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.
```

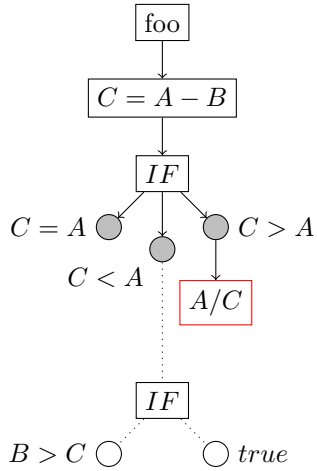

Figure 1: Example module and corresponding path selection

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.