Ludwig Maximilian University of Munich

Bachelor's Thesis in Informatics

Test-based Fault Localization in the Context of Formal Verification: Implementation and Evaluation of the Tarantula Algorithm in CPAchecker

Schindar Ali

Supervisor:	Prof. Dr. Dirk Beyer
Mentor:	Thomas Lemberger
Submission Date:	25.08.2020

SoSy-Lab
Software Systems

Eidesstattliche Erklärung

I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.

Hiermit versichere ich, dass ich diese Bachelorarbeit selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe und alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, als solche gekennzeichnet sind, sowie dass ich die Bachelorarbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt habe.

Munich, 25.08.2020 Schindar Ali

Acknowledgments

First of all, I would like to thank Professor Beyer for the opportunity to write this thesis in his group and thank you also goes to the group at the chair of Software and Computational Systems Lab for reviewing my code and for allowing me to use the great framework CPAchecker – without which this thesis would not have been possible. Special thanks go to my Mentor Thomas Lemberger for the excellent support. Whenever I needed help or had a question, I could rely on good advice, which was received almost instantly via Slack. I would also like to thank my parents, who raised me despite the many difficulties in an educated environment. I hope I can meet them again. I also thank my country Syria, who took care of me in its schools. My hope that one day the prosperity will return and I can visit it again. Many thanks to the Germans and Germany who recognized me as a refugee and who gave me the residence permit so I could continue studying normally.

Abstract

Many current fault localization approaches use test information to identify bugs in source codes, assuming that test cases satisfying a certain test adequacy criterion can provide adequate information for fault localization. Unfortunately due to the different purposes of fault locating and testing, test information is not sufficient for fault localization and can have a significant impact on performance. An alternative approach is a formula-based technique to fault localization, that's an algorithmic method that could provide distinctive facts for the recognized root causes.

This paper presents a novel idea of automatic fault localization by exploiting counterexamples generated by a model checker. We decided to evaluate the Tarantula algorithm for efficiency by using an Abstract reachability graph (ARG) instead of test suites, the key idea to find correlations between execution events and path outcomes — those events that correlate most highly with failure are suggested as places to begin an investigation. We (1) applied test-based tarantula to software verification and implemented it in a state of the art formal verification framework (CPAchecker), and (2) we conducted the first experimental evaluation between formal-based Tarantula against formal-based DStar and Ochiai and then against test-based Tarantula approach using Klee and VeriFuzz test suites generators on a set of benchmarks.

The results show that the formula-based approach on ARG can be significantly more effective in locating faults than the other considered techniques based on test-suite generators, showing that our approach is promising and capable of quickly and precisely localizing faults.

Contents

List of Figures

List of Tables

1 Introduction

There is no such thing as 100 percent bug-free software. Also, Edsger Dijkstra[1] was already aware of the risk of software bugs

> 'If debugging is the process of removing bugs, then programming must be the process of putting them in[2]'

However, the volume and severity of existing errors, as well as their impact on users, can be reduced through two processes: (1) **formal verification**, i.e., checking whether the software design satisfies some requirements (properties) (2) **debugging**, i.e., localizing the faults that are the causes of the failures and fixing the program.

In the software development process, debugging is a difficult task that required a lot of effort and time from programmers. However, finding causes for these bugs cannot be skipped, it must be analyzed to endure the software quality.

As the complexity of software systems is increasing, there is also a growing demand for tools that assist the programmer in the debugging process. Therefore, many techniques and methods have been developed over the years to automatically detect the location of faults in various ways in software. Among these methods, coverage-based or spectrum-based debugging [JH05] is considered a promising method. It is an empirical method that calculates ranking orders between the program statements or spectrum to show that a particular fragment of code is more suspicious than the others. However, the method requires many successful and failed executions to calculate the statistical metrics.

To illustrate how the Tarantula technique works, we provide a simple faulty C program and a test suite, given in Table 1.1. Program $max(int\ line_1, int\ line_2)$ takes two integers as input and outputs the max value between them. The program contains a fault in $line_{10}$. And this check is not necessary since the actual function says that if two numbers are the same, then one of them must be returned. To the right of each line of code is a set of four test cases, their input is shown at the top of each column, their coverage is shown by the black dots, and their $pass/fail$ status is shown

[1] Edsger Dijkstra: was a Dutch computer scientist, programmer, software engineer, systems scientist, science essayist, and pioneer in computing science.

[2] `https://www.goodreads.com/author/quotes/1013817.Edsger_W_Dijkstra`

at the bottom of the columns. To the right of the test case columns are one column labeled *suspiciousness* shows the suspiciousness score that the technique computes by Equation (3.2)

For example, consider $statement_1$ in $line_7$, which is executed by all four test cases that contain both passed and failed test cases. The Tarantula technique assigns a suspiciousness rating of 0.5 to this statement because one failed test case executes it out of a total of one failing test case in the test suite (with a ratio of 1), and three passed test cases run from a total of three passed test cases in the test suite (with a ratio of 1). Hence, we get $1/(1+1)$ or 0.5. Using the suspiciousness score, the covered entities of the program are sorted. The set of entities that have the highest suspiciousness value ($statement_9$ and its block 10) are the set of entities to be considered first by the programmer when looking for the fault.

Table 1.1: Example of Tarantula technique

Code Line	Input Values				Suspiciousness
	(4,3)	(3,2)	(5,6)	(3,3)	
3: int num1	•	•	•	•	0.5
4: int num2	•	•	•	•	0.5
5: int result =0;	•	•	•	•	0.5
7: if(num1>num2)	•	•	•	•	0.5
8: result = num1	•	•			0.0
10: else if (num1 == num2)				•	1.0
11: ERROR:__VERIFIER_error();				•	1.0
12: else			•		0.0
13: result = num2;			•		0.0
15: return 0;	•	•	•	•	0.5
Pass/Fail Status	P	P	P	F	

If we consider the example above Table 1.1, we can see a problem that we try to solve through model-checking in this paper. In simple words, if the user tries to enter two different numbers all the time and did not try to enter two equal numbers. Then we cannot know if there is a bug in the function $max()$. Or if we want to test the function automatically, how many test cases do we have to generate to achieve the failed test case?

The purpose of testing is to reveal as many failures as possible with as few test cases as possible. Therefore, the following two cases are used:

- The more test cases that are provided for fault localization, the more relevant information needs to be provided for fault localization in order to achieve good

results. However, to keep the cost as low as possible, the number of test cases from the test should be as low as possible. Therefore, getting a large number of test cases from a test is somewhat difficult for fault localization.

- Fault localization usually focuses on program information related to the fault. The location of the failing program is the root cause of the failure, so you need enough test cases to cover the location of the failing program.

The two cases above indicate that the purpose of fault localization and testing are different and may provide insufficient information for fault localization.
Therefore, the number of required test cases explodes and forces tests to consume more than 50% of the development costs for embedded software and 60% to 70% for safety-critical software [BH13]. Complete execution of all test cases in a test suite can be slow; execution times of up to seven weeks have been reported in the literature [Rot+01]. For larger test suites, it may not be realistic to expect the engineers to wait for all test cases to be executed before they can start working on fault fixing. Also, it may not be worth executing all test cases, once the first failure has been detected. In many scenarios, as soon as the first failure is detected, the engineer will want to switch from testing to fixing [Rot+01].

However, the error detection rate is too low to prevent common errors and incidents, especially with security-critical software. Therefore, the dilemma has to be solved differently. The use of formal methods is an approach that has sparked great interest in the industry [Woo+09].

Model-checking is alluring for two primary reasons. First, it does not require the user to provide annotations such as preconditions or loop invariants. Second, when a property violation is detected, a witness to the violation is produced in the form of an error trace (a counterexample) at the source level [BNR03].

The formula-based strategy is more precise than the cover-based approach. Typically because it has a logical establishment developed in the model-based diagnosis (MBD) hypothesis [Rei87].

Furthermore, existing formula-based strategies do not guarantee that all root causes are covered. This is partly because the complete list of the root causes requires a lot of computing effort. Its complexity increases exponentially with the size of the program and the number of errors in the program.

This paper focuses on fault localization and presents a method for determining the cause of errors in C programs. We introduced the Tarantula algorithm as part of the verification of software models instead of information on test coverage. We assume we are given a program written in C and specification. If the program is faulty, then we have a counterexample that indicates that the specification does not hold. We use the

counterexample as input to the Tarantula algorithm, and we get as output set of fault candidates for the given traces. Otherwise, Tarantula reports *"no bugs found"*.

Finally, we evaluate our implementation on one hand against other ranking metrics such as DStar and Ochiai, on the other hand against the test-based Tarantula algorithm using test suite generators Klee and VeriFuzz and show which of the two approaches (Formal verification vs testing) is more efficient in finding bugs.

Contributions. We present the following contributions:

- We design Tarantula, as a new module of fault localization based on the Abstract Reachability Graph (ARG).
- We implement Tarantula to software verification in the open-source verification framework CPAchecker, to establish a baseline for comparison with new ideas for improvement.
- We implement Tarantula to work with the tool *TestCov* for robust running test suites and measuring test coverage.
- We conduct an experimental study to compare two important CPAchecker configurations, Predicate Analysis against Symbolic to show which of them is more efficient in fault localization.
- We have made a comparison with Tarantula against the other ranking metrics, DStar and Ochiai methods to show if the technique shown in testing as better/-worse than Tarantula can still give better/worse results in formal verification environment.
- We collected and adjusted a set of small examples of faulty C-Programs as a benchmark set for evaluation setup.
- We developed an evaluation metric which helped us to evaluate all available techniques in this study.
- Finally, we conduct a large experimental study to compare test-based Tarantula against software verification using ARG (considered ARG paths instead of test cases) to highlight the strengths and weaknesses of our implementation in the domain of software verification.

2 Related Work

There are various techniques for locating the causes of faults that complement the approach presented in this paper. As a counterexample, fault localization for traces generated by model checkers has been an active research area in recent years. [GSB07; BNR03].

Program slicing [Wei] was introduced to help with debugging, and this has been empirically proven to be effective [Kus+14]

The formula-based automatic error localization method essentially combines the SAT-based formal verification techniques [PBG05] with the model-based diagnosis (MBD) theory. The MBD theory establishes a logical formalism of the fault localization problem [Rei87]. The model is presented as a formula that is expressed inappropriate logic. The formula is not satisfactory because it is an artifact that contains faults.

MBD theory distinguishes between conflict and diagnosis. Conflicts are the wrong situations that are represented by unsatisfactory minimal subsets (MUS) of the unsatisfactory formula. Conflicts represent a series of program fragments that contain errors and should therefore be compared to the sectors identified in the program split approach. Diagnostics are defects to identify and minimal correction subsets (MCS). The MBD theory states that the MUS and the MCS are linked by the relationship of the stroke set. So the problem is to list all MUS or all MCS. Such quantities can be calculated automatically if the formula is presented in decidable fragments of first-order theory.

An alternative approach to obtaining MCS has been used in the fault localization of *"Very Large Scale Integration"* or (VLSI) circuits [Saf+07]. The procedure reduces the problem of error localization to the maximum feasibility of the non-feasible formula in the statement logic and calculates maximum feasible partial quantities (MSS). An MCS is a complement to one MSS [LS08].

This idea was applied to the problem of locating errors in imperative programs and was implemented in a tool, BugAssist[JM02]. It uses an iterative location algorithm to obtain the MCS from which the CSR is calculated. CSR is smaller than with program cut approaches. However, the algorithm does not guarantee the enumeration of all MSS, which means that some errors may be overlooked. If multiple fault location passes are required, *BugAssist* combines MCS by combining all atomic elements (clauses) into a single sentence. For example, the set of MCSes $\{\{1,2,3\},\{4,5,6\}\}$ becomes

$\{1, 2, 3, 4, 5, 6\}$. Such a combination loses information contained in the calculated MCS because the MCS obtained for the same error-inducing inputs are merged. This makes it difficult for software developers to debug programs with multiple errors.

The most closely work related to the current research is that of Ball, Naik, and Rajamani [GSB07], who use multiple calls to a model checker and compare the counterexamples to a successful trace. The faults are those transitions that do not appear in a correct trace. Our approach is similar: Where is required to comparing safe paths to failed paths and outputs after analyzing the possible coverage by a specific function, a potentially ranked list of suspicious program elements.

An alternative approach to reducing the cognitive load of debugging is the use of delta debugging [Zel02]. Multiple runs are used to minimize the *"relevant"* part of the input, and data and control dependency information is used to remove statements that are irrelevant to the cause of the bug.

3 Background

3.1 Automated Software Testing

Software developers spend a lot of time checking the accuracy of the software. Software testing is the most widely used technique to accomplish this task. Most software test cases are created manually and may not always cover all paths of software execution. If important test cases are not executed, there is still the possibility of software errors. With tools such as Klee and VeriFuzz[1], we introduced a test-based tarantula so that software tests for checking and validation can be automated.

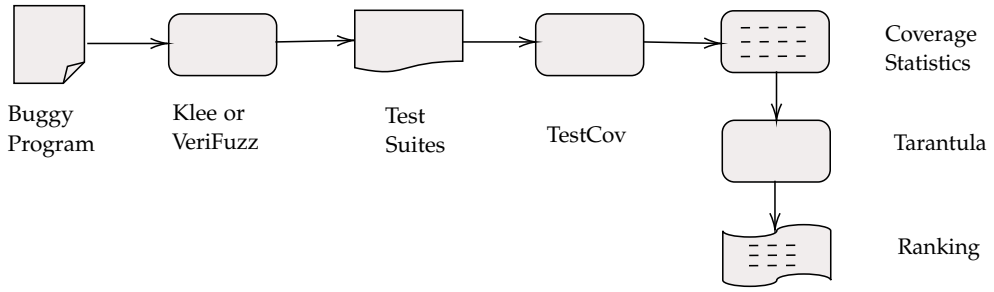


Figure 3.1: Workflow of how the test-based Tarantula works in combination with Klee,VeriFuzz and TestCov

Figure 3.1 shows the workflow of the walking tarantula on TestCov. As we can see, we first put a buggy program into test-generator, Klee or VeriFuzz which creates the corresponding test suites. Afterwards we run the output by TestCov[BL19], a tool that runs the test suite and measures the possible coverage. TestCov creates output files such as *.info* files which includes the coverage information for each test case in test suites and *result.json* which includes the status of the ran tests, where *Returncode* gives the status of the coverage. If a test has $returncode = 1$, it has hit the error. We use all this information in our implemented Tarantula Python script in order to automate the generation of ranking information about possible fault code lines.

[1] `https://gitlab.com/sosy-lab/test-comp/archives-2020/-/tree/master/2020`

3.1.1 Klee

KLEE [CDE08] is a symbolic execution tool for automatic test generation. It is open source. It performs a combination of concrete and symbolic execution for C applications to generate test cases with high line coverage. This tool works with the publicly available LLVM compiler for GNU C. The source code of an application is first compiled in bitcode with the LLVM compiler, and then performs symbolic execution of the LLVM bitcode. KLEE models memory with bit-level precision and uses heuristics to reduce the number of execution paths to be examined, resulting in increased line coverage.

There are two important goals of KLEE:

1. Hits every line of the program's executable code.
2. All dangerous operations (dereferences, requests, etc.) to determine if there are any input values could cause an error.

When the symbolic execution finishes, KLEE uses the STP constraint solver to resolve the current path conditions and create test cases that follow the same execution path when running the unmodified version of the source code. This tool has proven to be a successful approach to automated software testing. KLEE detected 56 fatal errors when applied to multiple UNIX utility applications. Some of them have been unrecognized for more than 15 years.

3.1.2 VeriFuzz

VeriFuzz [BMV19] is a coverage-driven automated test entry generation tool. VeriFuzz based on Gray Box Fuzzing analysis to provide meaningful information about program behavior extract that can help generate test inputs based on fuzzing to quickly reach coverage targets. VeriFuzz is based on evolutionary algorithms to generate newer test inputs from an initial population of test inputs. A key evolutionary algorithm is the selection of the most suitable candidates from a population and the generation of a source by applying crossover and mutation operations to them. The newer sources are checked for their suitability against a target. The population develops by adding the source of the source to the existing population.

VeriFuzz's core strength is the ability to find test input that can cause program execution to quickly reach the points of failure.

VeriFuzz's weakness is that the text input that leads to execution reaching a fault location may not always be recognized because VeriFuzz examines the specific program paths randomly due to its evolutionary approach.

3.1.3 TestCov

TestCov [BL19] is a tool for running robust test suites and measuring test coverage in C programs. TestCov is based on the existing benchmarking tool BenchExec, it uses an overlay file system and a Linux control group to protect the file system from changes and prevent unexpected resource usage during test runs.

TestCov takes as input the C-Program under test, the coverage criterion to verify and the test set, and creates an executable program that can be used to feed the tests to the program under test, coverage statistics about the test set and a set of tests that achieve the same coverage as the original set of tests. TestCov reads and writes test suites in XML-based exchange format for test suites, which consists of two parts: A metadata file that describes the test suite and a set of test case files, each defining a single test case.

3.2 Automated Software Model Checking

A complementary approach to testing is software model checking (SMC). This has many advantages over testing: it can firstly (symbolically) explore all of the program executions, and consequently, it can prove the absence of bugs and find them. The disadvantage of model checking is that it is generally more complex to define a specification for model testing than for a test case (for similar problems). This is because specification languages must provide the ability to define expected results for different program execution paths. In contrast, a test case may only consider one program execution path [Cla08]

Model checking is a very effective technique for verifying temporal specifications and the correct functioning of hardware and software systems.

The essential idea behind model checking is shown in Figure 3.3.

A *model-checking* tool accepts model/program and system property (specification) that the final system is expected to satisfy. The tool then outputs *"yes"* if the given model satisfies the given specifications, or otherwise generates a counterexample. The idea is that by ensuring that the model satisfies enough system properties, confidence in the correctness of the model is increased. Model-checking is therefore considered to be a very powerful approach to finding a large number of bugs [JM07].

Examples of properties are simple assertions, (e.g., "the variable x is positive whenever control reaches l"), global invariants, (e.g., "each array access is within bounds"), or termination properties, (e.g., "the program terminates on all inputs"). In general, properties are classified as safety and liveness:

1. **Safety:** stipulates that nothing bad will ever happen.

2. **Liveness:** stipulates that something good will eventually happen.

One way to check for safety is by using a reachability analysis, which consists of deciding whether a given system configuration (a.k.a. target) can ever be reached from a given initial configuration.

All of this means that model checkers are a powerful technique for applying many fault-localization algorithms, such as Tarantula, set union, and set intersection or even nearest neighbor [JH05].

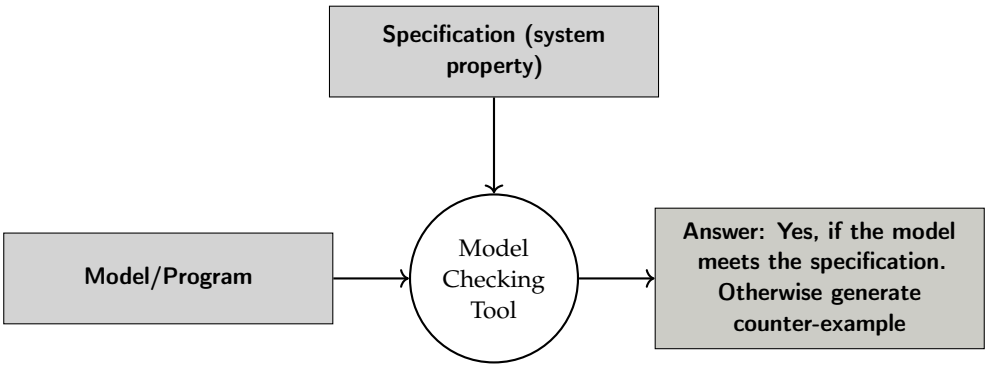


Figure 3.3: The model checking approach

3.2.1 Simple Programs and Control Flow Automata (CFA)

To understand how the Tarantula algorithm can be applied to the abstract reachability graph, we need to examine how source code can be represented as a graph.

The internal representation of the model (the program to be verified) in CPAchecker is a control-flow automaton (CFA) [Bey+09].

We limit the presentation to a simple imperative programming language, where all operations are either assignments or assume operations, and all variables range over integers.

A CFA $A = (L, l_0, G)$ consists of a set L of control (program), an initial location $l_0 \in L$ models the program entry point, and a set G of edges between the nodes. Each edge $g \in G$ is a tuple (l, Ops, l'), where $l, l' \in L$ are control locations (nodes). Each edge is identified with operation Ops that is executed when control flows from one program location to another. The sets of nodes and edges naturally define a directed labeled graph, called *the control-flow (CF)* of the program.

If a node has no successors, it is called *"final"* and represents *a program exit*. And if a node is an error location this is called *target program location* $l_e \in L$. A path σ [BLW15]

is a sequence $\langle (op_1, l_1), ..., (op_n, l_n) \rangle$ of pairs of an operation and a location. The path σ is called program path if for every i with $1 \leq i \leq n$ there exists a CFA edge g = (l_{i-1}, op_i, l_i) and l_0 is the initial program location, i.e., the path σ represents a syntactic walk through the CFA.

Example 1 (Simple Program and Control-Flow Automaton) Figure 3.4 shows an example C program and the corresponding CFA. The CFA has ten program locations (Nodes) $L = \{3,4,5,7,8,10,11,13,15,16\}$, such that l_0 is the initial location of this program and contains three variables $(X = \{num_1, num_2, result\})$, such that num_1, num_2 are both initialized to $non - deterministic$ values and $result$ initialized to 0. At $line_{10}$ num_1 and num_2 are checked for equality. If the variables are equal, control flows to the error location 11, otherwise flows to the exit location 16.

```
1  int main()
2  {
3    int num1 =_VERIFIER_nondet_int();
4    int num2 =_VERIFIER_nondet_int();
5    int result = 0;
6
7    if(num1 > num2){
8      result = num1;
9    }
10   else if(num1 == num2){
11     ERROR:__VERIFIER_error();
12   }else{
13     result = num2;
14   }
15   return 0;
16 }
```

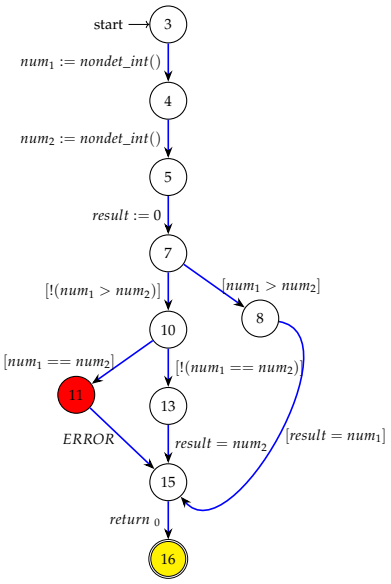


Figure 3.4: A simple C program (Left) and a CFA representing it (Right)

3.2.2 Concrete State

A concrete data state of a program [BLW15] is a variable assignment $cd : X \rightarrow Z$, which assigns to each program variable an integer value. The set of integer values is denoted as Z. A concrete state of a program is a pair (l, cd), where $l \in L$ is a program location, and cd is a concrete data state. The set of all concrete states of a program is denoted by C, a subset $r \subseteq C$ is called a region. A region of concrete states that violate a given specification is called target region σ.

3.2.3 Abstract State and Abstract Domain

Abstract state: Abstraction uses abstract states domain instead of concrete states domain and checks the properties the domain of the abstract states. It is probably the most significant technique for reducing the complexity of model checking. This is usually done by a method called data abstraction. The key point is to construct a function mapping concrete state to abstract states.

Abstract domain: The main problem with the program semantics is that there are potentially infinitely many data states and the program paths cannot be analyzed individually. Analyzing the set of program paths together (abstract domain) is a solution for that:

1. Group concrete states $\Rightarrow$ abstract states.
2. Define (abstract) semantics for abstract states.

The abstract domain [BHT07] $D = (PC, \epsilon, [\![.]\!])$ is defined by a set PC of concrete states, a semi-lattice $\epsilon = (E, \top, \sqsubseteq, \sqcup)$ that describes the abstract states and their possible relationship to each other, and a concretization function $[\![.]\!] : E \to \overset{C}{2}$ assigns to each abstract state its meaning, i.e., the set of concrete states that it represents.

A *semi-lattice* $\epsilon = (E, \top, \sqsubseteq, \sqcup)$ consists of a set E of elements, a top element $\top \in E$, a partial order $\sqsubseteq \subseteq (E \times E)$ and the total function $\sqcup : (E \times E) \to E$ called join operator. The elements $e \in E$ of an abstract domain are called abstract states.

3.2.4 Symbolic Execution

The tool which is implemented in CPAchecker for symbolic execution called *CPA-SymExec* [BL18] and is based on abstraction and counterexample-guided abstraction refinement (CEGAR) [Cla+03], and uses a constraint-interpolation approach to detect symbolic facts. The symbolic execution is a static program analysis [BL18] that uses symbol values as input values instead of actual data and to display the values of the program variables as symbolic expressions. As a result, the output value calculated by the program is expressed based on the value of the input symbol.

The state of a symbolically executed program includes the value of the program variable (symbol), the path condition (PC) and the program counter. Path conditions are Boolean formulas (without quantifiers) for symbolic input. Accumulates the restrictions the input must meet for the execution to follow a particular related path.

3.2.5 Predicate Abstraction and Predicate CPA

Predicate abstraction [GS97; BWK12] is an important method for checking software. It abstracts data by tracking only certain predicates on the data. Each predicate is represented in the abstract program by a Boolean variable while the original data variables are removed. The verification of a software system with predicate abstraction consists of the construction and evaluation of a system with finite states, which is an abstraction of the original system with respect to a series of predicates. When model checking of the abstract program fails it may produce a counterexample that does not correspond to a concrete counterexample. This is usually called a *spurious counterexample*. When a spurious counterexample is encountered, refinement is performed by adjusting the set of predicates in a way that eliminates this counterexample. The abstraction refinement process has been automated by the Counterexample Guided Abstraction Refinement paradigm or CEGAR for short. In CPAchecker the predicate uses the predicate abstraction called predicate CPA [BGS; BHT08] to compute abstract states from a formula ϕ and a set π of predicates. The cartesian predicate abstraction $(\phi)^C_\pi$ is used in predicate CPA and is the strongest conjunction of predicates from π that is implied by ϕ.

The predicate CPA $P = (D_{\mathbb{P}}, \Pi_{\mathbb{P}}, \rightsquigarrow_{\mathbb{P}}, merge^{sep}, stop^{sep}, prec)$ consists of:

1. The Abstract domain $D_{\mathbb{P}} = (C, \rho, [\![\blacksquare]\!])$
2. The set of precisions $\Pi_{\mathbb{P}} = 2^p$ models a precision of an abstract state as a set of predicates
3. The transfer relation $r \overset{g}{\rightsquigarrow}_{\mathbb{P}} (r', \pi)$
4. Merge operator: there are two options at predicateCPA:
   - $merge^{sep}(r, r', \pi) = r'$ for all $r, r' \in \rho$. This is that we used in our evaluation, because it considers each path of CFA separately Figure 6.3b.
   - The other merge operator is the default merge which combines two paths, or makes two possible paths into a single path Figure 6.3a
5. Termination check $stop^{sep}$ defind as $stop^{sep}(r, R, \pi) = \exists r' \in R : r \sqsubseteq r'$
6. The presicion $prec$ defined by: $prec(r, \pi, R) = (r, \pi)$

3.2.6 Abstract Reachability Graph (ARG)

In this section, we explain what ARG is and what this graph consists of.

If the number of states is large, it can be very difficult to determine whether such a program is correct or if it's possible to extract potential error paths. For programs with infinite states, the symbolic reachability analysis cannot be ended or it can take an excessive amount of time or memory to complete. Abstract model checking trades off precision of the analysis for efficiency. When checking abstract models, a reachability analysis is carried out for an abstract domain, which uses some abstract semantics of the program to collect some, but not necessarily all, information about an execution. Proper choice of the abstract domain and semantics ensures that the analysis is sound (i.e. proving the safety property in the abstract semantics implies the safety property in the original, concrete, semantics) and efficient.[CGL83] This is often done by construction of an *abstract reachability graph* (ARG); predicate abstraction is one of the preferred abstract domains. The ARG [BDW17] represents unrolling of the control-flow of the program.

The definition of ARG looks like the following:

Let $P = (L, l_0, G)$ be a CFA and $A = ((C, (E, \sqsubseteq, \sqcup, \top), [\![\cdot]\!]), \rightsquigarrow)$

An ARG for P and A is ARG = $(N, root, \varrho)$, such that

For some abstract domain with abstract states $N \subseteq E$, $root \in N$ is an initial abstract state, $\varrho \subseteq (N \times G \times N)$ and fulfills two very important properties:

1. **Rootedness:** Root considers all initial states i.e., $\{c \in C | c(pc) = l_0\} \subseteq [root]$
2. **Completeness:** No concrete successor is forgotten: $\forall n \in N, g \in G :$
   $(\exists c, c' \in C : c \in [\![n]\!] \wedge c \xrightarrow{g} c') \Rightarrow (\exists (n, g, n') \in G : c' \in [\![n']\!])$

The nodes of an ARG are abstract states that contain more domain-specific data such as control flow location, call stack information, and a path formula that represent the state of the data. The edges of the ARG represent the program operations that correspond to the edge that was followed by the CFA.

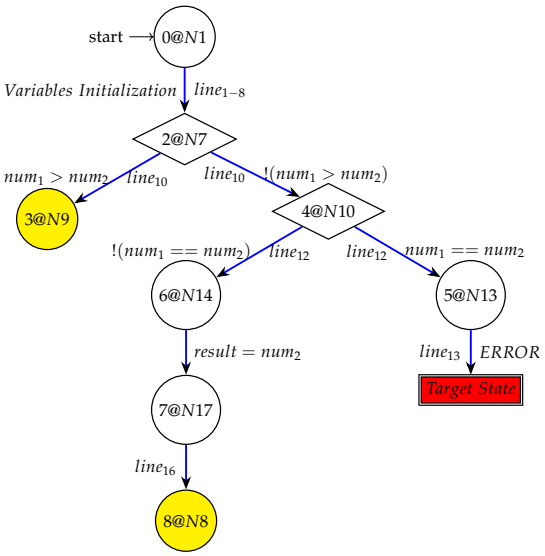


Figure 3.6: ARG construction by execution of predicate analysis by CPAchecker for the example program shown in Example Figure 3.4. The red rectangle is target state and yellow states (3@N9 and 8@N8) are safe states.

Figure 3.6 is an example ARG for the domain of predicate analysis. In this paper we differ from two different nodes, target state/error state (red-colored node): is the node which present the mouth of the error path (by CPAchecker generated counterexample) and safe states (yellow states): are the nodes which presents the leaves of the safe paths.

3.2.7 Counterexample

A major strength and one of the most important features of model checking is the ability to generate counterexamples in case a property is violated. The counterexample [Gen+18] details why the model doesn't satisfy the specification. By studying the counterexample, we can pinpoint the source of the error in the model and providing a user with information on how to debug their system and/or specification, correct the model, and try again.

The Figure 3.8 is an example of a generated counterexample in ARG of Figure 3.6 by CPAchecker. The abstract state "Error" that belongs to the error location is found and the concrete program path that leads to this state is reconstructed from the ARG and checked for feasibility. If the error path is feasible, the program is unsafe and the analysis terminates. Otherwise, the error path is infeasible, and the precision of the analysis will be refined to be precise enough to eliminate this infeasible error path from the ARG. Then the analysis is restarted, and the steps are repeated until

either a concrete error path is found, the generated counterexample report will include information about the program path leading to the error - the error path or the abstract model (and thus the program) is proven safe. In this paper, we considered this error path $\sigma = \langle(\xrightarrow{op_0} ... \xrightarrow{op_{n-1}})\rangle$ as *failed path* and its leaf as *target state* $op_{n-1} \in \sigma$.

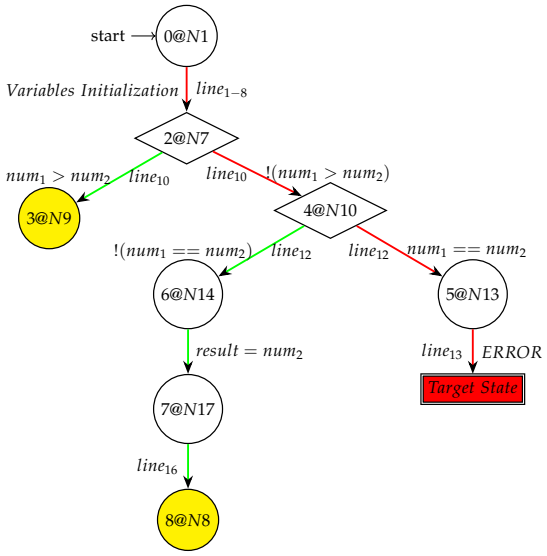


Figure 3.8: ARG construction with a generated colored with red counter example by execution of predicate analysis by CPAchecker for the example program shown in Example Figure 3.4. The red rectangle is target state and red path is the error path and yellow states (3@N9 and 8@N8) are safe states. and green paths are safe paths (Starts from root and ends at safe states).

3.2.8 CPAchecker

CPAchecker [BK11] is a *configurable software verification* [BHT07; CW09] tool that expresses different approaches to program analysis and model checking in a single formalism. The main algorithm is configurable to perform a reachability analysis for any combination of the existing configurable program analysis (CPA). One application of CPAchecker is the checking of Linux device drivers.

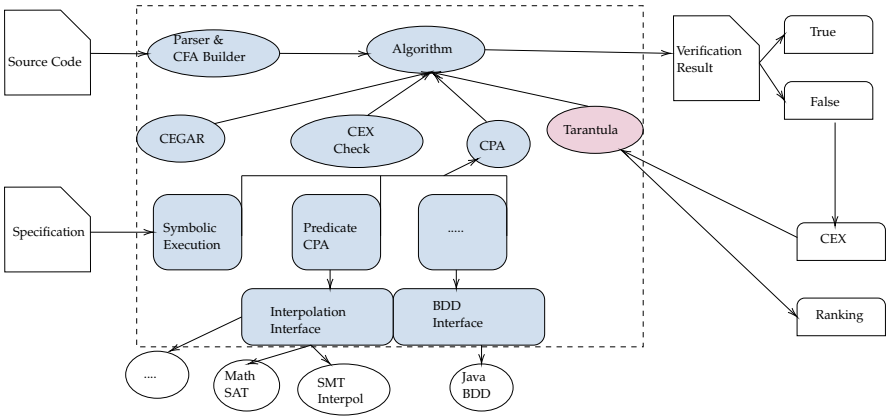


Figure 3.10: *CPAchecker* Architecture, after Beyer and Keremoglu [BK11]

The Figure 3.10 above is the overview of CPAchecker's architecture. The central data structure consists of a series of control-flow automata (CFA). Before a program analysis starts, the input program is transformed into a syntax tree, and further into CFAs. The current version of CPAchecker uses the parser from the CDT[2], a fully functional C and C++ IDE plug-in for the Eclipse platform.

The framework provides interfaces to SMT solvers and interpolation procedures, such that the CPA operators can be written concisely and conveniently. From the picture, we know that they use MathSAT as an SMT solver and CSIs at and MathSAT as interpolation procedures. They also use JavaBDD as a BDD package and provide an interface to an Octagon Library as well. Important to mention that CPAchecker generates HyperText Markup Language (HTML) reports that represent the verification outcome. A *Report.html* file is generated if there is no error found by the verification run and a *Counterexample.html* file is generated for each counterexample found by the system. If a counterexample is found by CPAchecker, the report will also include a table-like representation of the error path, the path that leads to the error as shown in the Figure 3.8. Next to the CPA algorithm, we can see the Tarantula algorithm.

[2]Available at `http://www.eclipse.org/cdt`

3.3 Fault Localization

Fault localization techniques aim to reduce the cost of debugging by automating the process of searching for the location of the fault in the program. In this section we first discuss some aspects of fault localization then we explain the Tarantula algorithm based on an example. Finally, the problem with this approach.

3.3.1 Preliminaries

3.3.1.1 Failures, Errors, and Faults

As defined in [Avi+04], we use the following terminology. A failure is an event that occurs when delivered service deviates from the correct service. An error is a system state that may cause a failure. A fault is the cause of an error in the system. To illustrate these concepts, consider the C function in Table 1.1. It is meant the maximum between two integer numbers, then returns the largest number between them. There is a fault (bug) in the equality statement: If the two integer numbers given by the user are equal, e.g. 3 and 3, which leads the execution to $\_\_VERIFIER\_error()$. In this case, the check is true and a failure occurs, and an error occurs after the code inside the conditional statement is executed, $num_1 == num_2$. Therefore, to detect and remove existing and potential errors (also called 'bugs') in a software code that can cause it to behave unexpectedly or crash, programmers mostly use manual debugging process.

3.3.1.2 Software Debugging

All definitions in this section are provided by the Glossary of computer science [90]. The test phase is defined as the process of exercising or evaluating a system or system component manually or automatically to check whether it is fulfills specified requirements or to identify differences between expected and actual results. The test phase includes different types of tests. *Unit testing*, *integration tests* and *Acceptance tests* are better known.

Unit testing: is a test of a single program module that can contain multiple functions or procedures in one isolated environment (i.e., isolated from all other modules) to see if the device meets the specified device design specification.

Integration tests: Test the interfaces between modules to determine if the system behaves like the specified design Specification.

Acceptance test: Is the validation of the system or program to the Requirements specification.

If failures are found during testing, the faults that cause the failures must be corrected. This activity is called debugging. Debugging can be seen as three successive activities:

(1) failure localization, during this process the failures which have been detected in the testing are reproduced. (2) fault localization: The location of the fault in the target program is identified. (3) fault correction: The fault is corrected.

3.3.1.3 Code Coverage

To detect and locate faults anywhere in the program, any diagnosis technique requires each statement to be covered by the tests.

Code coverage is a measure that describes the extent to which the source code of a program is tested. It is a form of white box testing that finds areas of the program that are not practiced by a group of test cases. It also creates some test cases to increase coverage and define a quantitative measure of code coverage.

In this work we will use the branch coverage.

Branch coverage. Branch coverage is a code coverage measure, which aims to ensure that each branch from each decision point (e.g., if statements, loops) is executed at least once during testing. (It is sometimes also described as saying that each branch condition must have been true at least once and false at least once during testing.) It helps in validating all the branches in the code making sure that no branch leads to abnormal behavior of the application.

The formula to calculate Branch Coverage:

$$\texttt{Branch Coverage} = \frac{\texttt{Number of Executed Branches}}{\texttt{Total Number of Branches}} \tag{3.1}$$

3.3.1.4 Program Entity

A program entity [SCK17] is a part of a program. It comprises any granularity of a program, from a statement to a subprogram. Program entities include *statements, blocks, branches, predicates, definition-use associations, components, functions, program elements, and program units*.

3.3.2 Ranking Metrics

Ranking metrics [SCK17] are utilized in fault localization to calculate the chance that program entities could be faulty. The studies on fault localization use special phrases to consult ranking metrics: *technique, threat assessment formulation, metric, heuristic, ranking heuristic, coefficient, and similarity coefficient*.

Several ranking metrics were suggested for error localization, which was created or adapted from other areas. Everyone has their specific characteristics.

In software development, a test suite is a collection of test cases that are intended to be used to test a software program to show that it has some specified set of behaviors.

Tarantula. A fault localization technique with Tarantula [JH05] takes as input a faulty program and a set of test cases with at least one failed test, and it generates as output a potentially ranked list of suspicious program elements.

The Idea behind Tarantula is, that when executing a test suite, execution events that correlate to errors are more likely to be the cause of the error. In other words, events that mostly occur in failed test cases but rarely in passed test cases are more likely to be the fault. This conclusion examines the event similarities between the failed test cases and distinguishes these similarities from the events that occur in the passed test cases.

The suspiciousness of a coverage entity e is defined as the following equation:

$$\texttt{suspiciousnessTarantula} = \frac{\left(\frac{\texttt{Failed(s)}}{\texttt{TotalFailed}}\right)}{\left(\frac{\texttt{Failed(s)}}{\texttt{TotalFailed}}\right) + \left(\frac{\texttt{Passed(s)}}{\texttt{TotalPassed}}\right)} \tag{3.2}$$

Where `Passed(s)` is the number of passing executions that execute statement(s), `Failed(s)` is the number of failing executions that execute statement(s), `TotalPassed` is the total number of passing test cases, and `TotalFailed` is the total number of failing test cases.

Ochiai. Ochiai differs from Tarantula in that the lack of a statement is taken into account in the case of failed runs. Ochiai metric [Ali12] defines the suspiciousness of a statement (s), as follows:

$$\texttt{suspiciousnessOchiai} = \frac{\texttt{Failed(s)})}{\sqrt{\texttt{TotalFailed} * (\texttt{Failed(s)} + \texttt{Passed(s)}))}} \tag{3.3}$$

Similar to the Tarantula algorithm, the Failed(s) is the number of failing test cases which execute s, Passed(s) is the number of successful test cases that execute, and TotalFailed is the number of failing test cases in the test suit. Unlike Tarantula, Ochiai does not require that a passed case should exist since the denominator of the equation will not be divided by 0. Therefore, this technique is considered an improvement for Tarantula.

DStar. DStar has the same variables in its suspicion equation that have the same explanation as the other mentioned techniques.

The function is defined as follows:

$$\texttt{suspiciousnessDStar} = \frac{\texttt{Failed(s)}^{\delta}}{\texttt{Passed(s)} * \left(\texttt{TotalFailed} - \texttt{Failed(s)}\right)} \tag{3.4}$$

For the variable $\delta > 0$. We used $\delta = 2$. This is the most thoroughly researched value that has been experimentally proven and is the best case where DStar can work efficiently [Won+14].

4 Theoretical Contributions

In this chapter, we explain in detail and theoretically which classes and methods we used in the framework and in TestCov as contributions.

4.1 Contribution in CPAchecker

CPAchecker is a framework and tool for formal software verification, and program analysis, of C programs. When executed, CPAchecker performs a reachability analysis, i.e., it checks whether a certain state, which violates a given specification, can potentially be reached.

However, the most difficult step was to apply an algorithm that was developed for testing on ARG. We, therefore, had to define what should be a pass case and what should be a fail case.

We will now explain the most important classes of the implementation:
Pass/Fail Case. Unlike the test-based algorithm, we used in CPAchecker the ARG to distinguish the fail from pass cases. Ranking Metrics use the information on ARG such as the pass/fail information for each path, the CFAEdge that were executed by each path (e.g. statements, branches, methods) and the source code for the program.

Each safe/error path consists of a series of control flow automaton edges (*CFAedge*) that describe the traces of the error/safe paths from the root to the target/safe state. Each CFAedge is an error/safe edge and contains the code line number from the source code and between which nodes the edge is located.

Both pass and fail-case classes have a very important verification method, which is used to check whether a safe/fail path is generated during the CPAchecker analysis since it is known that some ranking techniques such as Tarantula or DStart ensure at least one safe one Path and an error path to need to work. Otherwise, *ArithmeticException* will be thrown since divided by zero will be possible.

- **Fail Case:** In our implementation, the fail case contains all error paths that start from the root and end in the target state. For example from the ARG in the Section 3.2.7 we can see the error path as the following set of CFAedges:

```
1
2 [[line 10: N16 -{[!(num1 > num2)]}-> N19, line 12:  N19 -{[num1 == num2
    ]}-> N22, line 13:N4 -{Label: ERROR}-> N5]]
```

Figure 4.1: All edges in error path. Contains code line numbers and node identities and the descriptions which presents the states.

- **Pass Case:** The pass case in other hand contains all safe paths, starts from root and end to *"safe states"* (not target state and is leaf).

Fault Localization Ranking. After getting all information about both cases and the coverage. The information about the coverage will be then stored in a key-value data structure and finally the suspicious will be calculated by the following equation:

Let us take the function of Tarantula algorithm Table 1.1. We have to redefine the function by replacing fail test case by *errorPath* and safe test case to $safePath$ and so for all other ranking metrics as following:

$$\texttt{suspiciousness} = \frac{\left(\frac{\texttt{errorPath(s)}}{\texttt{totalErrorPaths}}\right)}{\left(\frac{\texttt{errorPath(s)}}{\texttt{totalErrorPaths}}\right) + \left(\frac{\texttt{safePath(s)}}{\texttt{totalSafePaths}}\right)} \tag{4.1}$$

Coverage Information. Coverage information is a metric that determines how often a CFAEdge was successfully visited by a safe/error path.

This can help us to understand how much of the source code is tested. It's a very useful metric that can help us to assess the quality of paths of ARG.

Table 4.1: Example of how the CFAEdges are covered by Safe(S)/Error(E) paths. The number after S or E means how many time is the corresponded CFAEdge covered by which kind of path

CFAEdge	Coverage
(num_1>num_2)	((S,1),(E,0))
[!(num_1>num_2)]	((S,1),(E,1))
[!(num_1==num_2)]	((S,1),(E,0))
((num_1==num_2)	((S,0),(E,1))
(result=num_2)	((S,1),(E,0))
ERROR	((S,0),(E,1))

The class has to be initialized with failedcase and shutdownNotifier.

- **FailedCase:** is class attribute is necessary because we need to use the method *existsErrorPath* More details on pseudo-code can be found in the upcoming Section 5.1.2.
- **ShutDownNotifier:** is designed for operations where you think that they could take longer, it should hand over the shutdown notifier and check regularly whether it has been activated - e.g. at the beginning of the loop of an algorithm.

Faults Ranking. Since TestCov does not provide *CFAEdges* as output, so that we can compare test suite generators with the ranking algorithms in the evaluation, but we only get coverage statistics in the art of code line numbers. We have sorted and summarized the CFAEdges in CPAchecker according to their code line number so that we can carry out a valid evaluation between the techniques.

Each line number has its corresponding maximum suspicion rating between all suspected CFAEdges, the corresponding line number, and a list of hints that represent all fault contributions. This is explained in detail later in this chapter.

For the output, we have used the fault data structure [Ket20] where the algorithms are embedded in a statistics structure that has been created explicitly for each fault localization algorithm. The structure is designed to be easily usable and extendable in any manner and to be easily tailored to all sorts of fault localization algorithms. Each time a counterexample is found, CPAchecker creates a visible record of the use of HTML, JavaScript and, CSS. The record already incorporates a visual illustration of the *CFA*, enables get entry to the supply code, and lists all applicable edges from the counter pattern in the correct order from the beginning of this system to the error.

We will now explain the most important classes of the data structure in which we were able to summarize CFAEdges under their code line numbers:

FaultLocalizationFault. This class allows us to call the most important methods. This has to be initialized with the default constructor. The methods in this class are:

- **sumUpFaults:** This function summarized the CFAEdges into a list of FaultInformation which is already explained at the beginning of this section.
- **faultsDetermination** This function converts faultinformation into a list of errors for use as input in the fault data structure. Besides, the function sorts the errors in reverse order so that the code line number with the highest suspicion is sorted first.

4.1.1 Example of Ranking Metric on ARG with Tarantula

To illustrate how the Tarantula technique with CPAchecker works, we applied our technique to the same faulty C program in Section 3.3.2. And as we can see in the Table 4.2 to the right of each Code Line is a set of three paths: two are safe and one is an error path which represents the counterexample, their coverage is shown by the black dots. To the right of the paths, column is one column labeled *suspiciousness* shows the suspicious score that the technique computes by the redefined equation Equation (4.1) for each statement.

Table 4.2: Example of Tarantula technique in CPAchecker using ARG

Code Line	Possible ARG Paths			Suspiciousness
	path1	path2	path3	
3: int num1	•	•	•	0.5
4: int num2	•	•	•	0.5
5: int result	•	•	•	0.5
7: if (num1 >num2)	•			0.0
8: result = num1;	•			0.0
10: else if (num1 == num2)			•	1.0
11: ERROR:__VERIFIER_error			•	1.0
12: else		•		0.0
13: result = num2		•		0.0
Safe/Error Status	S	S	E	

If we take a closer look at both tables, we can see the difference that ARG to CFAEdges can give us more precise information about where the fault is located. In contrast to the test-based technique, which covers the entire line of code, so we do not know exactly what could be the cause of the error. Besides, the suspiciousness is better distributed, and we have fewer cases than the test-based tarantula.

5 Implementation of Tarantula Algorithm

In this section, we explain the most important algorithms which helped us to apply Tarantula algorithm on ARG.

5.1 Implementation in CPAchecker

5.1.1 All Possible Paths

Since there was no specific function in CPAchecker where you can get all paths to form root to a specific state (whether error or safe state), we had to implement an algorithm to returns all needed paths. The algorithm takes the following inputs:

1. **reachedSet:** is a ReachedSet data type which is an interface representing a set of reached states, including storing a precision for each one. In all its operations it preserves the order in which the state was added. All the collections returned from methods of this class ensure this ordering, too.

2. **chosenState:** is an ARGstate data type which is a class representing the state of the Abstract reachability graph (ARG).

As output, we expected a set of ARGPath from ARGPath data type and which is a class that contains a non-empty path through the ARG consisting of both a sequence of states and the edges between them.

We need to remember all the paths that we had to traverse in all the different ways on the graph, therefore we need a path list. For every path, we will make it longer by one if it has one parent, and if it has two or more we will duplicate this list and add

the parent to each one like in Algorithm 1.

Algorithm 1: findAllPaths algorithm for finding all error/safe paths

```
1  function findAllPaths (reachedSet, chosenState);
   Input  : reachedSet is set of all ARGstates and chosenState is the selected state
            whether safe state or error state and has datatype ARGstate
   Output: A set of all paths as ARGPath
2  root ⟵ firstState of reachedSet;
3  states ⟵ empty list;
4  results ⟵ empty set of state lists;
5  paths ⟵ list of lists of states;
6  add chosenState to states;
7  add states to paths;
   /* This is assuming from each node there is a way to go to the start
      */
   /* Go on until all the paths got the start                          */
8  while the paths is not empty do
       /* Expand the last path                                         */
9      curPath ⟵ remove the last element of paths;
10     if no more to expand then
11         results ⟵ curPath ;
12         continue
13     end
14     foreach parentElement ∈ parntes of curPath do
15         tmp ⟵ copy curPath;
16         tmp ⟵ parentElement;
17         paths ⟵ tmp;
18     end
19 end
```

Time Complexity: We should think about the time complexity with terms of E and V since the algorithm is *(Breadth-first search) BFS* which is a graph algorithm. Usually, the time complexity of *BFS* is O(E + V) but now we need to save all the different paths O(E $\star$ V) in the complexity.

5.1.2 Coverage Information

We implemented an Algorithm called *"calculateCoverageInformation"* which counts how many times a safe/error path has visited each CFAEdge as already explained theoretically in Section 4.1. The algorithm takes only one input and its ARGpath and outputs a Map of CFAEdge as key and *"TarantulaCasesStatus"* as its corresponding value.

The "FaultLocalizationCasesStatus" is a data structure that includes two integer values: *failedCases* represent the number of CFAEdge that is visited by an error path and *passedCases* represent the number of CFAEdge that is visited by a safe path.

Algorithm 2: calculate the coverage information of each CFAEdge in the paths

```
1  function calculateCoverageInformation (paths);
   Input  : paths is set of all ARGpaths both safe paths and error paths
   Output : coverageInfo: A Map data structure of CFAEdge and
            TarantulaCasesStatus
2  coverageInfo ⟵ empty Map;
3  foreach path in paths do
4      foreach cfaEdge in the full path of paths do
5          caseStatus ← empty TarantulaCaseStatus;
6          if cfaEdge not in coverageInfo then
7              coverageInfo ⟵ key : cfaEdge and Value : new empty
                TarantulaCaseStatus;
8          end
9          caseStatus ← element at index of cfaEdge from coverageInfo
10         if path is error path then
11             Increase the failedCase of caseStatus
12         else
13             Increase the passedCase of caseStatus
14         end
           /* skip 0 line number                                    */
15         if line number of cfaEdge is not 0 then
16             coverageInfo ⟵ key : cfaEdge and Value : caseStatus;
17         else
18         end
19     end
20 end
```

Time Complexity: The outer loop executes N times. Every time the outer loop executes, the inner loop executes M times. As a result, the statements in the inner loop execute a total of N * M times. Thus, the total complexity for the two loops is $O(N^2)$.

5.1.3 Options

We added one option that can be set to select a ranking metric algorithm:

faultlocalization.type The option accepts three inputs: *"TARANTULA"*, *"OCHIAI" and "DSTAR"*. Via this option the algorithm for the further analysis is set. Each one executes its corresponding ranking strategy. If this option is not set, we run TARANTULA by default.

Example Configurations We give example configurations for each of the implemented algorithms. The option *alwaysStoreCounterexample* is required to perform error localization. The *cpa.predicate.merge* is required to separate the paths. With The option *-[configuration analysis e.g. predicateAnalysis]* we can choose configuration analysis e.g. Predicate or symbolic execution. The following commands can be used to locate errors using the implemented ranking algorithms:

TARANTULA:
-[configuration analysis e.g. predicateAnalysis]
-setprop analysis.algorithm.FaultLocalization=true
-setprop analysis.alwaysStoreCounterexamples=true
-setprop faultlocalization.type=TARANTULA
-setprop cpa.predicate.merge=SEP
<path to program>
DSTAR:
-[configuration analysis e.g. predicateAnalysis]
-setprop analysis.algorithm.FaultLocalization=true
-setprop analysis.alwaysStoreCounterexamples=true
-setprop faultlocalization.type=DSTAR
-setprop cpa.predicate.merge=SEP
<path to program>

OCHIAI:
-[configuration analysis e.g. predicateAnalysis]
-setprop analysis.algorithm.FaultLocalization=true
-setprop analysis.alwaysStoreCounterexamples=true
-setprop faultlocalization.type=OCHIAI
-setprop cpa.predicate.merge=SEP
<path to program>

5.2 Implementation in TestCov

In this section, we explain the idea of our implementation of finding faults in source code from coverage information of TestCov.

Information Extraction. Since most fault localization algorithm with ranking metrics has four important inputs e.g. *TotalFailed, TotalPassed, Passed(s) and Failed(s)*. We have to extract the important information from the output of TestCov. As mentioned in Section 3.1.3 the *.info* files include the failed cases and passed cases.

The python Script has a very important method, namely *data_extrac* This function Extracts data from all info files in the '$info\_files$' folder and '*results.json*' as well and combine the result in the '$output\_tar/result\_tar.info$ This result is then sorted in a collection data structure called *OrderedDict*. This is then the input data to tarantula suspicious function where the four already mentioned parameters are defined.

The structural design with TestCov is implemented in such a way that it can be easily used in every possible way and allows easy adaptation to find errors in all types of test suites that are generated by each test suite generator.

6 Experimental Evaluation

In this section, we show the capabilities of Tarantula in CPAchecker with some experiments made on a benchmark provided by *Bekkouche* and *sv-benchmarks*. The section concludes with a discussion of the obtained results and lessons learnt.

Table 6.1 shows the types of errors that our benchmarks contain. This type of bug is taken from BugAssist's evaluation [JM02].

All the experiments were carried out using an Intel Core i7, 2.4 GHz with 8 GB of RAM on the operating system macOS High Sierra version 10.14.6 using Ubuntu20 virtual-Box version 6.1, base Memory 4869 MB. All data of these experiments are stored and available in GitLab[1]. The CPU and Wall times are measured by BenchExec version 2.6 [2] a modern framework for reliable benchmarking and resource measurement, to run the experiments, and verification witnesses to validate the verification results. It measures CPU time, wall time, and memory usage of a tool, and allows specifying limits for these resources.

For all experiments, we have set for each technique 900 seconds for the CPU time limit. Within this, the technique should deliver results, otherwise, we consider it "*timeout*", and we do not have an explicit memory limit, but it is limited by the memory of the virtual machine.

[1] `https://gitlab.com/Schindar/fault_localization_tarantula/-/tree/master/evaluation_results`

[2] `https://github.com/sosy-lab/benchexec/releases/tag/2.6`

6.1 Benchmark

Benchmarks play an important role in evaluating the efficiency and effectiveness of solutions to automate several phases of the software development life cycle.
The benchmark provided by Bekkouche[3] consists of several C programs of 15 to 100 lines of code. It includes programs with arithmetic operations, and the faults in these programs were injected specifically for experimenting with automatic fault localization methods. To expand our benchmarks with more kinds of programs, we add one section of sv-benchmarks to this set, which include while, for loops, to cover as much as possible of type of errors.

We also use these benchmarks and replace the function assertion *sniper_assert* by the function used by CPAchecker $\_\_VERIFIER\_assert$ and replaced the cover assumption by non-deterministic values with the function $\_\_VERIFIER\_nondet\_int()$ to cover all possible options.

The SV-benchmarks structure was developed for the International Competition on Software Verification SV-COMP.[4] We take some safes versions and compared them to the unsafe versions and considered the difference between them as a bug and marked this in the program to check the effectiveness of the Tarantula.

Table 6.1: Overview of used type of error in bekkouche benchmark and sv-benchmark

Error Type	Explanation for the error
assign	Wrong assignment expression
op	Wrong operator usage e.g.: <=instead of <
init	Wrong value initialization of a variable
branch	Error in branching due to negation of branching condition
assign-for-loop	Wrong assignment inside loop
if-for-loop	Wrong check inside loop
add-unecessary	Wrong assignment which should be removed
assign-while	Wrong assignment inside while loop
index-for-loop	Use of wrong array index
index-while	Use of wrong array index inside while loop

In addition, we add more examples to the Bekkouche set called MiddleNumber and maxLoop. And inject those manually with bugs using the same principle of Bekkouche. We used this time the equality technique between the correct version and the incorrect version to confirm the correctness of the output.

$\_\_VERIFIER\_assert(wrongResult == correctResult)$.

[3] `http://capv.toile-libre.org/Benchs_Mohammed.html`
[4] `https://github.com/sosy-lab/sv-benchmarks`

Since our real goal is to find the bug, so we need the unreach-call specification, therefore we ran symbolic execution and predicate-merge-sep with the property *unreach-call* by the command line **config/properties/unreach-call.prp**.

6.2 Evaluation Metric

There are various metrics to evaluate the fault localization techniques, we developed in this section our evaluatoin metric which helped us to evaluate all available techniques in this study which is based on ranking metrics.
Our evaluation metric is called omega. Where we look at the accuracy and efficiency of each fault localization technique when the developer tries to use the technique to analyze his code and find bugs accordingly. With omega-percentage, we can determine which fault localization technique is "better" than others. Therefore, consider the worst case, leaving no gaps in the evaluation.
Ω is the percentage of the measure worst-case-step divided by the total code lines of the corresponding program and it indicates how many lines of code, its suspiciously higher or equal to the suspect of the bug location, so the lower the omega result Ω the better the technique.
The worst-case-step is defined as the following:

```
worst-case-step:=|{codeLine;rank(codeLine)<=rank(faulty codeLine) && codeLine != faulty codeLine}|
```
(6.1)

Equation (6.1) shows the worst case step definition. This is a cardinality of a set of code lines, whose rank is less than or equal to the rank of the actual error code line and this set should not contain any faulty code line.

6.2.1 Example

Let us consider the ranking result in Table 6.2 of the program **while_infinite_loop_1** from the used benchmark: The bug position is colored with gray. The suspicious column shows the suspiciousness score that the technique computes for each statement applied on this example. The ranking column shows the maximum number of statements that would have to be examined.
If we apply our evaluation metric on this example then we will have:

worst-case-step = $|\{5, 6, 1, 2, 11, 14, 4, 12\}|$ = 8 since there are 2 code lines with 1.0 suspicious and 6 code lines with 0.5 suspicious is because we take the worst case.

The result of the $\Omega = 8/20 = 0.400$ that is because we have 20 total code lines of the program.

Table 6.2: The represented example of *while_infinite_loop_1* from used Benchmarks after analyzed by *predicate-merge-sep*

codeline	suspicious	rank
5	1.0	1
6	1.0	1
1	0.5	2
16	0.5	2
2	0.5	2
11	0.5	2
14	0.5	2
4	0.5	2
12	0.5	2

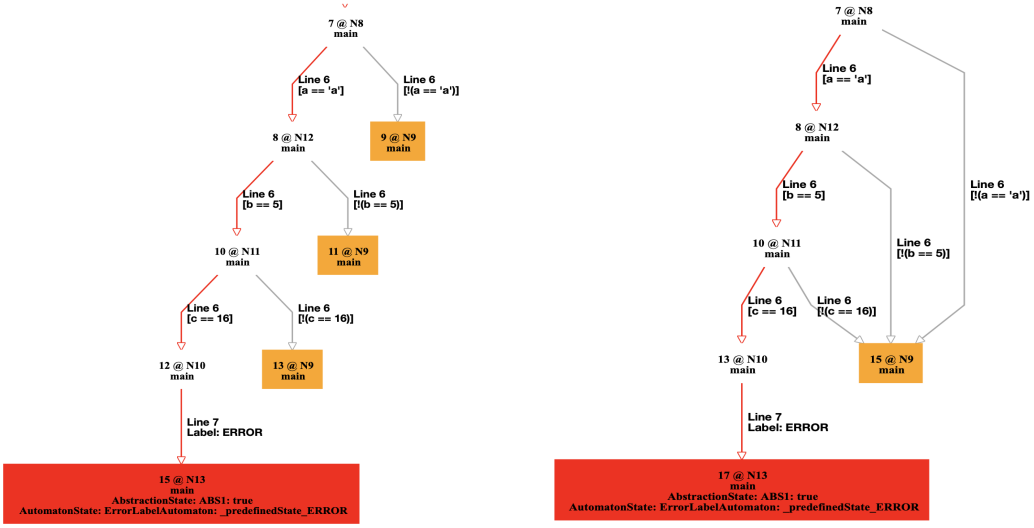
6.3 Experimental Setup

CPAchecker provides many analysis configurations, e.g. the symbolic execution and predicate analysis. We compared both techniques in our experiment to choose the best of them and compare Tarantula with the chosen configuration against other ranking metrics, DStar and Ochiai then against test-based Tarantula algorithm using test suites generators, Klee and VeriFuzz.
We have used the implementation from Section 3.1.3 to evaluate Klee and VeriFuzz. We used branch coverage as a test goal for both techniques. We first ran the programs using our algorithm (Tarantula) on CPAchecker, and since predicate analysis in CPAchecker with default merge provide merged paths as defined at Section 3.2.5., the number of error paths always corresponds to the number of safe paths, therefore the suspicion is always 0.5 for all edges. In this case, it is not possible to determine the error position. As a solution, we use in our experimental the command line: **cpa.predicate.merge=SEP** which ensures that PredicateCPA uses the $merge_{sep}$. This gives us an advantage that each path case is separated from its neighbor path and therefore the result is more efficient. But this technique is very expensive and slows down the analysis, and even runs the analysis for certain large programs infinitely.

```c
int main() {
  char a = __VERIFIER_nondet_char();
  char b = __VERIFIER_nondet_char();
  char c = __VERIFIER_nondet_char();

  if (a == 'a' && b == 5 && c == 16) {
    ERROR:__VERIFIER_error();
  }
}
```

Figure 6.1: A simple C-Program



(a) $merge_{sep}$ (b) Default Merge

Figure 6.3: Comparison between ARGs created by PredicateCPA with $merge_{sep}$ (left) and with default merge (right) operators generated by CPAchecker

Figure 6.1 shows a simple C program example and Figure 6.3 shows two ARGs applied at that example. The left one is generated using $merge_{sep}$. From the diagram, we can find out how three paths end up at one node has identity 15. After applying the Tarantula formula to this ARG, we can easily see that all edges in the circle of the graph will get suspicious of 0.5, while on the right graph we can see that the paths are separate and run to different nodes, which ensures that Tarantula is more efficient.

6.4 Results

6.4.1 Symbolic Execution vs Predicate-merge-sep

This subsection explains the results of running Tarantula on CPAchecker using symbolic execution without CEGAR and predicate analysis without merging the paths.

Table 6.3: Results of running Tarantula on CPAchecker using symbolic execution and predicate abstraction with *merge=SEP* command line on our benchmark. Note: The lower the omega result the better the technique.

Programs	#TL	Tarantula with predicate-merge-sep analysis					Tarantula with symbolic execution				
		Rank	#WC	Ω	CPU	Wall	Rank	#WC	Ω	CPU	Wall
MinmaxKO	50	3 of 8	3	0.060	9.05	7.40	3 of 5	4	0.080	6.98	7.16
middleNumber	95	2 of 8	4	0.042	8.01	9.58	2 of 6	5	0.053	7.47	7.66
middleNumber1	95	3 of 8	5	0.053	7.70	9.75	2 of 5	3	0.032	7.09	8.76
middleNumber2	95	5 of 5	27	0.284	7.14	9.83	5 of 5	27	0.284	7.11	9.34
maxLoop1	52	6 of 7	7	0.135	271	291	9 of 10	23	0.442	16.3	16.5
maxLoop2	52	4 of 6	5	0.096	11.9	16.3	5 of 5	20	0.385	8.66	9.41
maxLoop3	52	4 of 4	19	0.365	8.55	13.8	6 of 6	17	0.327	7.93	13.0
AbsMinusKO	48	3 of 6	3	0.063	7.85	9.68	4 of 4	4	0.083	7.39	8.38
AbsMinusKO2	48	6 of 6	14	0.292	7.57	8.57	4 of 4	13	0.271	7.10	7.38
AbsMinusKO3	46	5 of 7	5	0.109	7.28	9.45	2 of 3	4	0.087	6.71	8.427
AbsMinusKO4	48	5of 7	4	0.083	7.16	10.1	2 of 3	5	0.104	6.78	9.20
sanfoundary_24-1	50	2 of 4	5	0.100	8.63	13.5	54 of 6	3	0.060	8.12	8.25
array-1	30	3 of 5	3	0.100	6.65	9.65	2 of 3	3	0.100	5.93	6.07
insertion_sort-1	28	3 of 6	6	0.214	6.43	9.46	3 of 5	5	0.179	9.14	9.28
gj2007b	26	3 of 7	8	0.308	7.26	7.75	6 of 9	14	0.538	8.36	8.94
jm2006	28	3 of 6	5	0.179	6.22	6.38	6 of 6	15	0.536	5.99	6.11
while_infinite_loop_1	20	2 of 2	8	0.400	6.02	8.77	no safe paths	no safe paths	-	6.29	6.74
sum01_bug02	20	3 of 6	4	0.200	7.58	8.07	no safe paths	no safe paths	-	7.18	7.33
brs.c	51	3 of 6	3	0.059	7.28	8.21	-	-	-	timeout	timeout
partial_lesser_bound-1	31	3 of 5	4	0.129	7.12	10.7	no safe paths	no safe paths	-	7.90	8.14
array_mul_init	33	3 of 6	6	0.182	7.84	8.26	4 of 4	18	0.545	11.2	11.4
TritypeKO	101	-	-		timeout	timeout	1 of 8	4	0.040	9.23	9.71
TritypeKO2	101	-	-		timeout	timeout	3 of 8	8	0.079	9.33	9.75
TritypeKO2V2	98	-	-		timeout	timeout	5 of 8	6	0.061	9.59	11.4
TritypeKO3	99	-	-		timeout	timeout	3 of 12	3	0.030	9.39	9.61
TritypeKO4	100	-	-		timeout	timeout	3 of 6	6	0.060	9.06	9.23
TritypeKO5	100	-	-		timeout	timeout	4 of 6	6	0.060	8.32	8.51
TriPerimetreKO	104	-	-		timeout	timeout	1 of 7	4	0.038	8.81	8.99
TriPerimetreKOV2	106	-	-		timeout	timeout	3 of 9	7	0.066	9.59	9.78
TriPerimetreKO2	104	-	-		timeout	timeout	3 of 8	8	0.077	9.14	9.32
TriPerimetreKO3	102	-	-		timeout	timeout	2 of 11	3	0.029	9.65	9.85
Maxmin6varKO	195	-	-		timeout	timeout	4 of 9	5	0.026	195	196
Maxmin6varKO2	196	-	-		timeout	timeout	5 of 11	5	0.026	97.8	98.2
Maxmin6varKO3	200	-	-		timeout	timeout	5 of 11	5	0.025	98.8	99.6
Maxmin6varKO4	195	-	-		timeout	timeout	3 of 5	5	0.026	199	200.

Table 6.3 lists the results obtained by running Tarantula in CPAchecker on each benchmark program. The first column of each technique of the table shows the program name. The column #*TL* shows the total number of lines of code that each program has, while the column #*Rank* shows the rank of the error position within all possible error positions in the corresponding program from all possible ranks, of which

each program contains. The #*CPU* column presented with three significant digits is the amount of time for which the CPU was used for processing in second. The #*Wall* column presented with three significant digits and is the actual time consumed in second. Important to mention that the cell "no safe paths" means that the technique has been done and despite the discovery of the counterexample but it did not provide any safe paths, and therefore the algorithm is not efficient because Tarantula needs at least one failed and one safe path to provide suspicious. The #*WC* and Ω columns show the *worst-case steps* and Ω measures and are defined as in the Section 6.2

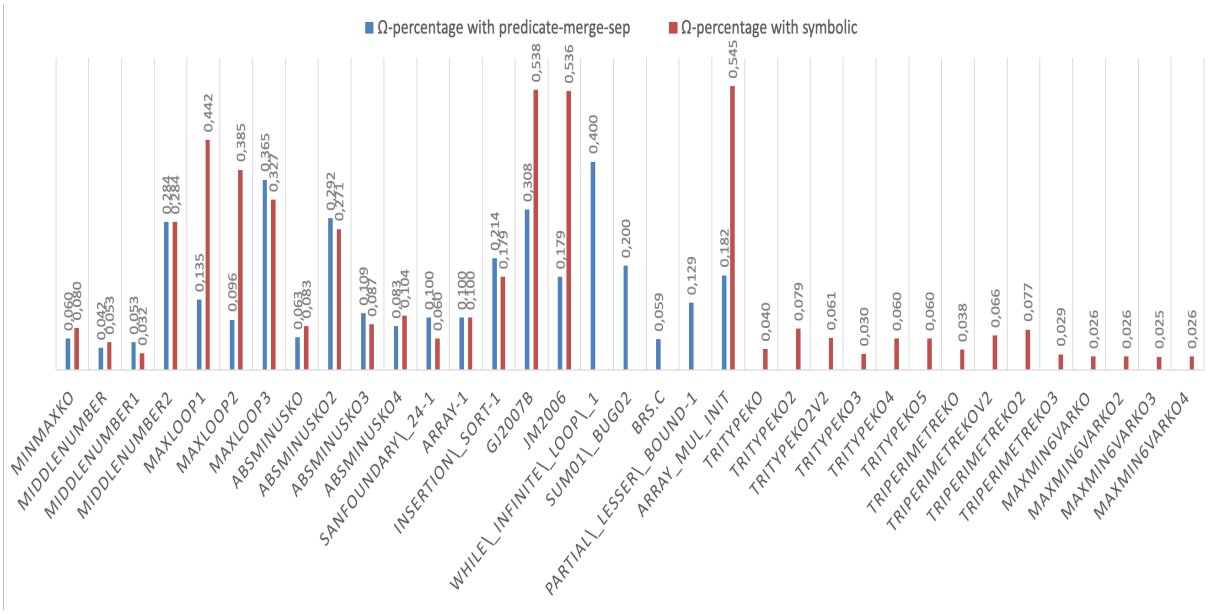


Figure 6.5: Results of running Tarantula on CPAchecker using symbolic execution and predicate abstraction with *merge=SEP* command line on our benchmark. Note: the lower the column the better the technique

Figure 6.5 reports an overview of the comparison between *symbolic execution* and *predicate-merge-sep* from the Table 6.3. The bars of the histogram in blue represent the omega Ω results of predicate-merge-sep on our benchmarks, while the red bars represent omega Ω of symbolic execution. we can easily deduce that both techniques are equally efficient in the first three programs. After that, symbolic execution begins to produce poor results with two programs $maxLoop_2$ and $maxLoop_3$, in contrast to predicate analysis, which has remained stable. After that, the two techniques maintain their effectiveness in finding bugs. Symbolic execution suddenly stops because the analysis does not provide any safe paths but returns a long vertical graph with a

single counterexample, unlike predicate analysis. In the large programs, the symbolic execution continues and the predicate analysis stops due to the high calculation of the separation of the paths.

6.4.2 Tarantula vs other Ranking Metrics with Symbolic Execution

This subsection explains the results of running Tarantula against Dstar and Ochiai on CPAchecker using symbolic execution without CEGAR.

In the following experiments, we did not measure the CPU and wall times at all, because DStar and Ochiai use the same technical structure as Tarantula, so we assumed that we did not see any differences.

The only thing we found interesting to measure is the omega percent to know which technique can work better in fault localization with ranking metrics principle.

Table 6.4: Results of running Tarantula against DStar and Ochiai ranking metrics using the CPAchecker configuration: Symbolic Execution

Programs	#TL	Tarantula		DStar		Ochiai	
		#WC	Ω	#WC	Ω	#WC	Ω
MinmaxKO	50	4	0.080	2	0.04	3	0.06
middleNumber	95	5	0.053	3	0.032	5	0.053
middleNumber1	95	3	0.032	2	0.021	3	0.032
middleNumber2	95	27	0.284	25	0.263	26	0.274
maxLoop1	52	23	0.442	24	0.462	23	0.442
maxLoop2	52	20	0.385	21	0.404	21	0.404
maxLoop3	52	17	0.327	20	0.385	21	0.404
AbsMinusKO	48	4	0.083	2	0.042	4	0.083
AbsMinusKO2	48	13	0.271	9	0.188	13	0.271
AbsMinusKO3	46	4	0.087	2	0.043	4	0.087
AbsMinusKO4	48	5	0.104	3	0.063	5	0.104
sanfoundary_24-1	50	3	0.060	5	0.100	5	0.100
array-1	30	3	0.100	1	0.033	3	0.100
insertion_sort-1	28	5	0.179	3	0.107	4	0.143
gj2007b	26	14	0.538	14	0.538	15	0.577
jm2006	28	15	0.536	14	0.500	16	0.571
while_infinite_loop_1	20	no safe paths	no safe paths	no safe paths	no safe paths	8	0.400
sum01_bug02	20	no safe paths		no safe paths		3	0.15
brs.c	51	timeout	timeout	timeout	timeout	timeout	timeout
partial_lesser_bound-1	31	no safe paths		no safe paths	no safe paths	3	0.097
array_mul_init	33	18	0.545	18	0.545	18	0.545
TritypeKO	101	4	0.040	sus = 0		4	0.040
TritypeKO2	101	8	0.079	6	0.059	8	0.079
TritypeKO2V2	98	6	0.061	6	0.061	8	0.082
TritypeKO3	99	3	0.030	1	0.010	3	0.030
TritypeKO4	100	6	0.060	6	0.060	6	0.060
TritypeKO5	100	6	0.060	6	0.060	6	0.060
TriPerimetreKO	104	4	0.038	sus = 0		4	0.038
TriPerimetreKOV2	106	7	0.066	5	0.047	10	0.094
TriPerimetreKO2	104	8	0.077	6	0.058	8	0.077
TriPerimetreKO3	102	3	0.029	1	0.010	4	0.039
Maxmin6varKO	195	5	0.026	5	0.026	5	0.026
Maxmin6varKO2	196	5	0.026	19	0.097	6	0.031
Maxmin6varKO3	200	5	0.025	19	0.095	6	0.030
Maxmin6varKO4	195	5	0.026	4	0.021	5	0.026

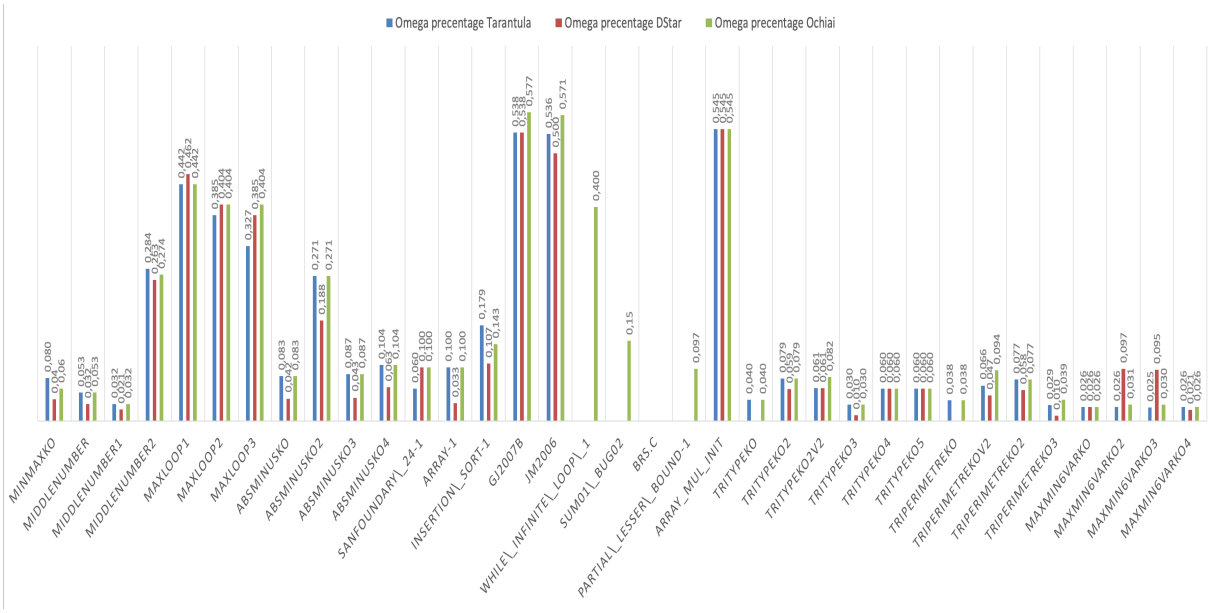


Figure 6.7: Comparison of the effectiveness of Tarantula Algorithm against other Ranking Metrics techniques, such as DStar and Ochiai using Symbolic Execution

Table 6.4 lists the results obtained by running Tarantula, DStar, and Ochiai in CPAchecker on each benchmark program. The tables contain only omega Ω percentage for the three techniques. The Cell *"no safe paths"* means that the CPAchecker using the configuration of Symbolic Execution was not able to generate any safe path so that the Tarantula and DStar can work better, but that does not affect Ochiai. The Cell *sus=0* means the Symbolic Execution analyzed well and the bug was also on the counterexample trace, but this technique assigned a suspicious of 0 to the code line.

Figure 6.7 reports an overview of the comparison between Tarantula, Ochiai, and DStar. The bars of the histogram in blue represent the omega Ω results from Tarantula, while the bars in green represent the omega Ω results of Ochiai and the red bars the results of DStar.

6.4.3 Formal Verification Tarantula vs Test-based Tarantula

This subsection explains the results of running Tarantula on test-suites generated by Klee and VeriFuzz and then comparing these to the obtained result of symbolic execution from Section 6.4.

Table 6.5: Results of running Tarantula on test suites generated by Verifuzz and Klee in TestCov on our benchmark. Note: The lower the omega result in Ω the better the technique

Programs	#TL	Verifuzz					Klee				
		Rank	#WC	Ω	CPU	Wall	Rank	#WC	Ω	CPU	Wall
MinmaxKO	50	2 of 4	9	0.180	902	902	1 of 3	9	0.180	0.870	1.59
middleNumber	95	1 of 5	9	0.095	901	902	1 of 3	11	0.116	1.17	1.39
middleNumber1	95	1 of 3	22	0.232	901	901	1 of 3	21	0.221	1.44	1.95
middleNumber2	95	3 of 4	36	0.379	901	901	-	no failing cases		1.30	1.83
maxLoop1	52	-	not in ranking		902	902	-	not in ranking		0.421	0.950
maxLoop2	52	1 of 3	7	0.135	902	902	-	not in ranking		0.883	1.45
maxLoop3	52	1 of 3	6	0.115	902	902	-	no safe cases		0.884	1.10
AbsMinusKO	48	1 of 4	6	0.125	901	901	-	no failing cases		0.957	1.17
AbsMinusKO2	48	-	no safe cases		901	901	-	no safe cases		0.793	1.00
AbsMinusKO3	46	3 of 4	19	0.413	901	901	-	no failing cases		1.07	1.31
AbsMinusKO4	48	2 of 4	19	0.396	901	901	2 of 3	16	0.333	0.863	1.40
sanfoundary_24-1	50	-	no failing cases		6.01	6.62	-	not in ranking		2.24	3.01
array-1	30	-	no failing cases		900.	901	-	no failing cases		0.788	0.981
insertion_sort-1	28	-	no failing cases		901	901	-	no failing cases		0.629	0.847
gj2007b	26	2 of 3	23	0.885	903	903	2 of 4	19	0.731	520.	525
jm2006	28	-	no test cases		902	903	-	no test cases		1.14	1.45
while_infinite_loop_1	20	-	no test cases		5.59	5.69	-	no test cases		0.513	1.18
sum01_bug02	20	-	no test cases		6.62	7.64	-	no test cases		1.16	1.97
brs.c	51	-	no safe cases		6.53	6.85	-	no safe cases		0.627	1.21
partial_lesser_bound-1	31	-	no test cases		5.39	7.40	-	no test cases		0.561	1.24
array_mul_init	33	-	no test cases		6.40	6.57	2 of 4	7	0.212	1.39	2.74
TritypeKO	101	6 of 6	last element rank		902	902	5 of 5	last element rank		1.74	2.27
TritypeKO2	101	7 of 7	last element rank		902	902	9 of 9	last element rank		1.89	2.74
TritypeKO2V2	98	8 of 8	last element rank		902	902	8 of 8	last element rank		1.86	2.12
TritypeKO3	99	-	not in ranking		902	902	-	not in ranking		1.90	2.32
TritypeKO4	100	7 of 7	last element rank		901	902	7 of 7	last element rank		1.71	2.49
TritypeKO5	100	1 of 7	15	0.150	901	901	5 of 6	63	0.630	1.71	2.29
TriPerimetreKO	104	-	not in ranking		902	902	-	not in ranking		1.77	2.04
TriPerimetreKOV2	106	7 of 7	last element rank		902	902	7 of 7	last element rank		1.76	2.00
TriPerimetreKO2	104	7 of 7	not in ranking		902	902	9 of 9	not in ranking		1.84	1.97
TriPerimetreKO3	102	-	not in ranking		902	902	-	not in ranking		1.92	2.47
Maxmin6varKO	195	1 of 5	10	0.051	904	905	-	no failing cases		10.4	11.4
Maxmin6varKO2	196	10 of 10	not in ranking		904	904	-	no failing cases		5.08	5.85
Maxmin6varKO3	200	12 of 12	not in ranking		904	905	-	no failing cases		5.35	5.46
Maxmin6varKO4	195	1 of 7	9	0.046	904	905	-	no failing cases		10.5	10.9

Table 6.5 shows the results obtained for each benchmark program by running Tarantula on generated test suites from Klee and Verifuzz. The column is already explained in subsection Section 6.4. However, it is important to note that the *"not in the ranking"* cell means that the technique provided a ranking of the possible error position, but the actual error position we were looking for was not shown in the ranking. The cell *"no test cases"* means that the technique could not generate test suites for this particular

program and therefore Tarantula cannot provide a ranking. The cells *"no failing cases"* and *"no safe cases"* mean that the technique could not provide any faulty/safe cases and therefore the Tarantula is not efficient, as previously mentioned.

The cell *"last element rank"* means that the technique has found the fault position as a sure case and gives this position the suspicious 0.0, which is considered the last element in the ranking.

Finally, the cell *"timeout"* means the technique could not be terminated within the set time, namely **900 sec/15 min**

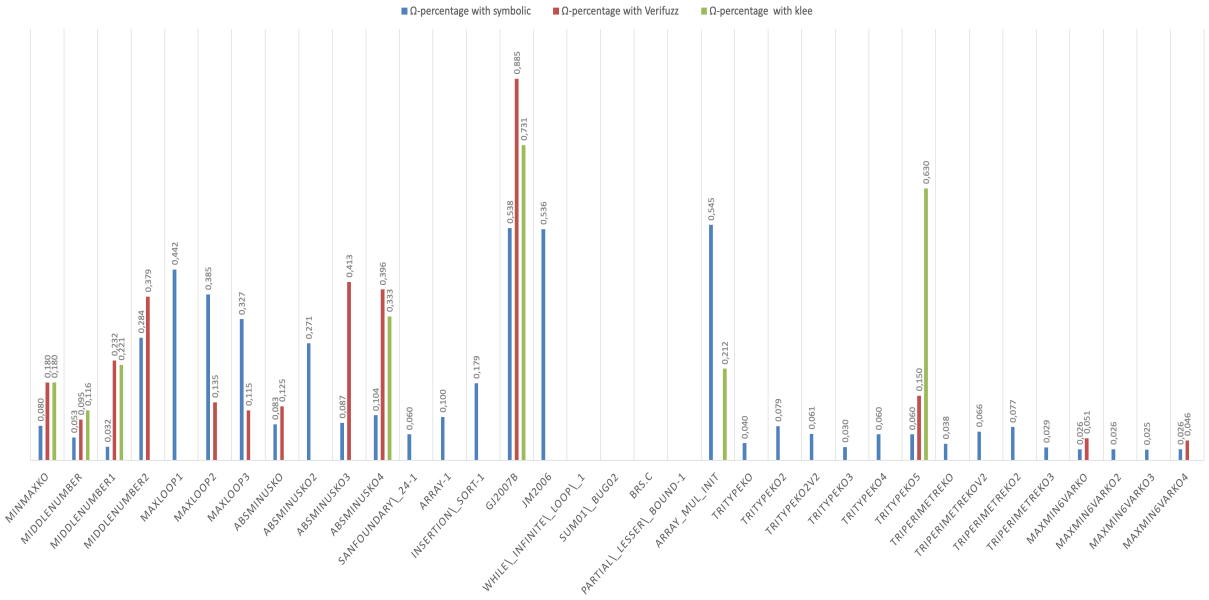


Figure 6.9: Comparison of the effectiveness of each technique: Symbolic execution against Klee and VeriFuzz

Figure 6.9 reports an overview of the comparison between symbolic execution and the test-based techniques Klee and VeriFuzz. The bars of the histogram in blue represent the omega Ω results of symbolic execution, while the bars in green represent the omega Ω results of Klee and the red bars the results of VeriFuzz.

In the first three programs, we can see that the three techniques have detected bugs but with different results of omega. After that, Klee and VeriFuzz start to deteriorate. But symbolic execution remains effective and continues to find bugs.

6.5 Threats to Validity

There are a few threats to validity about generalizing the results presented in this paper.

First, all techniques were evaluated using only the Tarantula metric as the basis of a probability distribution. The use of other error localization measures may have different results. Similarly, using other configurations of CPAchecker or other test generators could lead to other interesting results which we did not expect, so only further research can prove the efficiency of formal verification as a better approach to test suites generation.

Second, in our evaluation, we introduced the measure Ω in Section 6.2 measure as the basis for evaluating all of the techniques against each other. However, other evaluation measures can also affect the performance of the results, so we get other unexpected results.

Finally, different faults or subject programs may affect the performance of our techniques. We have tried to include various types of faults across multiple versions of subject programs, but only additional studies can further reduce this threat. Additionally, the used benchmark set contains only C programs because this is the only language supported by all the evaluated tools. Therefore, there is no guarantee that our results can be transferred to programs written in other programming languages, for example, software written in functional programming languages.

Another limitation of the experiment is that the results presented in this experiment apply only to the case where the subjects used in the study each contain a single fault. We cannot generalize these results to these or any programs that have multiple faults.

6.6 Discussion

In this section, we summarize and provide some observations about the results that we obtained.

Our fault localization algorithm depends on the application of the test-based algorithm on ARG. In most cases, a single incorrect entry was sufficient to localize the exact location of the fault.

The effectiveness of the algorithm is much better when using symbolic execution, since this technique gave results, regardless of the size of the programs, on the contrary to predicate analysis without merge where it failed due to large programs and ran out of time. Especially effective were both techniques, if the fault is within the block of the conditional statement unlike when it is in the conditional statement itself or the for-loop, the fault suspicious is then often very small, and the reason behind this is that the bug-position can be belonged to the safe part more than its affiliation to the wrong

part thus it is considered more safe than dangerous.

We have concluded through experience that Symbolic execution is better than Klee and VeriFuzz. The three techniques were especially effective when the fault was within the block of the conditional statement, but less so when it was in the conditional statement itself or in the for-loop, the same conclusion as mentioned before. Klee and VeriFuzz very often generated bad analyse through the whole program, so the bug sometimes suspected 0.0. Quite often both techniques delivered only counterexamples but no safe cases, so Tarantula can work perfectly well, thus the suspicious is 1.

Klee always generated test suites very quickly as it's shown in the columns "Wall time" and "CPU time" of Table 6.5, we thought it was because of the configuration of the coverage branch, then we tried other properties, but no better results were found.

VeriFuzz almost always took a long time to generate test suites, no wonder, since this works with genetic algorithms and takes a lot of time to mutate and generate new generations that are stronger than the previous one, see Section 3.1.2. However, the results were again not very good, and often fail to deliver good results.

The second weakness that we found in this algorithm is when the error is in the definition of a local variable in the function its self as in the second *absMinusKO4*, then the error suspicious is very small and the reason for this is that the number of error paths is always equal to the number of safe paths and therefore we get an error suspicious of 0.5 in most cases.

The last weakness is in the case of predicate analysis when analyzing large programs then the algorithm is not able to finish its analysis because is too complicated for predicate analysis without merge. There are extremely many paths through the many branches.

From these results, we can find out that symbolic execution is much better and faster to analyze the program. That's why we choose symbolic execution to compare with Klee and VeriFuzz.

We can also see that the DStar technique performed better than the other techniques, namely Tarantula and Ochiai. The reason for this is that DStar does not take TotalPaths into account in its suspicious form, which has the advantage that the fault location is more often on the fault path than on the safe path, which increases the suspicion of the fault position. We found it particularly interesting that Ochiai's Ω percentage was almost the same as Tarantula's, but Ochiai analyzed more test programs than Tarantula. The reason for this is that Ochiai does not need at least one failure path and at least one safe path in contrast to Tarantula.

To improve the usability of our tool, we need to create an Eclipse plugin to use our algorithm interactively during the development process. The plugin highlights potential bugs in the code that is under development.

7 Future Work

Future work should include the use of more advanced fault localization analysis on CPAchecker for the look-ahead method, with an emphasis on reducing the suspiciousness to choose the best fault localization technique or to design a new ranking method and use it as a default feature in CPAchecker. Besides, in our experiment, we compared just three ranking metrics against each other, but the work on more ranking metrics, such as *Barinel and Op2* is still open and can be analyzed. Moreover, we plan to improve CPAchecker to be able to not only analyze C-programs but also java and JavaScript programs.

8 Conclusion

Techniques such as model checking and data flow analysis can find subtle bugs in programs. However, the problem of finding the cause of the error is referred to as the user.

In this work, we have shown how to apply the test-based tarantula algorithm to model checking using ARG. The key idea is to find transitions in the error trace that do not appear in a correct trace and show that techniques based on model checking can be effective in locating bugs (and identifying potential fixes).

Regarding the experimental results, ranking algorithms, *DStar and Ochiai* are improvements and work better than Tarantula as expected due to the reasons which mentioned before. The proposed algorithm with Tarantula in CPAchecker using *symbolic execution* was able to identify potential faults, 88.57% of the chosen benchmarks with a very good percentage of Ω, while the same algorithm using *predicate-merge-set* found 60% of the total benchmarks with very good results from Ω, and failed to obtain useful faulty lines for fourteen of the adopted benchmarks due to the height calculation of the separating the paths from merge which led the algorithm to timeout.

Our approach has also shown that it delivers better results than test-based approaches. We compared symbolic searches to *Klee* and *VeriFuzz* test generators and found that Klee was only successful in 17.14% of all benchmarks used, while VeriFuzz was better than Klee but not CPAchecker in 37.14%.

We have found that our Tarantula technique works best with more error paths as well as more safe paths. With our technique, we can observe its results with any subset of the ARG as long as you have at least one error path and at least one safe path. We have discovered that using the information from multiple error paths allows the technique to take advantage of the richest information base.

Several interesting research questions remain open:

1. Is it possible (in some cases) to suggest a fix for a buggy program?

2. What other types of information can be used to locate the cause of the error? Algorithmic debugging provides information in the form of dynamic data dependencies and user input. Dynamic data dependencies from error tracking track the flow of values between statements and can be very helpful in returning from an assertion error to the variable definitions that caused it. Likewise, user input

about which functions in a program can be *"trusted"* (e.g. library functions) could be used to guide the search for a cause.

Bibliography

[90] "IEEE Standard Glossary of Software Engineering Terminology." In: (1990).

[Ali12] M. A. Alipour. "Automated fault localization techniques: a survey." In: (2012), pp. 6–7.

[Avi+04] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. "Basic concepts and taxonomy of dependable and secure computing." In: (2004), pp. 11–33.

[BDW17] D. Beyer, M. Dangl, and P. Wendler. "A Unifying View on SMT-Based Software Verification." In: (2017), pp. 305–309.

[Bey+09] D. Beyer, A. Cimatti, A. Griggio, M. E. Keremoglu, and R. Sebastiani. "Software Model Checking via Large-Block Encoding." In: (2009), pp. 1, 11.

[BGS] D. Beyer, S. Gulwani, and D. A. Schmidt. "Combining Model Checking and Data-Flow Analysis." In: (), pp. 515, 516.

[BH13] R. Baker and I. Habli. "An Empirical Evaluation of Mutation Testing for Improving the Test Quality of Safety-Critical Software." In: (2013), 2, 23, and 24.

[BHT07] D. Beyer, T. A. Henzinger, and G. Théoduloz. "Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis." In: (2007), pp. 504–518.

[BHT08] D. Beyer, T. A. Henzinger, and G. Théoduloz. "Program Analysis with Dynamic Precision Adjustment." In: (2008), pp. 29–38.

[BK11] D. Beyer and E. Keremoglu. *CPAchecker: A Tool for Configurable Software Verification*. 2011, pp. 184–190.

[BL18] D. Beyer and T. Lemberger. "CPA-SymExec: Efficient Symbolic Execution in CPAchecker." In: (2018).

[BL19] D. Beyer and T. Lemberger. "TESTCOV: Robust Test-Suite Execution and Coverage Measurement." In: (2019).

[BLW15] D. Beyer, S. Löwe, and P. Wendler. "Sliced Path Prefixes: An Effective Method to Enable Refinement Selection." In: (2015), pp. 228–243.

[BMV19] A. Basak Chowdhury, R. Medicherla, and R. Venkatesh. "VeriFuzz: Program Aware Fuzzing: (Competition Contribution)." In: Apr. 2019, pp. 244–249. ISBN: 978-1-4939-9100-6. DOI: 10.1007/978-3-030-17502-3_22.

[BNR03] T. Ball, M. Naik, and S. K. Rajamani. "From symptomto cause: localizing errors in counterexample traces." In: (2003), pp. 97–105.

[BWK12] D. Beyer, P. Wendler, and M. Keremoglu. "Predicate Abstraction with Adjustable-Block Encoding." In: (2012).

[CDE08] C. Cadar, D. Dunbar, and D. Engler. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs." In: (2008).

[CGL83] E. M. CLARKE, O. GRUMBERG, and D. E. LONG. "Model checking and abstraction." In: (1983), pp. 1, 23.

[Cla+03] E. Clarke, O. Grumberg, S. K. Jha, S. Jha, Y. Lu, and H. Veith. "Counterexample-guided abstraction refinement for symbolic model checking." In: (2003), pp. 752–794.

[Cla08] E. M. Clarke. "The birth of model checking." In: (2008), pp. 3–4.

[CW09] M. Chechik and M. Wirsing. *Fundamental Approaches to Software Engineering: 12th International Conference, FASE 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software*. ETAPS 2009, York, UK, 2009, pp. 485–487.

[Gen+18] J. Gennari, A. Gurfinkel, T. Kahsai, J. A. Navas, and E. J. Schwartz1. "Executable Counterexamples in Software Model Checking." In: (2018), pp. 504–518.

[GS97] S. Graf and H. Saidi. "Construction of Abstract State Graphs with PVS." In: (1997), pp. 72–83.

[GSB07] A. Griesmayer, S. Staber, and R. Bloem. "Automated Fault Localization for C Programs." In: (2007), pp. 95–111.

[JH05] J. A. Jones and M. J. Harrold. "Empirical Evaluation of the Tarantula Automatic Fault-Loc alization Technique." In: (2005).

[JM02] M. Jose and R. Majumdar. "Cause Clue Clauses: Error Localization using Maximum Satisfiability." In: (2002), pp. 49, 76.

[JM07] R. JHALA and R. MAJUMDAR. "Software Model Checking." In: (2007).

[Ket20] M. Kettl. *Fault Localization for Formal Verification. An Implementation and Evaluation of Algorithms based on Error Invariants and UNSAT-cores*. Bachelor's Thesis, LMU Munich, Software Systems Lab. 2020.

[Kus+14] S. Kusumoto, A. Nishimatsu, K. Aishie, and K. Inoue. "Experimental Evaluation of Program Slicing for Fault Localization, Empirical Software Engineering." In: (2014), pp. 49, 76.

[LS08] M. H. Liffiton and K. A. Sakallah. "Algorithms for Computing Minimal Unsatisfiable Subsets of Constraints." In: (2008), pp. 75, 86.

[PBG05] M. P. Prasad, A. Biere, and A. Gupta. "A survey of recent advances in SAT-based formal verification." In: (2005), pp. 156, 173.

[Rei87] R. Reiter. "A Theory of Diagnosis from First Principles." In: (1987), pp. 57, 95.

[Rot+01] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. "Prioritizing Test Cases For Regression Testing." In: (2001), pp. 929–948.

[Saf+07] S. Safarpour, H. Mangassarian, A. Veneris, M. H. Liffiton, and K. A. Sakallah. "Improved Design Debugging using Maximum Satisfiability." In: (2007), pp. 13, 19.

[SCK17] H. A. de Souza, M. L. Chaim, and F. Kon. "Spectrum-based Software Fault Localization: A Survey of Techniques, Advances, and Challenges." In: (2017), pp. 2–3.

[Wei] M. Weiser. "Programmers Use Slices When Debugging." In: (), pp. 521, 531.

[Won+14] W. E. Wong, V. Debroy, R. Gao, and Y. Li. "The DStar Method for Effective Software Fault Localization." In: *IEEE Transactions on Reliability* 63.1 (2014), pp. 290–308.

[Woo+09] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald. *Formal Methods: Practice and Experience*. ACM, 2009, pp. 2, 3.

[Zel02] A. Zeller. "Isolating cause effect chains from computer programs." In: (2002), pp. 1–10.