Acta Cybernetica 25 (2021) 205–221.

Towards Version Controlling in RefactorErl*

Jenifer Tabita Ciuciu-Kiss[ab], Melinda Tóth[ac],
and István Bozó[ad]

Abstract

Static source code analyser tools are operating on an intermediate representation of the source code that is usually a tree or a graph. Those representations need to be updated according to the different versions of the source code. However, the developers might be interested in the changes or might need information about previous versions, therefore, keeping different versions of the source code analysed by the tools are required. RefactorErl is an open-source static analysis and transformation tool for Erlang that uses a graph representation to store and manipulate the source code. The aim of our research was to create an extension of the Semantic Program Graph of RefactorErl that is able to store different versions of the source code in a single graph. The new method resulted in 30% memory footprint decrease compared to the available workaround solutions.

Keywords: Erlang, RefactorErl, graph version control, optimisation

1 Introduction

Static program analysis [3] is a method of debugging of the source code of a program performed before the execution. It is used early in development before any other testing. The benefits of static analysis need to be mentioned: speed (compared to manual code reviews automated tools are way faster and the early found coding errors are less costly to fix); depth (static code analysers can cover every possible code execution path); accuracy (human errors can be eliminated).

*The research has been supported by the project "Integrált kutatói utánpótlás-képzési program az informatika és számítástudomány diszciplináris területein (Integrated program for training new generation of researchers in the disciplinary fields of computer science)", No. EFOP-3.6.3-VEKOP-16-2017-00002. The project has been supported by the European Union and co-funded by the European Social Fund. The research is part of the "Application Domain Specific Highly Reliable IT Solutions" project that has been implemented with the support provided from the National Research, Development and Innovation Fund of Hungary, financed under the Thematic Excellence Programme TKP2020-6 (National Challenges Subprogramme) funding scheme.

[a]ELTE, Eötvös Loránd University, Budapest, Hungary
[b]E-mail: kuefmz@inf.elte.hu, ORCID: 0000-0002-3170-6730
[c]E-mail: toth_m@inf.elte.hu, ORCID: 0000-0001-6300-7945
[d]E-mail: bozo_i@inf.elte.hu, ORCID: 0000-0001-5145-9688

DOI: 10.14232/actacyb.289386

Version control [18] (also known as revision control or source control) is a component of software configuration management. It is used for managing changes to programs, documents and other data. The idea of storing the differences between different versions of a document is as old as writing is but it became more important in the last two decades when the area of computing has started. In software engineering, version control is any kind of tracks that provides control over the changes between the different version of source code. Version controlling tools make the process easier and much faster, so it is very convenient to use them. One of the most famous tools for that is Git [8], which is known by most software developers.

RefactorErl [4, 19] is a static source code analyzer and transformation tool for Erlang programs. Beyond that RefactorErl does thorough static analysis on the given source code, it has many other features to support code comprehension: it finds the dependencies between the modules, provides a query language to gather semantic information about the source code, performs clustering, etc. Some source code analysers have their own built-in version controlling systems. RefactorErl does not provide version controlling, it works on a given snapshot of a software.

The internal representation of RefactorErl gives us the opportunity to build the version control over that. The tool represents the source code in a Semantic Program Graph (SPG) [12] which will be explained in detail later. The graph is stored in a database to make the results of the source code analysis permanently available. The tool can work with different databases (i.e. the distributed, relational Mnesia database [14], or the key-value Kyoto database [10]), but it builds the same internal representation, the SPG, in any cases. All the analyses and the transformations are performed on the graph layer, thus our version controlling is defined on the top of the SPG and not on the database layer.

The main contribution of this paper are the algorithms as an extension of RefactorErl to handle different versions of the source code. Our first tests on open source repositories provided a memory saving for about 30%.

In Section 2, we introduce some information about the RefactorErl tool and the data structure used by the tool. In Section 3, the algorithm of finding the differences between different versions is presented which is followed by Section 4, where the algorithm of storing the found differences is discussed. Then, in Section 5, we present some usage of the version control. In Section 6 the evaluation is explained. Section 7, contains information about the related work. Finally, Section 8 concludes the paper.

2 Background

In our work we focus on the RefactorErl framework that was designed to analyse Erlang source code.

Erlang The Erlang [6, 1] programming language is a general-purpose, competitive, dynamically typed programming language. Other important features include being a greedily evaluated, non-purely functional and open source language. It is

designed to develop systems with high fault tolerance and real-time robust size. Erlang programs run inside a virtual machine (Erlang VM or *node*), so programs written in Erlang are platform-independent. Erlang's standard library is called OTP (Open Telecom Platform [13]), which was originally designed to write telecommunications software, but can be used more widely. The Erlang runtime environment and OTP are collectively referred to as Erlang/OTP.

RefactorErl Static source code analyser tools are heavily used in software development and maintenance processes. RefactorErl [4, 17, 19] is an open-source static source code analyzer and transformer tool for Erlang source codes. Despite the fact that RefactorErl is still considered as a prototype tool, its usefulness in industrial usage has been proved already. It was initially developed at the request of Ericsson and it is still used by developers there. Apart from static analysis it has many other features: it is compatible with the most famous code editors; it can find the dependencies between the modules and it also represents this information in well designed graphs; it gathers many different kind of information about the source code; the found bugs can be investigated. These features manifest the significance of RefactorErl.

Semantic Program Graph During the initial analysis, the tool builds a Semantic Program Graph (SPG) [12]. In order to make the source code analysis results available for further use, we need to save the graph. The SPG is stored in a database. Different databases can be used to store the SPG, but all the analyses and transformations are manipulating the SPG itself. If the source code is changed, the tool updates the graph, so in the new graph, only the new information is available. Any kind of backups about different versions are really expensive at the current implementation, because we can use the backuping features of the used database only to backup the whole SPG. Therefore we would like to design a much more effective way of version handling on the SPG level. An adequate solution to this problem is to save the differences in the same graph. In this way, we do not need to store database backups and load them once an older version is needed.

The SPG is a three-layered rooted, directed, labelled graph structure that contains lexical, syntactic and semantic information about the source code. The different kinds of data are stored in the nodes of the graph (in the attributes of the node) that are linked through tagged edges. These edges contain structural information about the nodes. The graph has a specified structure. It starts with a root node, which is followed by a node containing the information about the file (the name, absolute path, etc.). They are linked with an edge tagged with the file. The file node is linked to its forms: the function definitions and attributes. The function definition forms are linked to its clauses, and so on. In addition, there are module and function semantic nodes to represent the semantic information as well. Based on the structure of the graph, we were able to define an algorithm to find the differences between two versions of the source code by traversing the representing syntax trees simultaneously.

Motivation During a software development in industry, versions occur quickly and many times the difference between them are important since the new change could lead to some error or the previous version was more effective, etc. Storing the versions in a static analyzer tool helps the developers to fasten the process of debugging, not to mention the analyses of the relations between the versions that the tool can provide.

As it was previously mentioned, any kind of backups are expensive since RefactorErl can store only one version of a source code at once. It is possible to manually create backups of the different versions on database level, but a single running instance of RefactorErl can handle only one version at the same time. Loading an old version of the source code requires to start a new RefactorErl instance and load the whole old SPG stored in the backups. That is an inconvenient, time-, memory- and resource-consuming task.

3 Detecting differences

As it was mentioned before, RefactorErl generates an SPG for each module and it stores the analyzed source code in this structure. When we re-analyze a module that has already been analyzed, the tool recognizes this and replaces the subgraph belonging to the modified source code part[1] with a new subgraph, but the old and the new version of the software have not been related to each other so far. The old subgraph is deleted after the new subgraph is created, however, there is a point in the analysis when the information for both graphs can still be found in the tool. We use this point to analyze the two subgraphs, detect differences, and perform version control.

In our version controlling, the primary goal is to somehow explore the differences between the two graphs, and then, in some way, to represent the information stored by the two subgraphs in a merged subgraph.

The differences can be categorised into three major groups that need to be recognized and addressed in different ways. These three groups are:

- Update
- Deletion
- Insertion

Let us look at how to handle each case properly. For that we will use an example seen on Figure 1. The example contains two function definitions. The first one, `foo`, checks whether its argument is `0` and returns the term zero. Function `op` multiplies its arguments.

[1] RefactorErl performs an incremental analysis on form level: once a form (e.g. function, attribute, etc.) changed in the source code, it reanalyses only the changed form

```
foo(0) -> zero.
op(A, B) -> A * B.
```

Figure 1: Example Erlang function

3.1 Update

We talk about an update when we had something in a version, and we changed something in it, without inserting a new part or deleting a whole part of the source code.

For a better understanding we make a modification on the example presented in Figure 1. As it can be seen in Figure 2, we made an update on the return value of the function **op**, the multiplication operation (`*`) has been changed to plus (`+`).

```
foo(0) -> zero.
op(A, B) -> A + B.
```

Figure 2: Example Erlang function with update

The algorithm of finding the updates is the most important, because after resolving those updates we will consider the remaining changes either as a deletion or an insertion depending on where the extra elements on the graph were found.

As a first step we make all combinations of function nodes and store them in a list[2]. We will go through these function pairs and check if they are the same function in different versions. For that we check whether the name and the arity of the functions are the same. If so, we say that these two functions are the same and we look for the differences between those functions. This algorithms works well for real code examples as the name of the function rarely changes after a commit[3].

The recursive function of finding the differences gets two nodes, one of them is from the SPG of the previous version and the another one is from the new version. The algorithm can be found in Figure 3. The list of pairs that the algorithm initially gets is the list of the investigated form nodes. The `TableOfDifferences` variable refers to the ETS table [7] in which the difference nodes are stored. While analyzing a new version of a file RefactorErl recognizes the new, deleted and modified forms of the source code based on the hash value of the forms. At this point we save the changed forms[4] in ETS tables for further analysis. The `equivalent` function checks whether the two nodes are the same in the required attributes (number and type of attributes, number of children, etc.), and if not, we say that we found an update at that node and we insert the node and the subgraph bellow it into the version

[2]We need to generate all function pairs because at this point we do not know their names yet, as the name is stored in a lower level in the graph.

[3]In the current implementation, we recognise renaming of function definitions as newly inserted and deleted functions. Some heuristics can be applied to change this behaviour.

[4]We save the sub-syntaxtrees representing the changed forms.

controlled graph with `insertUpdate` function. If we did not find an update, we prepare the list of pairs of the children nodes with `makePairs` function and check whether the children contain updates. Each edge label has a name and a serial number in the SPG representing the order of the syntax tree elements. Thus the `children` function organizes the nodes in this way, so we are able to create the pairs for the next step in the recursive investigation.

```
selectUpdate(ListOfNodePairs)
    for NodePair in ListOfNodePairs do
        Children1 = children(first(NodePair))
        Children2 = children(second(NodePair))
        if NodePair in TablesOfDifferences then
            if !equivalent(Children1, Children2) then
                insertUpdate(Children1, Children2)
            else
                ListOfChildrenPairs =
                     makePairs(Children1, Children2)
                selectUpdate(ListOfChildrenPairs)
            end if
        end if
    end for
    return ok
```

Figure 3: `selectUpdate` function

3.2 Deletion

A deletion is when we had something in the source code in a previous version, and we deleted it in a version after that.

For finding the deletions, we go through the nodes that have been present in the previous version of the source code but they are not present any more. That means that we go through the SPG and if we find a node that does not have a version in the new graph then we conclude that a deletion is found.

In Figure 4, one can see that we made a deletion by deleting the function `foo`.

```
op(A, B) -> A * B.
```

Figure 4: Example Erlang function with deletion

3.3 Insertion

An insertion is, when we insert a brand new part in the source code, which did not exist in the previous version.

We use a similar algorithm here as for deletion, the difference is that when we look for insertions we look for those nodes that are present in the new graph but they had not been present in the old graph.

As an example, we made an insertion to the code seen in Figure 4, so after making a deletion we do an insertion on that code. We add the function `bar` and the result code can be seen on Figure 5.

```
op(A, B) -> A * B.
bar(1) -> one.
```

Figure 5: Example Erlang function with insertion

4 Updating the graph

After finding and identifying the different types of changes, we need to represent them in the SPG. The basic idea is that we introduce two different types of tags for the edges. Updates, deletions and insertions are handled in a similar way. Thanks to RefactorErl, we can easily and efficiently insert new nodes and edges. In the following, we will demonstrate the algorithm for each case.

Let us consider the examples used in the previous section to present the method we used for updating the graphs. To make the differences more illustrative, first we take a look at the non version controlled graph which is generated by the RefactorErl. You can find that in Figure 6.

4.1 Update

In the case presented in Figure 2, we find the difference presented in the previous section. We insert the whole subgraph under the found difference between the operations with a versioned edge. An edge is versioned when it has the same name as before but it has a v at the beginning of its name. In the resulting graph under the difference node, we can find the content of both versions in the different subgraphs. One can see in Figure 7 the edge *vbody* which marks the discussed update.

In this way, we can easily find the versioned part in the graph and we can keep all the previous features of the tool. An obvious question may arise, whether it is possible to store more than two versions in this way. Yes, it is possible, but we need to make it customisable by the users.

4.2 Deletion

The handling of this difference is very similar to the algorithm from the previous subsection. Let us consider the modified example in Figure 7 again. We can see that the difference seen in Figure 4 discussed in the previous section is released.

Figure 6: The not version controlled SPG

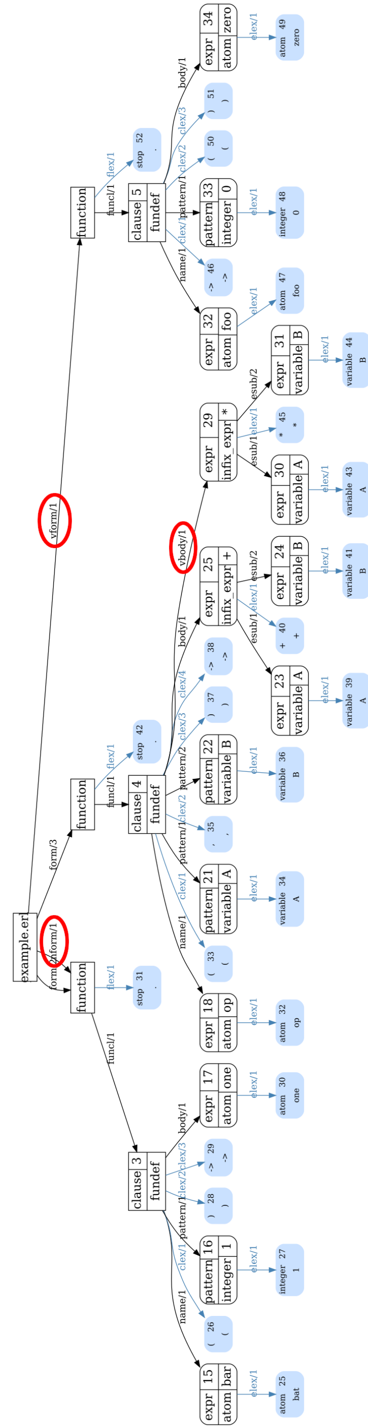

Figure 7: The version controlled SPG

To handle this, we insert the deleted subgraph under its parent node (the original function node) with a *vform* versioned edge seen on Figure 7.

4.3 Insertion

Considering the example seen on Figure 5, we have an inserted function in the code and we want to represent that in the version controlled graph. For that, we insert the difference in the same way but for the tag of the edge, we insert an n at the beginning instead of the v, one can see the inserted *nform* edge on the Figure 7. When we do that, we keep the nonversioned edge as well, because it does not bother us and it is needed to have a syntactically correct graph. Now, we can easily find the insertions since it had a different concept than the deletions and updates.

5 Gathering information

At this point we have the version controlled SPG saved in the tool. We would like to extract information from the graph and for that we use the query language which was already part of the tool and we extended it for the version controlled graph.

The semantic query language [11] was designed to query information about the analyzed Erlang code. The concepts of the query language are defined according to the semantic units and relationships of the Erlang language, e.g. functions and function calls, records and their usage, etc.

The elements of the language are the following entities: module, function, variable, etc. Each of them has several selectors and properties. A selector selects a set of entities that meet the given requirement. A property describes some properties of an entity type.

One of the needed information pieces could be to define whether the modules are version controlled or not. For that we performed the query seen in Figure 8. The query returns whether the uploaded modules are version controlled or not, in other words, its return value is a Boolean which is true if a module contains information from a previous version and false otherwise.

```
ri:q("mods.is_versioncontrolled").
ri:q("mods.is_versioned").
```

Figure 8: "Is the module version controlled" query

For a huge repository, this information might not be needed for all the modules or we would like to know which specific functions are version controlled. For that we realized a query found in Figure 9. It is quite similar with the first one apart from that it defines whether the functions in the modules are version controlled or not in the same way.

```
ri:q("mods.funs.is_versioncontrolled").
ri:q("mods.funs.is_versioned").
```

Figure 9: "Is the function version controlled" query

Also we might need the names of the version controlled functions and the query presented in Figure 10. It is built on the query presented in Figure 9. The return value of this query is a string which contains the name of the version controlled function in case that it is version controlled and `not_versioncontrolled` otherwise.

```
ri:q("mods.funs.versname").
ri:q("mods.funs.versionname").
ri:q("mods.funs.versionedname").
ri:q("mods.funs.vname").
```

Figure 10: "Name of version controlled functions" query

It is possible that we are not sure about the stored version on the SGP. This problem can be solved using the query presented in Figure 11. This query gives the number of the included versions on graph for each module. The returned value is an integer which is one if only the actual version is stored, two if the previous version is also included on the graph, etc.

```
ri:q("mods.version_counter").
ri:q("mods.vercount").
ri:q("mods.versions_number").
ri:q("mods.vernum").
```

Figure 11: "Number of stored versions" query

6 Evaluation

We made some measurement for the algorithm using a public GitHub repository [5] and two of its versions, the used old version can be found with change-id `499de032d7c6fc294aff977bea1405e09a1a3eae` and the new version can be found with change-id `5b7363c14071abe92d4039a7fc96371bd6eee91e`. The number of lines that differ between the two versions is 646.

To investigate the measurement, we check the tool's memory footprint[5] in five different states of the tool: when the tool is empty; when only the first version is loaded into the tool; when only the second version is loaded to the tool; when both

[5]We used the $erlang : memory()$. function.

Table 1: Results for version controlling a public repository

	ets	**total**
Empty	1399208	**23318920**
Old version	90707888	**119560720**
New version	94655240	**123790536**
Version controlled	106574824	**177795384**
Old- + New- version	183963920	**220032336**

of the version are loaded to the tool separately; when the versions are loaded to the tool using version control.

It is important to mention that the results are strongly dependent from the analyzed data. If there are almost no differences between the versions, we will get almost 100% efficiency. Meanwhile, when the source code has totally changed, the version control is not efficient at all, as both of the source codes have to be saved.

The results for the measurements presented above can be seen in Table 1 and Figure 12. As one can see, the tool's memory footprint is the smallest when the tool is empty. When only the first version is loaded, it has a smaller footprint than the second version. It is generally true as the code is developed over time. When both of the versions are loaded separately, the memory requirements are around twice as big as for the versions separately. When the algorithm of version control is used we can see that the memory requirement for the tool is less than for the case without using version control. This means that the algorithm works as it was expected at the first place. Quantifying the results in this case $32,52\%$ memory gain was achieved.

At the same time we also checked the overhead of the version controlling algorithm on the runtime: we experienced 4% overhead.

7 Related work

Version control of trees [2] presents everywhere in software development; generally it is solved by serializing the tree, not via methods operating on the tree itself.

Ralph Hinze and Ross Paterson have published a data structure called finger tree [9], which is purely functional. It can be used to efficiently implement other functional data structures. The finger tree, amortized, provides access to the "fingers" (leaves) of the tree where it stores the data. Depending on the size of the given part, we can concatenate or split data in logarithmic time in the size of the smaller piece. Internal nodes store what associative action their descendants have performed. The data stored in internal nodes can be used to provide its functionality to other non-tree data structures.

In the finger tree, the finger is the part where you can access a part of the data structure, in imperative languages this is called pointers. Fingers are structures

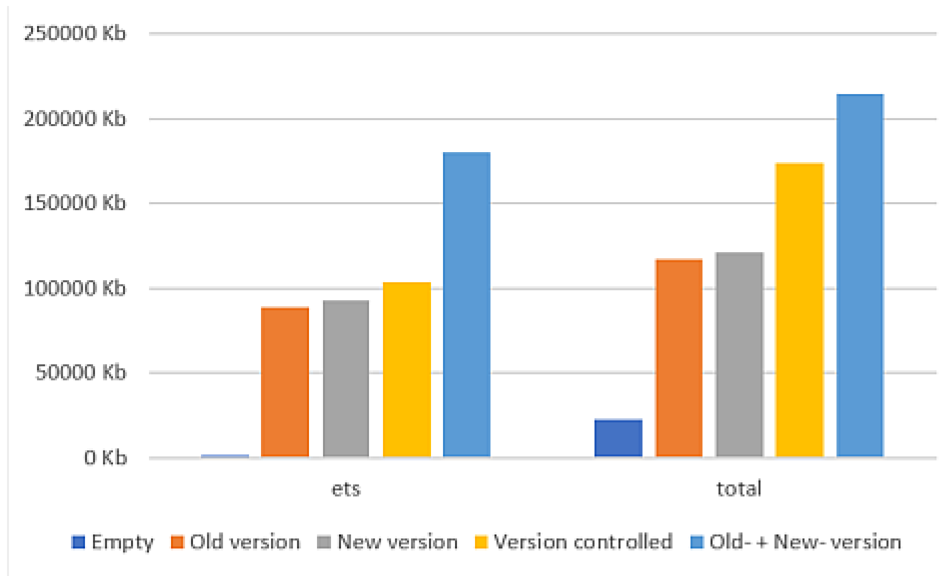

Figure 12: Diagram for results version controlling a public repository

that point to the end of a series or a letter to *node*. The fingers are part of the original wood, thus ensuring constant access over time. It also stores in each internal node the result of applying some associative operation to its descendants. This ”summary” data stored in the internal nodes can be used to provide the functionality of data structures other than trees. Finger trees are inherently persistent which means that older versions of the tree are always preserved. This data structure is often used for for handling the undo/redo operation in editors because it makes easy to find the differences in the text. The undo/redo actions can be interpreted as version control, but there some issues that need to be mentioned. The inside representation of RefactorErl does not make it possible to use the undo/redo method for version control without any changes, because the SPG (Semantic Program Graph) structure does not have the ”summary” data property, thus the finger tree change detection mechanism cannot be adopted easily for our use case.

UML diagrams also have version control issues [15]. The logic of the version control solution for UML diagrams is very similar to the version control implemented in RefactorErl. First, let us look at how a UML diagram is built. When designing UML diagrams, we distinguish between two types of diagrams, those that are semantically relevant and those that are not. It is important that UML diagrams also have a tree structure. *Node* has a well-defined type, and there can be different types of connections between different types of *node*. Some *node* is complex, they can carry several different types of information, but one can also store an entire subchart.

Differences need to be classified into different groups, as each difference needs to be treated differently. Let us look at what types of differences there can be between UML diagrams. First, we divide the differences into two groups, there are differences within *node* when only the content of *node* has changed, and there are structural differences when there has already been a change in the structure itself. Differences within *node* can be simple when only one attribute has changed, it can be multivalued when multiple attributes have changed, or it can be a change within an attribute reference when only the reference has changed.

In the event that there has also been a change within the structure, we must also divide them into several groups.

- Internal design difference: when there was no change within the elements, only the relationships between them changed.

- Shifting within a structure: when the structure remains the same as it was, only one or more edges moved.

- Internal node shifting: when we put operations or attributes from one class to another.

- Position shifting: when rearranging the order of attributes or operations within a class.

As the inside representation of RefactorErl does not force us to divide the handling of version control in as many different groups, it is important to know how different data structures are version controlled.

The difference computation of the version control algorithm of the UML diagrams works in a very similar way as in RefactorErl. Both of the algorithms go through the nodes in a breadth-first order and compare them to each other. The difference is that we consider changes inside nodes and for UML diagrams the differences are identified comparing the subgraph for each node. This would not be effective for us as the elements can be reorganized easily without making any significant change so we decided to compare each node pair only.

8 Conclusion

In this paper we presented how version control works in RefactorErl. After defining the different cases of modifications between different versions we explained how those differences can be found and represented in the tool. We also showed some use-cases and the results of the analysis of its efficiency.

The proof of concept implementation of the algorithm is integrated within the RefactorErl environment. In general, it has an overhead on the run-time, but it has a smaller memory footprint. The previous graphs for previous versions do not need to be saved, because the annotated new graph is storing all the required information

about the different versions. The algorithm can be optimized for different resources and different projects. We are investigating the possible usage of the versioned graph now.

For treating more than one version, an extra parameter needs to be introduced for counting the version number next to versioned edges. By this we can easily differ the different versions from each other and it also makes possible to store all the version in one graph.

8.1 Future work

The next step is the extension of the implementation to handle more than two versions. For this an index could be added for the version controlled edges (i.e. $vbody_1, vbody_2, \dots$). This way we can differ the versions easily. We would like to note here that only the representation of the changes need to be changed at this stage, the defined algorithm for finding and categorizing the differences is the same in all cases.

The algorithm could also we extended with some extra heuristic algorithms, so we could find the differences in as low level as possible. The heuristic algorithms could be defined after collecting some data from industrial usage.

The designed generic difference detection algorithm can be used in other projects as well. It would be a nice proof of the reusability of our approach to try it on the representation of the SSQSA [16] language independent source code analyser framework.

References

[1] Armstrong, Joe. Erlang. *Commun. ACM*, 53(9):68–75, September 2010. DOI: 10.1145/1810891.1810910.

[2] Asenov, Dimitar, Guenat, Balz, Müller, Peter, and Otth, Martin. Precise version control of trees with line-based version control systems. In *International Conference on Fundamental Approaches to Software Engineering*, pages 152–169. Springer, 2017. DOI: 10.1007/978-3-662-54494-5_9.

[3] Bellairs, Richard. What is static analysis (static code analysis). *Perforce Software*, 2020.

[4] Bozó, I., Horpácsi, D., Horváth, Z., Kitlei, R., Köszegi, J., M., Tejfel., 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, October 2011.

[5] Chesneau, Benoit. hackney - HTTP client library in Erlang. https://github.com/benoitc/hackney. [Accessed: 25 August 2020].

[6] Ericsson AB. Homepage of the Erlang Programming Language. http://www.erlang.org [Accessed: 2019.08.07].

[7] Fritchie, Scott Lystig. A study of Erlang ETS table implementations and performance. In *Proceedings of the 2003 ACM SIGPLAN Workshop on Erlang*, ERLANG '03, page 43–55, New York, NY, USA, 2003. Association for Computing Machinery. DOI: 10.1145/940880.940887.

[8] Github. https://github.com/ [Accessed: 20 December 2020].

[9] Hinze, Ralf and Paterson, Ross. Finger trees: a simple general-purpose data structure. *Journal of functional programming*, 16(2):197–217, 2006. DOI: 10.1017/S0956796805005769.

[10] Horák, Ales and Rambousek, Adam. The KYOTO database system. Technical report, Masarykova univerzita, 2010. http://www.kyoto-project.eu/.

[11] Horváth, Zoltán, Kozsik, László Lövei Tamás, Király, Roland, Tóth, Melinda, Kitlei, Róbert, Horpácsi, Dániel, and Bozó, István. Extended semantic queries on Erlang programs and comprehensive testing of RefactorErl. Technical report, Tech. Report 2010. Ericsson Hungary, 2010.

[12] Horváth, Zoltán, Lövei, László, Kozsik, Tamás, Kitlei, Róbert, Víg, Anikó Nagyné, Nagy, Tamás, Tóth, Melinda, and Király, Roland. Modeling semantic knowledge in Erlang for refactoring. In *Knowledge Engineering: Principles and Techniques, Proceedings of the International Conference on Knowledge Engineering, Principles and Techniques, KEPT 2009*, volume 54(2009) Sp. Issue of *Studia Universitatis Babeş-Bolyai, Series Informatica*, pages 7–16, Cluj-Napoca, Romania, Jul 2009.

[13] Logan, Martin, Merritt, Eric, and Carlsson, Richard. *Erlang and OTP in Action*. Manning Publications Co., 2010.

[14] Mattsson, Håkan, Nilsson, Hans, and Wikström, Claes. Mnesia - A distributed robust DBMS for telecommunications applications. In *International symposium on practical aspects of declarative languages*, pages 152–163. Springer, 1999. DOI: 10.1007/3-540-49201-1_11.

[15] Ohst, Dirk, Welle, Michael, and Kelter, Udo. Differences between versions of UML diagrams. In *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 227–236, 2003. DOI: 10.1145/940071.940102.

[16] Rakić, Gordana, Budimac, Zoran, and Savić, Milos. Language independent framework for static code analysis. In *Proceedings of the 6th Balkan Conference in Informatics*, BCI '13, page 236–243, New York, NY, USA, 2013. Association for Computing Machinery. DOI: 10.1145/2490257.2490273.

[17] RefactorErl. Static source code analyser and refactoring tool for Erlang. https://plc.inf.elte.hu/erlang. [Accessed: 8 August 2020].

[18] Sink, Eric. *Version Control by Example*. PYOW Sports Marketing, 1st edition, 2011.

[19] Tóth, Melinda and Bozó, István. Static analysis of complex software systems implemented in Erlang. In *Central European Functional Programming School*, pages 440–498. Springer, 2011. DOI: `10.1007/978-3-642-32096-5_9`.