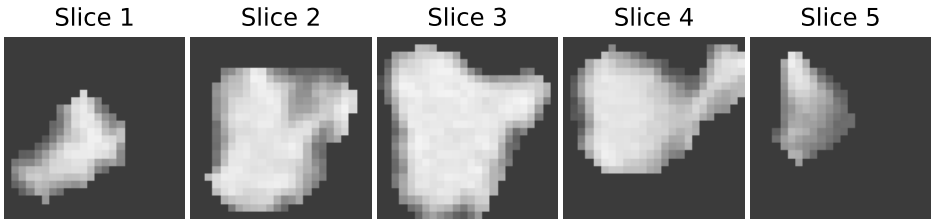


(c) Order 4 (MSRE: 0.072)

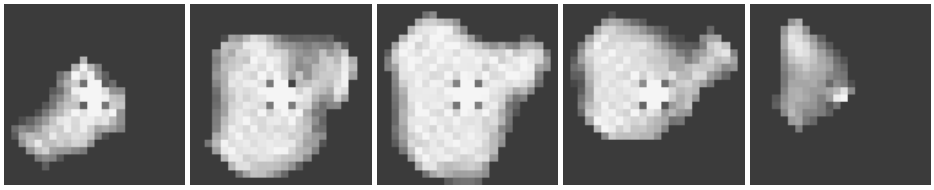
(d) Order 6 (MSRE: 0.057)

(e) Order 8 (MSRE: 0.055)

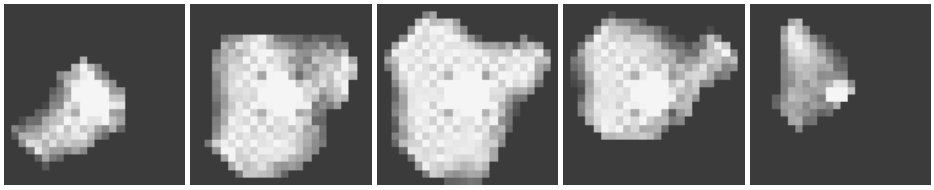
Figure 7: Reconstructed Images from RHFMs (order 2 to 8)



(a) Original CT Image



(b) Order 10 (MSRE: 0.058)



(c) Order 12 (MSRE: 0.061)

(d) Order 14 (MSRE: 0.071)

(e) Order 16 (MSRE: 0.086)

Figure 8: Reconstructed Images from RHFMs (order 10 to 16)

4.3 Prognosis Analysis

Generally, prognosis refers to the expected course and outcomes of a disease over time. In medical science, healthcare professionals make the assessment considering the likely course of a condition and its potential results based on their scientific knowledge, clinical experience and the circumstance of individual patient. This assessment helps the doctor to guide treatment decisions and set expectations for recovery or disease progression. In clinical oncology, the valid and prominent question is which radiomics features of radio images are correlated with the prognosis. The answer to this question is solved through the survival analysis which predicts the time to death by establishing a connection between radiomics features and the time to death. The difference of survival analysis from the traditional regression model is that the survival model can work on the partially observed data called censored data. A popular robust mathematical model of survival analysis is Cox proportional hazard model which is expressed in terms of hazard model formula as

$$h(t, X) = h_0(t)e^{\sum_{i=1}^{p} \beta_i X_i}. \tag{8}$$

That is, Cox survival model is the product of two quantities: $h_0(t)$, baseline hazard function and exponential expression of the linear sum of $\beta_i X_i$ for p radiomics features X_i. Here the hazard means the probability of death. The Cox proportional hazard model does not assume about the distribution for the outcome variable (time to death), but it assumes that the hazard proportion between different subjects is constant over time [8]. The assumption helps to estimate the regression coefficient β_i without considering the full hazard function. This Cox proportional hazard regression model can be used to assess the radiomics features by measuring the predictive discrimination ability [6]. Due to the presence of censored data, this assessment is performed by Concord index (C-index or CI) proposed by Harrell et al in 1982 [6]. The C-index is the measure of how well the patients are sorted according to the event occurrence. The index explains the ability of a radiomics feature to order subjects by estimating the proportion of correctly ordered pairs among all usable pairs in the dataset where the patient pairs have at least one died patient. In our study, as only Lung1 dataset has clinical data having survival time and death status, its four 419 patients are considered to measure the prognostic power through C-index for each feature of 456 radiomics features. In our experiment, 378 RHFM and 430 wavelet features are above 0.5 and thus show prognostic value. The mean and median CI is approximately 0.54 and 0.55 for both the RHFM and wavelets, which proves a similar prognostic value.

5 Conclusion and Future Work

We investigated the application of orthogonal moments known as RHFMs for radiomics analysis of lung CT images of NSCLC patients, and compared the extracted RHFMs features with state-of-the-art wavelet features. We proposed a reconstruction framework for RHFM-based tumour representation, and optimized the feature

selection by the analysis of reconstruction error. Statistical tests were performed to determine the stability, reliability, and prognostic value of the proposed features, which aspects play important roles in clinical oncology. We conclude that orthogonal moments are promising for radiomics analysis, since they show comparable behavior compared to wavelets, while they are more flexible, possibly adaptive, and preferable due to their shift-invariant property. In the future, we plan to further investigate the application of orthogonal moments in radiomics in several respects. This includes the adoption of various orthogonal bases, adaptive transformations, and discretization approaches. We plan to extend the RHFM model to 3D, and also direct utilization of transformation invariant moments as radiomic features. Here the feature selection was optimized for the whole databases, but the optimal order is worth to be explored in a patient-wise adaptive manner as well. Furthermore, we will extend our work to build a radiomics signature for associating reliable, stable and prognosis radiomics features and gene expression profiles.

References

[1] Abu-Mostafa, Y. S. et al. Recognitive Aspects of Moment Invariants. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-6(6):698–706, 1984. DOI: 10.1109/TPAMI.1984.4767594.

[2] Aerts, H. et al. Decoding tumour phenotype by noninvasive imaging using a quantitative radiomics approach. *Nature Communications*, 5:4006, 2014. DOI: 10.1038/ncomms5006.

[3] Chun-peng, W. et al. Geometrically invariant image watermarking based on fast Radial Harmonic Fourier Moments. *Signal Processing: Image Communication*, 45:10–23, 2016. DOI: 10.1016/j.image.2016.03.007.

[4] Coroller, T. P. et al. CT-based radiomic signature predicts distant metastasis in lung adenocarcinoma. *Radiotherapy and Oncology*, 114(3):345–350, 2015. DOI: 10.1016/j.radonc.2015.02.015.

[5] Efthymia, N. *Osteoarchaeology: A Guide to the Macroscopic Study of Human Skeletal Remains*. Academic Press, 2016. ISBN: 9780128040973.

[6] Harrell, F. E. et al. Evaluating the yield of medical tests. *JAMA*, 247(18):2543–2546, 1982. DOI: 10.1001/jama.1982.03320430047030.

[7] Hu, M.-K. Visual pattern recognition by moment invariants. *IRE Transactions on Information Theory*, 8(2):179–187, 1962. DOI: 10.1109/TIT.1962.1057692.

[8] Kleinbaum, D. G. et al. *Survival Analysis: A Self-Learning Text*. Statistics for Biology and Health. Springer New York, New York, NY, 2012. DOI: 10.1007/978-1-4419-6646-9.

[9] Koo, T. K. et al. A Guideline of Selecting and Reporting Intraclass Correlation Coefficients for Reliability Research. *Journal of Chiropractic Medicine*, 15(2):155–163, 2016. DOI: 10.1016/j.jcm.2016.02.012.

[10] Lambin, P. et al. Predicting outcomes in radiation oncology–multifactorial decision support systems. *Nature Reviews. Clinical Oncology*, 10(1):27–40, 2013. DOI: 10.1038/nrclinonc.2012.196.

[11] Li, R. et al. *Radiomics and Radiogenomics: Technical Basis and Clinical Applications*. CRC Press, 2019. ISBN: ISBN: 9780367779580.

[12] Liu, Y. et al. Accurate quaternion radial harmonic fourier moments for color image reconstruction and object recognition. *Pattern Analysis and Applications*, 23:1–17, 2020. DOI: 10.1007/s10044-020-00877-6.

[13] Pereira, G. C. et al. The Role of Imaging in Radiation Therapy Planning: Past, Present, and Future. *BioMed Research International*, 2014:e231090, 2014. DOI: 10.1155/2014/231090.

[14] Ping, Z. et al. Image description with Chebyshev–Fourier moments. *JOSA A*, 19(9):1748–1754, 2002. DOI: 10.1364/JOSAA.19.001748.

[15] Prokop, R. J. et al. A survey of moment-based techniques for unoccluded object representation and recognition. *CVGIP: Graphical Models and Image Processing*, 54(5):438–460, 1992. DOI: 10.1016/1049-9652(92)90027-U.

[16] Ren, H. et al. Multidistortion-invariant image recognition with radial harmonic Fourier moments. *Journal of the Optical Society of America. A, Optics, Image Science, and Vision*, 20(4):631–637, 2003. DOI: 10.1364/josaa.20.000631.

[17] Sheng, Y. et al. Orthogonal Fourier–Mellin moments for invariant pattern recognition. *Journal of the Optical Society of America A*, 11(6):1748–1757, 1994. DOI: 10.1364/JOSAA.11.001748.

[18] Tang, V. H. et al. Wavelet radiomics features from multiphase CT images for screening hepatocellular carcinoma: Analysis and comparison. *Scientific Reports*, 13(1):19559, 2023. DOI: 10.1038/s41598-023-46695-8.

[19] Teague, M. R. Image analysis via the general theory of moments. *Journal of the Optical Society of America (1917-1983)*, 70:920, 1980. DOI: 10.1364/JOSA.70.000920.

[20] Tian, J. et al. *Radiomics and Its Clinical Application: Artificial Intelligence and Medical Big Data*. Academic Press, 2021. ISBN: 978-0-12-818102-7.

[21] Van Griethuysen, J. J. M. et al. Computational Radiomics System to Decode the Radiographic Phenotype. *Cancer Research*, 77(21):e104–e107, 2017. DOI: 10.1158/0008-5472.CAN-17-0339.

[22] Zhao, B. et al. Evaluating variability in tumor measurements from same-day repeat CT scans of patients with non-small cell lung cancer. *Radiology*, 252(1):263–272, 2009. DOI: 10.1148/radiol.2522081593.

Acta Cybernetica **27** (2025) 197–219.

Smart Contract in the Loop: Fault Impact Assessment for Distributed Ledger Technologies*

Bertalan Zoltán Péter[ab], Zsófia Ádám[ac], Zoltán Micskei[ad], and Imre Kocsis[ae]

Abstract

Due to their decentralized and trustless nature, blockchain and distributed ledger technologies are increasingly used in several domains, including critical applications. The behavior of such blockchain-integrated systems is typically driven by smart contracts. However, smart contracts are application-specific software and may contain faults with severe system-level impacts. This is especially true in the case of the extensively used Hyperledger Fabric (HLF) platform, where smart contracts are written in general-purpose languages (Java, among others), and applications can go far beyond handling virtual-currency-like assets. In this work, we present a novel formal-verification-based approach to smart contract verification and a high-level empirical model of the HLF platform. Our *Smart Contract in the Loop (SCIL)* method uses a model checker (Java Pathfinder) to check whether specific error properties hold for a given smart contract, while a predefined combination of platform-level fault modes is active. We facilitate the checking of HLF smart contracts without modification and enable the propagation or non-propagation of platform faults through the smart contracts to the system failure level.

Keywords: distributed ledger technology, blockchain, formal verification, model checking, Java Pathfinder, Hyperledger Fabric

1 Introduction

Distributed ledger technologies (DLTs) – especially blockchains – provide high-integrity distributed databases without requiring a trusted party. Initially developed with financial applications in mind and powering cryptocurrencies, blockchain

*This paper was supported by multiple programs detailed in the Acknowledgments section.

[a]Critical Systems Research Group, Department of Artificial Intelligence and Systems Engineering, Faculty of Electrical Engineering and Informatics, Budapest University of Technology and Economics; Műegyetem rkp. 3, H-1111 Budapest, Hungary

[b]E-mail: bpeter@edu.bme.hu, ORCID: 0000-0002-5577-1369

[c]E-mail: adamzsofi@edu.bme.hu, ORCID: 0000-0003-2354-1750

[d]E-mail: micskei.zoltan@vik.bme.hu, ORCID: 0000-0003-1846-261X

[e]E-mail: kocsis.imre@vik.bme.hu, ORCID: 0000-0002-2792-3572

DOI: 10.14232/actacyb.312501

technology now has a variety of use cases, including supply chain management, healthcare, and telecommunication.

Blockchains & Smart Contracts Blockchains have powerful properties, such as immutability, distribution, decentralization, and high security that make them fit for cross-organizational (enterprise) applications. Where high integrity is paramount, they are already widely used, even in critical applications; e.g., in the nuclear [9] or the railway [14] industry (although, importantly, not in safety-critical functions). Typically, such use cases are backed by permissioned platforms, such as R3 Corda [12] or Hyperledger Fabric [1], but Ethereum [5] can also power permissioned networks. However, where other extra-functional properties, such as timeliness, age-of-information, dependability, or availability are also matters of concern, the system-level analysis of critical applications is still largely an open challenge.

Smart contracts, introduced with Ethereum [5], are akin to stored procedures and describe computations executed on the blockchain with effects that are persisted on-chain. They extend the original accounting "ledger" functionality of permissionless blockchains with rich, self-executing business logic. Smart contracts have since become ubiquitous and are widely used in most blockchain frameworks, enabling decentralized collaboration among the participants.

However, smart contracts are pieces of software and thus susceptible to faults with potentially devastating consequences. The beneficial properties of blockchains may also pose some issues; e.g., even if a bug is found, the ledger's immutability inherently prevents fault removal. Because of these risks, verifying smart contracts has been a central research topic in recent years, bringing about several approaches for fault removal and prevention [22]. These are mostly design-time methods, such as static analysis, and primarily target Ethereum and the Solidity programming language.

Hyperledger Fabric (HLF) [1] is a widely used, mature, enterprise-grade permissioned blockchain platform maintained by Linux Foundation Decentralized Trust (LFDT). It offers pluggable consensus mechanisms, identity management, flexible "subnetting" features, and privacy mechanisms. Fabric powers several projects in both development and production in various domains[1]. In HLF, the network *must* have smart contracts (called "chaincode") for any meaningful transactions to be able to occur.

Lack of Cross-Organizational V&V Support There is significantly less support for verifying enterprise smart contracts, even though recent developments show that the Ethereum Virtual Machine (EVM) is no longer the only available smart contract execution environment; known alternatives include:

- WebAssembly (WASM) [19] (used by Polkadot [23])
- the Berkeley Packet Filter (BPF) [15] VM (used by Solana [24])
- the Move [2] VM (used by the Aptos Blockchain [2])

[1] See the use case tracker at https://www.hyperledger.org/learn/use-case-tracker/.

Enterprise smart contracts necessitate developing different techniques from its public counterparts for several reasons, such as the usage of general-purpose programming languages instead of domain-specific ones to write smart contracts or additional variable features that have to be taken into account in the enterprise case, such as deployment. Further complications arise from the fact that enterprise solutions are often not openly available, lowering the number of available case studies and evaluations, thus hindering research efforts in this area.

One cannot follow the same methodologies for the verification and validation (V&V) of cross-organizational platforms and smart contracts that are already widely available in the literature. The main reason is that while in public platforms such as Ethereum, a canonical set of platform events and relevant attacks can be defined, there is much more variability in these aspects on consortial networks. We explain this notion in detail in Section 2.

Nevertheless, some formal verification approaches can be employed to verify smart contracts both in public and consortial applications [11, 3]. However, to our knowledge, no tooling enables the impact assessment of platform-level faults given an unmodified smart contract implementation, especially for enterprise platforms such as HLF – even though the differences in deployment and the platform greatly influence the possible fault modes. To take all of that into account, verification methods either need to divide all of these components into small parts and verify them separately, or they need to experiment with methods that can handle several of these layers together. We believe that the latter cannot be disregarded, as issues emerging from these systems as a whole must be considered.

While we have little information about cross-organizational smart contracts, as they are typically kept private, we can hypothesize that the majority of them are more or less direct translations of existing contracts from the public blockchain world. Furthermore, as the general-purpose languages used by most consortial platforms were not designed with smart contract development in mind, we also postulate that there are more types of faults to consider for these programs than for those written for the EVM. Unfortunately, we do not have a library of such common faults, as no suitable corpus of consortial contracts is available. At the same time, faults in cross-organizational smart contracts may be more consequential, as the potential damage is not limited to losses in financial assets.

Based on all this, we recognize a lack of V&V methods specialized to enterprise solutions, even though they are necessary based on both the use cases and the individual characteristics of the world of enterprise blockchain platforms.

Contributions & Paper Structure In this work, we propose the application of model checking to show whether a smart contract may develop errors in the presence of certain platform-level faults. To this end, we present a simplified model of the HLF blockchain platform with its primary components and configurable fault modes. This model implementation enables the user to define several aspects of deployment (e.g., the number of peers per organization or the channel's endorsement policy) and specify what faults or attacks can arise.

We demonstrate the viability of our approach with a Java-based prototype capable of simulating network faults in the context of a (hypothetical) safety-critical application. This prototype can be verified using the model checker Java Pathfinder (JPF) [16]. Our method provides the means for one to *plug in* their Java HLF smart contract to the framework and determine whether a predefined property holds while select platform-level faults are active. We dub this approach *Smart Contract in the Loop (SCIL)*. The prototype implementation and all other artifacts related to this paper are open-source and available online on GitHub[2].

In the next section, we briefly overview the application of formal verification to smart contracts and DLTs and what motivates this research. In Section 3, we present our model of the HLF platform, and we describe our Smart Contract in the Loop approach, followed by an overview of our prototype implementation in Section 4, and a worked out case study in Section 5. Finally, we conclude and discuss future work in Section 6.

2 V&V of Cross-Organizational Smart Contracts

Smart contracts are programs that run on blockchains, and as with any other software, they are prone to contain faults ("bugs"). Unfortunately, while traditional software can usually be patched and thus its faults can be removed, smart contracts are inherently immutable; i.e., platforms are typically unprepared to support patching or upgrading these programs because they follow an append-only paradigm. The public blockchain world quickly recognized the need for verification and validation (V&V) activities in the smart contracts development process to prevent these faults from making it into the deployed contracts, proposing several diverse approaches. That being said, cross-organizational (consortial) blockchain applications and smart contracts have unique aspects that warrant different V&V techniques. These different techniques are still largely unexplored.

Smart contract faults may result in the loss of (commonly financial) assets in permissionless systems and the cryptocurrency world (see, for example, the infamous DAO hack [7]). The potential effects are arguably far more devastating in the context of permissioned and especially critical applications. While smart contracts can be enhanced with various defenses (including runtime verification mechanisms or techniques such as n-version programming (NVP) [18]), faults of the platform itself may still induce unintended behavior.

2.1 An Overview of Smart Contract V&V Approaches

Since the initial release of Ethereum [5] and the quick recognition of the need for V&V techniques in smart contract development, hundreds of research papers have been published about various verification tools and approaches. [22] collected 202 papers that are concerned with blockchain V&V techniques in general, such as model checking, theorem proving, program verification, symbolic execution, and

[2] https://github.com/ftsrg/scil

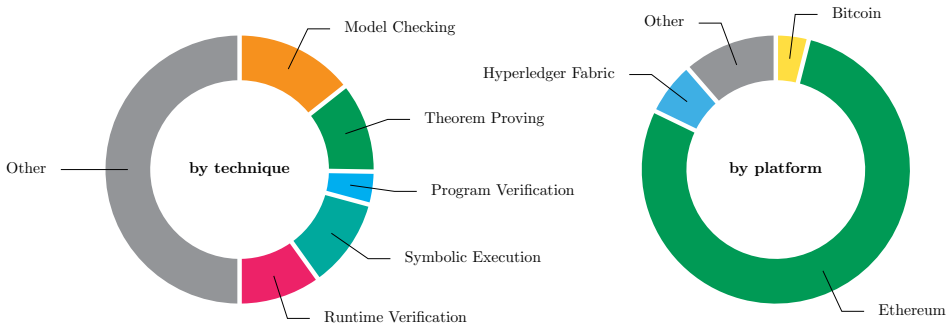


Figure 1: Distribution of V&V techniques in the underlying corpus of [22]

runtime verification. We have summarized the distribution of these techniques among the papers in Figure 1. The diagrams were created by filtering the papers listed at the website[3] created by the authors and counting the results.

It is clear from the results that there are significant research efforts towards smart contract V&V, but methods targeting Ethereum far outweigh those proposed for enterprise platforms. Indeed, Ethereum smart contracts are publicly available, and their common problems are already well-known. On the other hand, enterprise smart contracts are seldom made public, and therefore, we know much less about incidents or common faults in these programs. Furthermore, there are several key differences in cross-organizational blockchain applications that we outline in the following subsection.

2.2 Enterprise and Public Smart Contract V&V Differences

Although not immediately apparent, applications and smart contracts on cross-organizational distributed ledger technology (DLT) platforms may be radically different from their public platform counterparts. The fundamental difference is that while the relevant failure modes and effects in public platforms are fairly canonical, they are much more varied in cross-organizational DLTs. This subsection overviews the most important differences that highlight why the V&V of consortial DLTs forms a different, largely unsolved problem set.

Deployment The deployment model of a consortial DLT and a public blockchain differs. First of all, the infrastructure is typically given and available for smart contracts on permissionless blockchains. Its potential failure modes and their associated risks are known; e.g., selfish mining [10] on Ethereum [5] before The Merge[4], but similar attacks have been identified [17] for the current PoS consensus, too. Conversely, the deployment of a cross-organizational DLT is *application-*

[3] https://ntu-srslab.github.io/smart-contract-publications/

[4] Ethereum [5] switched from proof of work (PoW) consensus to proof of stake (PoS) on 2022-09-15.

dependent. It depends on the number of participating organizations, their relationships, the consensus protocol (especially the endorsement policy in Hyperledger Fabric (HLF)), and the underlying physical infrastructure, among others. Applications based on smart contracts may be affected by these parameters in unforeseeable ways.

Further, some consortial DLTs have capabilities that simply do not exist in the case of public networks. For instance, in HLF's model, smart contracts is installed independently to a number of peers in such a way that it is theoretically possible to have different implementations of the same smart contract specification installed onto different nodes within an organization or even across organizations. This idea is explained in further detail in [18].

Programming Model While there has been a recent, noticeable shift towards other programming languages and execution environments even in the public blockchain space (e.g., Rust in Solana [24], Move on Aptos [2], Python on Algorand [6], etc.), the vast majority of smart contracts have been written for the Ethereum Virtual Machine (EVM) [5] and in Solidity. The common Solidity vulnerabilities, weaknesses, and code smells are known; some examples include reentrancy, arithmetic over- and underflow, frontrunning, and access control [21]. On the other hand, smart contracts on consortial platforms are typically written in general-purpose programming languages. HLF [1], for example, currently supports Go, Java, and JavaScript. Corda [12] smart contracts are written in Java (or Kotlin).

The significance of this is twofold. First, as these languages were not developed with smart contracts in mind, we hypothesize that their usage may imply a more extensive yet unexplored set of potential software faults ("bugs"). As there is practically no publicly available corpus of enterprise smart contract written in these languages, we do not know the statistically most common problems (as opposed to contracts written in Solidity, where sizable corpora exist). Second, while Solidity is still a relatively new and unique language, extensive research has already been done regarding V&V techniques for pieces of software written in ubiquitous programming languages like Java.

Besides the language, the way the world state is stored often also differs; e.g., in HLF, the underlying database is a simple (versioned) key-value store. This greatly affects how smart contracts must be written, especially since serialization and key management issues also become matters of concern.

Execution Model Both consortial and public systems rely on consensus among participants to establish a world state agreed upon by all parties. However, the way this consensus is reached is radically different between the two types of systems.

Finality is a crucial difference; e.g., HLF offers immediate, absolute transaction finality, meaning that a transaction accepted by the network will deterministically end up in a block. Consensus mechanisms in public networks are different. In PoW, finality is *probabilistic:* as the block height grows, an accepted transaction is more and more likely to become final. In PoS, there is *economic* finality: a transaction is

final when "reversing" it would be financially infeasible due to the collateral losses of validators. As a corollary, temporary forks can form on these public platforms, but usually not on permissioned ones (like Fabric).

Another difference is how the smart contract halting problem is solved. In Ethereum [5] and its derivatives, *gas* is used for this purpose. In consortial DLTs, timeout mechanisms are employed since it often does not make sense to track money-like assets on the ledger.

Calls to external services from within smart contracts may be supported and even desired in consortial DLTs, while it is only possible through oracles in Ethereum.

Variability of Platform Events and Failure Effects Since deployment and configuration aspects need not be considered for individual applications, the expected platform-level events, attacks, and failure effects in public networks are fairly canonical and thus can be anticipated. All of these are variable regarding cross-organizational DLTs.

For example, in a HLF network, depending on the endorsement policy, the downtime of a peer may result in unintended behavior even when the smart contracts are entirely fault-free. Or, malicious behavior of a HLF channel's ordering service may also lead to issues regardless of smart contract quality. We detail an example in our case study in Section 5, where the ordering service reorders transactions (a kind of frontrunning attack), resulting in an incorrect final state of the ledger state and a real-world accident occurring as an effect.

Based on the above, we propose that due to the radically different models of execution, programming, and expected failure effects, the V&V of cross-organizational blockchain applications requires different, specialized approaches from those developed for public platforms. In fact, there are several traditional fault-tolerance techniques (such as NVP or runtime verification) that are not practically applicable to public networks (usually because of the added costs) but could be employed in consortial settings.

Because there are so many other "moving parts," one cannot rely on simple, direct V&V of smart contracts (like testing or formal, static analysis). Instead, we suggest a holistic approach where the smart contract is verifiable in the context of the entire target network, including deployment, configuration, higher-level applications (dApps) using on the smart contract as a backend, and any potential defenses. To this end, we have modeled the components of HLF to be able to run simulations. However, instead of requiring the smart contract to be modeled, we have developed a framework (for Java smart contracts) where the contract code can be plugged in *as is*. We elaborate on this framework in the Section 3.

2.3 Related Work

As shown in Subsection 2.1 there has been significantly more research on V&V techniques for public, permissionless platforms. Still, there are some papers aiming at the permissioned HLF platform, too.

In [3], the authors present an approach for the formal verification and deductive verification of HLF smart contracts using the KeY prover. They define the formal specification of Java smart contracts using Java Modeling Language (JML) that is translated into Java Dynamic Logic (JavaDL) [4] and then perform static analysis to ensure the specification's rules are fulfilled. The paper states explicitly that "verifying the correctness of the Fabric framework itself (e.g., communication between peers and orderer) [...] is not within the scope" of their work.

The BCVerifier [20] tool for HLF checks the integrity of the ledger itself to detect local modifications and to ensure transaction executions are valid. A Hyperledger Labs project[5] was created for the tool but has been since archived.

We were unable to find any such literature about other permissioned platforms, such as R3 Corda [12]. The only related research paper is [13], where the authors show a model-driven engineering (MDE) methodology that includes validation.

These existing solutions focus on specific elements of DLT-based applications, such as the smart contracts or the ledger state. The model-checking-based Smart Contract in the Loop (SCIL) approach presented in this paper is more comprehensive in the sense that it does not only verify smart contract correctness or state changes but also considers various platform-level events (faults) and potentially deployed defenses (e.g., smart contract NVP).

3 The Smart Contract in the Loop Approach

As explained in Subsection 2.2, we propose a holistic treatment of cross-organizational distributed ledger technology (DLT) systems for the verification and validation (V&V) of smart-contract-based consortial applications. Concretely, we have developed our Smart Contract in the Loop (SCIL) approach that performs *model checking* of a configurable Hyperledger Fabric (HLF) network model instance, given a smart contract and an error property to check for. We have visualized the core elements of our approach in Figure 2. At the core of our framework, there is an executable HLF model (written in Java), which we describe in the next subsection. In Subsection 3.2, we describe the further components of the SCIL framework for HLF.

3.1 Executable Model of Hyperledger Fabric

Hyperledger Fabric (HLF) [1] is a permissioned, highly configurable, modular enterprise blockchain platform maintained by Linux Foundation Decentralized Trust. Among R3's Corda [12] and Canton [8], it is among the few widely used consortial (cross-organizational) DLT platforms. Fabric is known to power several enterprise use cases[6].

Even with state-of-the-art verifiers, out-of-the-box model checking of a large project, such as the implementation of HLF, is still not feasible due to numerous

[5] https://github.com/hyperledger-labs/blockchain-verifier
[6] See footnote 1.

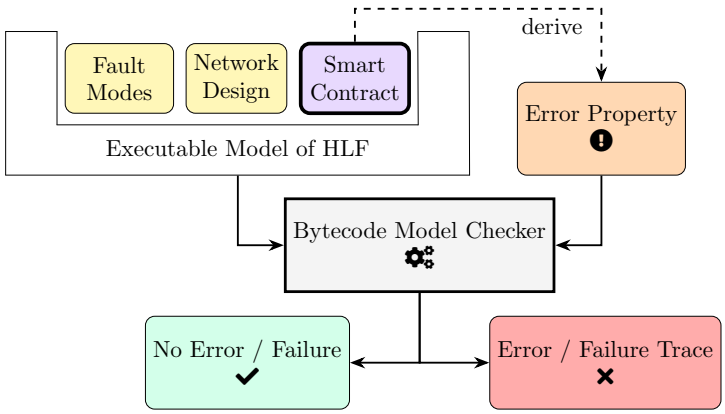


Figure 2: Verification Process

factors, such as scalability issues, libraries, and the distributed nature of the project. Therefore, we have instead created our simplified implementation-independent model of HLF with a level of abstraction that enables meaningful formal analysis but does not generate an overly complex state space.

The difficulty of this approach lies in the empirical nature of modeling the network – verification outcomes are hard to trust on an abstract model based on informal documentation and some code. We identified abstraction as a key point regarding the quality of the model; i.e., finding the right abstraction to catch all relevant aspects to the faults and the error property while keeping the model and its limitations clear.

Please refer to Figure 3 for a high-level overview of our model that incorporates both structure and dynamics (i.e., the messages between the components).

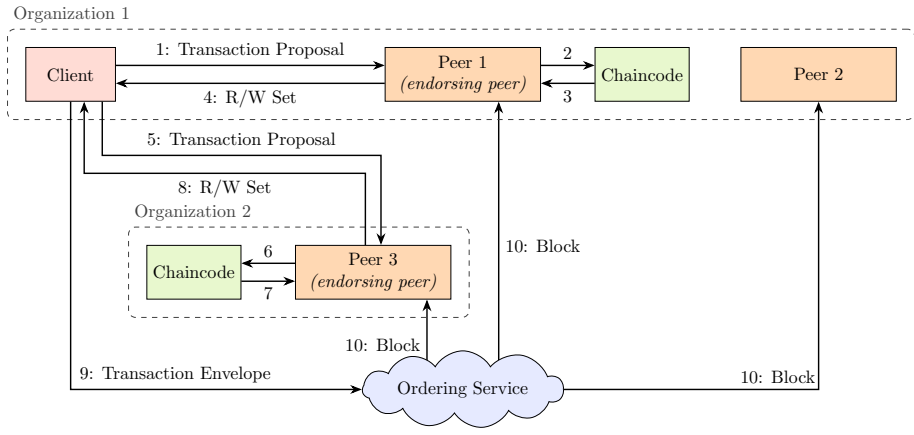


Figure 3: High-level overview of Fabric's architecture

3.1.1 Main Components

In the following, we elaborate on how the main components illustrated on Figure 3 have been modeled, including how they are mapped to the "real" HLF platform's components and their fault modes and interactions with other components.

Unlike common public blockchain platforms such as Ethereum [5], HLF is not designed to provide a de-facto network to be used by participants. Rather, a HLF network comprises several independent channels used by independent consortia. Figure 4 provides a high-level overview of this "consortial architecture."

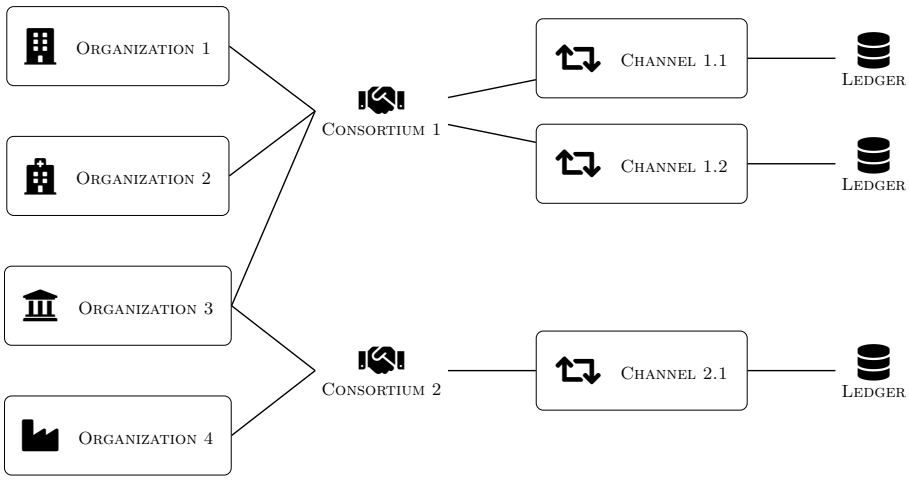


Figure 4: Overview of organizations, consortia, and channels in HLF

Organizations In HLF (and so-called "consortial" platforms and networks in general), the participants are collaborating *organizations.* Organizations form consortia, and each consortium maintains one or more channels.

Most other components, such as peers, orderers, and clients, belong to organizations. Although not explicitly modeled at this stage, it is important to mention that consortium member organizations agree on a per-channel so-called *endorsement policy* that defines the number of organizations required to *endorse* transactions for them to be accepted.

Ordering Service The ordering service is an abstraction formed by all *orderer* nodes in a channel, responsible for establishing a total order over the transactions and creating new blocks. Typically, an independent organization (or several independent organizations) provide ordering services.

We did not model individual orderer nodes at this phase, mainly due to the high complexity of the consensus mechanisms employed during ordering. We did model, however, the critical fault modes of ordering services:

1) **Dropping Transactions** An erroneously or maliciously behaving ordering service may occasionally ignore transactions, refusing their inclusion in new blocks.

2) **Reordering Transactions** Transaction reordering is usually done to perform a *frontrunning*-type attack; i.e., unfairly moving certain favored transactions ahead of others for some business advantage.

Peers Peers maintain the distributed ledgers for the channels they are in. Furthermore, peers receive and simulate client transaction requests and validate blocks published and broadcast by the ordering service.

As we have focused on ordering, we have not yet modeled fault modes of peers, but there are some ways they may misbehave – although the cause of these faulty behaviors would likely not be malicious intent. For instance, a peer may simply become unavailable, either due to issues with its physical host or network infrastructure problems. If the number of reachable peers is insufficient, clients will not be able to gather enough transaction endorsements, and desired state changes may be delayed.

Ledgers Each channel has its own *ledger,* where the world state is being stored. In HLF, the ledger is a simple key-value store. Accordingly, smart contracts (chaincode) are provided a *stub* through which they can read/write values from/to keys (but more complex operations, such as range queries, are also usually available).

Channels Channels group some peers to form a "subnet" in the HLF network with its own isolated and independent ledger. Newly created blocks are broadcast to the peers in the channel.

If endorsement policies were also modeled, they would significantly affect the behavior of the channels. Inappropriately chosen policies can have significant system-level effects – in fact, problems with endorsement policy configuration are fundamental fault modes of channels. However, in our current simplified implementation, there is no specific *endorsement* step; thus, we did not model the endorsement policy.

Application Clients Clients are the most user-facing components of the network, who submit transaction requests to peers. There may be additional logic embedded within clients, but in our current model's scope, clients can do nothing more than submit basic transactions (function names and arguments) to select peers.

Smart Contracts Smart contracts define the business logic of the cross-organizational collaboration a channel enables. In HLF 's terminology, smart contracts are typically referred to as *chaincode* – although more accurately, a piece of chaincode is a group of smart contracts.

Chaincode is installed on one or more peers in the channel. When clients submit transactions to peers, they in turn invoke the chaincode installed on them. The chaincode may read and write state (key-value pairs) from/to the ledger and return a so-called *read-write set* to the peer. The process is described in more detail in the Transaction Flow subsection.

Network The collection of all independent channels, all participating peers and orderers, the ledgers maintained by the peers, as well as the chaincode installed on them, form a HLF *network*.

3.1.2 Transaction Flow

A simplified version of HLF 's transaction flow is part of our model. Clients first submit transaction requests to endorsing peers, who respond with their endorsements and corresponding read/write sets (based on the results of the chaincode execution simulation). Then, the client sends the endorsed requests to the ordering service. Figures 3 and 5 visualize the process.

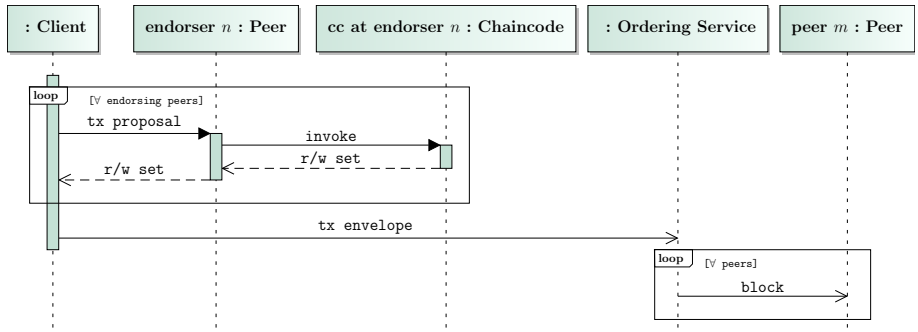


Figure 5: Transaction flow in our HLF model

3.2 Components of the SCIL Framework

We have already described our executable HLF model in detail in the previous subsection. In the SCIL framework, there are three configurable elements of the model:

1) Fault Modes

The fault modes of the individual components can be toggled before simulation. For example, in HLF, if a malicious ordering service intentionally reorders and selectively accepts (i.e., occasionally drops) transactions, ledger updates may not always reflect the expected world state. Platform-level faults in HLF include:

- malicious orderer behavior (transaction dropping, reordering)
- network faults (e.g., traffic congestion)
- host-level faults (e.g., a peer becomes unavailable)
- incorrect configuration (e.g., unsuitable endorsement policies)
- other malicious or unintentional behavior (e.g., client issues)

In a correctly configured network, Fabric protects against some of the potential faults. For example, endorsement policies can be designed to tolerate the downtime of some peers.

2) Network Design
The network design is an instantiation of the modeled components and describes the deployment of the network. This includes aspects such as how many organizations there are, how many peers do these organizations maintain, where are smart contracts installed, and what operation-time defenses have been employed.

3) Smart Contract
Our approach's core idea is to include the smart contract in the simulation as is. After defining the network and selecting the fault modes, one simply needs to *plug in* their existing smart contract code.

Error Property The error properties to consider during model checking can be derived from the smart contract and the application. This would usually be an undesired world state after a series of transactions or some erroneous results returned by the smart contract. We should note that while model checking can prove that a particular error property cannot be satisfied in a given configuration (an error state cannot be reached), it does not guarantee an utterly fault-free system.

Model Checking Given the network design, the enabled fault modes, and most importantly, the smart contract, a model checker can determine whether the specified error property can be satisfied. If so, the model checker also provides a failure trace – a list of events leading to the undesired state. If the property cannot be satisfied, we have formal proof that a certain error state is unreachable (if the initial model was correct).

The "In The Loop" Aspect We have dubbed our approach Smart Contract in the Loop because the smart contract source code is given to the model checker "as is." We do not expect smart contract developers to employ model-driven engineering (MDE) methods, and thus, a formalized model of the smart contract is likely not available. Furthermore, even if such a model exists, it is still worthwhile to test the concrete implementation in a simulation. The advantage of this approach is efficiency for the user: they simply need to provide the network configuration once, then plug in their existing smart contract implementation, and run the model checker.

4 Prototype Implementation

Our Java prototype implementation contains the Hyperledger Fabric (HLF) model described in Subsection 3.1, as well as a framework for model checking using the Smart Contract in the Loop (SCIL) approach. In the following, we describe the key elements of the prototype in more detail.

4.1 Implementation of the HLF Model

We have visually represented the most important classes of the model in Figure 6. A class exists for all major components, and there are also some additional utility classes; e.g., we package the invoked method and the arguments passed in `InvocationRequest` objects and the results returned from smart contracts (more accurately, chaincode) in `InvocationResult` instances. The latter includes the arbitrary result returned by the smart contract method and the read/write set resulting from the transaction simulation.

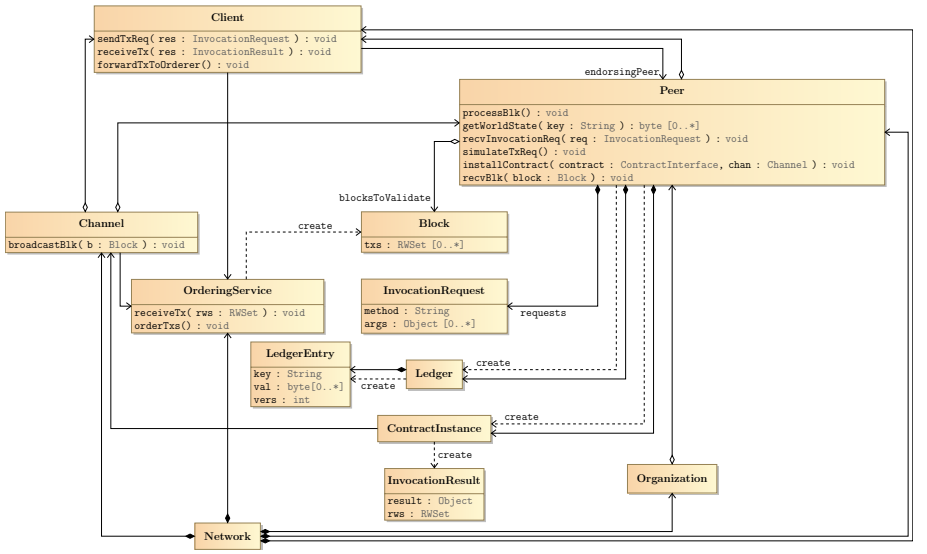


Figure 6: Simplified class diagram of the prototype

Most of Figure 6 follows from the conceptual HLF model presented in Subsection 3.1, but there are a few bespoke classes needed for network simulation and enabling smart contract "in the loop." For instance, the `ContractInstance` class does not, in fact, refer to a concrete instance of a specific smart contract class. Rather, it represents a smart contract installed at a peer, but it does have a reference to a `ContractInterface` object that is going to be an actual instance of the plugged smart contract class.

The ledger is stored in memory; the `Ledger` class contains a list of `LedgerEntry` objects, which are, in turn, versioned key-value plain old Java objects (POJOs). Peers have their local copy of the ledger and provide smart contracts with ledger data during transaction "simulation."

The `Client` class is not a concrete client implementation either, but an abstract, logical application client that can be used to send parameterized transaction requests (proposals) to the network.

There are several more classes present in the prototype that we have omitted for brevity and simplicity. Many of these facilitate the network simulation explained in the following subsection.

We should note that this model deliberately does not accurately reflect how the real HLF works. In the actual HLF implementation, transaction processing is much more complex as it uses shims, stubs, context providers, etc. We have done some simplifications and abstractions to improve model checking performance (by avoiding unnecessarily increasing the state space) and to keep our code concise and maintainable.

4.2 Network Simulation

To perform model checking, we simulate predefined transaction requests' execution on the user-configured abstract HLF network. The sequence diagram in Figure 7 concisely models how the framework simulates the network.

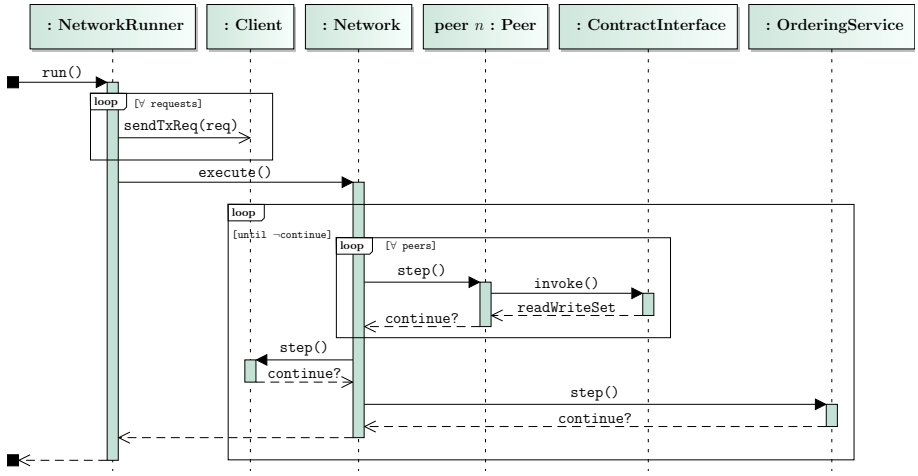


Figure 7: Network simulation

The primary logical entry point is the `NetworkRunner#run(String, OrderingService.FaultMode, List<InvocationRequest>)` method, which first instantiates all network components according to the supplied design and configuration. For now, network design is hardcoded into `NetworkRunner`'s source code,

but in a later iteration of the tool, we plan to offer a lightweight configuration language in which it can be specified at runtime. The method also *dynamically* instantiates the smart contract (chaincode) class based on a fully qualified class name passed as the first argument. We did not illustrate these instantiations in Figure 7 to make the diagram more readable.

The next step is sending what is essentially a call sequence to a client defined in the network. This call sequence is provided to the `run` method as its third argument. Each call represents an invocation with a method name and a list of arbitrary-type arguments.

The network is then ready for simulation. `NetworkRunner` calls `Network#execute`, which in turn begins a simulation loop. In each iteration, the `Network` class sequentially calls the `step` method of the individual components: peers first, then clients, then the ordering service. All components implement the `Participant` interface that requires them to define such a `step` method. During `Peer#step`, peers simulate an invocation of their local chaincode by calling `ContractInterface#invoke` – which will finally delegate the call to the actual smart contract implementation. The network simulation loop ends when no component indicates that there is still more to do.

At this point, all (virtual) ledger updates have occurred, and it is time to check the error property. For now, this is also hardcoded into `NetworkRunner#run` as a simple Java assertion.

4.3 Plugging in Smart Contracts

As explained in the previous subsection, our prototype framework dynamically instantiates chaincode classes based on fully qualified names passed as input arguments.

To make these pieces of chaincode work in the simulation without any additional modifications, we have developed a "shadowing," mock HLF Java package with stubbed versions of the classes required by smart contracts. These stubbed classes are in the `org.hyperledger.fabric` package and are intended to be found on the classpath before the classes in the *real* package supplied with HLF. Key mocked classes include `Context`, `ContractInterface`, `ChaincodeStub`, and annotations offered by HLF such as `@Transaction` and `@Contract`. This setup enables the seamless specification of existing pieces of chaincode to the framework.

4.4 Model Checking with JPF

Our framework uses Java Pathfinder (JPF) [16] (developed by National Aeronautics and Space Administration (NASA)) for model checking. JPF uses a properties file for configuration where the target main class, command line arguments, classpath, and other JPF-specific options. can be set. Listing 1 shows a partial example configuration where the ordering service's `CAN_DROP` fault mode is enabled (meaning it randomly ignores transactions), the "Smart Contract in the Loop" is a

minimal implementation of the train crossing example described in Section 5, and invocations are read from a `traincrossing.invocations` file.

```
target = hu.bme.mit.ftsrg.scil.Cli.CommandLineInterface
target.args = -f,CAN_DROP,-i,traincrossing.invocations,hu.[...].TrainCrossing
classpath = ./[...]/app-0.1.0-all.jar:./examples/train-crossing/[...]
cg.enumerate_random = true
+vm.assert = enable
```

Listing 1: Example (simplified) JPF configuration

5 Case Study

In the following, we demonstrate the viability of our approach in the context of a (hypothetical) safety-critical application where an autonomous vehicle may cross an unguarded railway intersection if, according to a (permissioned) decentralized application (dApp), it is safe to do so; i.e., no train is approaching. Figure 8 shows a simple schematic of this scenario.

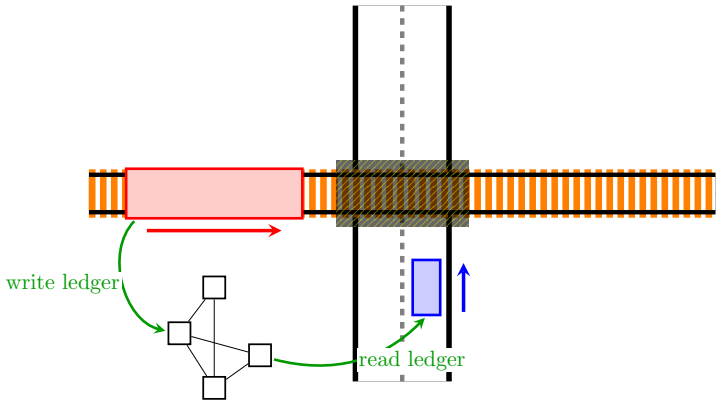


Figure 8: Visualization of the train crossing scenario

The precise operation of this illustrative system (implemented for Hyperledger Fabric (HLF)) is the following:

1. The two network participants are the railway company and the operators of the autonomous vehicles. For simplicity, let us assume there is only a single *train* and a single self-driving *car* – this is inconsequential from the application's perspective but simplifies the explanations in this paper.

2. There is a single world state entry that is updated and checked by the participants in any transaction: the value at the `canGo` key, which is either `"true"` or `"false"`.

3. The chaincode (smart contract) has a single `updateState` function that takes a boolean parameter and sets `canGo` to that parameter's value.

4. The train invokes `updateState` before entering the intersection (to set `canGo` to `"false"`). After the train has left the intersection, it again invokes `updateState` (this time parameterizing it with `"true"`).

5. Upon approaching the intersection, the car queries the ledger content and decides to cross or wait depending on the current latest value of `canGo`.

One way to phrase the fundamental safety-critical requirement in this elementary system: *the value of `canGo` MUST NOT be `"true"` when a train is in the intersection.*

The following describes how the individual elements first introduced in Section 3 are parameterized for the case study.

Network Design As mentioned in Section 4, the tool does not yet support dynamic network definitions at runtime; the network design must be specified programmatically before compiling to bytecode. For our case study, the network has the following components:

- two organizations R_1 and R_2
- a single channel C_1 with an ordering service O_1
- one peer at each organization (P_1 at R_1 and P_2 at R_2), both in C_1
- the smart contract is installed on P_1
- there is a single client in O_1 that connects to P_1

The programmatic setup of this network can be seen in Listing 2.

Fault Modes For illustration, let us assume that the O_1, the ordering service of the channel, behaves maliciously (or at least in a faulty manner): it randomly drops some transactions received. This behavior is enabled by passing the `-f CAN_DROP` option to the program, as seen in Listing 1.

Smart Contract We have attached the implementation of the train intersection smart contract in the Appendix, in Listing 3. It must be available on the classpath at runtime, and its fully qualified class name is specified as the first (and only) positional argument to the command line interface (CLI) of the framework.

Invocation Sequence We use an arbitrary sequence of a few transactions that update the state of `canGo`. The final invocation in the simulation invokes the smart contract with a `"false"` parameter, meaning there is a train entering the intersection, and it is not safe to cross.

Error Property Based on the safety-critical requirement and derived from the smart contract being checked (as well as the client-side invocations used for testing), we have specified the error property as an assertion of `canGo` being equal to `"false"` at the end of the simulation. A `"true"` value at the end represents a potential accident where the train has entered the intersection, but the ledger was not updated accordingly.

Model Checking To run the model checking, one needs to run the `jpf` binary on the properties file (Listing 1). Extensive logging is implemented throughout the framework so that, in case the error property is violated, there is a readily available execution trace. For example, due to the `CAN_DROP` fault mode of the ordering service in our example, it is possible to reach the error state when the orderer places a transaction with a `"true"` parameter last. See Listing 4 for the resulting execution trace leading up to the error.

6 Conclusion

In this paper, we have presented the novel Smart Contract in the Loop (SCIL) method and framework in detail that can be used for the comprehensive verification of a Hyperledger-Fabric-based [1] distributed ledger technology (DLT) application. In contrast to the few other verification and validation (V&V) tools available for permissioned (consortial) platforms, the approach presented in this paper does not only take the smart contracts or the ledger state into consideration but also aspects such as deployment and potential component-level faults.

SCIL contains a high-level model of HLF's key components and their interactions and performs model checking (based on Java Pathfinder (JPF) [16]) to determine whether a predefined error property can be fulfilled in the defined network and having the smart contract implementation (given to the tool "as is") installed. In other words, the tool can show how platform-level behavior can impact service-level behavior through smart contracts.

We have described our approach in theory and our prototype implementation created for the Hyperledger Fabric (HLF) platform and Java smart contracts. We have also presented a case study to demonstrate the viability of our approach on a theoretical safety-critical DLT application.

The framework can already be used, but there are still implementation efforts for future work, such as allowing the specification of the network's design at runtime (through the command line interface (CLI)) and implementing the possibility to define runtime *defenses* as well as faults. Furthermore, our model currently includes a limited set of fault modes for the ordering service component, while there are several other fault modes to consider.

We believe SCIL can also be extended to platforms other than HLF. Indeed, while Solidity smart contracts and the Ethereum Virtual Machine (EVM) can still be considered "common denominators" in the blockchain space, more and more new platforms support general-purpose programming languages and other domain-specific

languages (DSLs) for smart contract development – see, for example, Corda [12], which also supports Java, or Substrate[7], where so-called *pallets* are written in Rust. The key idea of reusing already existing, ready-to-use tools for analyzing smart contract bytecode also applies to other platforms wherever a virtual machine (VM) is used for execution. This includes Corda, as well as a few other platforms, such as Algorand [6].

Acknowledgments

The work of Bertalan Zoltán Péter, partially supported by the Doctoral Excellence Fellowship Programme (DCEP), is funded by the National Research Development and Innovation Fund of the Ministry of Culture and Innovation and the Budapest University of Technology and Economics under a grant agreement with the National Research, Development and Innovation Office.

The work of Bertalan Zoltán Péter was partially created under, and financed through, the Cooperation Agreement between the Hungarian National Bank (MNB) and the Budapest University of Technology and Economics (BME) in the Digitisation, artificial intelligence and data age workgroup.

The research of Zsófia Ádám was partially funded by the EKOP-24-3 New National Excellence Program under project number EKÖP-24-3-BME-288, and the Doctoral Excellence Fellowship Programme under project number 400434/ 2023; funded by the NRDI Fund of Hungary.

References

[1] Androulaki, E. et al. Hyperledger Fabric: A distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery. DOI: 10.1145/3190508.3190538.

[2] The Aptos Blockchain: Safe, scalable, and upgradeable Web3 infrastructure. Whitepaper, Aptos Foundation, 2020. URL: https://aptosfoundation.org/whitepaper/aptos-whitepaper_en.pdf.

[3] Beckert, B., Herda, M., Kirsten, M., and Schiffl, J. Formal specification and verification of a Hyperledger Fabric chaincode. In *3rd Symposium on Distributed Ledger Technology*, page 44–48. Institute for Integrated and Intelligent Systems, 2018. DOI: 10.5445/IR/1000092715.

[4] Beckert, B., Klebanov, V., and Weiß, B. *Dynamic Logic for Java*. In *Deductive Software Verification – The KeY Book – From Theory to Practice*, page 49–106. Springer, 2016. DOI: 10.1007/978-3-319-49812-6_3.

[7] https://substrate.io/

[5] Buterin, V. Ethereum: A next-generation smart contract and decentralized application platform. Whitepaper, Ethereum, 2014. URL: https://github.com/ethereum/wiki/wiki/White-Paper/.

[6] Chen, J. and Micali, S. Algorand: A secure and efficient distributed ledger. *Theoretical Computer Science*, 777:155–183, 2019. DOI: https://doi.org/10.1016/j.tcs.2019.02.001.

[7] Dhillon, V., Metcalf, D., and Hooper, M. *The DAO Hacked*. In *Blockchain Enabled Applications: Understand the Blockchain Ecosystem and How to Make it Work for You*, page 67–78. Apress, Berkeley, CA, 2017. DOI: 10.1007/978-1-4842-3081-7_6.

[8] Digital Asset. Canton Network: A network of networks for smart contract applications. Whitepaper, Digital Asset, 2024. URL: https://www.digitalasset.com/hubfs/Canton/CantonNetwork-WhitePaper.pdf.

[9] Díaz, M., Soler, E., Llopis, L., and Trillo, J. Integrating blockchain in safety-critical systems: An application to the nuclear industry. *IEEE Access*, 8:190605–190619, 2020. DOI: 10.1109/ACCESS.2020.3032322.

[10] Feng, C. and Niu, J. Selfish mining in Ethereum. In *Proceedings of the 2019 IEEE 39th International Conference on Distributed Computing Systems*, page 1306–1316, 2019. DOI: 10.1109/ICDCS.2019.00131.

[11] Garfatta, I., Klai, K., Gaaloul, W., and Graiet, M. A survey on formal verification for Solidity smart contracts. In *Proceedings of the 2021 Australasian Computer Science Week Multiconference*, ACSW '21, New York, NY, USA, 2021. Association for Computing Machinery. DOI: 10.1145/3437378.3437879.

[12] Gendal Brown, R., Carlyle, J., Grigg, I., and Hearn, M. Corda: An introduction. Whitepaper, R3, 2016. URL: https://docs.r3.com/en/pdf/corda-introductory-whitepaper.pdf.

[13] Górski, T. and Bednarski, J. Applying model-driven engineering to distributed ledger deployment. *IEEE Access*, 8:118245–118261, 2020. DOI: 10.1109/ACCESS.2020.3005519.

[14] Kuperberg, M., Kindler, D., and Jeschke, S. Are smart contracts and blockchains suitable for decentralized railway control? *Ledger*, 5, 2020. DOI: 10.5195/ledger.2020.158.

[15] McCanne, S. and Jacobson, V. The BSD packet filter: A new architecture for user-level packet capture. In *Proceedings of the USENIX Winter 1993 Conference*, USA, 1993. USENIX Association. DOI: 10.5555/1267303.1267305.

[16] National Aeronautics and Space Administration (NASA). Java Pathfinder, 2005. URL: https://github.com/javapathfinder/.

[17] Neu, J., Tas, E. N., and Tse, D. Two more attacks on proof-of-stake GHOST/Ethereum. In *Proceedings of the 2022 ACM Workshop on Developments in Consensus*, page 43–52, New York, NY, USA, 2022. Association for Computing Machinery. DOI: 10.1145/3560829.3563560.

[18] Péter, B. Z. and Kocsis, I. N-version programming as a mitigation for smart contract faults in execute-order-validate blockchain systems. In *Proceedings of the 30th Minisymposium of the Department of Measurement and Information Systems*, page 33–36, Budapest, Hungary, 2023. Budapest University of Technology and Economics. DOI: 10.3311/minisy2023-009.

[19] Rossberg, A. WebAssembly core specification. W3c recommendation, W3C, 2019. URL: https://www.w3.org/TR/wasm-core-1/.

[20] Shimosawa, T., Sato, T., and Oshima, S. BCVerifier: A tool to verify Hyperledger Fabric ledgers. In *Proceedings of the 2020 IEEE International Conference on Blockchain*, page 291–299, 2020. DOI: 10.1109/Blockchain50366.2020.00043.

[21] Soud, M., Liebel, G., and Hamdaqa, M. A fly in the ointment: an empirical study on the characteristics of Ethereum smart contract code weaknesses. *Empirical Software Engineering*, 29(1):13, 2024. DOI: 10.1007/S10664-023-10398-5.

[22] Tolmach, P., Li, Y., Lin, S.-W., Liu, Y., and Li, Z. A survey of smart contract formal specification and verification. *ACM Computing Survey*, 54(7):1–38, 2021. DOI: 10.1145/3464421.

[23] Wood, G. Polkadot: Vision for a heterogeneous multi-chain framework. Whitepaper, Ethereum & Parity, 2016. URL: https://whitepaper.io/document/596/polkadot-whitepaper.

[24] Yakovenko, A. Solana: A new architecture for a high performance blockchain. Whitepaper, Solana Foundation, 2017. https://solana.com/solana-whitepaper.pdf.

Appendix

```
Network network = Network.builder()
  .addOrganization("R1")
  .addOrganization("R2")
  .addPeer("P1", "R1")
  .addPeer("P2", "R2")
  .addOrderingService("O1", blockSize, faultMode)
  .addChannel("C1")
  .registerPeersToChannel(List.of("P1", "P2"), "C1")
  .installContract(contract /* <- SCIL */, "P1", "C1")
  .registerOrderingServiceToChannel("O1", "C1")
  .addClient("Client", "P1", "O1")
  .build();
```

Listing 2: Programmatic network design setup for the train crossing case study

```
package hu.bme.mit.ftsrg.chaincode.traincrossing;

import org.hyperledger.fabric.contract.Context;
import org.hyperledger.fabric.contract.ContractInterface;
import org.hyperledger.fabric.contract.annotation.*;
import org.hyperledger.fabric.shim.ChaincodeException;

@Contract(
  name = "TrainCrossing",
  info = @Info(/* contract metadata */))
public class TrainCrossing implements ContractInterface {

  @Transaction
  public void updateState(Context ctx, String value) {
    if (!(value.equals("true") || value.equals("false"))) {
      throw new ChaincodeException("Value must be 'true' or 'false'");
    }

    ctx.getStub().putStringState("canGo", value);
  }
}
```

Listing 3: Implementation of the train crossing scenario's chaincode

```
$ jpf model.jpf
JavaPathfinder core system v8.0 [...]

====================================================== system under test
hu.bme.mit.ftsrg.scil.cli.CommandLineInterface.main(
  "-v","-v","-v","-f","CAN_DROP",
  "-i","updateState false! updateState true! updateState false! updateState false! updateState
  ↪  true! updateState false",
  "hu.bme.mit.ftsrg.chaincode.traincrossing.TrainCrossing"
)

====================================================== search started: 9/30/24, 6:25 PM
[INFO | Peer#P1 @ 2024-09-30 18:25:03.164] simulating transaction request [...]
[INFO | Client#Client[connected to P1] @ 2024-09-30 18:25:03.335] forwarding transaction to
↪  orderer [...]
[INFO | OrderingService#O1 @ 2024-09-30 18:25:03.384] building a new block [...]
[INFO | Peer#P2 @ 2024-09-30 18:25:03.663] transaction applied to ledger, world state in peer
↪  updated
[INFO | network @ 2024-09-30 18:25:03.665] Network stopped

====================================================== results
error #1: gov.nasa.jpf.vm.NoUncaughtExceptionsProperty "java.lang.AssertionError: canGo should
↪  be false in..."

====================================================== statistics
elapsed time:       00:00:03
states:             new=10,visited=1,backtracked=2,end=3
search:             maxDepth=9,constraints=0
choice generators:  thread=3 (signal=0,lock=1,sharedRef=0,threadApi=0,reschedule=2), data=6
heap:               new=5543,released=2117,maxLive=3316,gcCycles=10
instructions:       274880
max memory:         248MB
loaded code:        classes=324,methods=5455
====================================================== search finished: 9/30/24, 6:25 PM
```

Listing 4: Execution trace leading up to the violation of the error property

Acta Cybernetica **27** (2025) 221–239.

Multi Model Recursion for Hungarian Electricity Load Forecasting

Mátyás Sebők[ab]

Abstract

Time series analysis and prediction is a difficult and complex problem. Many Machine and Deep Learning methods exist with better and better results. This paper proposes a strategy called Multi Model Recursion. It uses separate Deep Learning models per feature that needs predicting. Another improvement is not predicting features which are easily calculated. Having extra models per feature helps in "simulating" a future environment since it predicts external variables otherwise unknown. The Multi Model Recursion developed is an improvement of the commonly used Recursive strategy. The paper compares this method with models and strategies frequently used in the field. The testing dataset is put together from publicly available Hungarian electricity load and weather data. The task was to predict the country's net electricity load for the next 3 hours.

Keywords: time series, deep learning, Multi Model Recursion, electricity load forecasting

1 Introduction & Related works

Short term electricity load forecasting is useful since the forecasting models can adapt better to the given situation and give more accurate predictions. The better predictions give the opportunity for participants to better exploit their resources and minimize their costs. A 3-hour forecast comparison of Hungary's net electricity load shows the different strengths of models at single step forecasting and also describes their longer range performance.

The difficulty is that while weather data is available at a large resolution, forecasts are not always available the same way. The focus of Multi Model Recursion is to create a simulated environment with the given exogenous variables and their respective models to further enhance the predictions of the target variable. Compared to the regular Recursive strategy, this architecture can optimize better since it doesn't have to directly take into account the exogenous and time-series variables when calculating the cost function.

[a]Eötvös Loránd University, Budapest, Hungary
[b]E-mail: sebokmatyas01@gmail.com, ORCID: 0009-0000-0725-4212

DOI: 10.14232/actacyb.311880

This paper primarily compares Multi Model Recursion with the regular Recursive strategy using recurrent deep-learning algorithms. It also compares it with the Multi Input Multi Output (*MIMO*) strategy using Convolutional, Temporal Convolutional and LSTM networks. Compares it with the very powerful Sequence to Sequence (*Seq2Seq*) strategy, which uses an encoder-decoder architecture. To justify the usage of such complex algorithms it also looks at the performance of a machine-learning algorithm known as Random Forests and looks at the advantages compared to a Statistical method known as Seasonal Autoregressive Integrated Moving Average (*SARIMA*). The comparison happens based on a dataset created from public data from OMSZ (*Országos Meterológiai Szolgálat*) for weather data and data from MAVIR (*Magyar Villamosenergia-ipari Átviteli Rendszerirányító Zrt.*) for electricity load data.

1.1 Electricity load forecasting

Nti et al. provides a review on electricity load forecasting [11]. The authors provide a comprehensive study on the used forecasting methods and evaluation metrics. This motivates the use of MAE, RMSE, MPE and MAPE metrics and the evaluation of ANNs as these are the most used algorithms in the field.

Azeem et al. explains the application of electricity load forecasting techniques including short term electricity load forecasting [1]. The forecast horizon of a couple of hours can be critical in the operation and financial decision-making of energy management systems. Such forecasts can be used to decide which resources to utilize, for e.g. gas, coal, solar or wind. Another decision may be to import electricity at a lower cost than the utilization of non-renewable resources. The authors also explain the optimization techniques where the forecasts are used.

While the paper focuses on national load forecasting all methods can be utilized at a lower resolution such as Virtual Power Plants. Ghavidel et al. explain that such VPPs aggregate many physical entities such as renewable and non-renewable power plants, batteries and pump storage [5]. Accurate forecasts help the operation of such VPPs.

Yazici et al. provide a case study for electricity load prediction for Istanbul [14]. The authors achieve an impressive 1% MAPE metric for one-hour-ahead predictions and 2.2% for 24-hour-ahead predictions. While not matching this paper's 3-hour-ahead forecast horizon they provide a baseline to verify the results of this paper.

1.2 Problem description

Gasparin et al. [4] describes the task of time series forecasting for the electricity load case where there is a given uniform resolution $s = [s[0], s[1]..., s[T]]|s \subset \mathbb{R}$ time series data vector. It is ordered by time and has an hourly resolution in this case. For machine and deep learning purposes it is helpful to work with equal n_T length time windows. The prediction window's length is specified as n_O. The paper explores supervised learning solutions which require input-output pairs. Let's specify at time step t the input vector as $x_t = [s[t - n_T + 1], ..., s[t]]$ and the output

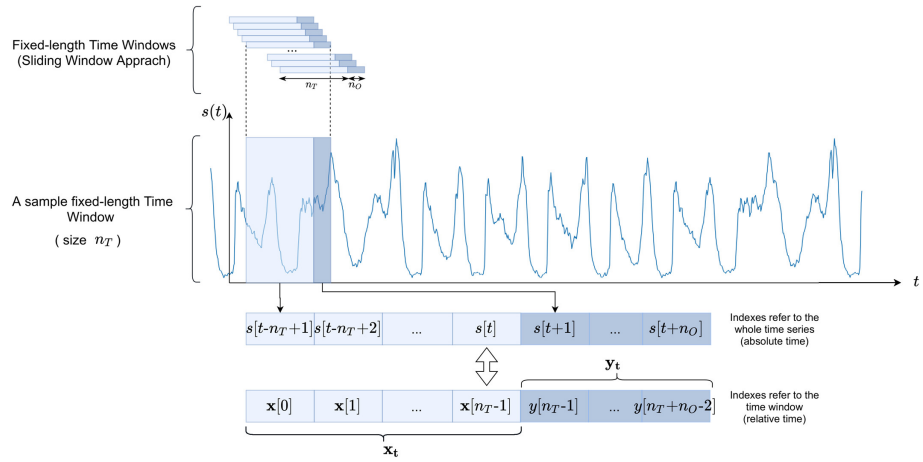


Figure 1: The sliding window approach [4]

vector as $y_t = [s[t+1], ..., [t+n_O]]$. This concludes in a sliding window type approach shown in Figure 1.

A model can be described as a parametric function f and its parameter vector θ, approximated by $\hat{\theta}$. With the above notation at time step t the model's output is $\hat{y_t} = f(x_t, \hat{\theta})$ which is an approximation of $y_t \in \mathbb{R}^{n_O}$. It is important to mention that in the practical application of such approaches the model is split into Machine or Deep Learning models and forecasting strategies where a strategy describes the steps of forecasting. These are discussed separately.

In the extension of the problem description $s \subset \mathbb{R}^d$ where $d-1$ is the number of exogenous or external features. $x_t[i][k]$ notates the kth feature of the ith element in the sequence where $i \in [0..n_T)$ and $k \in [0..d)$. In layman's terms this means that the forecast is helped by including additional features such as the weather, time of day or year.

1.3 Models

This section discusses the commonly used neural network and machine learning architectures for time series forecasting. These models are used in conjunction with the following forecasting strategies to provide predictions. The models are used later for the new Multi-Model Recursion strategy.

The Random Forest model is a classical machine learning method that uses the splitting rule for optimization. Probst et al. explains the optimization and training of such models [12]. This model may be used in any but the Sequence-to-Sequence strategy.

Gu et al. discusses the advancements made in convolutional neural network development [7]. By applying the architecture to 1D data like a time series the model can learn patterns in time that impact the prediction of the next time step.

[4] show the usage of causal convolution which applies the padding from only 1 side. This can further be expanded with dilated causal convolution ensuring a large receptive field for each layer.

Bai et al. shows the architecture and advantages of a temporal convolutional network [2]. This architecture, shown in Figure 2, employs dilated causal convolutions in addition to residual connections. Applying residual connections is beneficial to combat the vanishing gradient problem where the gradient gets progressively smaller as the optimization reaches the early layers. Through residual connections the gradient doesn't get affected by the weights ensuring a more stable descent.

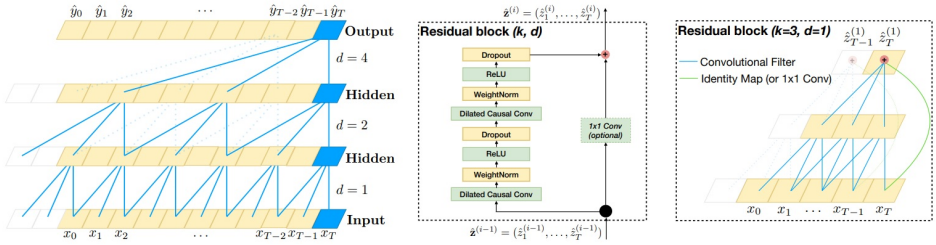


Figure 2: Temporal Convolutional Network Architecture: dilated convolutional layers, residual block and example for a residual block [2]

Masum et al. describes the LSTM model's architecture and its application to time series forecasting [10]. The advantage of RNNs (*Recurrent Neural Network*) is that they can process inputs of different lengths. This architecture can be applied to the Sequence-to-Sequence strategy that is mentioned and evaluated later in this paper. Shen et al. describes the workings of GRU based networks [13]. It is a newer approach compared to the LSTM aiming to resolve the same problem. It is generally not obvious as to which will perform better for a given task out of LSTM and GRU based networks. This is the reason both are evaluated in this paper.

1.4 Forecasting Strategies

Forecasting strategies describe how AI models are used for time series prediction. Strategies have to describe how many models are used and what the output dimensions are. It is possible to have strategies that complete predictions in a single or multiple steps. The following sections describe the forecasting strategies used and compared against Multi Model Recursion.

1.4.1 Multi-Input Multi-Output

Taieb et al. finds that multi-output strategies have good performance on time series forecasting tasks [3]. MIMO (*Multi-Input Multi-Output*) uses the entire input in one step at time t to produce the entire output vector y_t. The strategy can simply be described by the equation below if the forecasting model is f.

$$\hat{y}_t = f(x_t), \text{ simple multi-output prediction} \tag{1}$$

1.4.2 Sequence-to-Sequence

Seq2Seq (*Sequence-to-Sequence*) architectures were designed originally because it can be difficult to provide arbitrary length outputs with RNNs. The encoder-decoder architecture this strategy follows is the basis of modern LLMs (although the SoTA models are decoder only at the time of writing).

It consists of 2 models of the same type of RNNs, usually LSTMs or GRUs. The encoder produces a hidden state (and a cell state in the case of an LSTM). The decoder then uses this hidden state and its own outputs to produce the output. This can go until a certain stop sequence or iteration count. Zaki et al. [9] uses an LSTM based Seq2Seq model for household electricity load prediction but in this paper more success was found using a GRU based approach. Algorithm 1 describes the strategy.

Algorithm 1 Seq2seq strategy

$h_t := f_{enc}(x_t, h_0)$ *only need the hidden state from the encoder*
$\hat{y}_t[-1] := SELECT(x_t[n_T])$ *value that corresponds to the target feature*
for $i = 0, \ldots, n_O - 1$ **do**
  $\hat{y}_t[i] := f_{dec}(\hat{y}_t, h_t)$ *y_t is extended step-by-step*
  if $random(0 \ldots 1) < teacher\_forcing$ **then**
    $\hat{y}_t[i] := y_t[i]$ *teacher forcing, only while training*
  end if
end for
return $\hat{y}_t[0 \ldots]$

Teacher forcing for training Seq2Seq architectures helps with generalization over longer sequences. Since previous predictions affect the new ones they are substituted at a random probability with the real values. The probability gets decreased as training goes on. This technique helps the model train for longer forecast horizons as the compounding effect of incorrect predictions is removed. If $\hat{y}_t[i]$ is swapped for $y_t[i]$ then when optimizing based on the ith prediction step $\hat{y}_t[i]$ is used since $y_t[i]$ would provide a gradient of zero even if the prediction is incorrect. This approach is also applicable to the Recursive and Multi Model Recursive strategy.

1.4.3 Recursive

Taieb et al. explains that the recursive strategy uses a single model that is trained for forecasting only 1 step [3]. Here the model forecasts all external features for the next time step. This approach is interesting as the model isn't necessarily trained for multi step forecasting, but with the strategy it is applicable as such. Algorithm 2 describes the strategy.

Algorithm 2 Recursive strategy

1: **for** $i = 0, \ldots, n_O - 1$ **do**
2: $\hat{y}_t[i] := f(x_t)$ *forecast 1 step*
3: $x_t := x_t[1..n_T)$ *removing the first element of the input window*
4: $x_t[n_T] := \hat{y}_t[i]$ *extend the window by 1 at the end*
5: **end for**
6: **return** $\hat{y}_t$ *this includes the external feature forecasts*

For each t time step the forecast goes for step $t + 1$. This is then viewed as the "truth" and the model forecasts step $t+2$. Iterating this approach gives the output vector. The disadvantage of this method is that external feature forecasting requires larger weight matrices increasing processing requirements. Usually the optimization is also not efficient since the model is optimized for features not relevant for an application.

2 Methodology

2.1 Multi-Model Recursion

This paper presents the MMRec (*Multi-Model Recursion*) strategy which is an enhancement of the previously mentioned Recursive strategy. It aims to keep the advantages of the Recursive approach such as the single step prediction which generally gives more accurate predictions at that step. Being able to predict any length regardless of the training specifications is also an advantage, although performance may not be desirable if the training and inference output lengths are different. It incorporates the advantage of the MIMO and Seq2Seq approaches which only predict and optimize for the target variable.

There are 4 major changes from the Recursive method. The first one is regarding loss calculation for the network. Instead of training for all features directly the loss is calculated at the target variable at each step. This way the optimization focuses on electricity load in this case. The backpropagation will make sure that external features aren't left out. This is especially important in the next steps where multiple models are introduced.

Following the example of Seq2Seq teacher forcing is also applicable for the strategy. The same principles apply with the only difference being that each step's output vector is larger than 1. The random probability shown in Seq2Seq is calculated per member instead of once for the vector. It also decreases over the training period just like the mentioned strategy.

The next difference is the calculation of external features that are simple to predict. Features like time, or the lag of electricity load are easy to calculate via equations. These features are either pre-calculated or implemented into the strategy and used as is. This way no processing power is wasted on features that we know the exact values of even for the future.

Algorithm 3 Multi-Model Recursion

for $i = 0 \dots n_O - 1$ **do**
  for $j = 0 \dots m - 1$ **do**
    $\hat{y}_t[i,j] := f_j(x_t)$ *forecast given feature*
    $x_t := x_t[1..n_T)$ *sliding the window*
    $x_t[n_T, j] := \hat{y}_t[i,j]$ *substitute in the forecasted features*
    if $random(0 \dots 1) < teacher\_forcing$ **then**
      $x_t[n_T, j] := y_t[i,j]$ *teacher-forcing*
    end if
  end for
  $x_t[n_T] := g(x_t)$ *g calculates the obvious variables*
end for
return $\hat{y}_t$

The last change is the Multi-Model part of MMRec. Each feature that isn't calculated with the previous change gets its own neural network. Each model forecasts 1 specific feature making the individual models smaller. It is also possible to vary the architectures of them. Algorithm 3 and Figure 3 describe the strategy where m is the number of features forecasted by Neural Networks.

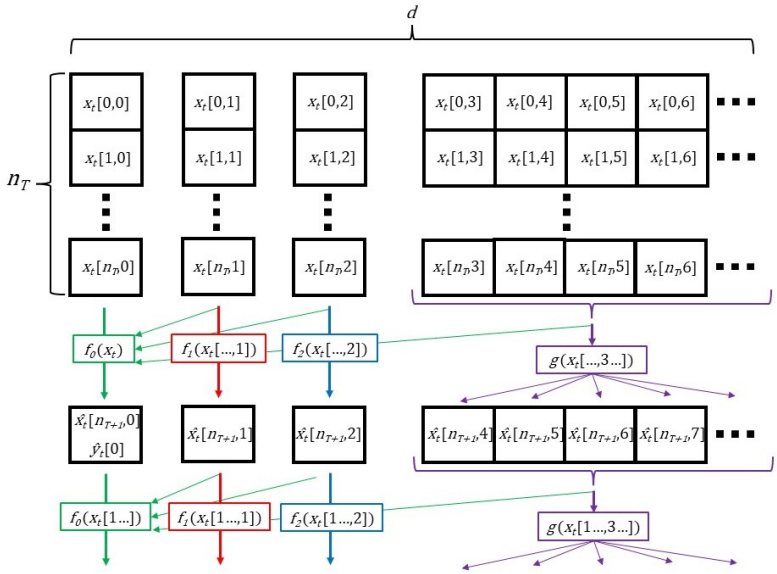
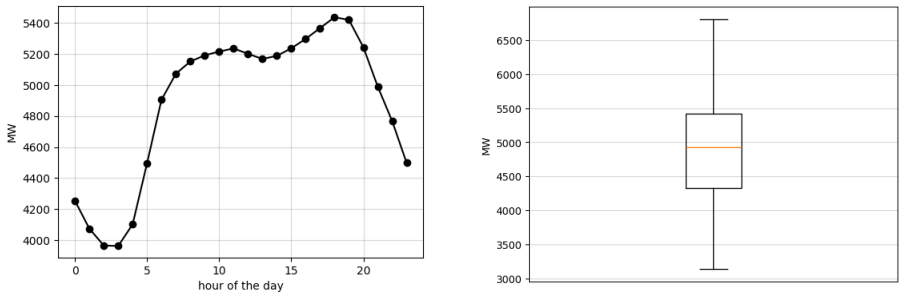


Figure 3: Multi-Model Recursion diagram, f_k are the different models, g calculates the obvious features, note that teacher forcing is not indicated here

2.2 Data

The dataset used for evaluating Multi Model Recursion against the mentioned methods consists of weather data downloaded from OMSZ's data publication [1] and electricity/system load data downloaded from MAVIR's dataset [2]. The observed time is from 01/01/2015 until 31/08/2023 resulting in an approximately 9 year long dataset describing Hungary. The timeframe was chosen due to OMSZ establishing many new weather stations in the year 2014.



(a) Hourly grouping (b) Box plot diagram

Figure 4: Net electricity load graphs

The source for the electricity load part of the dataset contains many fields from which "net electricity load (MW)" was chosen as the target and only feature describing electricity load. This is due to another feature existing in the original dataset named "MAVIR forecast" predicting net electricity load at the time step. This gives a baseline to justify the usage of machine and deep learning methods. Figure 4 (a) shows the hourly grouping of net electricity load for the observed timeframe. These type of graphs vary heavily by country. The box plot diagram in 4 (b) shows the high standard deviation of the dataset making forecasting tasks difficult.

OMSZ's weather stations all measure many different weather features like precipitation, temperature, relative humidity, global radiation and wind speed just to name a few. These are measured at over 100 stations giving a resolution that is too large for country scale predictions. So every weather feature was averaged over all stations resulting in 1 feature for each that describes the entire country. For example instead of 100+ temperature measurements there is only 1 describing Hungary.

Intuitively, many of these weather features don't make a difference for electricity load forecasting. By applying automatic sequential feature selection using Random Forests precipitation and global radiation proved to be the most descriptive. The feature selection algorithm was applied to time describing features at the same

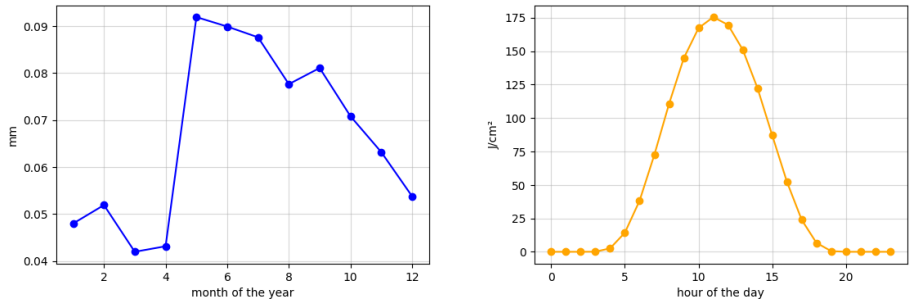
[1] https://odp.met.hu/
[2] https://www.mavir.hu/web/mavir/rendszerterheles

time. This means the process not only gave the useful weather features but time features as well. The best performance was observed at 11 features (the algorithm ran until 13 but after 11 the performance decreased). The chosen features are:

- electricity load and its 24-hour lag
- precipitation and global radiation
- holiday and weekend indicators
- hour, day of the week, day of the year, month, year

The feature selection chose global radiation over temperature for the best results. This is likely due to the fact that global radiation refers to the solar radiation that falls on a horizontal surface. This is supported by a correlation factor of 0.55 when observing temperature and global radiation.

The main point of reducing feature count in this way is that Multi Model Recursion is best applied in cases where only a couple external features are present since they all require separate models. Graphs for precipitation and global radiation from the dataset can be found in Figure 5.



(a) Precipitation monthly grouping (b) Global radiation hourly grouping

Figure 5: Weather feature graphs

The chosen features were re-evaluated at a later stage while training the MIMO LSTM approach and the 11 features chosen performed better. At this stage it's important to mention the choice of $n_T = 24$ for most strategy model pairs. This was made after choosing features and testing 12-, 24-, 36- and 48-hour lookbacks where 24 performed the best. This was re-evaluated for certain models, changes are mentioned where they were made.

2.3 Training and Evaluation method

When evaluating strategy model pairs it is important to find close to the best hyperparameter configurations. In this paper this is done using the Grid Search

algorithm where each hyperparameter gets a specified set of values. All combinations are then tried and the one with the best metrics and/or loss is chosen. Here RMSE was used since its strong reaction to outliers provides a clear picture of performance. RMSE is always calculated on the test set.

For machine and deep learning approaches it's usually important to use some form of cross validation technique. This means that multiple training loops are run using different parts of the data. Time series forecasting differs from other tasks since it wouldn't make sense to use future data in training while predicting the past. Due to this when evaluating in this paper, time series k-fold cross validation (Figure 6) is used. While finding hyperparameters the splits were limited to $k = 6$ (to save computational resources) and for final evaluation it was limited to $k = 9$. The validation set is always separated from the training set, taking up 1/8th of a single fold. For example if observing the 1st fold in Figure 6, the validation set is the last 1/8th of the training set. Furthermore, it is the same length for all other splits but always taken from the end of the training set.

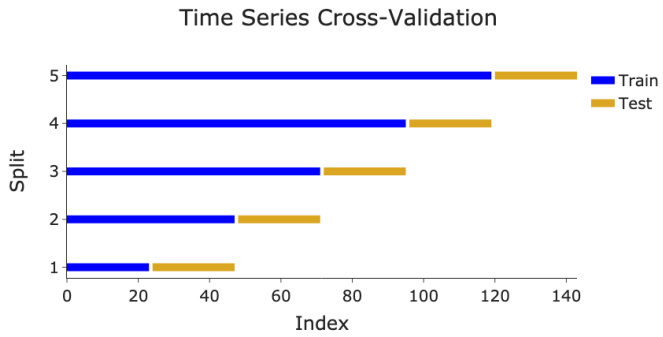


Figure 6: Time series cross validation [8]

After the hyperparameters are found each strategy model pair is trained and evaluated 6 times. These are done with the same hyperparameters but since the weights were initialized randomly the results vary. This is taken into account and the standard deviation of results through each fold and training cycle is displayed in the final table. Observed metrics are the following: MAE (Mean Absolute Error), MSE (Mean Squared Error), RMSE (Root Mean Squared Error), MAPE (Mean Absolute Percentage Error), and MPE (Mean Percentage Error).

Some technical details for the Grid Search algorithm are listed here. For each strategy pair the size and number of layers were searched for. For convolutional networks different lookback lengths were also observed such as $n_t = 48$. Learning rates, batch sizes and dropout ratios were also searched for. For most approaches not all combinations were observed at once but 2–3 hyperparameters were searched for once. This was iterated until a satisfying result was reached. After reaching a good point small changes were tested and if they didn't yield better results the parameters were chosen.

2.4 Chosen strategy model pairs

This section lists the chosen approaches for evaluation to compare with Multi Model Recursion. Other than SARIMA all of them are listed as strategy – model with abbreviations that are present in the final evaluation.

SARIMA (*Seasonal Autoregressive Integrated Moving Average*) This statistical method proved to be ineffective for this dataset since its parameters optimized at 1 time step didn't mean it was good for other time steps. At any one point the forecasts were comparable to MAVIR's predictions but it required a new parameter search to be effective. Due to this it isn't listed in the final evaluation. Parameter search may take over an hour which is not acceptable for this application where the neural network based models show no degradation of performance up to a year after training.

MIMO – RF (*Random Forest*) A classical machine learning approach to compare neural networks with. Decent performance on certain folds but heavily degrades at others.

MIMO – CNN (*Convolutional Neural Network*) A 1D Causal Convolution approach which didn't perform well on the dataset.

MIMO – TCN (*Temporal Convolutional Neural Network*) A 1D Temporal Convolutional approach performing well on short term forecasting.

MIMO – LSTM (*Long Short-Term Memory*) An approach that used to be one of the most popular since Recurrent networks work quite well for short sequence understanding.

Seq2Seq – GRU (*Gated Recurrent Unit*) A newer approach that may be viewed as the ancestor to SoTA transformer models.

Recursive – GRU The simplest forecasting strategy used with a GRU model. LSTM was also tested but GRUs proved to be more effective.

MMRec – 1 layer GRU Uses CNN and TCN for external features. Larger hidden state than the following approach.

MMRec – 2 layer GRU Uses CNN and TCN for external features. Smaller hidden state, when searching for hyperparameters its performance was very close to the previous one.

MMRec – FULL GRU Uses GRUs for external features too. A comparatively large and slow model to view what kind of performance MMRec can reach on the dataset.

The models observe a 24 horizon and forecast 3 hours ahead. The only exception to this are the MIMO - CNN and MIMO - TCN methods as these benefit from observing a 48-hour horizon. The other methods were also tested with shorter and longer horizons but the 24 hour performed best.

2.5 Training details of MMRec

To make the hyperparameter search faster each model for external features were first tuned as a Recursive strategy model for the given feature. The external feature models tried for precipitation and global radiation were CNN, TCN, GRU and LSTM networks. For global radiation CNNs performed better than TCNs and for precipitation the opposite was true. When GRUs were tested the addition of external features in relation to precipitation or global radiation slightly outperformed the Convolutional counterparts.

During the hyperparameter search 2 configurations proved to be powerful in forecasting. One which used a GRU with 1 layer but a larger hidden state and one with 2 layers using a smaller hidden state. Due to this both configurations are part of the final evaluation in addition to the FULL GRU approach.

3 Results

The final evaluation of strategy model pairs listed in Section 2.4 happened according to Section 2.3. Each pair is trained starting with their respective found "best" hyperparameters 6 times for 9 splits each. An interesting observation made during this is that most approaches using LSTMs performed noticeably worse in the first 2 splits. This was lessened by GRUs but the first splits were generally worse than anything else. This is likely due to the amount of data required by these models when training. Due to this the final evaluation lists the best performance of strategy model pairs while excluding the first or first and second splits, whichever gives better results.

Table 1 shows the final results for each of the mentioned metrics. Each metric also has the standard deviation listed over training iterations and the 9 folds per iteration. This gives more insight into certain models that may perform differently over the given folds such as CNNs and the Random Forest approach. MAVIR predictions have a standard deviation of 0 since it is impossible to observe training iterations or folds because it is given as is by MAVIR. MAVIR's predictions provide a higher resolution than 1 hour and are made with a 12-hour forecast horizon. This heavily affects accuracy at 3 hour predictions as forecasts for longer horizons generalize more.

From the table it's possible to observe that in terms of the MIMO strategy LSTMs perform the best closely followed by TCNs. TCNs struggle more in the

later steps of forecasting (steps 2 and 3 in this comparison). The Seq2Seq approach proved to be the most performant for this dataset in terms of metrics but also in approach size. The Recursive method performs quite poorly in comparison to others due to the shortcomings mentioned in Section 1.4.3. The presented MMRec strategy is clearly better than the Recursive strategy it is based on. With the additional ideas taken from Seq2Seq such as teacher forcing its performance is comparable with the MIMO LSTM approach and starts closing the gap on the Seq2Seq strategy as well.

The scores in Table 1 show a MAPE score for the presented approaches of about $1.2 - 2.1\%$ which is in line with the case study mentioned in the Introduction done by Yazici et al. [14]. Although the authors use a different dataset, time horizons and find that 1D CNNs perform best the performance of the forecasts are comparable. This shows that the results in this paper compare to an existing real world study.

Table 1: Table of evaluation results

Strategy Model	**MAE** (MW)	**RMSE** (MW)	**MAPE** $(\%)$	**MPE** $(\%)$
MAVIR prediction	252.58 ± 0	300.81 ± 0	4.97 ± 0	-4.70 ± 0
MIMO RF	69.96 ± 15.29	104.32 ± 24.0	1.42 ± 0.32	0.017 ± 0.25
MIMO CNN	103.13 ± 21.5	146.69 ± 27.55	2.12 ± 0.46	0.196 ± 0.416
MIMO TCN	63.92 ± 10.46	92.96 ± 15.57	1.31 ± 0.22	-0.001 ± 0.198
MIMO LSTM	62.62 ± 6.05	88.97 ± 9.19	1.28 ± 0.13	0.05 ± 0.175
Seq2seq GRU	$\mathbf{58.75 \pm 6.22}$	$\mathbf{84.21 \pm 9.83}$	$\mathbf{1.21 \pm 0.13}$	$\mathbf{0.08 \pm 0.168}$
Recursive GRU	94.41 ± 14.41	128.39 ± 17.39	1.94 ± 0.28	0.119 ± 0.744
MMRec GRU 1L	65.4 ± 7.68	92.43 ± 10.88	1.34 ± 0.17	0.114 ± 0.292
MMRec GRU 2L	64.79 ± 7.31	90.85 ± 10.27	1.32 ± 0.16	-0.029 ± 0.295
MMRec FULL GRU	62.17 ± 7.95	88.42 ± 11.25	1.27 ± 0.17	0.098 ± 0.261

Table 2 shows how much time is required for training and predicting with the models. Average and standard deviation can be understood the same way as for Table 1 described above. The Recursive GRU strategy training times and the MMRec GRU 1L/2L ones are similar. Even though multiple models are used for MMRec, they are much smaller and thus train faster than a large GRU for the Recursive strategy. Prediction times are the worst for MMRec as it uses multiple

Table 2: Table of training and prediction times

Strategy Model	**Training** (minutes)	**Prediction** *circa 7500 entries* (seconds)
MIMO RF	2.48 ± 1.45	0.082 ± 0.026
MIMO CNN	1.99 ± 1.09	0.136 ± 0.011
MIMO TCN	2.82 ± 1.51	0.258 ± 0.045
MIMO LSTM	2.99 ± 1.81	0.153 ± 0.015
Seq2seq GRU	6.16 ± 3.66	0.25 ± 0.019
Recursive GRU	7.16 ± 3.75	0.731 ± 0.035
MMRec GRU 1L	6.16 ± 4.69	2.242 ± 0.285
MMRec GRU 2L	6.89 ± 4.94	1.984 ± 0.044
MMRec FULL GRU	10.9 ± 6.45	1.884 ± 0.072

models. The shown times are for circa 7500 entries, so a single prediction is much faster. The time it takes is insignificant if we consider that it would only be made once an hour in an application.

For reproducibility the implementation of all strategies, the evaluation suite and the dataset can be found in the referred repository[3].

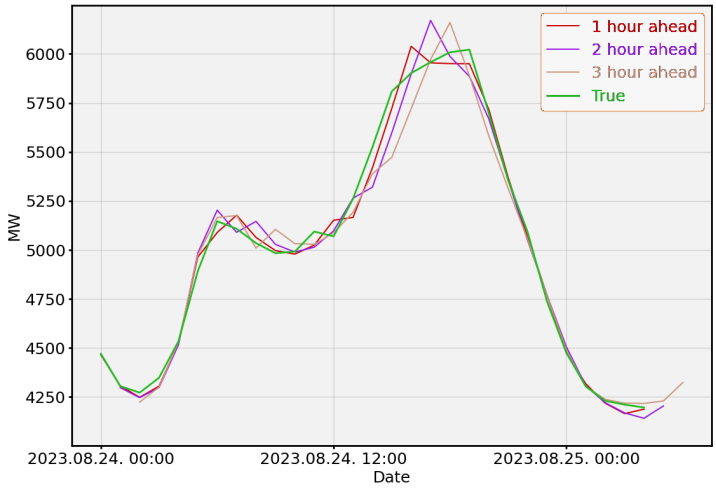
3.1 MMRec vs Seq2Seq

It was shown that MMRec outperforms the Recursive strategy but lacks the performance at its current stage to perform better than the Seq2Seq strategy on this dataset. This section provides a detailed explanation of the differences.

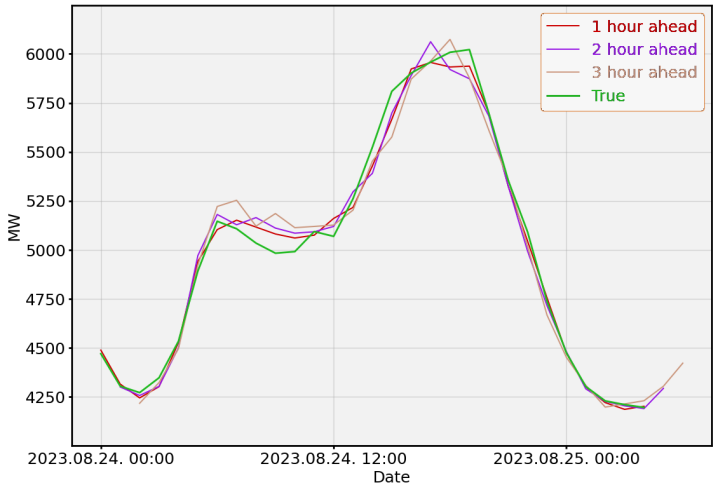
MAE and RMSE show a small difference between the two approaches. MPE varies more for MMRec, but this metric usually depends on the specific training run for these approaches. It isn't indicative of performance in this comparison. In terms of speed, the Seq2Seq model trains faster than MMRec - FULL GRU. Against the GRU 1L and 2L variants it doesn't have a clear advantage. MMRec however is a lot slower in prediction which can be critical for certain applications but isn't for this one. A comparison of the exact predictions for a specific date can be seen on Figure 7. In this example MMRec performs better in predicting the afternoon and Seq2Seq performs better in the morning.

Figure 8 shows the RMSE of the Seq2Seq and MMRec (FULL GRU approach here) strategies by how many hours they predicted. The main issue with MMRec that this graph displays is the first step being inaccurate in comparison to Seq2Seq (a). Interestingly MMRec accumulates less error as the prediction reaches farther distances (b). This hints at MMRec maybe performing closely to or better than Seq2Seq at longer prediction lengths. It could be argued that the algorithm of MMRec may cause this. For the first step the external feature models are not

[3] https://github.com/MeepOwned13/es_load_fs_HUN



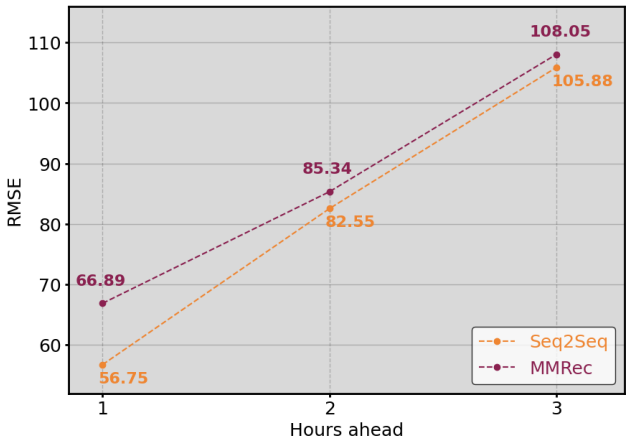
(a) Seq2Seq predictions example



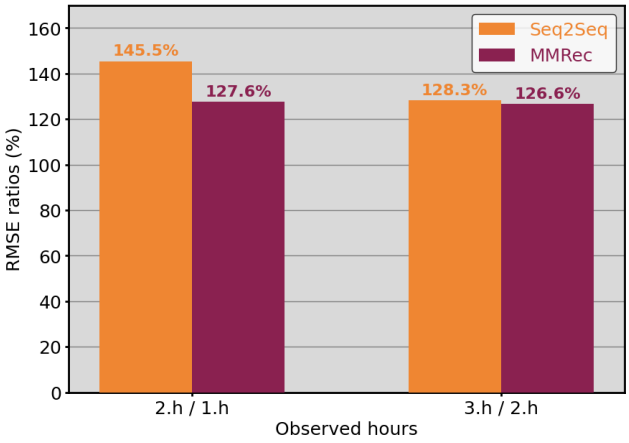
(b) MMRec predictions example

Figure 7: Comparing Seq2Seq and MMRec predictions for a specific date

involved since the real values are known at that stage. This being a 3-step prediction the optimization algorithm by default will prefer to optimize for the overall best average. Since all 3 models only get involved on step 2 and 3 it may lean heavier on the prediction of external feature models. This can cause a performance difference at step 1. This may be addressed by taking $n_T = n_T + 1$ for the input and using the external feature models to predict the ones it would already know.



(a) Error by hours predicted ahead



(b) Error accumulation bar plot

Figure 8: Comparing Seq2Seq and MMRec by RMSE per hour ahead prediction

4 Conclusion

This paper presented the MMRec (*Multi Model Recursion*) strategy for short term time series forecasting. Comparisons with existing time series forecasting strategies and models reveal that it outperforms the Recursive strategy it is based on. For the dataset constructed from Hungarian electricity load and weather data with the task of electricity load prediction it isn't the best performer in the comparison. The Seq2Seq strategy outperforms it in terms of MSE, RMSE, MPE and MAPE metrics.

MMRec shows promise in longer forecast horizons because it accumulates less error over time than Seq2Seq. This is counteracted in the 3-hour horizon by MMRec's worse performance at the first forecasting step. The conclusion is thus that MMRec may become a competitor to the mentioned strategies on some datasets given further refinements. This is backed by the observed error metrics being fairly close for MMRec and Seq2Seq.

4.1 Possible applications

Apart from the application to electricity load forecasting MMRec may be applied to any time series forecasting task where the external features that need to be forecasted by deep learning models are few. An interesting future application is choosing a task and dataset where external features more heavily influence the target feature. An example to this would be solar or wind electricity production. These heavily correlate with external weather features where getting a decent prediction for them could make a difference.

MMRec can also be used if forecasts of future external variables are sparse, for example if a better forecast for precipitation can be provided by outside models in certain cases but not always. In this case the model can take external forecasts by not using its own for that specific feature. Disruptions could also be caused by unforeseen events like hardware failures. In this case MMRec can operate without the need for its external feature models when everything is working as intended but use the external ones in the event that some forecasts are unavailable. For this use case MMRec doesn't require additional changes where other strategies would. The Recursive strategy also has these advantages but it was shown that MMRec outperforms it.

4.2 Outlook

The MMRec strategy has shown some advantages and disadvantages against the Seq2Seq strategy on the given dataset. A single dataset doesn't provide the full picture in these kinds of cases. Thus, the following future works can be specified:

- Apply the strategy model pairs to other datasets in electricity load forecasting.
- Apply the strategy model pairs to different time series forecasting tasks such as solar electricity production.
- Compare the strategies at longer forecast horizons such as 6, 12 and 24 hours.
- Compare the strategies at higher resolution on any time series forecasting task.
- Compare the strategy with more advanced methods such as Spacetimeformers introduced by Grigsby et al. [6].

References

[1] Azeem, A., Ismail, I., Jameel, S. M., and Harindran, V. R. Electrical load forecasting models for different generation modalities: A review. *IEEE Access*, 9:142239–142263, 2021. DOI: 10.1109/ACCESS.2021.3120731.

[2] Bai, S., Kolter, J. Z., and Koltun, V. An empirical evaluation of generic convolutional and recurrent networks for sequence modeling. arXiv, 2018. DOI: 10.48550/arXiv.1803.01271.

[3] Ben Taieb, S., Bontempi, G., Atiya, A. F., and Sorjamaa, A. A review and comparison of strategies for multi-step ahead time series forecasting based on the nn5 forecasting competition. *Expert Systems with Applications*, 39(8):7068–7070, 2012. DOI: 10.1016/j.eswa.2012.01.039.

[4] Gasparin, A., Lukovic, S., and Alippi, C. Deep learning for time series forecasting: The electric load case. *CAAI Transactions on Intelligence Technology*, 7(1):3–14, 2022. DOI: 10.1049/cit2.12060.

[5] Ghavidel, S., Li, L., Aghaei, J., Yu, T., and Zhu, J. A review on the virtual power plant: Components and operation systems. In *IEEE International Conference on Power System Technology (POWERCON)*, pages 1–6, 2016. DOI: 10.1109/POWERCON.2016.7754037.

[6] Grigsby, J., Wang, Z., and Qi, Y. Long-range transformers for dynamic spatiotemporal forecasting. arXiv, 2021. DOI: 10.48550/arXiv.2109.12218.

[7] Gu, J., Wang, Z., Kuen, J., Ma, L., Shahroudy, A., Shuai, B., Liu, T., Wang, X., Wang, G., Cai, J., and Chen, T. Recent advances in convolutional neural networks. *Pattern Recognition*, 77:354–357, 2018. DOI: 10.1016/j.patcog.2017.10.013.

[8] Howell, E. How to correctly perform cross-validation for time series. Towards Data Science, 2024. URL: https://towardsdatascience.com/how-to-correctly-perform-cross-validation-for-time-series-b083b869e42c.

[9] Masood, Z., Gantassi, R., Ardiansyah, and Choi, Y. A multi-step time-series clustering-based Seq2Seq LSTM learning for a single household electricity load forecasting. *Energies*, 15(7):5–6, 2022. DOI: 10.3390/en15072623.

[10] Masum, S., Liu, Y., and Chiverton, J. Multi-step time series forecasting of electric load using machine learning models. In Rutkowski, L., Scherer, R., Korytkowski, M., Pedrycz, W., Tadeusiewicz, R., and Zurada, J. M., editors, *Artificial Intelligence and Soft Computing*, pages 151–153, Cham, 2018. Springer International Publishing. DOI: 10.1007/978-3-319-91253-0_15.

[11] Nti, I. K., Teimeh, M., Nyarko-Boateng, O., and Adekoya, A. F. Electricity load forecasting: a systematic review. *Journal of Electrical Systems and Information Technology*, 7(1):13, 2020. DOI: 10.1186/s43067-020-00021-8.

[12] Probst, P., Wright, M. N., and Boulesteix, A.-L. Hyperparameters and tuning strategies for random forest. *WIREs Data Mining and Knowledge Discovery*, 9(3):1–7, 2019. DOI: 10.1002/widm.1301.

[13] Shen, G., Tan, Q., Zhang, H., Zeng, P., and Xu, J. Deep learning with gated recurrent unit networks for financial sequence predictions. *Procedia Computer Science*, 131:897–898, 2018. DOI: 10.1016/j.procs.2018.04.298.

[14] Yazici, I., Beyca, O. F., and Delen, D. Deep-learning-based short-term electricity load forecasting: A real case application. *Engineering Applications of Artificial Intelligence*, 109:104645, 2022. DOI: 10.1016/j.engappai.2021.104645.

Acta Cybernetica **27** (2025) 241–264.

Multithreading Atomicity: Static Analysis Checkers

Patrik P. Süli[a], Judit Knoll[b], and Zoltán Porkoláb[c]

Abstract

Ensuring thread safety in applications is crucial for preventing subtle and challenging bugs in concurrent programming. This paper presents two algorithmic approaches to improve thread safety through static analysis and to demonstrate their benefits in real life, the authors also implemented them as two detectors in SpotBugs static analyzer. These checkers are designed to identify unsafe usages of shared resources and improper atomic operations in concurrent Java programming, aiming to mitigate common multithreading issues such as race conditions. By emphasizing consistent locking strategies and the correct use of atomic types, the study offers insight into how to improve the reliability of multithreaded applications.

Keywords: Java, concurrency, atomic operations, static analysis

1 Introduction

Static analysis in software development can detect several types of issues, such as runtime errors and security violations in the code, without running the program itself, so developers could be informed about bugs in early stages of development [6]. There are many ways to analyze source codes, for example, Control Flow Analysis examines the execution, revealing infinite loops, unreachable codes, and improper usages of control structures [9]; Data Flow Analysis focuses on data tracking to identify issues like uninitialized variables, null pointer dereferences, and potential memory leaks [1]; Pattern-Based Analysis uses predefined rules to look for common issues or antipatterns in the code [13, 16].

Guidelines have been created to assist developers in producing code that is secure and reliable. The Software Engineering Institute (SEI) of the Carnegie Mellon University has its own, called CERT Coding Standards[1]. It has many rules

[a]Doctoral School of Applied Informatics and Applied Mathematics, Obuda University, Budapest, Hungary, E-mail: suli.patrik@uni-obuda.hu, ORCID: 0009-0001-9481-3664

[b]Sigma Technology Hungary Ltd., E-mail: judit.knoll@sigmatechnology.com, ORCID: 0009-0004-2400-6391

[c]Department of Programming Languages and Compilers, Institute of Computer Science, Faculty of Informatics, ELTE Eötvös Loránd University, Budapest, Hungary, E-mail: gsd@inf.elte.hu, ORCID: 0000-0001-6819-0224

[1] https://wiki.sei.cmu.edu/confluence/display/seccode/SEI+CERT+Coding+Standards

DOI: 10.14232/actacyb.311949

covering various aspects of coding practices, including memory handling, proper use of concurrency, input validation, and more, with the aim of preventing software vulnerabilities such as buffer overflows, race conditions, and injection attacks.

The SEI Cert Coding Standard contains several rules for atomicity, which is essential in software that work with parallel threads. This paper focuses on two specific rules of these, which are concerned with thread-safe usage of shared data.

There are several tools that can analyze code and make suggestions to improve it, one of them is SpotBugs[2]. SpotBugs is an open source tool that looks for possible issues in Java programs using Apache BCEL (Byte Code Engineering Library)[3], so it can handle binary *.class* files and understand instructions and methods at the bytecode level. When analyzing classes, SpotBugs reads and understands the structure of the bytecode, looking for specific patterns, coding practices, or known issues.

In this paper, we present two algorithms we designed and implemented as new detectors that cover possible concurrent programming problems in the Java language, which are described in the Visibility and Atomicity SEI Cert Rule group, focusing on the *VNA03-J*[4] and *VNA04-J*[5] rules. These rules provide practical guides with examples of both correct and incorrect usage, making it easy to identify common programming mistakes in connection with threads and shared resources, such as race condition, when a computation depends on timing or interleaving of multiple threads by the runtime. The *VNA03-J* rule highlights that a group of calls to independently atomic methods may not be atomic. *VNA04-J* underscores the method chaining mechanism, where the methods used in the chain can be atomic, but the chain overall is inherently non-atomic.

The rest of the paper is organized as follows: Section 2 introduces the concept of atomic types in different programming languages and details cases of non-thread-safe usage of this type. Section 3 presents the technical background used as a basis for our algorithms. The current state of the art is shown in Section 4 as a benchmark of a few static analyzer tools. The outline of the algorithms and the detectors developed are detailed in Section 5. The results are presented in Section 6, which are obtained from open source projects, with a comparison of their effectiveness with other static analysis tools. Furthermore, Section 7 highlights known limitations and opportunities for further development to improve the accuracy of the implemented detectors. The paper concludes in Section 8.

2 Related work

The concept of atomic types, also known as atomic operations or atomic classes, was introduced in concurrent programming to manage and manipulate shared data

[2] https://spotbugs.github.io

[3] https://commons.apache.org/proper/commons-bcel/

[4] https://wiki.sei.cmu.edu/confluence/display/java/VNA03-J.+Do+not+assume+that+a+group+of+calls+to+independently+atomic+methods+is+atomic

[5] https://wiki.sei.cmu.edu/confluence/display/java/VNA04-J.+Ensure+that+calls+to+chained+methods+are+atomic

safely and efficiently without the need for complex synchronization mechanisms. The term `atomic` in this context refers to operations that are completed as a single, indivisible, and unbreakable unit.

This idea was first proposed and explored in low-level hardware and assembly languages where atomic instructions such as "test-and-set" [3] or "compare-and-swap" (CAS) [11] were implemented directly by the CPU to facilitate safe concurrent access to shared memory.

2.1 Atomic types in programming languages

As multithreading and parallel processing have become more prevalent, the significance of atomic operations in high-level programming languages has grown. Although, this challenge is not unique: similar issues occur in a wide range of programming languages. Therefore, it is crucial to extend this analysis to different languages to gain a comprehensive understanding of how usable the atomic type is in different environments. By comparing the atomic types in Java (and other JVM based languages like Kotlin and Scala), C++, Python, and Rust, we can understand how different languages approach the challenge of concurrency and atomic operations, highlighting the strengths and trade-offs of each approach.

Java introduced `atomic` types with the release of Java 5 in 2004. Java's `java.util.concurrent.atomic` package[6] includes several atomic classes such as `AtomicInteger`, `AtomicLong`, `AtomicBoolean`, and `AtomicReference`. These classes leverage low-level atomic instructions to offer thread-safe operations on single variables without the overhead of locks[7]. For instance, `AtomicInteger` provides methods like `incrementAndGet()`, `decrementAndGet()`, and `compareAndSet()`, which are implemented to ensure that operations are completed without interruption.

Kotlin and Scala as JVM-based languages leverage the same concurrency mechanisms and atomic types provided by the Java platform. Developers can use the `java.util.concurrent.atomic` package directly and create atomic-typed classes in the same way as in Java.

```
1 AtomicInteger atomicInteger = new AtomicInteger(0);
2 atomicInteger.incrementAndGet();
```

Code 1: Example usage of Java's `AtomicInteger` typed variable

One of the major high-level programming languages, the C++ programming language, was lack of proper solution for atomics until the C++11 standard defined the C++ memory model and introduced atomic classes[8] in the C++11 Standard Library [12]. The `std::atomic` template encapsulates the complexity of atomic instructions and provides a standardized interface for the developers.

[6] https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/package-summary.html

[7] https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/package-summary.html

[8] https://cplusplus.com/reference/atomic/atomic/

```
std::atomic<int> counter{0};
counter.fetch_add(1);
```

Code 2: Example of C++ `std::atomic` variable

As C++ is a highly performance critical programming language, the instantiations of the `std::atomic` template, where the hardware provides atomic instructions handling the type parameter (e.g., `std::atomic<int>`) are compiled without any run-time overhead, while more complex template parameters, like structs have an external locking mechanism to provide atomicity.

Python is known for its simplicity and readability, making it a favorite for tasks ranging from web development to data science. However, Python's Global Interpreter Lock (GIL) presents challenges in concurrent programming[9]. To address atomic operations, Python relies on external libraries like *'atomicx'*[10] or built-in threading primitives[11] to simulate atomicity.

```
from atomicx import AtomicInt

atomic_int = AtomicInt(0)
atomic_int.inc()
```

Code 3: Example of an atomic operation using Python's `atomicx` library

In contrast, Rust is designed with a strong emphasis on memory safety and concurrency without sacrificing performance. Rust's ownership system ensures memory safety, while its standard library provides built-in atomic types[12] like `AtomicBool`, `AtomicIsize`, and `AtomicUsize`. These types support thread-safe lock-free operations, making Rust an excellent choice for programming systems and applications that require high reliability.

```
use std::sync::atomic::{AtomicUsize, Ordering};

let atomic_usize = AtomicUsize::new(0);
atomic_usize.fetch_add(1, Ordering::SeqCst);
```

Code 4: Example of an `AtomicUsize` variable in Rust

Despite the differences in syntax, all these languages share a common foundation when it comes to atomic types. They provide atomic operations to manage shared resources in multithreaded environments. However, the same challenges persist across these, namely avoiding race conditions and managing the complexity of lock-free programming, so the algorithms that are detailed in this paper could be applicable for each programming language mentioned before.

[9] https://wiki.python.org/moin/GlobalInterpreterLock
[10] https://github.com/RuneBlaze/atomicx
[11] https://docs.python.org/3.10/library/threading.html
[12] https://doc.rust-lang.org/std/sync/atomic/index.html

2.2 Thread safety issues with Java concurrency types

When working with atomic types or synchronized collections in Java, it is important to understand that while individual method calls on these variables are atomic, combining these operations within a thread introduces potential thread safety issues. If an operation in a thread involves multiple atomic variables, proper synchronization is necessary to ensure that the entire operation remains atomic.

To bring attention to this issue, the *SEI Cert VNA03-J* rule – titled *"Do not assume that a group of calls to independently atomic methods is atomic"* – was created to avoid wrong usages of atomic typed variables and collections, which can lead to difficult-to-detect concurrency bugs.

```
1 private AtomicInteger a = new AtomicInteger(10);
2 private AtomicInteger b = new AtomicInteger(15);
3 // ...
4 a.get().add(b.get()); // Combination is not thread-safe
```

Code 5: Combine Java atomic typed variables unsafe

In code snippet 5, the `get()` method calls on both variables are atomic *separately*, but the `add()` operation itself is not thread safe, because meanwhile another thread may make changes on one or both variables, as it is shown on Figure 1.

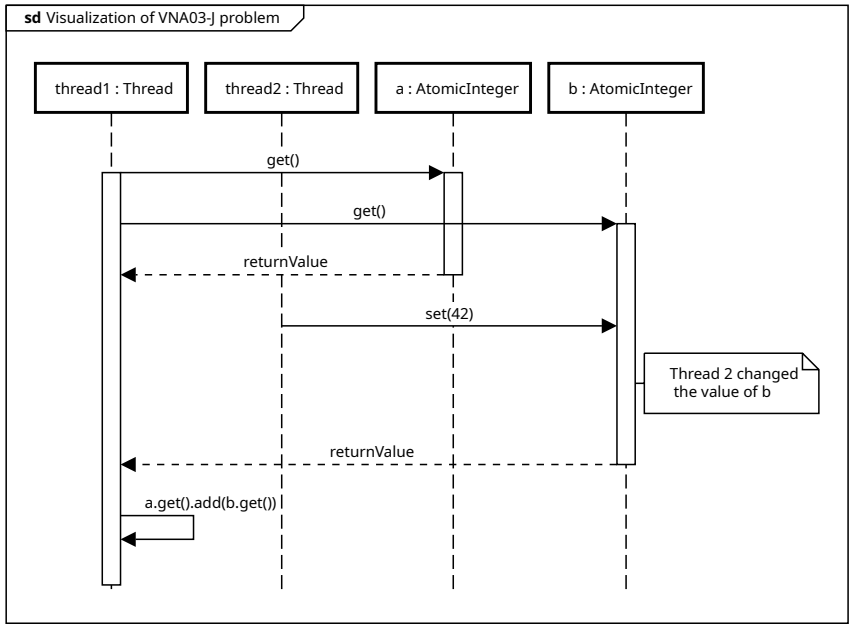


Figure 1: Sequence diagram of unsynchronized combination of atomic typed variables

2.3 Multiple atomic operations in threads

If a method contains more than one operation for an atomic variable without additional synchronization, then they are not atomic overall and could cause race condition between the threads, just like in section 2.2. So, to handle this, synchronization of code blocks or functions is necessary to guarantee that multiple threads cannot simultaneously modify or access shared resources.

```
1 private AtomicInteger number = new AtomicInteger(0);
2 // Thread 1, combined atomic method calls are not atomic together
3 number.get();
4 // ...
5 number.get();
6 // Thread 2
7 number.getAndIncrement(); // It is safe standalone
```

Code 6: Unsafe multiple operations on atomic variable

`Thread 1` in Code 6 is not thread-safe, because between the two atomic method calls `Thread 2` can change the value of the variable.

2.4 Unsynchronized concurrent collection elements

The usage of thread-safe collections such as `SynchronizedList` or `ConcurrentHashMap` is not sufficient to ensure thread safety in itself, because any access to the collection's elements is not synchronized. The operations on these collections, such as adding or removing elements, are basically thread-safe, but accessing or modifying their elements themselves are not inherently synchronized. Consequently, any operations performed on the elements retrieved from these collections must be properly synchronized to avoid concurrent modification issues.

```
1 private final Map<Integer, Integer> counterMap =
2       Collections.synchronizedMap(new HashMap<Integer, Integer>());
3
4 public void incrementCounter(int id) { // Called by multiple threads
5     Integer count = counterMap.get(id);
6     counterMap.put(id, count + 1);
7 }
```

Code 7: Unsafe access to thread-safe collection elements

If multiple threads run the counter increment lines in Code 7 at the same time, it results in race condition, because the operation on the collection's element is not synchronized.

2.5 Unsafe usages of shared resources in multiple threads

The *SEI Cert's VNA04-J* rule - titled *"Ensure that calls to chained methods are atomic"* - is about the nonatomic-typed variable usages in threads. It focuses on a special case: method chaining.

Method chaining is a mechanism that allows multiple method calls on the same object in a single statement. It consists of a series of methods returning the `this` reference, allowing chained method invocations using the return value of the preceding method.

This style is often used in classes that employ the *Builder Pattern*[13] to set up objects with multiple parameters. For a common example, the `StringBuilder` class[14] uses this kind of mechanism.

```
StringBuilder sb = new StringBuilder();
String result = sb.append("Hello, ").append("World!").toString();
```

Code 8: Example usage of `StringBuilder` class in Java

Although individual methods in a chain can be atomic, the chain as a whole is not. Consequently, callers must ensure sufficient locking for the chain's atomicity.

While the *VNA04-J* rule specifically deals with chained methods with builder pattern, its underlying principle can be extended to a wider range of scenarios. By generalizing this rule, a broader set of cases can be covered: any method can be found that modifies the state of a shared resource across multiple threads and may lead to race conditions. Overall, the checker is more flexible, it can recognize thread safety issues in various contexts, beyond just method chaining.

In Code 9 multiple threads modify the `User` object's state and when the execution reaches the `getName()` method call, the state of the `name` property is unambiguous.

```
public class User {
    private String name;

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}

public class ExampleClient {
    private User user = new User();

    public ExampleClient() {
        new Thread(() -> {
            user.setName("Jane");
            System.out.println("New name: " + user.getName());
        }).start();

        new Thread(() -> {
            user.setName("Bob");
            System.out.println("New name: " + user.getName());
        }).start();
```

[13] https://refactoring.guru/design-patterns/builder
[14] https://docs.oracle.com/javase/8/docs/api/java/lang/StringBuilder.html

```
26      }
27  }
```

Code 9: Unsafe operation in multiple threads to a shared resource

The visualisation of the problem in the *Code 9* can be seen in *Figure 2*.

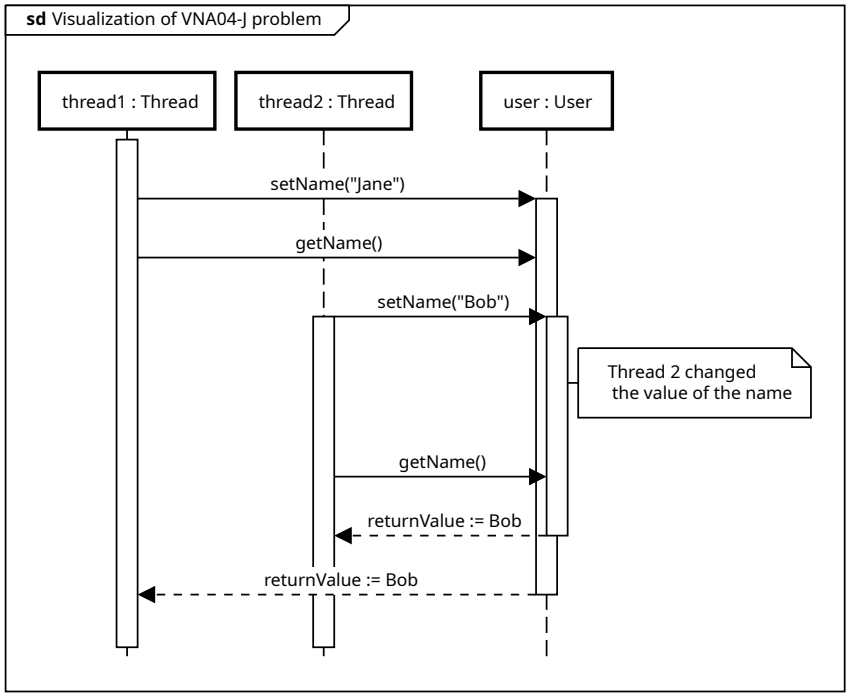


Figure 2: Sequence diagram visualization of Code 9

2.6 Similar concurrency issues in C/C++

Concurrency bugs involving atomic operations are not unique to Java. The SEI CERT guidelines for C and C++ also address atomicity and threading issues through rules such as *CON40-C* (*"Do not refer to an atomic variable twice in an expression"*)[15] and *CON43-C* (*"Do not allow data races in multithreaded code"*)[16].

These rules highlight problems analogous to *VNA03-J* and *VNA04-J* in Java, noting that while individual atomic operations (e.g., `atomic_load()`, `atomic_store()`, and compound assignments) are guaranteed to be thread-safe, combining multiple atomic reads or writes in a single expression or code block can result in race conditions without a careful locking mechanism. The recommendation in both languages

[15] https://wiki.sei.cmu.edu/confluence/display/c/CON40-C.+Do+not+refer+to+an+atomic+variable+twice+in+an+expression

[16] https://wiki.sei.cmu.edu/confluence/display/c/CON43-C.+Do+not+allow+data+races+in+multithreaded+code

to adopt explicit locking (e.g., using mutexes or synchronization) when performing compound operations.

Static analyzers in the C/C++ ecosystem, such as CodeSonar[17] or Coverity[18], provide support for detecting atomicity violations and data races according to the SEI Cert rules. Although each language's memory model differs in detail, the overarching principle remains the same: atomic types ensure thread safety only for *single* operations, and automated tools can help developers to detect and handle cases where multiple atomic operations compose a non-atomic sequence. By drawing these parallels, we emphasize that detecting improper atomic usages is a cross-language challenge that requires consistent locking strategies and thorough static analysis approaches.

3 Technical background

Contrary to testing and dynamic analysis methods, *static analysis* works at compile time, based only on the source code of the system, and does not require any input data [4]. Most of these methods are fast enough feasibly integrated into the continuous integration (CI) loop providing a positive impact on speed up the development-bug detection-bug fixing cycle. As the earlier a bug is detected, the lower is the cost of the fix [5], therefore, static analysis is a useful and relatively cheap supplement to testing.

All static methods apply heuristics, which means that sometimes they may *underestimate* or *overestimate* the program behavior [14]. In practice this means static analysis tools sometimes do not report existing issues which situation is called as *false negative*, and sometimes they report correct code erroneously as a problem, which is called as *false positive*. Therefore, all reports need to be reviewed by a professional who has to decide whether the report stands.

During the last two decades various static analysis techniques evolved. The most simple, but surprisingly strong method is *pattern matching*. First, the source code is transformed into some canonical format (e.g., all loops are converted to `while` and the body of the loop to a single line) and then predefined regular expressions are applied against this code. While context-sensitive problems (as divergence between the declaration and the use of a variable) are impossible to detect, many programmer's mistakes are detectable. As a huge advantage, this method does not require the successful construction of the Abstract Syntax Tree (AST), therefore applicable for non-compiling or partial code fragments too. Earlier versions of CppCheck[19] used pattern matching to find issues in C and C++ programs.

Most of the available static code analysis tools, however, are based on the analysis of the Abstract Syntax Tree (AST). The AST is a usual internal representation of a program or at least a translation unit used by the compiler [2]. Various versions of the AST can represent only the structure of the parsed tokens or may hold semantic

[17] https://codesecure.com/our-products/codesonar/
[18] https://scan.coverity.com
[19] http://cppcheck.sourceforge.net/

information too. It encodes the structure of the program, the declarations, the variable usages, selection and loop statements, function calls. Thus, AST-based static analysis is capable to detect complex errors, like erroneous implicit conversions, inconsistent design rules, and many others. Such checks are relatively fast, some of them may be implemented using a single traversal of the AST. These features make the AST-based method the most frequently used type of static analysis with notable examples as the Clang Tidy[20] for C++, SpotBugs for Java and PyLint[21] for Python.

While the AST-based method is more powerful than the regular expression based one, seeing only the *structure* of the program it still lacks of the reasoning on the *possible values* of the variables at a certain point of the program. *Symbolic execution* [10] is a path-sensitive abstract interpretation method. During symbolic execution we interpret the source code, but instead of using the exact (unknown) run-time values of the variables we use symbolic values and gradually build up constraints on their possible values. Symbolic execution is the most powerful, but also the most expensive method for static analysis, and requires a precise modeling of the language semantics and the representation of the memory usage [7].

3.1 SpotBugs overview

SpotBugs is designed to detect bugs in Java programs by analyzing bytecode. It is the successor to FindBugs and maintains compatibility with many of its features and plugins. SpotBugs can identify a wide range of potential issues in Java code, including but not limited to concurrency problems, performance bottlenecks, and potential security vulnerabilities.

SpotBugs primarily operates by analyzing the Abstract Syntax Tree (AST) generated from Java bytecode. It uses various detectors, which are specialized components designed to identify specific types of bugs. These detectors can be visitor-based, which analyze the bytecode in a straightforward manner, or CFG-based, which utilize control flow graphs to perform more sophisticated analysis. CFG-based detectors are particularly powerful, but come with higher computational costs.

One of the strengths of SpotBugs is its extensibility. Developers can create custom detectors through a plugin architecture, allowing SpotBugs to be tailored to specific project needs. The tool is capable of integrating into continuous integration (CI) pipelines, providing ongoing feedback on potential issues as code is developed.

3.1.1 Applying SpotBugs to Kotlin and Scala: FindSecBugs

SpotBugs, while originally designed for Java, can also be applied to other JVM-based languages like Kotlin and Scala. This is particularly useful in projects where multiple JVM languages are used, allowing for consistent static analysis across different parts of the codebase.

[20] https://clang.llvm.org/extra/clang-tidy/
[21] http://pylint.pycqa.org/en/latest/

To facilitate security-focused static analysis in these languages, the Find Security Bugs (FindSecBugs)[22] plugin extends SpotBugs' capabilities. FindSecBugs is a SpotBugs plugin that specializes in detecting security vulnerabilities in Java, Kotlin, and Scala code. Identifies potential security issues such as SQL injection, cross-site scripting (XSS), and improper validation of input data.

When applied to Kotlin and Scala, FindSecBugs leverages the underlying bytecode analysis capabilities of SpotBugs, adapting them to handle the syntactic and semantic differences of these languages. While Kotlin and Scala introduce language-specific constructs that may not map directly to Java, the bytecode they compile to is still compatible with SpotBugs' analysis techniques.

However, it is important to note that, due to differences in the way Kotlin and Scala handle certain programming concepts, such as lambdas and coroutines [8, 15], there may be limitations in the accuracy and coverage of the analysis. Despite this, FindSecBugs and the SpotBugs itself remain valuable tools for enhancing security in multi-language JVM projects, providing a unified approach to identifying and mitigating security risks across Java, Kotlin, and Scala codebases.

4 State of the art

With the help of test cases focusing on the above-mentioned issues which we developed for the SpotBugs testing framework, we performed a comparative analysis with other existing static analyzers for Java.

On our test cases (which are detailed in Section 6) there should be *22* hits on the *VNA03-J* cases and *10* hits on the *VNA04-J*, but it seems like the static analysis tools we tested do not detect these multithreading atomicity rules, as can be seen in Table 1.

Table 1: Result of static analyzer tools hits on *VNA03-J* and *VNA04-J* test cases

Name of the tool	VNA03-J hits	VNA04-J hits
PMD v7.0.0 [23]	0	0
SonarQube v9.9.5.90363 [24]	0	0
The Checker Framework v3.43.0[25]	0	0
Google's Error Prone v2.27.1[26]	0	0
SpotBugs v4.8.6[27]	0	0

Although all tools report atomicity-related issues, these are limited to other aspects, such as do not use the `volatile` keyword[28], and suggests replacing it with

[22] https://github.com/find-sec-bugs/find-sec-bugs

[27] This version of SpotBugs does not yet include the detectors detailed in this paper.

[28] https://docs.pmd-code.org/latest/pmd_rules_java_multithreading.html#avoidusingvolatile

a Java-built-in atomic type. These tools do not detect issues related to the complex usage of atomic types or synchronized collections[29].

In conclusion, our analysis indicates that the static analysis tools we tested do not currently detect the specific multithreading atomicity issues described in the *VNA03-J* and *VNA04-J* rules. SpotBugs is open source and free to use, it allows the detection of these bugs to be distributed to a large community of developers.

5 Detector algorithms

The *VNA03-J* and *VNA04-J* SEI Cert rules focus on the proper use of locking with synchronization as it is described in Section 2.2 and 2.5 in detail. We designed two algorithms and implemented them as detectors in the SpotBugs static analyzer to find unsafe usages of common references between threads, and make sure the proper usage of fields with Java `atomic` types. The algorithm that covers rule *VNA04-J* works with references of types which are not related to the Java Concurrent API, and the algorithm using the rule *VNA03-J* ensures the proper usage of `atomic` type-based classes.

The source code and test cases of these detectors are publicly accessible in the official SpotBugs repository, where our contributions are submitted as two pull requests: #2919 – *VNA03-J Sequence of calls on a synchronized abstraction may not be atomic*, and #2986 – *VNA04-J. Ensure that calls to chained methods are atomic.* VNA04-J is already available in SpotBugs from version 4.9.0.

5.1 Finding unsafe reference usages in multiple threads

The algorithm implementing *VNA04-J* rule works in class context, which means that it scans class bytecode, but it does not see the relations between classes and can only work with the code inside the currently analyzed class. This limitation is inherited from the SpotBugs Framework, as detailed in Section 7.2. The detector searches and collects methods that are in a call hierarchy that starts with a lambda (anonymous) or referenced method passed directly to a `Thread` object, but takes place in the current class. In Java, a `Thread` object[30] requires a method in its constructor that implements the `Runnable` functional interface.

We developed and tested a variation of the algorithm, in which the detector was designed to include all methods in its scan, meaning that it also works with the operations running on the main thread. When testing this solution on large open source projects (these are introduced in *Section 6*, with the final detection results) it had many false positive hits, so it was less useful on real world projects. Because of this we decided to use the stricter version of the algorithm, to only cover a subset of the original problem, but have more useful, accurate hits.

[29]The *VNA03-J* SEI Cert Rule Wiki page mentions that the Coverity and Parasoft Jtest (`https://www.parasoft.com/solutions/static-code-analysis/`) tools cover the rule, however, being proprietary tools and not freely available for research we do not cover them in our evaluation.

[30]`https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html`

Algorithm 1 Collect methods used in threads

```
1: methodsInThread ← ∅
2: for all method invokation bytecode instruction in methods do
3:     if method invokation implements "java.lang.Runnable" and is passed to
   "java.lang.Thread" then
4:         methodsInThread ← invokedMethod          ▷ This is a starting point
5:     else if CONTAINS(methodsInThread, currentMethod) and
   currentClass = invokedClass then
6:         methodsInThread ← invokedMethod
7:     end if
8: end for
```

When detecting the issue the class context is scanned twice, this is necessary, because the methods are visited in the order of definition, not in call order, so every method is visited once during one scan. The issue can only happen in methods which are used by threads, so in the first scan these relevant functions are gathered (as it is shown in Algorithm 1).

The second time the algorithm visits the code (see Algorithm 2), it looks for the usages of variables inside the stored methods. It collects the operations and groups them by variables which are performed on variables with not atomic or synchronized types.[31]

Every operation on the referenced fields - which meet the type constraint - is processed and the following boolean flags are saved about each variable:

- `onlySynchronized` is `true`, if all modifier operations are under proper synchronization.
- `onlyPutField` is `true`, if the threads only assign new values to the field.
- `modified` is `true`, if a thread modifies a variable or assigns a new value to it either directly or via a method call.

The `onlySynchronized` flag is necessary because, in the end, only those variables are relevant that have at least one operation not properly synchronized.

An acceptable solution could be that the threads only assign new values to fields and don't perform any other operations on them. E.g., construct the `message` variable with the help of `Builder` class, the construction of the object is finalized by calling the `build()` method. This pattern makes the `Message` class immutable and, consequently, thread-safe. The `onlyPutField` flag helps to identify this special case.

The `modified` flag is used to decide if there are multiple threads with only reading operations, since then it does not lead to race condition, but if at least one thread modifies the referenced object's state, then the state is not ambiguous in the threads.

[31] Improper usages of variables with Java's built-in atomic types and synchronized collections are handled by the VNA03-J algorithm.

A bug is reported, if a field is modified in multiple threads, accessed outside of synchronized blocks, and is neither a synchronized collection nor an atomic typed field. The algorithm marks instructions as a bug if shared data is used in multiple threads, with at least one modifying its state without a consistent locking policy.

The algorithm marks all instances of not thread-safe field accesses as a potential bug, because it helps the developer to identify which statements require synchronization, so in general it makes easy to locate and accurately determine the appropriate scope of the `synchronized` block necessary to ensure thread safety in a method.

Algorithm 2 Collect operation data in threads

```
1: List methodsInThread                                   ▷ Inherited from Algorithm 1
2: Map⟨Field, FieldData⟩ fieldInThreads ← ∅
3: for all bytecode instructions in methodsInThread do
4:     if (field assignment or method call on field and
   CONTAINS(methodsInThread, currentMethod)) then
5:         data ← GETORCREATE(fieldsInThreads[field])
6:         data.onlySynchronized ← data.onlySynchronized ∧ isSynchronized
7:         data.onlyPutField ← data.onlyPutField ∧ isFieldAssign
8:         data.modified ← data.modified ∨ isFieldAssign ∨ looksLikeSetter
9:         fieldsInThreads[field] ← PUTORUPDATE(data)
10:    end if
11: end for
```

5.2 Finding non-atomic usages of the Java Concurrent types

The `atomic` typed fields and collections have atomic methods that are inherently atomic. The algorithm based on the rule *VNA03-J* searches for scenarios where these atomic methods are used in a combined or sequential manner. If a shared data is used more than once, the operations together are not atomic, so these accesses are marked as bug. There may be the possibility that all shared data are used just once in a method, but if combined (for example `a.get().add(b.get())`) then it is a bug too, because these two resource accesses must be atomic not only individually. It is important to note that, if a shared data is used by multiple methods and at least one accesses it more than once, all methods that work with it need synchronization for consistent locking to avoid race conditions between the threads that are using the same shared resource at the same time while parallel running.

The algorithm has the following base logic: analyzing functions in a class context and mark each method call and field assignment of common objects which are not `synchronized`. If a method contains a `synchronized` block, the detector logs only once for every different object inside the block, no matter how many times they are accessed; because of the synchronization, it is considered an atomic operation. If a private method performs an unsafe operation without proper synchronization, but all the methods that call it have proper synchronization, then the private function does not need another one.

Shared data could be a field of the class, a function argument, or a local variable containing a reference for a shared resource, for example, an element of an atomic collection.

The algorithm also visits the class two times: first, the `atomic` or synchronized collection typed fields of the class are collected. For fields with types inherited from the `atomic` package (such as `AtomicInteger`, `AtomicLong`, `AtomicBoolean`, and `AtomicReference`) only the type needs to be checked, but finding the synchronized collections is a bit more complex: the fields only has a general `List`, `Set` or `Map` type and the algorithm must look for the field assignments to determine the concrete type. For example, Code 10 shows a synchronized list assignment:

```
1 List<String> lst = Collections.synchronizedList(new ArrayList<>());
```

Code 10: Create a synchronized list

Overall, a method which creates a synchronized collection can be recognized by being in the `Collections` class[32] (of the `java.util.concurrent.atomic` package), and its name starting with "synchronized" followed by the concrete collection type's name (e.g. `synchronizedSet`, `synchronizedMap`). The checker stores the variables, which are assigned the return value of these methods.

Algorithm 3 Check if a Class Member's type is `atomic` or a synchronized collection

```
1: function IsAtomicTypedField(classMember)
2:     methodNames ← GetMethodNamesReturningSyncCollections()
3:     className ← GetClassName(classMember)
4:     isCollectionsClass ← "java.util.Collections" = className
5:     isAtomicClass ← className.StartsWith("java.util.concurrent.atomic")
6:     isNameInteresting ← Contains(methodNames, className)
7:     return (isCollectionsClass ∧ isNameInteresting) ∨ isAtomicClass
8: end function
```

It is possible that a collection typed field has more than one assignment, and not all of them are synchronized collection assignments, for example, in the constructor a `List` collection is only assigned a simple `ArrayList` value, but after the application starts running, it is assigned a `synchronizedList` value. In this case, the checker treats this field as a synchronized collection, it assumes that the variable is used in concurrent operations.

In the second visit using the data of the collected variables, the detector looks for operations on these fields in every method in the class, except the constructor and synchronized methods. Constructors (as well as the static initializer) only run on object creation once, they do not appear in parallel operations, they may initialize fields, but this is part of the life-cycle of the object and can run only once. In addition to the stored fields, there may be local variables or method arguments with `atomic` types, and the checker also has to mark operations on these variables.

[32] https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html

Overall, the following operations are logged:

- A value assigned to a stored variable.
- A method called on a stored field.
- An operation on an `atomic` typed local variable or method argument.
- Multiple `atomic` typed variables combined.

When the algorithm has logged every instruction that meets the above list, it cumulates the results to determine which operations need to be reported as a bug. First of all, logged private methods are removed if they have proper synchronization on the call site in every method. Code 11 shows an example for this:

```
1  private AtomicInteger count = new AtomicInteger(0);
2
3  public void modifyCountSafely() {
4      synchronized (count) { // Every other caller methods call the
           private method with proper synchronization
5          incrementAndPrint();
6      }
7  }
8
9  private void incrementAndPrint() {
10     count.incrementAndGet();
11     System.out.println("Current count: " + count.get());
12 }
```

Code 11: Synchronized private method on the call side

This optimization only works with private methods because, with higher visibility, these methods can be accessed from outside the current class, and their usages are unknown (see Subsection 7.2).

After this, the algorithm has the information, which `atomic` typed fields are accessed multiple times by multiple methods without proper locking strategy. If operations are performed in multiple methods, these are marked as a bug, and it informs the developer to put these lines under a synchronization or refactor the usage strategy of the shared resource.

5.3 Generalization possibilities

Although our current algorithms are explained through the example of the Java language and contain language specific details, such as the `Atomic*` classes and standard library synchronized collections, our underlying detection logic can be extended to other programming languages, if they follow similar usage patterns (e.g. those mentioned in Section 2).

The core principle of identifying multiple potentially conflicting operations on shared data applies generally to any abstraction that offers atomic operations, but it can be composed unsafely if not synchronized consistently. However, in other programming languages, the same algorithmic idea remains valid, it must be adapted

to detect their particular locking primitives – for example, `std::mutex` in C++, or `threading.Lock` in Python – instead of Java's `synchronized` blocks.

In SpotBugs, many Java-specific base type names are hardcoded rather than making them project-configurable. Since these elements are essentially part of the standard library, it is typically more practical and efficient to hardcode some parts of the detection logic than to parameterize it for every possible project. The synchronized collections used by the algorithms are specific to Java, but this approach could be extended to any language. For example, in C# the .NET Framework[33] offers `ConcurrentDictionary`, `ConcurrentQueue`, and `ConcurrentBag`, all of which ensure thread safety without requiring explicit external locking.

By enumerating these known synchronization and atomic constructs in other languages, the algorithms could be extended beyond Java to automatically detect non-atomic compositions of supposedly *atomic* operations.

6 Results

To validate our checkers, we implemented a considerable number of unit test cases to eliminate potential bugs and filter out possible false positive cases. After that, we evaluated our checkers on large, modern, open source Java projects, which were selected based on the following criteria:

1. **Concurrency intensity:** The project needs to use multithreading or concurrent data structures extensively.
2. **Codebase size and activity:** The project should be large and actively maintained, ensuring real-world relevance.
3. **Popularity and community participation:** The project should have a diverse contributor base and a significant user community, allowing meaningful feedback on potential bugs.

The unit tests are written in the SpotBugs testing framework, which makes them suitable to be used as integration test. Every test runs without issues, as expected.

For the *VNA03-J* there are *46* test cases overall: *22* positive and *24* negative to cover all possible mechanisms, such as edge cases like if there are multiple synchronized blocks in a method, but not every operation is inside, or lambda or anonymous method is passed to an atomic field's method call as argument, but this method itself performs additional operations on that same atomic field. Every test case has its own example class (which may have inner classes depending on the test's complexity) with a unique usage of atomic field(s).

For *VNA04-J* there are *14* test cases, with *10* positive and *4* negative. It has fewer test cases than the other checker, because in this case it is not necessary to include several atomic based types, it just works with any type that is not in Java's `atomic` package. However, it also includes some special cases like handling that if

[33] https://learn.microsoft.com/en-us/dotnet/standard/collections/thread-safe/

a `Thread` is passed to Java's `Runtime`[34] as a shutdown hook, or to verify that the checker can also work and detect bugs correctly with nested classes.

6.1 Results of detection the VNA03-J rule

The test results on the projects (can be seen in Table 2) show that the *VNA03-J* detector has low hit rate. We performed a *manual review* of each found bug by examining the relevant code regions, verifying whether the detected pattern could indeed lead to a race condition or data inconsistency. We confirmed that the reported issues were legitimate concurrency pitfalls. We found no code usage that was mistakenly classified as problematic, and we believe that *all the identified hits were true positive.*

Table 2: VNA03-J Measurements on large, open source projects

Project	Lines of Java Code	Atomic variables	Combined access bugs	Simple access bugs
Bt[35]	78 483	25	3	7
MATSim-Libs[36]	679 033	47	24	9
OpenGrok[37]	132 290	1	0	0
Kafka[38]	980 184	213	40	75
ElasticSearch[39]	3 149 220	339	71	99

Combined atomic accesses are reported where atomic variables are accessed multiple times in the same function without synchronization and marked cases of *simple atomic accesses*, when the access needs synchronization due of the existence of the combined resource usages in other methods.

Code 12 is a code snippet, a simplified version of the `Counter` class[40] originally from the MatSim-Labs open source repository, represents a real true positive finding. The class-level variable `nextCounter` is accessed multiple times – once with a `get()` call and again with a `compareAndSet()` call in the `incCounter()` method. These calls constitute an unsynchronized *combined atomic accesses* bug. Consequently, the `reset()` method, which also modifies `nextCounter`, can overlap with `incCounter()`, resulting in a *simple atomic access* bug.

[34] https://docs.oracle.com/javase/8/docs/api/java/lang/Runtime.html
[35] https://github.com/atomashpolskiy/bt/commit/6041303
[36] https://github.com/matsim-org/matsim-libs/commit/1c6779d
[37] https://github.com/oracle/opengrok/commit/077089f
[38] https://github.com/apache/kafka/commit/b436499
[39] https://github.com/elastic/elasticsearch/commit/44c92715
[40] https://github.com/matsim-org/matsim-libs/blob/1c6779d/matsim/src/main/java/org/matsim/core/utils/misc/Counter.java

```
1  private final AtomicLong counter = new AtomicLong(0);
2  private final AtomicLong nextCounter = new AtomicLong(1);
3
4  public void incCounter() {
5      long i = this.counter.incrementAndGet();
6      long n = this.nextCounter.get();
7      if ((i >= n) && (this.nextCounter.compareAndSet(n, n*multiplier)
           )) { // combined atomic access bug, multiple accesses
8          log.info(this.prefix + n + this.suffix);
9      }
10 }
11
12 public void reset() {
13     this.counter.set(0);
14     this.nextCounter.set(1); // simple atomic access bug
15 }
```

Code 12: Example of the relation between the bug types

While our research was more to find out the possibilities of detecting concurrency related errors with static analysis, we intend to apply our tool for solving practical problems. We initiated discussions with the maintainers of the projects where we find possible problems and we are looking for their feedback whether the findings were true positives. We hope a more intensive communication with theses developers when the new version of SpotBugs including our checkers will be available for the larger community.

6.2 Results of detection the VNA04-J rule

The test results for the *VNA04-J* rule on large, open source projects (with the same versions that are noted in Table 2) can be seen in Table 3. We found no hits on the evaluated projects because we opted to use the algorithm variation that excludes main thread analysis. This decision was made to avoid the checker being so noisy that it is unusable (see the details in Section 5.1).

Table 3: VNA04-J Measurements on large, open source projects

Project	Lines of Java Code	`Thread` starts	Unsafe access bugs
Bt	78 483	1	0
MATSim-Libs	679 033	1	0
OpenGrok	132 290	4	0
Kafka	980 184	1	0
ElasticSearch	3 149 220	0	0

It is very rare that in one class more than one `Threads` run parallel, which are using common resources. The detector has a serious limitation; it only works within the class context, and cannot see the relations, method calls outside of a class. This limitation is inherited from the SpotBugs framework, as discussed in Section 7. Is it possible that the public functions of a class are run in different threads in parallel way, but the checker cannot detect that usage.

6.3 Performance and integration with SpotBugs

SpotBugs keeps the analyzed application classes in memory (via BCEL) and performs separate passes on each classes for all enabled detectors, which are completely independent of each other, there is no shared data or any connection between them. This design simplifies the analysis process and avoids unintended interactions between detectors.

Our newly introduced concurrency checkers are implemented as additional detectors in SpotBugs. The *VNA03-J* algorithm, due to certain implementation details, is a CFG-Based detector, which means it is more expensive to run, than the visitor-based detectors, like the *VNA04-J*. To quantify runtime overhead, we ran SpotBugs on the *Kafka* codebase (same commit as noted in Table 2) on an Apple MacBook M2 Pro, repeating each run five times. We used the following command for benchmarking:

```
./gradlew clean core:spotbugsMain core:spotbugsTest -x test \
    --rerun-tasks --no-build-cache --profile
```

Table 4: Benchmark results on Kafka, run on an Apple MacBook M2 Pro

Configuration	Avg. Runtime
No concurrency detectors	15.21s
VNA03-J enabled	18.46s
VNA04-J enabled	15.54s
VNA03-J and VNA04-J enabled	18.79s

These results align with SpotBugs' documented tendency for CFG-based detectors to incur more overhead than visitor-based ones.

Similar to other SpotBugs detectors, our checkers rely on SpotBugs' in-memory class repository and do not add extra complex data structures. Consequently, we anticipate negligible memory overhead even when analyzing large-scale projects. Compared to existing detectors, our concurrency checkers primarily store per-class metadata for pattern matching and analysis and follow similar design principles and performance characteristics.

7 Known limitations and possibilities for further development

This section highlights limitations in the current implementation of the SpotBugs detectors and suggests some possible opportunities to improve them.

7.1 Use of atomic typed variables in public methods

Atomic types are typically used to ensure thread safety. However, if they appear in public methods not utilized in a threaded context, hits of the algorithm based on the rule *VNA03-J* are false positives. The algorithm assumes that, if the developer used atomic-typed variables, then it is because it is used in threads. This limitation could be solved by analyzing the relations of the classes (see Section 7.2), and exclude those public methods from the analysis which are not run in parallel way.

7.2 SpotBugs' class context analyzing limitation

SpotBugs operates within a single-class context, as such can only analyze operations within a class. This restriction may lead to potential false negatives in both detectors in systems where class interactions play a crucial role in the application's concurrency logic.

Without these limitations, the checker implementation of the algorithm based on the rule *VNA03-J* could exclude those nonprivate methods, which are not running in parallel threads, and the algorithm with rule *VNA04-J* could include those nonprivate methods which are passed to a `Thread` in another class, or just called by another method outside of the analyzed class that is running in parallel thread.

7.3 False positives in single-function modifications

Another issue is related to the scenario where multiple threads modify a common field, such as incrementing a counter. While this might technically represent a concurrency issue, if each thread's modification is self-contained and thread-safe (like atomic increments), it currently triggers a false positive.

```
1  private AtomicInteger count = new AtomicInteger(0);
2
3  // Thread 1:
4  public void incrementByOne() {
5      count.incrementAndGet();  // Safe atomic operation
6  }
7
8  // Thread 2:
9  public void incrementByTwo() {
10     count.addAndGet(2);  // Another safe atomic operation
11 }
```

Code 13: Example of safe usage of parallel modifications

It is hard to determine by method names, what are those operations that only modify the common data, so since there were no hits like this in our evaluations of large open source projects, we chose to leave this false positive chance in the algorithm because it is not noisy for the developers.

7.4 Challenges with Lambda Expressions

The introduction of Lambda Expressions[41] into Java 8 marked a significant milestone in the evolution of the language. This feature was one of the most anticipated additions to Java and fundamentally changed the way Java programmers write code, especially when dealing with collections and concurrency.

Technically, a lambda expression in Java is an instance of a functional interface[42], an interface with a single abstract method (SAM interface). The Java compiler infers the type of lambda expression from the context in which it is used, allowing simpler and more concise syntax.

Lambda Expressions are implemented under the hood as bootstrap methods using the `invokedynamic` bytecode instruction. With the SpotBugs framework, there is some limitation to analyze lambda methods, because the calls and operations on these kinds of method are different due to the specialized bytecode instruction, and certain features are either not implemented or implemented in an alternative manner in the current version of SpotBugs, resulting in the loss of some information during analysis.

8 Conclusion

In concurrent programming, it is crucial to use shared resources in a thread-safe way. To achieve this, it is recommended to use a consistent locking policy, which could be even necessary, when a program works with Java `atomic` based types or with synchronized collections. Static analysis is a very useful tool to look for mistakes and make sure the developers identify and rectify potential errors, and implement their concurrent logics in a proper way.

We analyzed practices in thread safety, not only in Java but also by reviewing methodologies in other programming languages, such as Python, Rust and C++.

We delved into the SEI Cert Coding Standards, which is pivotal in guiding developers toward safer coding practices. Our research into this guideline was not just theoretical; we applied part of these standards practically by designing an algorithm and implementing corresponding checkers (which cover the *VNA03-J* and *VNA04-J* rules) in SpotBugs Static Analyzer Tool.

By integrating new detectors, our research has directly contributed to the enhancement of this tool, allowing it to identify unsafe resource usage across concurrent threads more effectively. The addition of these detectors extends SpotBugs' capa-

[41] https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html
[42] https://docs.oracle.com/javase/8/docs/api/java/lang/FunctionalInterface.html

bilities, enabling it to catch subtle bugs that could otherwise lead to inconsistent states or even system failures in production environments.

The enhancements in SpotBugs that we implemented offer practical benefits to developers by reducing the time and effort required to identify concurrency issues. This not only increases productivity, but also improves the overall reliability of software applications. By detecting potential problems in the early stages of the development cycle, developers can address issues before they manifest in deployed systems, reducing downtime and maintenance costs. These advantages also enable managers to reduce the use of project resources and financial expenditures.

Furthermore, our work underscores the value of community-driven open source projects in the evolution of software development tools. Our contributions to the SpotBugs project exemplify how individual efforts can lead to significant improvements in tools that are widely used by the developer community. The advanced capabilities of SpotBugs, enriched with more robust detectors for concurrency issues, render it a valuable tool for developers aiming to write safer and more reliable Java applications.

References

[1] Aghav, I., Tathe, V., Zajriya, A., and Emmanuel, M. Automated static data flow analysis. In *2013 Fourth International Conference on Computing, Communications and Networking Technologies (ICCCNT)*, pages 1–4, 2013. DOI: 10.1109/ICCCNT.2013.6726670.

[2] Aho, A. V., Sethi, R., and Ullman, J. D. *Compilers principles, techniques, and tools*. Addison-Wesley, Reading, MA, 1986. ISBN: 9780201100884.

[3] Anderson, T. The performance of spin lock alternatives for shared-money multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, 1990. DOI: 10.1109/71.80120.

[4] 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.

[5] Boehm, B. and Basili, V. R. Software defect reduction top 10 list. *Computer*, 34(1):135–137, 2001. DOI: 10.1109/2.962984.

[6] Bíró, P., Kádek, T., Kósa, M., and Pánovics, J. A new method to increase feedback for programming tasks during automatic evaluation. *Acta Polytechnica Hungarica*, 19(9):103–116, 2022. DOI: 10.12700/aph.19.9.2022.9.6.

[7] Clang SA Static Analyzer, 2019. URL: https://clang-analyzer.llvm.org/.

[8] Coroutines, K. O. D. https://kotlinlang.org/docs/coroutines-overview.html. Accessed: 08 2024.

[9] Halim, V. H. and Dwi Wardhana Asnar, Y. Static code analyzer for detecting web application vulnerability using control flow graphs. In *2019 International Conference on Data and Software Engineering (ICoDSE)*, pages 1–6, 2019. DOI: 10.1109/ICoDSE48700.2019.9092687.

[10] Hampapuram, H., Yang, Y., and Das, M. Symbolic path simulation in path-sensitive dataflow analysis. *SIGSOFT Software Engineering Notes*, 31(1):52–58, 2005. DOI: 10.1145/1108768.1108808.

[11] Herlihy, M. Wait-free synchronization. *ACM Transactions on Programming Languages Systems*, 13(1):124–149, 1991. DOI: 10.1145/114005.102808.

[12] ISO/IEC. N4917 post-summer 2022 C++. Working draft, International Organization for Standardization (ISO), Geneva, Switzerland, 2022. URL: https://isocpp.org/std/the-standard. Accessed: 07 2024.

[13] Palša, J., Hurtuk, J., Chovanec, M., and Chovancová, E. Using machine learning algorithms to detect malware by applying static and dynamic analysis methods. *Acta Polytechnica Hungarica*, 19(7):177–196, 2022. DOI: 10.12700/aph.19.7.2022.7.10.

[14] Rice, H. G. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74:358–366, 1953. DOI: https://doi.org/10.2307/1990888.

[15] Scala Coroutines. URL: https://scala-coroutines.github.io/coroutines/. Accessed: August 2024.

[16] Zhang, X., Zhou, Y., and Tan, S. H. Efficient pattern-based static analysis approach via regular-expression rules. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 132–143, 2023. DOI: 10.1109/SANER56733.2023.00022.

Contents

ISSN 0324—721 X (Print)
ISSN 2676—993 X (Online)

Editor-in-Chief: Tibor Csendes