DEPARTMENT OF INFORMATICS

LUDWIG MAXIMILIAN UNIVERSITY OF MUNICH

Bachelor's Thesis

Fault Localization in Model Checking

Implementation and Evaluation of Fault-Localization Techniques with Distance Metrics

Angelos Kafounis

Supervisor:	Prof. Dr. Dirk Beyer
Mentor:	Thomas Lemberger
Submission Date:	15.09.2020

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

Munich, 15.09.2020 Angelos Kafounis

Abstract

A very often occurrence in the world of software developers are faulty programs. The definition of a faulty program is a program that entails an error in its code. It is the developer's job to search, locate, and fix the fault that caused the error. However, this process can be extremely time-consuming. Model checking is a method that can be used to find out whether a program is faulty or not. This work discusses the use of distance metrics for automatic fault-localization in model checking. When a program doesn't fulfill its specification, the model checker produces a counterexample that represents an instance of an undesirable behavior of the program. We take the generated counterexample of a failed program and we compared it with a program execution that is as close as possible to the counterexample and does not lead to an error. The difference between the failed run and the closest to it successful run, should be the fault. However, how close two program executions can be with each other depends on what kind of metric is used. The metrics that are discussed in this work belong to the category of distance functions that work with predicate abstraction, which makes the states of a program abstract instead of concrete, and that makes the techniques more scalable and enables a more detailed and understandable explanation of why the error occurred. To compare the considered techniques, we implemented them in an unifying framework. We conducted an experimental evaluation using 34 benchmarks and compared the three fault-localization techniques with each other in regards to their time and efficiency. The results of the evaluation show that using an automated method for fault-localization purposes can be very promising and can save for the developer a lot of time.

Contents

List of Figures

List of Tables

1 Introduction

Debugging can be notoriously difficult and is one of the most time-consuming processes in software development [13]. The average developer spends 75 % of their time debugging [1]. Usually, developers have to go manually through the code searching for the faulty statement that caused the violation of the specification. Researchers have used model checking for fault-localization purposes. The application of the model checking technology to isolate and understand errors has been vastly used in the past years. Model-checking is a method, through which a program is checked to make sure that a given specification is fulfilled. If the specification is not fulfilled, then a counterexample will be generated. The counterexample is an instance of a flow of the input program that ends up violating the specification.

An example is the following minmax.c program. The specification that must not be violated is in line 13, `assert(least<= most);`. This program currently doesn't fulfill this specification. The reason is the assignment on line 10, `most = input2;`.

```
1  int main (){
2    int input1, input2, input3;
3    int least = input1;
4    int most = input1;
5    if (most < input2)
6      most = input2;
7    if (most < input3)
8      most = input3;
9    if (least > input2)
10     most = input2; // ERROR
11   if (least > input3)
12     least = input3;
13   assert (least <= most);
14 }
```

Listing 1.1: minmax.c with the error in the 10-th line

In this work, we used fault-localization techniques based on distance functions to localize the source of the fault. In other words, instead of going line to line through the code, trying to find out what caused the error, this method allows someone to apply a fault-localization technique and either point out the exact location of the bug or eliminate most lines that are not important for the error, having at the end only a few suspicious lines to investigate. We used various distance metrics to achieve that. The basic concept is, first to find all the program executions that do not lead to an error. Afterwards, the already generated counterexample is being compared with each successful execution. This comparison happens through a distance function. The function calculates how close the successful execution is to the counterexample. The goal is to find a successful execution that is as close as possible to the counterexample. After finding the closest to the counterexample successful run, then the fault should be the difference.

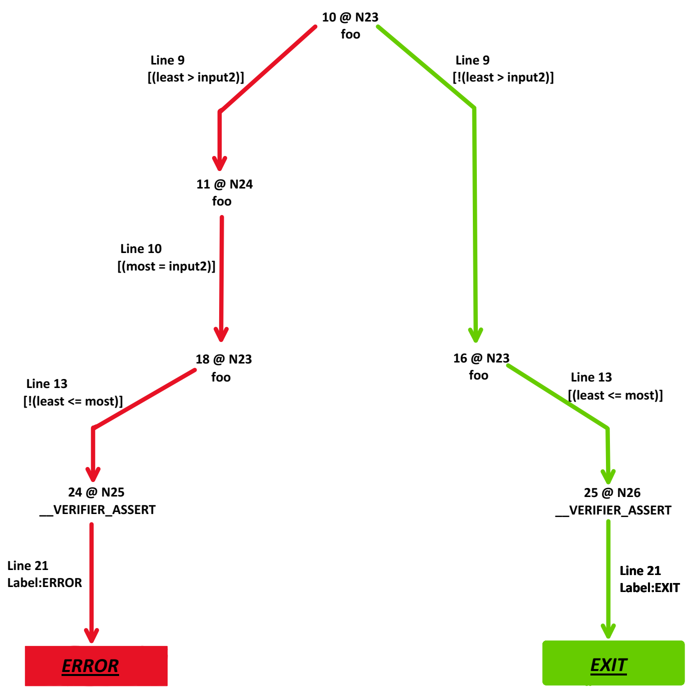


Figure 1.1: The red control flow path is the counterexample and the green path is the closest successful execution. At node `@N23` the two paths take separate ways and the red path executes the line 10 while the green path does not. This is the only difference between the successful execution and the counterexample, which is exactly the fault.

Looking at the figure 1.1 we observe that the difference between the counterexample and the closest successful execution is exactly the line 10, which is the faulty code that caused the error.

2 Related Work

Tarantula [16], [10] This fault localization technique utilizes coverage information of passed and failed test suites to find the location of the faulty statement. After the successful execution of the algorithm, the result will be in the shape of a table, which ranks in descending order the statements based on their suspiciousness score (%). The developer has to go through that table and examine the source code based on their level of suspiciousness, in contrast to our technique in which the developer has to review the differences between the failed program execution and the closest to its successful execution.

Delta Debugging [10] Delta debugging is a divide and conquer method by which our original debugging task is simplified by minimizing the input. It accepts two different inputs, one which provides a correct result and another which causes a failure. The algorithm is minimizing the difference between the two inputs while trying to look for the most successful outcome. The goal is to approach systematically a large failing input, simplify and reduce it into its minimum form. This technique is quite different from the ones that we applied in our work but the result that is being delivered to the developer is quite similar. In both cases, the result is a set of the most suspicious for the error-lines of code. In our case are the differences between the failed execution and its closest successful run while the result of the Delta debugging is the minimum form of the original failing input that was calculated.

Similar Distance Metrics [5], [8] Distance metrics that examine concrete error explanations are related to our metrics. In both cases, a distance function, which calculates the distance between two program executions, is defined, but the calculation is completely different. The metrics in this paper make alignments between parts of the two program executions to calculate how different the two executions are from each other. However, the related metrics make use of the SSA Index (a table, which enumerates all the variables of a node and their assigned values), and then compare each variable between the two executions and the sum is the number of differences.

SSA-based vs Alignment-based Metrics [5], [8] The biggest downside of the SSA is that the distance metric uses values from all possible control flow paths. This could lead to false explanation of the error, because program executions with similar values in portions of the control flow may not be executed in either successful program execution or the counterexample. Nevertheless, the process of alignments has its drawbacks. Most of the time, the process of alignments leads to some serious performance problems as a result of the time needed to define the aligned states between the two executions.

3 Background

3.1 Formal Verification

3.1.1 Data Flow Analysis

The control-flow automaton (CFA) [2, 4], is an illustration of a program as a directed graph. The CFA nodes represent program locations. The first node is the entry of the program. The nodes with no children are the ones that represent the program exits. Furthermore, CFA edges connect two nodes, if transfer control can be executed from one location to the next. The control-flow edges are labeled based on their action that transfer control. There are the assume edges that represent all the statements that depending on their value can change the flow of the program (For example "if ()" can be true or false) and there are also the statement edges that represent an assignment of a statement (For example "a = 0"). Those types of edges are a simpler form of programs written in C.

3.1.2 Abstract Reachability Graph

Abstract Reachability Graph (ARG) [11] is a formal representation of the unrolled analyzed program. The program is portrayed as a directed acyclic graph. An ARG for a control-flow automaton and a specified abstraction is a graph that consists of ARG Nodes (representing the abstract program states, e.g., including control-flow location and call stack) and edges which connect those nodes and model the transfer that leads from one abstract state to the next one [12]. The nodes of an ARG are called ARG states and the edges are the edges of a control-flow automaton.

3.1.3 Predicate Abstraction

Predicate Abstraction [7] is an abstract interpretation technique, which unwinds the CFA in an Abstract Reachability Graph until a certain point is reached. The representation of the abstract states is constructed through a given set of predicates over program variables.

3.1.4 Counterexample-Guided Abstraction Refinement

Counterexample-guided abstraction refinement (CEGAR) [6] is a methodology in which, the primary abstract model is generated through the automatic analysis of the control structures in the program to be verified. The analysis creates an abstract model based on some precision, which is an approximation of the program. In case a state of the abstract model which is affiliated to the error location is found, the initial non-approximated program path that ends up in this state is reconstructed from the abstract reachability graph and checked for feasibility. Thereupon, providing that the error (target) path is feasible, the program is marked as unsafe and the analysis is terminated. On the other hand, if the error path is not feasible, then the precision of the analysis has to be refined for it to become precise enough for the successful elimination of the error path from the abstract reachability graph.

3.1.5 CPAchecker

CPAchecker [3, 4] is an open-source framework for software verification that implements the configurable program analysis concept (CPA). CPA specifies the abstract domain that is used for the analysis of the program. CPAchecker supports the verification of C programs (GNU C and C 11). The first step of an analysis with CPAchecker is the creation of a CFA for the given program. The core of CPAchecker is the CPA algorithm. This algorithm is a reachability analysis algorithm that uses a CPA to analyze the given program. Furthermore, the CPA algorithm can be wrapped inside further algorithms that implement approaches such as CEGAR or conditional model checking. The verification result typically consists of the set of all reachable abstract states as determined by the CPA algorithm.

3.2 Fault-Localization with Distance Metrics

A distance metric [14] for program executions is a function d(a, b) (where **a** and **b** are executions of the same program) that satisfies certain properties:

1. Nonnegative property: $\forall a.\ \forall b.\ d(a,b) \geq 0$
2. Zero property: $\forall a.\ \forall b.$ d(a,b) = 0 $\iff$ a = b
3. Symmetry: $\forall a.\ \forall b.$ d(a,b) = d(b,a)
4. Triangle inequality: $\forall a.\ \forall b.\ \forall c.$ d(a,b) + d(b,c) $\geq d(a,c)$

3.2.1 Distance Metrics in the Context Of Fault-Localization

The use of distance functions to find faulty code makes it easier for the developer to understand what went wrong. The idea behind the metrics, Abstract Distance Metric and Control Flow Distance Metric, in this work consists of five steps:
The first step is to find a feasible counterexample that ends up to the violation of the specification. Then, we have to find all the successful program executions that are going

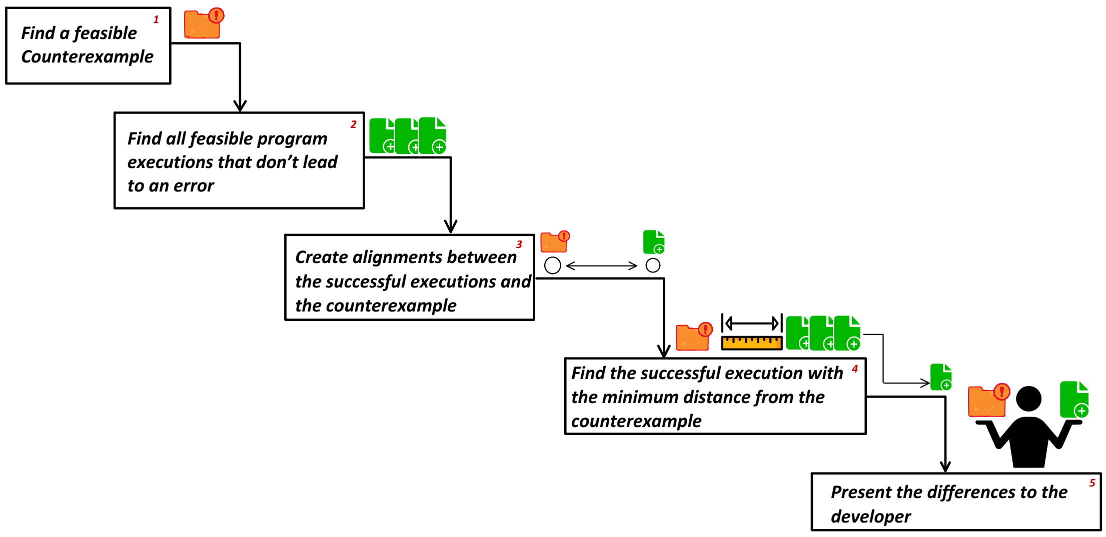

Find a feasible Counterexample
1
Find all feasible program executions that don't lead to an error
2
Create alignments between the successful executions and the counterexample
3
Find the successful execution with the minimum distance from the counterexample
4
Present the differences to the developer
5

Figure 3.1: Five Step Process of Distance Metrics in fault-localization

through the same cfa nodes as the counterexample does. Once this step is completed, then we continue to make alignments between the nodes of the counterexample and the nodes of each successful execution. After the alignments have been done, we have to measure how different the counterexample from each successful execution is using a chosen distance metric. After the comparison of paths is completed, we proceed in finding the successful execution, which is closer to the counterexample than the others. For the final step, we compare the closest to the counterexample successful run that was found in the previous step and we present to the user, what changes have to be made in the counterexample to become this successful execution.

3.2.2 Alignments

In fault-localization, a distance function is used to find the differences between two program executions. For this purpose, it must first be defined which parts of the two executions have to be compared with each other for the distance to be accurate. This step can be done using a mapping function, called **alignment**. This function maps the CFA nodes of an execution **a** with the CFA nodes of an execution **b** that are most fit for each other.

Each node has a control location number **c**. Furthermore, each node is represented through the statements **s** that it contains. Control-location number $\mathbf{c}(s_i^a)$ is the i-th statement of the program execution **a**.

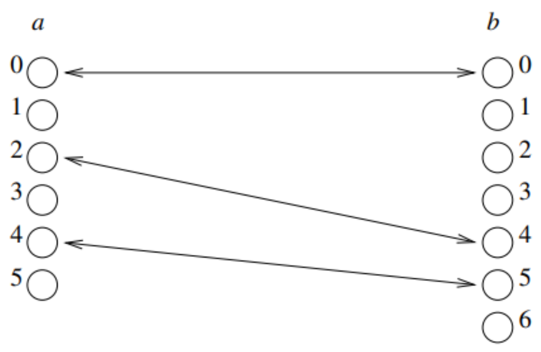


Figure 3.2: Example alignment of two generic runs a and b [5]

Definition 1 *(ALIGNMENT, align(i,j))*

$$align(i,j) = \begin{cases} 1, & if\ c(s_i^a) = c(s_j^b) \\ & \wedge\ \forall k. \neq j.\ align(i,k) = 0 \\ & \wedge\ \forall l \neq i.\ align(l,j) = 0 \\ & \wedge\ \forall m > i, n < j.\ align(m,n) = 0 \\ & \wedge \forall m < i, n > j.\ align(m,n) = 0 \\ 0, & \quad otherwise \end{cases}$$

where $i, l, m < |\boldsymbol{a}|$ *and* $j, k, n < |\boldsymbol{b}|$.

For the successful alignment of two Nodes there are some requirements that have to be fulfilled.

- Two CFA nodes can be aligned, if and only if the have the same control location.
- Alignments are unique, meaning that each Node at **a** can be aligned with maximum one Node in **b**, and in reverse.
- Alignments preserve ordering: for instance, if **i** is aligned with **j**, then no earlier Node than **i** in a is allowed to be aligned with a later than Node in b than **j**.

Preservation of ordering: See Figure 3.1: It is not allowed for two lines two cross each other.

Another important information for the metric is the number of the CFA nodes of the program executions which aren't paired with another CFA node from the other execution. Definition 2 describes formally, when two different program segments, from two different program executions, are **not** aligned with each other (unaligned).

Definition 2 *(UNALIGNED, $unalign(a/b)$(i/j))*

$$unalign_a(i) = \begin{cases} 1, & if\ \forall j.\ \neg align(i,j) \\ 0, & otherwise \end{cases}$$

$$unalign_b(j) = \begin{cases} 1, & if\ \forall i.\ \neg align(i,j) \\ 0, & otherwise \end{cases}$$

where $i < |a|$ and $j < |b|$.

3.2.3 Abstract Distance Metric

In fault-localization is easier to spot a faulty code by seeing:

Predicate changed:
was: var1 < var2
now: var1 <= var2

instead of seeing an actual concrete value of a variable that was changed, for instance:

Value of **var1** changed:
from 2147483615 to 2340552562

This technique aims to find the fault using abstract states in order to produce a better explanation to the developer about the source of the fault. Once a counterexample has been found and all the successful executions have been calculated, we have to define our distance function to find a successful execution with the minimum distance from the counterexample. The closest to the counterexample successful execution is going to be determined using the distance metric which is defined as follows.

Given an unsuccessful execution **a** and a successful execution **b**, we define the distance $d(a,b)$ [5] through the calculation of the atomic changes needed to turn **a** into **b**. The atomic changes are to be found gradually. In general we have to compute:

1. Changes in predicates
2. Changes in actions
3. Number of unaligned states of the executions

We define here the **i**-th predicate of a state **s** of the execution **a**, as $p(s_i^a)$ and respectively for the execution **b** as $p(s_i^b)$

Definition 3 *(Δp(i,j,v) Δp(a,b))* ***Predicate Distance***

$$\Delta p(i,j,v) = \begin{cases} 1, & if\ align(i,j) \wedge p_v(s_i^a) \neq p_v(s_j^b) \\ 0, & otherwise \end{cases}$$

where $i < |a|$, $j < |b|$ *and* $v < |p_v(s_i^a)|$

The predicate distance is defined :

$$\Delta p(a,b) = \sum_{i=0}^{|a|-1} \sum_{j=0}^{|b|-1} \sum_{v=0}^{|p(s_i^a)|-1} \Delta p(i,j,v)$$

Definition 4 *$(\Delta\alpha(i,j), \Delta\alpha(a,b))$* ***Changes in actions of the execution***
Changes in actions are defined in a similar alignment-based manner:

$$\Delta\alpha(i,j) = \begin{cases} 1, & if\ align(i,j) \wedge \alpha_i^a \neq \alpha_j^b \\ 0, & otherwise \end{cases}$$

where $i < |\boldsymbol{a}|$, *and* $j < |\boldsymbol{b}|$.

$$\Delta\alpha(a,b) = \sum_{i=0}^{|a|-1} \sum_{j=0}^{|b|-1} \Delta\alpha(i,j)$$

Definition 5 *$(\Delta c(a,b))$* ***Number of Unaligned States of the Execution***
This equation describes in mathematical form the operations needed to be executed for the calculation of the number of unaligned states.

$$\Delta c(a,b) = \sum_{i=0}^{|a|-1} unalign_a(i) + \sum_{j=0}^{|b|-1} unalign_b(j)$$

Definition 6 *(d(a,b))* ***Distance***
Combining the predicate distance, the number of changes in actions of the execution and the number of unaligned states we get the final distance metric (d(a,b)):

$$d(a,b) = min_{align}(W_p \cdot \Delta p(a,b) + W_a \cdot \Delta\alpha(a,b) + W_c \cdot \Delta c(a,b))$$

W_p and W_c are the weights for the distance between the predicates and the distance between the number of unaligned states. (In our case, the weights W_p and W_c are initialized with values 1 and 2 respectively.)

Right after the alignment process is complete and the closest to the counterexample successful program execution has been found, we proceed to the presentation of the differences between the closest to the counterexample successful execution and the counterexample.

```
Control location deleted (step #5):
  10:  most = input2
  {most = [ $0 == input2 ]}
------------------------
Predicate changed (step #5):
  was:  most < least
  now:  least <= most
Predicate changed (step #5):
  was:  most < input3
  now:  input3 <= most
------------------------
Predicate changed (step #6):
  was:  most < least
  now:  least <= most
Action changed (step #6):
  was:  assertion_failure
------------------------
```

Figure 3.3: Presentation of the Differences between the counterexample of the program 1 and the closest to the counterexample successful run [5]

Figure 3.3 shows which statements were and which statements weren't executed in the successful execution. We observe that line 10 wasn't executed in the successful execution, which leads us to conclude that it is possibly the faulty code that we are looking for.

3.2.4 Control Flow Distance Metric

Equally this metric follows the same process that we saw in figure 3.1. However, this distance function is slightly different from the previous one, as it focuses more on the control flow of the program execution. The process remains the same, the only difference is the way of calculating the distance between executions. This distance function [9] is defined as follows.

Let π and π' be two executions of a program. We define the differences between π and π' as follows:

$$diff(\pi, \pi') = < e^{\pi}_{i_1}, ..., e^{\pi}_{i_k} >$$

1. each event **e** in $diff(\pi, \pi')$ is a branch event occurrence drawn from run π.

2. the events in $diff(\pi,\pi')$ appear in the same order as in π, that is, for all $1 \leq j < k, i_j < i_{j+1} (event\ e^{\pi}_{i_j}\ appears\ before\ event\ e^{\pi}_{i_{j+1}}\ in\ \pi)$

3. for each e in $diff(\pi,\pi')$, there exists another branch occurrence e′ in run π' such that align(e,e′)=true. Furthermore, the outcome of e in π is different from the outcome of e′ in π'.

4. all events in π satisfying criteria (1) and (2) are included in $diff(\pi,\pi')$.

As a special case, if execution runs π and π' have the same control flow, then we define $diff(\pi,\pi') = < e^{\pi}_0 >$

Definition 7 *(Comparison Of Differences)*
Let π,π',π'' be three execution runs of a program. Let

$$diff(\pi,\pi') =< e^{\pi}_{i_1}, e^{\pi}_{i_2}, ..., e^{\pi}_{i_n} > \text{ and } diff(\pi,\pi'') =< e^{\pi}_{j_1}, e^{\pi}_{j_2}, ..., e^{\pi}_{j_m} >$$

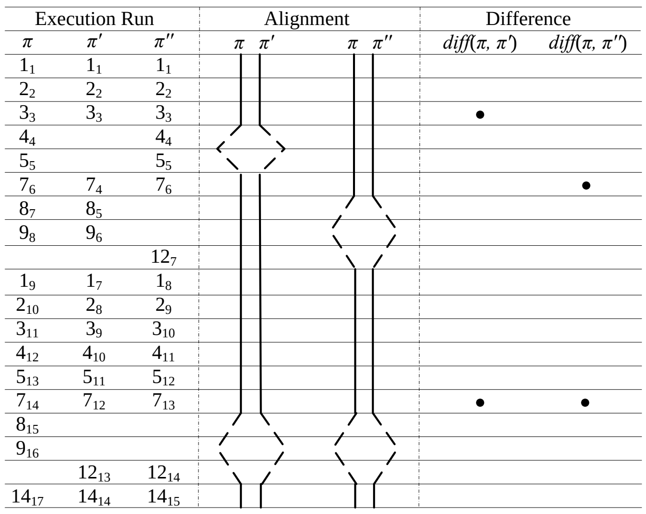
We define $diff(\pi,\pi')$ < diff(π,π'') iff there exists an integer $K \geq 0$ s.t.

1. *$K \leq m$ and $K \leq n$*
2. *the last K events in $diff(\pi,\pi')$ and diff(π,π'') are the same, that is, $\forall 0 \leq x < K$: $i_{n-x} = j_{m-x}$*
3. *one of the following two conditions holds*
   - *either $diff(\pi,\pi')$ is a suffix of diff(π,π''), that is $K = n < m$*
   - *or the (K+1)th event from the end in $diff(\pi,\pi')$ appears later in π as compared to the (K+1)th event from the end in diff(π,π''), that is $i_{n-K} > j_{m-K}$*

Given a failing run π and two successful runs π' and π'' we say that $diff(\pi,\pi') < diff(\pi,\pi'')$ based on a combination of the following criteria:

- Fewer branches of π need to be evaluated differently to get π' as compared to the number of branches of π that need to be evaluated differently to get π''. This is reflected in the condition K = n < m of Definition 7 (Comparison of Differences)
- The branches of π that need to be evaluated differently to get π' appear closer to the end (see Figure 3.4) of π (where the error is observed), as compared to the branches of π that need to be evaluated differently to get π''. This is reflected in the condition $i_{n-K} > j_{m-K}$ of Definition 7.

Figure 3.4 shows an example of a comparison of which execution, π' or π'', is closer to the counterexample π. The first column has the CFA nodes of the three executions. The second column shows the alignments between the execution π' and π as well as the alignments between the run π'' and the counterexample π. If two nodes are aligned with each other, then the line remains vertical (i.e. |). If not, then we use diagonal bars (/ or \) to symbolize that two nodes are not aligned with each other. The third and final column shows the differences of the executions to the counterexample. Each difference is symbolized with a dot. We notice that the two executions have both exactly two differences to the counterexample. Execution π' is not going through the CFA nodes 4_4 and 5_5. Thus, we see that the difference between the two executions lies in the CFA node 3_3, because the outgoing edge of this node that is part of the run π' is not the same as the outgoing edge of this node that belongs to the run π. Finally, we conclude that the execution π'' is closer to the counterexample π than the execution π' is. This happens because the branches of π that need to be evaluated differently to get π'' are closer to the end (to the error) than the ones to get π'.



Execution Run			Alignment		Difference	
π	π'	π''	π π'	π π''	*diff*(π, π')	*diff*(π, π'')
1_1	1_1	1_1				
2_2	2_2	2_2				
3_3	3_3	3_3			●	
4_4		4_4				
5_5		5_5				
7_6	7_4	7_6				●
8_7	8_5					
9_8	9_6					
		12_7				
1_9	1_7	1_8				
2_{10}	2_8	2_9				
3_{11}	3_9	3_{10}				
4_{12}	4_{10}	4_{11}				
5_{13}	5_{11}	5_{12}				
7_{14}	7_{12}	7_{13}			●	●
8_{15}						
9_{16}						
	12_{13}	12_{14}				
14_{17}	14_{14}	14_{15}				

Figure 3.4: Example of two different executions being compared to the counterexample [9]

After calculating correctly the aligned nodes and finding the closest to the counterexample successful execution, we present to the developer the differences that have to be made in order for the counterexample to become the closest successful execution that was found.

3.2.5 Path Generation Technique

Automated Path generation [15] is a fault-localization technique that produces automatically the closest, to the error, successful execution and it is based on the process of Control Flow Distance Metric. This technique has a slightly different approach than the two we defined before. When the number of successful executions is huge, then the computation of all the distances between every successful execution from the counterexample is going to take a lot of time and very likely lead to performance issues. The Path Generation technique helps us overcome this problem in case of a big amount of successful executions. Instead of computing for every safe path the aligned nodes and the distance to the counterexample, it builds the closest to the target path successful execution.

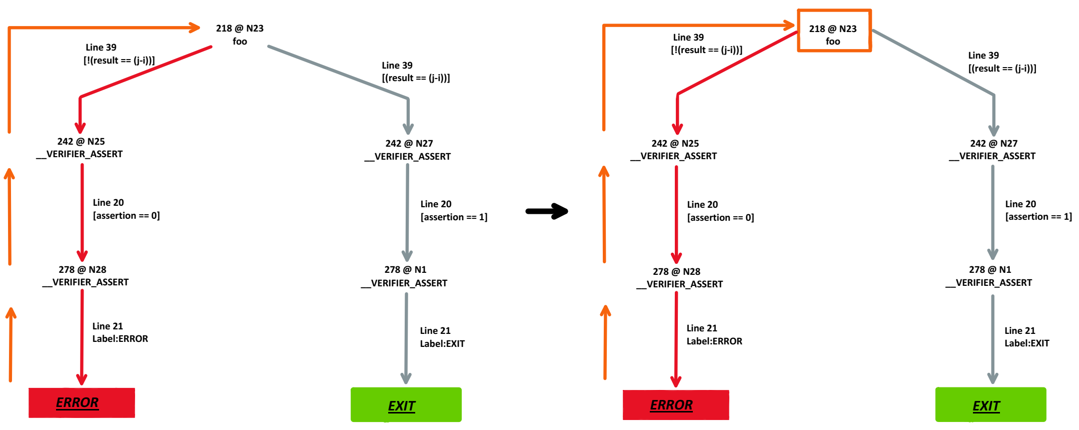


Figure 3.5: Searching for the last branch

In more detail, the algorithm starts from the target state (the control location that the error occurred) and goes backwards until it finds the first branch 3.5. When the branch changes its control flow, it is led to a feasible successful execution 3.6. Although this technique doesn't require either to compute all the successful runs or measure their distance form the counterexample, the presentation of the differences step remains the same.

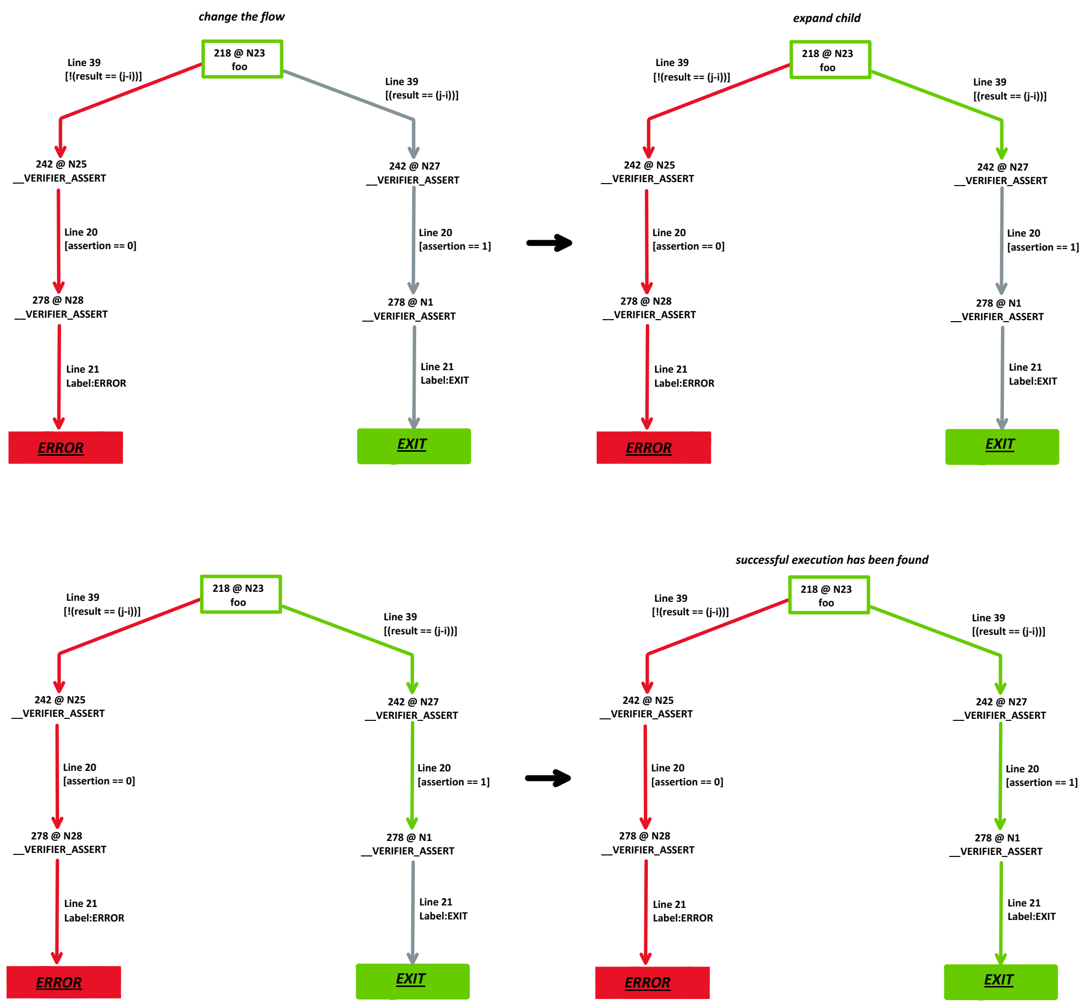


Figure 3.6: Search for possible successful runs

4 Theoretic Contributions

Adaptations of Abstract Distance Metric

In order to compute the final distance we have to compute:

1. Changes in predicates
2. Number of unaligned states of the executions

We define a new function, $contains_a(i, j, v)$, which controls for every alignment of the **i**-th state (from a) with **j**-th state (from b), if the v-th predicate of the state j is contained in the set of predicates that are part of the state **i** and equally $contains_b(j, i, v)$.

$$contains_a : \mathbb{N} \times \mathbb{N} \times \mathbb{N} \longrightarrow \{0, 1\}$$

$$contains_b : \mathbb{N} \times \mathbb{N} \times \mathbb{N} \longrightarrow \{0, 1\}$$

Furthermore, we define as $\Theta(a^i)$ and $\Theta(b^j)$ the set of predicates of the i-th state of the execution **a** and respectively, the set of predicates of the j-th state of the execution **b**.

Definition 8 *($\Delta p'(i,j,v)$ $\Delta p'(a,b)$) **Predicate Distance***

$$contains_a(i, j, v) = \begin{cases} 1 & if\ align(i, j) \wedge p_v(b^j) \notin \Theta(a^i) \\ 0 & otherwise \end{cases}$$

$$contains_b(j, i, v) = \begin{cases} 1 & if\ align(i, j) \wedge p_v(a^i) \notin \Theta(b^j) \\ 0 & otherwise \end{cases}$$

where $i < |a|$, $j < |b|$ and $v < |p_v(s_i^a)|$

$$\Delta p(a, b) = \sum_{i=0}^{|a|-1} \sum_{j=0}^{|b|-1} \sum_{v=0}^{|p(s_i^a)|-1} contains_b(i, j, v) + \sum_{j=0}^{|b|-1} \sum_{i=0}^{|a|-1} \sum_{v=0}^{|p(s_j^b)|-1} contains_a(j, i, v)$$

We notice that the definition above is slightly different from Definition 3 in section Abstract Distance Metric. Definition 3 checks if the v-th predicate of the i-th CFA node of the execution **a** is equal to the v-th predicate of the j-th CFA node of the execution **b**. Definition 8 has a slightly different approach to the calculation of the predicate distance. It checks for each execution if the v-th predicate of the i-th CFA node of execution **a** is contained in the set of predicates that were used in the j-th CFA node of the execution **b** and vice versa.

The equation below describes in mathematical form the operations needed to be executed for the calculation of the number of unaligned states.

Definition 9 *($\Delta c(a,b)$)* ***Number of Unaligned Nodes***

$$\Delta c(a,b) = \sum_{i=0}^{|a|-1} unalign_a(i) + \sum_{j=0}^{|b|-1} unalign_b(j)$$

Combining the two distances that we defined above, we get the final distance metric (d(a,b)):

Definition 10 *(DISTANCE, d(a,b))*

$$d(a,b) = min_{align}(W_p \cdot \Delta p(a,b) + W_c \cdot \Delta c(a,b))$$

W_p *and* W_c are the weights for the distance between the predicates and the distance between the number of unaligned states.
Finally, notice that the **changes in actions of the execution** are not a part of our final distance d(a,b). The changes in actions of an execution means that two aligned CFA nodes have different outgoing edges, which is already covered in the Δc(a,b), because if two CFA nodes have different outgoing edges then they are unaligned.

5 Implementation

This chapter describes in detail the implementation (in Java 11) of the three distance metrics that were implemented in CPAchecker.

5.1 Running Example

```
extern int __VERIFIER_nondet_uint();
extern void __VERIFIER_error();


void __VERIFIER_assert(int cond) {
  if (!(cond)) {
    ERROR: __VERIFIER_error();
  }
  return;
}

/* returns |i-j|, the absolute value of i minus j */
int foo (int i, int j) {
    int result;
    int k = 0;
    if (i <= j) {
        k = k+1;
    }
    if (k == 1 && i != j) {
        result = i-j; // error in the assignment : result = i-j instead
of result = j-i
    }
    else {
        result = i-j;
    }
    __VERIFIER_assert( (i<j && result==j-i) || (i>=j && result==i-j));
}


int main()
{

  foo(__VERIFIER_nondet_int(),__VERIFIER_nondet_int());
    return 0;
}
```

Figure 5.1: Example of an input program with a typical structure (6.1)

The program showed in Figure 5.1 consists of 3 methods:

1. __VERIFIER_assert(int cond)
2. $int foo(inti, intj)$
3. int main()

The first method checks if the specification holds. If not, then it throws an error. The second method is where the code that needs to be evaluated stands. The last one just calls the method `foo`.

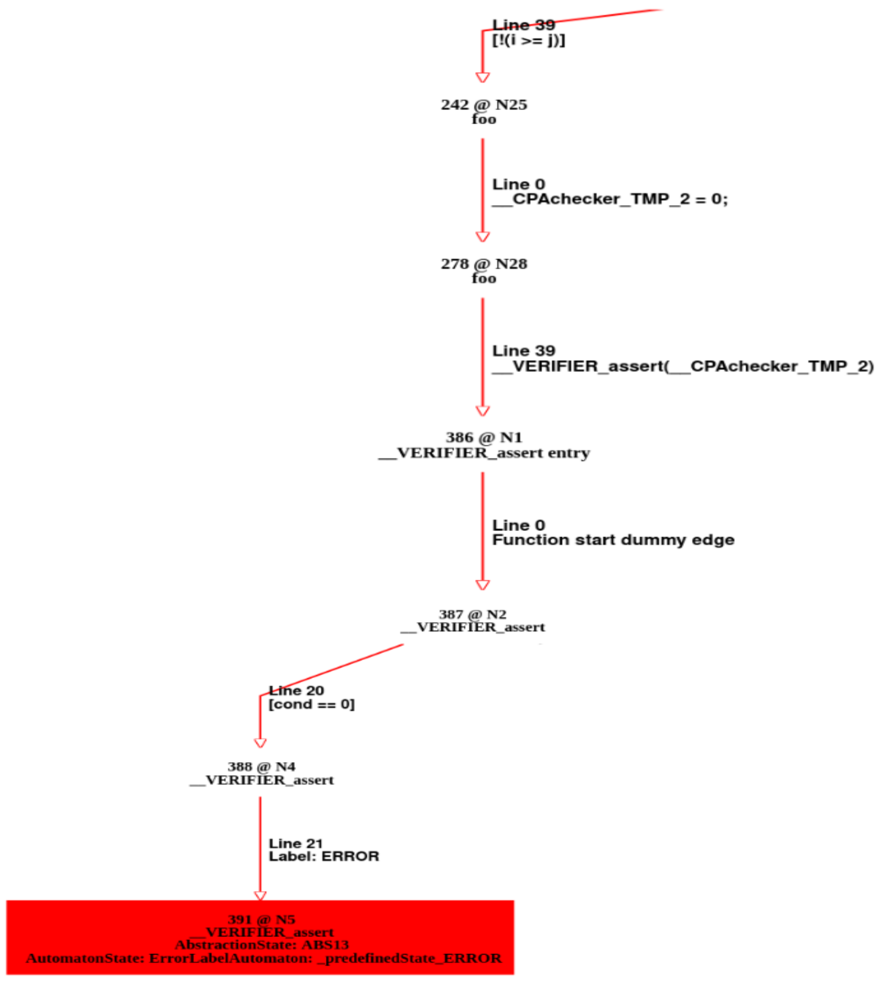


Figure 5.2: Counterexample in Abstract Reachability Graph

We notice that in Figure 5.2 the CFAEdge which represents line 39, between Nodes `N28` and `N1`, the method `__VERIFIER_assert(int cond)` is being called. The relevant information for the computation of the distance between two program executions lies before this control flow edge. For this purpose we have implemented an algorithm, `cleanPath(ARGPath)` as part of the class `DistanceCalculationHelper`, which takes as input a path and filters it so that only the relevant information remains in it. In other words, we go the path down, starting from the root, until we find the first CFA edge that is in the same line with the `__VERIFIER_assert`. We keep all the nodes and CFA edges from the root until that point and we then get rid of the rest because they are irrelevant for the localization of the fault.

Background Concept	Class Name
ARG state	ARGState
ARG path	ARGPath
CFA edge	CFAEdge
CFA node	CFANode
Assume edge	AssumeEdge
Statement edge	StatementEdge
Event	Event
W_p	predicateWeight
W_c	unalignedStatesWeight

Table 5.1: Assignment of Background Concepts to Class Names

5.2 Explainer

The Explainer is the main class of all three techniques. It implements the Interface Algorithm and is a subclass of the class NestingAlgorithm. Firstly it creates an instance of the CEGAR Algorithm. Then, CEGAR performs predicate analysis on the input program. After the successful execution of the predicate analysis we get a set with all the ARGStates that were reached during the analysis (i.e. reachedSet). The reachedSet is examined for any possible target states. If no target state is found then the explainer reports that the program does not violate the specification.

Find a Feasible Counterexample

If a target state is indeed found, then the explainer proceeds to find the complete ARGPath from the entry state until the state where the error (Target State) occurred. Thereupon, we retrieve the complete ARGPath (see Section 3.1.3) that leads to the target state, which we use as a counterexample.

Find all Successful Executions

The next step is to find all the feasible safe paths (the ARGPath's that do not end up to an Error state). This is achieved through the following process. First, we have to find all reached ARGStates that are not target states. Then, we implemented an algorithm `createPath` which accepts as input a set with all the safe ARGStates and builds a list with ARGPaths, which are all the successful program executions that could be built from the set of the reached states.

```
private List<ARGPath> createPath(StatesOnPathTo,root)
```

The idea behind this algorithm is that we create a list in which we are going to save all the safe paths that we build and we also create a wait-list that contains different lists with ARGStates (<List<ARGState>>) (see Algorithm 1). The purpose of this wait-list is to put inside all the not-expanded paths (In this particular algorithm we represent a path as a list of ARGStates) to get examined later. In the beginning, the wait-list contains only the entry state of the program, which is then expanded. By expanding a state we mean that the state has been searched for possible children. If the state has a single child, then the child is added in the to-be-built path as the last element and it is the next in line state to be examined. In case that an expanded state has more than one children, a new path (List<ARGState>) is created which contains the same states that the current expanded path contains but now the next child is added at the end of the new path and the whole path is added in the wait-list.

Algorithm 1: Case that the expanded Node has more than one children that need to be expanded

```
if children.size() > 1 then
    for every child in children do
        $newPath = copyOf(currentPath);$
        $newPath.addAtTheEnd(child);$
        $waitList.add(newPath);$
    end for
end if
```

The current expanded path just adds at the end the other children and continues until a state is found with no children (end state). The algorithm terminates when the wait-list is empty and there are no other paths to be processed.
When all the safe paths have been found, the explainer forwards the counterexample and the list of the safe paths to the distance function for the closest to the target path successful path to be found. The configuration option the user chooses is going to determine which distance function is going to get forwarded to.

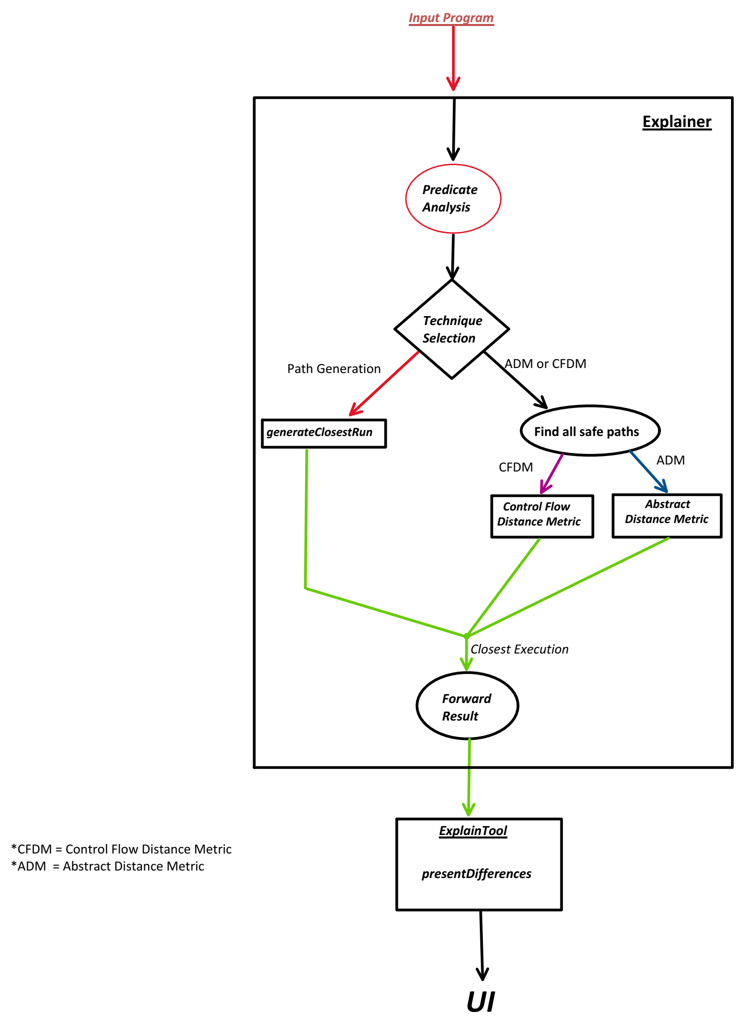


Figure 5.3: Flow of the fault-localization process using distance metrics in CPAchecker

5.3 Abstract Distance Metric (ADM)

This metric is a weighted distance function that consists of 2 sub-functions (see section 3.2.3 and chapter 4), the predicate distance, and the unaligned function (the number of unaligned states).

Once the list of the safe paths and the counterexample are passed to the distance metric, all of them go through the filter function (see Section 5.2). Once this process is complete, we shift our focus to the construction of the distance metric as described in chapter 4. We initialize the two weights, predicateWeight and unalignedStatesWeight here with values 1 and 2 respectively. After testing all the possible values for the weight, we found out that the optimal weights are 1 for `predicateWeight` and 2 for `unalignedStatesWeight`.

Create Alignments

However, the computation of the distance is not possible if there are no aligned states (section 3.2.2), so the next step is to find out which states from the successful executions are aligned with which states from the counterexample. For this purpose we have implemented a generic class called Alignment. We are going to use this class for both CFAEdges and later for ARGStates. It contains two different Lists of the same Type, one for the aligned elements of counterexample and the other for the aligned elements of the safe path.

Two edges are aligned with each other when those 2 conditions hold:

1. The predecessor CFANode of the first CFAEdge is the same with the predecessor CFANode of the second CFAEdge.
2. The successor CFANode of the first CFAEdge is different than the successor CFANode of the second CFAEdge.

When two CFAEdges can be aligned (see Definition 1) with each other we add them in the Alignment class as pair:

```
alignment.addPair(counterexampleEdge, safePathEdge)
```

The `Alignment` class guaranties that the size of the two lists in it, the aligned nodes of the counterexample and the aligned nodes of the successful execution, is always the same. For the calculation of the predicate distance, another alignment process has to take place but this time for ARGStates. It is almost the same as before, the only difference is that here we check only if the predecessor CFANode is the same for both CFAEdges.

Find the successful execution with the minimum distance

When the alignments are available, we proceed to calculate the distance between the safe path and the counterexample. First, we have to compute the predicate distance, then the number of the CFAEdges that are **not** aligned with any other CFAEdge, and put them together to construct the final distance (see 3.2.3 and 4).

Predicate Distance

We need to find all the predicates that are in the counterexample and all the predicates that are in the safe path. The predicates of the program execution, which are represented as a `BooleanFormula`, are coupled with `AND` and OR operators. To split this formula into the individual predicates, we implemented in the `DistanceCalculatorHelper` class a method, `splitPredicates(BooleanFormula)`, which takes as input the coupled BooleanFormula, uncouples it and returns a set with all the predicates that are in the program execution. This happens through iterative checking if the coupled `BooleanFormula` is coupled with an `AND` or an `OR` operator. For the first case, it calls the function `isConj(BooleanFormula)` and for the second the function `isDisj(BooleanFormula)`. If the formula is coupled with a conjunction in the front, then we call the `toConjunctionArgs(BooleanFormula, flatten=true)`, which is an internal method of the `BooleanFormulaManager` class. Correspondingly, if there is a disjunction in the front of the coupled `BooleanFormula` then we call the other method of the `BooleanFormulaManager`, called `toDisjunctionArgs(BooleanFormula, flatten=true)`. Those two methods unroll the predicate. When the predicate is completely uncoupled, a set with every single predicate is returned.

Finally, we are now able to calculate the predicate distance between the two executions the same way that we described in Theoretic Contribution, Definition 8.

Unaligned Edges

The second essential component that we need for the construction of the final distance is the number of CFAEdges that are not aligned. To compute this number we need to calculate the absolute of the subtraction of the aligned CFAEdges from the total number of CFAEdges.

Final Distance

At this point, we have calculated everything that we need to get the final distance. We put together the predicate distance multiplied by its weight and we add it with the number of unaligned edges multiplied by its weight also and we get our final result.

We do this process iteratively for every single safe path and we save our distances in a List called `distances`. After all the distances are computed we check the `distances` List for the smallest distance. However, we need to do one more step to get an accurate result.

Elimination of the zero distances

The elimination of the distances with value 0 is essential to the accuracy of the result. Safe paths that have zero differences from the counterexample are useless to us because the step no. 5 (Present the differences, Section 3.1) is going to give back nothing. As a result of that, if there are no differences to be evaluated, we are not in a position to find out, what went wrong in the failed program execution. For this reason, we erase all the safe paths that have distance 0 and then we search for the successful execution with the minimum distance from the counterexample and return it to the `Explainer`.

5.4 Control Flow Distance Metric (CFDM)

This distance function, as we already saw in Section 3.2.4, does not return a distance as an integer like the previous metric but as a List of Events (branches that were evaluated differently). Branches here are the CFAEdges of AssumeEdge Type.

The first step this algorithm does is the same as the previous one, i.e. filter the program executions so that we keep only the relevant information (Section 5.2). When this step is completed we proceed at the next one, which is to find all the branches of the program executions (counterexample and all the safe paths). Once this is done, the branches of each program execution are CFAEdges. For this distance function, we implemented a new Type, called `Event`. An `Event` consists of all the important information that is needed for the successful calculation of a distance between two program executions. In more detail, it has stored the CFAEdge (as `execution`) that was originally, the path that the CFAEdge belongs to, because we need it to calculate how far the CFAEdge from the end is (Definition 7) and the executions outgoing CFANode.

Create Alignments

Now we need to specify which events are going to be compared with which Event from the opposite path. This happens by comparing the CFANode of the two Events. Have they the same node number then they can be aligned to each other, otherwise not.

Find the successful execution with the minimum distance

As mentioned earlier the result of this metric is not a number but a list of Events. The next task in line is to calculate this list for all the successful program executions. For this purpose two things are controlled between the two aligned Events that need to be compared:

- Does the line of the execution of the first Event **equals** to the line of of the execution of the second ?
- Is the statement of the execution of the first Event **unequal** to the statement of the execution of the second ?

If both answers are yes, then the Event of the counterexample is added in the list of the events (i.e. the distance). This process is repeated for all the successful executions until we have a list of all the distances.

Elimination of zero distances

Before we proceed to compare the distances with each other to find the smallest, we have to eliminate all the zero-distances (see Section 5.4, Elimination of Zero Distances). The only difference here is that instead of looking for a distance with value 0, we are searching for a distance-list of Events with length 0.

Now, for finding out which distance is the smallest, we have implemented a function called `closestSuccessfulRun` that represents the definition of the distance metric as written in Section 3.2.4. It goes through the final list with all the distances and we compare each distance (i.e. list of Events) with each other. The comparison that takes place between two distances consists of checking (i) whether the Events of the first distance are closer to the end (target state) than the Events of the second distance and (ii) comparing the number of Events in the distance-list (Definition 7). When this process is over and we have successfully calculated the closest to the counterexample successful execution, the result is being returned to the `Explainer`.

5.5 Path Generation (PG)

For the implementation of this technique we used the `ControlFlowDistanceMetric` class, which is the class that also contains the previous metric, `Control Flow Distance Metric`, but it gets triggered with a different method - `generateClosestSuccessfulExecution`. However, unlike the previous two techniques, here we do not need all the successful executions but only the counterexample. The reason for this is that we are going to generate the successful execution which is closest to the counterexample. This means that we skip in `Explainer` the process of finding all the safe paths.

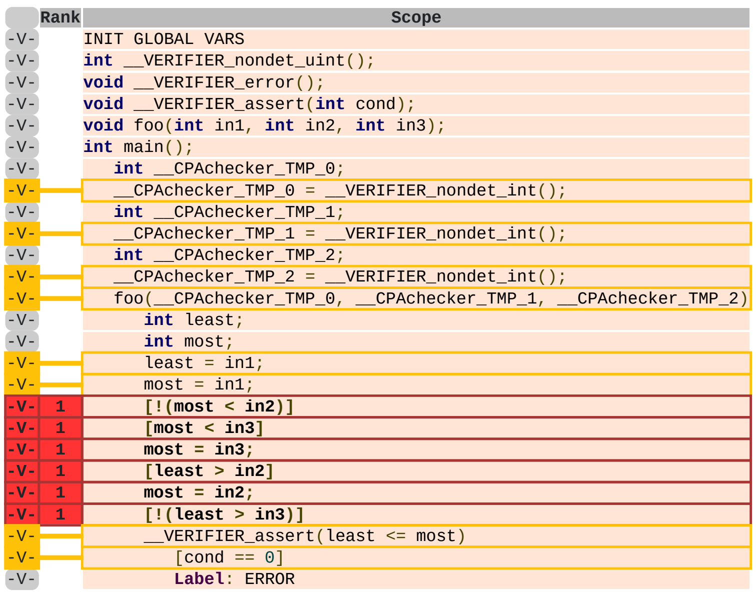
After finding all the control flow branches, like at the beginning of the previous section, we are going backwards through those branches, searching for the first branch that when evaluated differently it leads to an error-free end node. This is achieved by calling the `createPaths` (5.2) method that was also used in the `Explainer` to find all safe paths. The difference here is that we give as starting point the branch that we changed its flow. Thus, here we have to construct much fewer paths than for the ADM and CFDM in `Explainer`. It is possible that the method returns more than one successful execution. After the construction of the safe paths is complete, we return those paths in a list. If, however, no successful execution can be found from that control-flow branch, then we repeat the same process using the next control-flow branch that is closer to the error of the counterexample

If more than one successful execution was generated, then we use the Control Flow Distance Metric to examine which one of those paths is closest to the counterexample. On the other hand, if there is only one generated successful execution, then we forward the constructed safe path and the counterexample to the `ExplainTool` to present the differences to the user.

5.6 Presentation Of The Differences

When the final successful execution has been found and returned to the explainer, the explainer uses the ExplainTool to find the differences between the failed and the successful program execution. The ExplainTool examines if an execution that has been made in the successful run has also been made in the failed run. If not, then it is a difference and a possible bug. The same process is repeated for the executions of the counterexample. Once this procedure is complete, we use an already existing infrastructure, FaultLocalizationInfo, to forward the information to the graphical user interface of the CPAchecker.

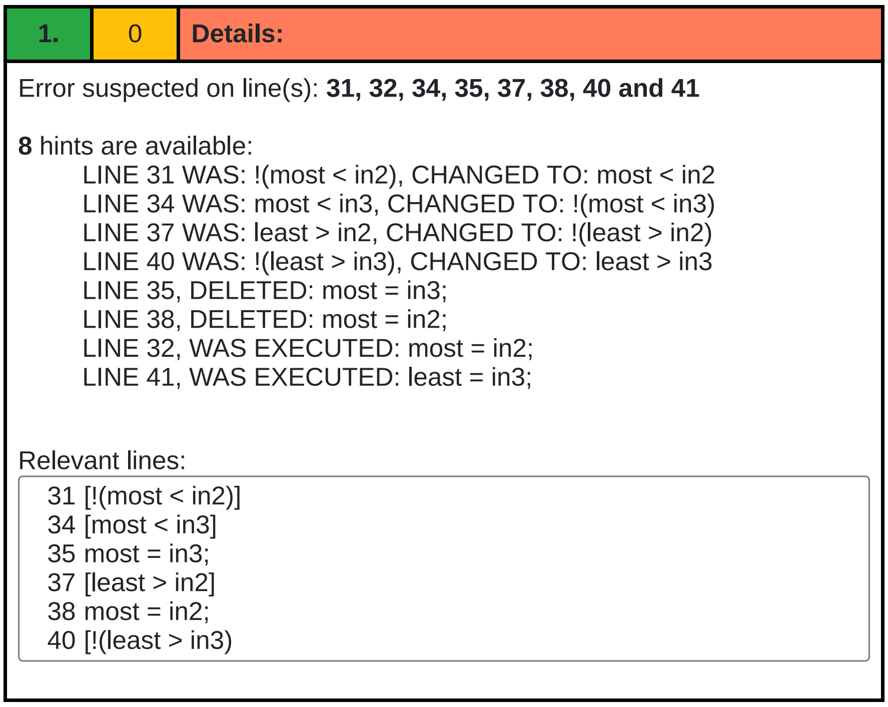
First, the developer has a brief overview of all the suspicious statements that are marked with red (5.4). Pressing the "Change View" button shows a table with all the changes that had to be made in the counterexample to become a successful execution. The changes look like Figure 3.3 that we saw in the Background. The developer scrolls down the table and investigates each suspicious statement.



	Rank	Scope
-V-		INIT GLOBAL VARS
-V-		int __VERIFIER_nondet_uint();
-V-		void __VERIFIER_error();
-V-		void __VERIFIER_assert(int cond);
-V-		void foo(int in1, int in2, int in3);
-V-		int main();
-V-		int __CPAchecker_TMP_0;
-V-		__CPAchecker_TMP_0 = __VERIFIER_nondet_int();
-V-		int __CPAchecker_TMP_1;
-V-		__CPAchecker_TMP_1 = __VERIFIER_nondet_int();
-V-		int __CPAchecker_TMP_2;
-V-		__CPAchecker_TMP_2 = __VERIFIER_nondet_int();
-V-		foo(__CPAchecker_TMP_0, __CPAchecker_TMP_1, __CPAchecker_TMP_2)
-V-		int least;
-V-		int most;
-V-		least = in1;
-V-		most = in1;
-V-	1	[!(most < in2)]
-V-	1	[most < in3]
-V-	1	most = in3;
-V-	1	[least > in2]
-V-	1	most = in2;
-V-	1	[!(least > in3)]
-V-		__VERIFIER_assert(least <= most)
-V-		[cond == 0]
-V-		Label: ERROR

Figure 5.4: Overview of the possible faults (marked with red) of the minmax.c (1)

Figure 5.5 shows a list **Hints** that are useful for tracking the faulty code of the **minmax.c** algorithm. We observe that the **6th Hint** in our result says that this piece of code was executed in the counterexample but not in the successful execution. This code is the faulty statement that we are looking for.



1. 0 **Details:**

Error suspected on line(s): **31, 32, 34, 35, 37, 38, 40 and 41**

8 hints are available:

```
LINE 31 WAS: !(most < in2), CHANGED TO: most < in2
LINE 34 WAS: most < in3, CHANGED TO: !(most < in3)
LINE 37 WAS: least > in2, CHANGED TO: !(least > in2)
LINE 40 WAS: !(least > in3), CHANGED TO: least > in3
LINE 35, DELETED: most = in3;
LINE 38, DELETED: most = in2;
LINE 32, WAS EXECUTED: most = in2;
LINE 41, WAS EXECUTED: least = in3;
```

Relevant lines:

```
31 [!(most < in2)]
34 [most < in3]
35 most = in3;
37 [least > in2]
38 most = in2;
40 [!(least > in3)
```

Figure 5.5: Successful localization of the bug

5.7 Options

The implemented in CPAchecker techniques that we researched are executable, if when starting the CPAchecker the following configuration is been set:

```
-explainer
```

The above configuration lets the CPAchecker know that the tool will be using the `Explainer` class to analyze the inputted program. Furthermore, we also have to declare what distance metric we would like to use for the localization of the fault. We do that by adding in our command line `-setprop` following by the metric that we want to use.

```
-setprop explainer.distanceMetric=ADM
-setprop explainer.distanceMetric=CFDM
-setprop explainer.distanceMetric=PG
```

6 Evaluation

6.1 Experimental Setup

In this chapter, the three techniques are evaluated for speed and accuracy. For the evaluation, we used a computer with 8 GB RAM DDR4, and an Intel Core i7 7-th Generation CPU. The operating system that was used is Linux Ubuntu 20.04 LTS. 34 different test programs, benchmark and a part of sv-benchmark [1] provided by Bekkouche [2], were used to test each technique in detail and the same hardware and operating system were used for all verification runs to make sure that the results are as precise as possible. Furthermore, the CPU-time limit was 900 seconds and no memory limit was set. For a reliable benchmarking we used BenchExec 2.3. Benchexec allows the individual isolation of each verification run, and thus reduces the risk of measurement errors. Furthermore, the test programs that we used contain two types of different faults. **Assignment errors** (`y = y-3;` should be `y = y-1;`) and **Conditional Errors** (`if (y % 2 == 1)` should be `if (y % 2 == 0)`).

Table 6.1: Explanation of the errors based on their type

Conditional Error	Assignment Error
wrong check **if** block	wrong method assignment
wrong number of iterations in **for**-loop	wrong assignment in nested **if**
wrong nested **if** in for-loop	wrong array assignment in **for**-loop
wrong check of arrays index in **for**-loop	array assignment in method
	unnecessary assignment in **while**-loop
	wrong assignment in **while**-loop

We notice in the table 6.1 that there are conditional and assignment errors inside and outside of loops, as part of extra called methods and inside of nested if blocks.

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

6.2 Ranking Function

In this section, we introduce a ranking function which we are going to use to compare the accuracy of those three techniques with each other.

The ranking function Rank accepts as input the lines of code that the program has, the number of differences that were reported back as a result of the used technique, and the overall success rate of the technique which we will see in the next section.

Our ranking function is defined as follows:

$$Rank\colon \mathtt{[0,1]} \times \mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{R}$$

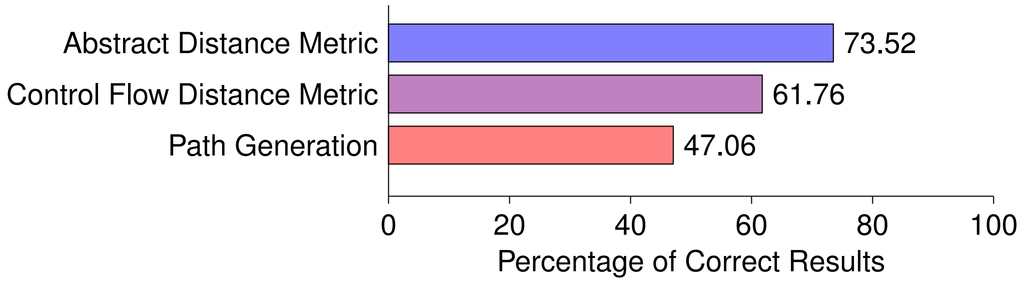
$$\boxed{Rank(\mathtt{successRate},\mathtt{Hotlines},\mathtt{Differences}) = \mathtt{successRate} * \frac{\mathtt{Hotlines}}{\mathtt{Differences}}}$$

The three input parameters of this function are all equally important for the successful comparison of the three techniques with each other. `successRate` is the overall success rate of the technique, `Hotlines` is the number of lines of code that the input program has and finally the `Differences` is the number of differences that the counterexample has from the closest successful execution. The main characteristic of those three techniques is that the result is a set of differences between the counterexample and the closest successful execution. This is directly related to the number of lines of code that the input program is. The more lines of code that the input program has, the more increases the number of differences between the successful and failed execution. Finally, the overall success rate of each technique plays also a major role, because it indicates the possibility of finding the location of the bug.

The result of the function is a real number. The higher the result, the more successful the technique. The lower the result, the less effective the technique. For example, a technique with Rank 999 is better than another technique with Rank 100 for the same input program. This is because the technique with 999 rank has a higher `Hotlines/Differences` ratio, which means that for the same program this technique has fewer differences than the other, thus the developer is going to investigate fewer suspicious statements which is better explanation.

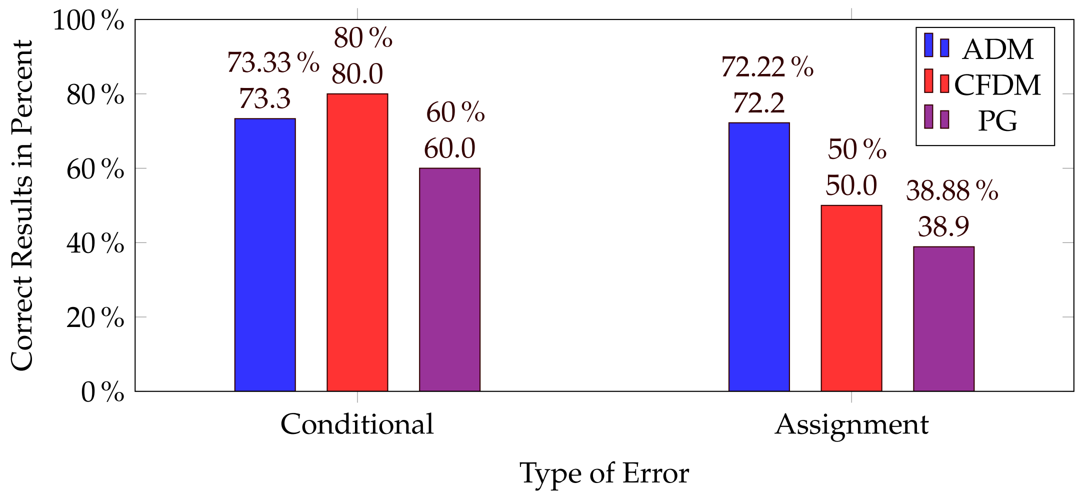
6.3 Quantitative Analysis

Overall Results



In this subsection, we see a brief overview of the results of the three techniques that we implemented into CPAChecker. In the above chart, we can see that the Abstract Distance Metric is the most promising with 73.52 % of the faults found. The next most efficient technique is the Control Flow Distance Metric with an overall result of 61.76 %. Last comes the Path Generation Technique, which has a significant difference in the results comparing with the other two. The Path Generation technique has located correctly the 47.06% of the faults, ruffly 14% less than the control flow distance metric.

Precision



Listing 6.1: Percentage bar chart of the success rate of each technique based on the type of error

The chart 6.1 illustrates how many conditional and how many assignment errors were located correctly by the three techniques. We observe that the Control Flow Distance Metric is the most promising for the localization of conditional errors and the Abstract

Distance Metric for the localization of the assignment errors. Abstract Distance Metric located 73.3 % of the conditional errors in a code, a success rate 6,67 % less than the Control Flow Distance Metric. The path generation technique comes last finding 60 % of the conditional errors.

On the right side of the chart, we notice that the Abstract Distance Metric has the lead with 72.22 % of the assignment errors found. We notice that the Control Flow Distance Metric has a success rate of 50 % here. It is 30 % less efficient than with the conditional errors. This result was expected as the main focus of this technique is the control flow of the program. This difference is much bigger than the one in Abstract Distance Metric, which is only 1.1 %. Finally, the path generation technique is the less promising technique out of all three, locating 60 % of the conditional errors and 38.88 % of the assignment errors.

6.4 Ranking

In this section, we compare the results of the three techniques with each other. For this purpose we are going to use the `Rank` function which we defined in Section **6.1**.

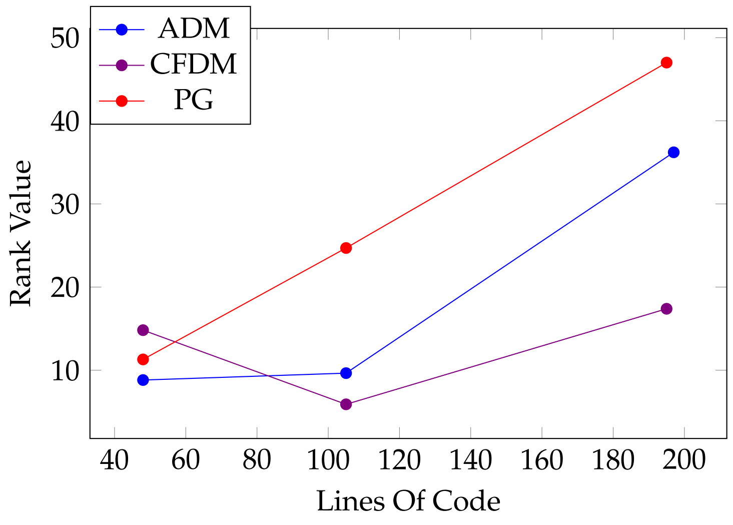


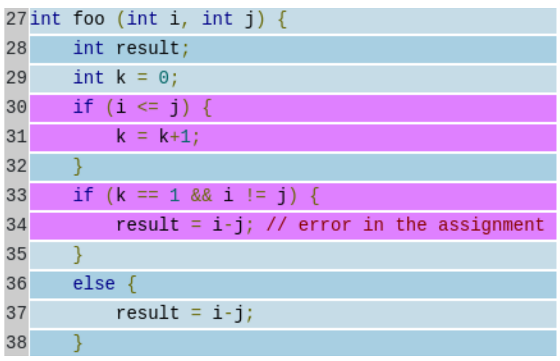
Figure 6.1: Ranking value of the three techniques

To compare, we used different programs with various number of lines. As we can see in the plot 6.1 the Path Generation technique has the highest-ranking out of all three of the programs with more than 100 hotlines. Taking a look at table 6.3 we observe that the Path Generation Technique has only 2 differences for all programs, regardless of

Table 6.2: Rank of Technique per No. of Hotlines

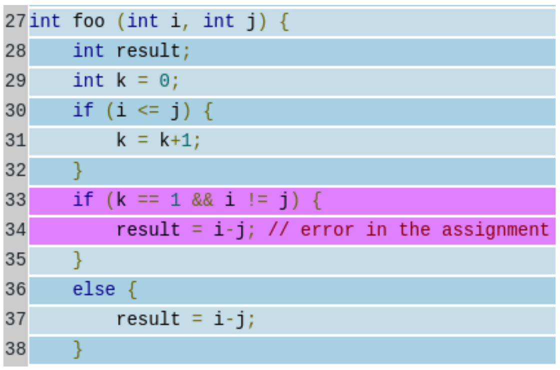
Hotlines	ADM	CFDM	PG
48	8.82	14.82	11.3
105	9.65	5.9	24.7
197	36.21	17.39	47

the number of lines of code. This means that the bug is exactly the difference between the counterexample and the successful run. On the other hand, we notice in table 6.2 that the Abstract Distance Metric for a program with 197 Hotlines ranks at **36.21**, which is much better than the **17.39** that ranks the Control Flow Distance Metric. However, it is less than the **47** that ranks the Path Generation technique. This is due to that the Abstract Distance Metrics (see Figure 6.2) reports 4 possible differences, which requires the developer to investigate double as much suspicious lines of code than with the (see Figure 6.3) Path Generation technique. Furthermore, we observe that the Control Flow Distance Metric for programs with more than 100 lines of code ranks less than the rest. This lies in the fact that the Control Distance Metric reports the biggest number of differences between the successful execution and the error path. However, for a program with fewer than 50 lines of code, the Control Flow Distance Metric has the lead with **14.82**, following by the Path Generation and Abstract Distance Metric with **11.3** and **8.82** respectively.



```
int foo (int i, int j) {
    int result;
    int k = 0;
    if (i <= j) {
        k = k+1;
    }
    if (k == 1 && i != j) {
        result = i-j; // error in the assignment
    }
    else {
        result = i-j;
    }
```

Figure 6.2: Possible faults using ADM



```
int foo (int i, int j) {
    int result;
    int k = 0;
    if (i <= j) {
        k = k+1;
    }
    if (k == 1 && i != j) {
        result = i-j; // error in the assignment
    }
    else {
        result = i-j;
    }
```

Figure 6.3: Possible faults using PG

Hotlines	ADM	CFDM	PG
48	4	2	2
105	8	11	2
197	4	7	2

Table 6.3: Average number of differences based on No. of hotlines

6.5 Runtime

Speed is also relevant for the evaluation of these three techniques. In the table below we can see the CPU time needed for each technique until a result is delivered.

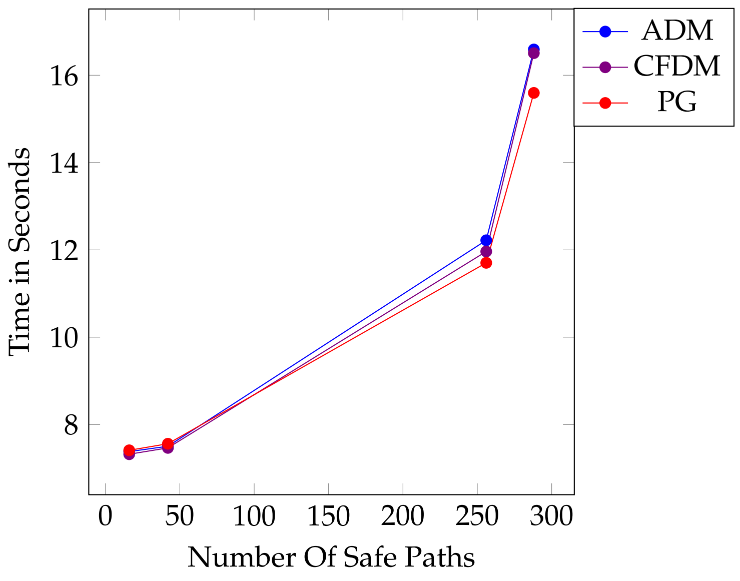


Figure 6.4: Runtime in seconds per Technique until a result has been found with maximum 300 safe paths

The plot 6.4 illustrates the seconds needed for each technique to complete an execution based on the number of possible candidates for the result. We observe that all three techniques need almost the same amount of time to find a result. For a small number of successful runs, 16-50, the path generation technique is the slowest of all. The Control Flow distance Metric is faster out of all three for that amount of safe paths. Furthermore, for 256 different successful executions, the Abstract Distance Metric is only 0.25 seconds slower than the Control Flow Distance Metric, while the Path Generation technique has the lead, needing only **11.70** seconds to locate the error. Finally, the gap is getting bigger between the Path Generation technique and the two distance metrics as the number of safe paths grows. For 288 safe paths, the Path Generation located the error in almost 1 second faster than the Control Flow Distance Metric and the Abstract Distance Metric.

In the plot 6.5, we continue the time comparison between the three techniques but only this time with a much larger number of successful executions. We notice the difference that makes the alignment and the computation of all safe paths in the time needed to find a result.

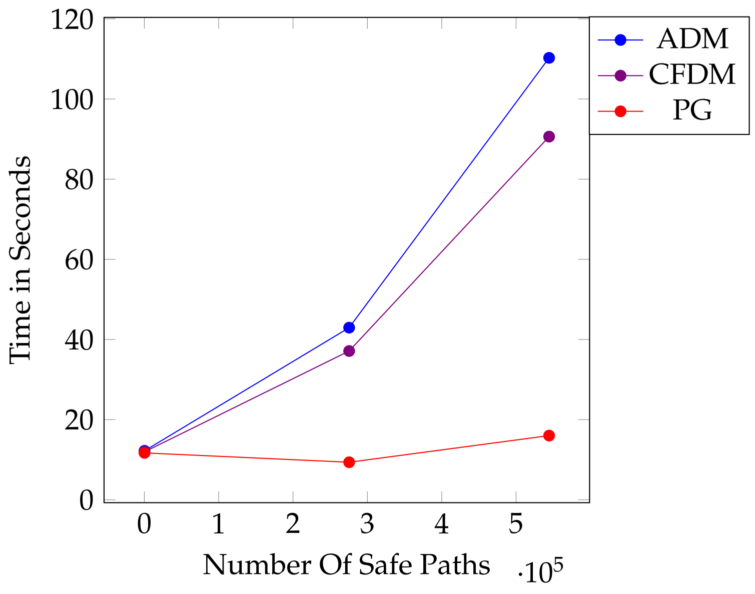


Figure 6.5: Runtime in seconds per Technique until a result has been found with at least 300 safe paths

The Path Generation technique maintains its runtime between **7-16** seconds. On the other hand, for more than 275470 possible for the solution candidates (safe paths) the Abstract Distance Metric and the Control Flow Distance Metric are much slower than the Path Generation technique. The Abstract Distance Metric is the slowest out of all three, requiring **42.95** and **110.27** for a program execution with 275470 and 544420 safe paths respectively. We notice that the Control Flow Distance Metric is a bit faster than the Abstract Distance Metric for a large number of safe paths. Finally, the Path Generation technique is the fastest out of all three. The reason is that the Path Generation technique doesn't require to find all the successful executions and then make alignments between them and the counterexample. This gives to the Path Generation technique a big advantage in the time needed to find the closest successful execution.

Table 6.4: Average Time in Seconds needed based on the number of safe paths, exact values of the plots 6.4 and 6.5

No. of Safe Paths	ADM	CFDM	PG
16	7.38	7.32	7.41
42	7.50	7.46	7.56
256	12.22	11.96	11.70
288	16.59	16.50	15.59
275470	42.95	37.12	9.37
544420	110.27	90.61	16.02

Tables 6.5 and 6.6 in the next page contain in detail how many seconds does each technique need to find a result based on the number of successful executions.

Table 6.5: CPU Time in Seconds per Benchmark task based on the number of safe paths, 1st Part

Benchmark Name	ADM	CFDM	PG	No. of safe paths
Minmax	7.38	7.33	7.41	16
AbsMinus1	7.52	7.56	7.28	42
AbsMinus2	7.41	7.49	7.78	42
AbsMinus3	7.56	7.54	7.64	42
AbsMinus4	7.53	7.29	7.60	42
MaxLoop	16.82	15.84	15.34	256
MaxLoop2	10.08	10.12	10.09	256
MaxLoop3	9.80	9.93	9.68	256
MaxMin	38.07	36.61	9.05	275470
MaxMin2	44.360	36.21	9.10	275470
MaxMin3	45.87	35.97	9.18	275470
MaxMin4	43.53	39.74	10.17	275470
MiddleNumber	9.49	8.10	7.85	36
MiddleNumber1	8.00	8.01	8.21	36
MiddleNumber2	7.92	8.03	7.98	36
TriPerimetreKO	100.55	86.07	10.59	544420
TriPerimetreKO2	114.55	87.19	10.95	544420
TriPerimetreKO3	115.40	84.31	10.36	544420
TriPerimetreKOV2	123.61	88.22	10.84	544420

Table 6.6: CPU Time in Seconds per Benchmark task based on the number of safe paths 2nd Part

Benchmark Name	ADM	CFDM	PG	No. of safe paths
TriTypeKO	102.50	95.82	10.46	544420
TriTypeKO2	126.06	86.02	11.75	544420
TriTypeKO2V2	106.79	88.70	10.66	544420
TriTypeKO3	108.24	99.03	10.53	544420
TriTypeKO4	105.45	88.52	15.54	544420
TriTypeKO5	112.66	85.66	15.03	544420
arrays_mul_init	20.15	19.35	19.30	8
brs	19.24	19.47	19.33	2
partial_lesser_bound	9.10	9.06	9.89	1
sanfoundary_24-1	21.91	21.36	22.14	4
while_infinite_loop_1	6.54	7.17	7.31	0
array-1	8.02	7.97	7.83	4
sum01_gub02	8.62	8.16	8.58	32
gj2007b	16.59	16.51	15.60	288
jm2006	8.26	8.05	7.91	2

7 Future Work and Conclusion

Future Work

There are numerous interesting avenues for future research. In this work, we conducted our research on distance metrics that use alignments and predicate abstraction. However, other metrics exist that are SSA-Index oriented. A combination of an alignment oriented and a SSA-Index oriented metric would be an interesting topic for research. Each metric has its pros and cons, however, combined they could produce a better explanation for the error. Furthermore, combining distance metrics with another fault-localization technique would be an interesting topic. The distance metric would be passing to the other fault-localization technique the closest to the counterexample successful execution and the other technique would investigate the path for the faulty statement and reporting back which line of code is more likely spurious.

Conclusion

This research aimed to identify how effective can distance metrics be for fault-localization purposes. Based on a qualitative and run-time analysis of the two different distance metrics and the path generation technique, it can be concluded that using a distance function to track down the faulty statement can be a great help to the developer. It may not always report the exact location of the bug, however, it proposes changes that when they are made, the code may not lead to an error anymore. The results indicate that the faulty code is often in the set of changes that the fault-localization technique suggests being made. The results of the Path Generation technique show that when the fault is contained in the set of differences between the closest to the counterexample successful execution and the error path, then the set of differences is exactly the fault. However, most of the time the Abstract Distance Metric and the Control Flow Distance Metric are more likely to contain in the set of differences the fault than the Path Generation technique. According to the results, the predicates of the CFA nodes and the control-flow edges of the program play a major role in finding the successful execution that will provide us with the correct information about the fault.

Bibliography

[1] A. Assaraf. This is what your developers are doing 75% of the time, and this is the cost you pay, 2020.

[2] D. Beyer, S. Gulwani, and D. A. Schmidt. Combining model checking and data-flow analysis. In E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, editors, *Handbook of Model Checking*, pages 493–540. Springer, 2018.

[3] D. Beyer, T. A. Henzinger, and G. Théoduloz. Configurable software verification: Concretizing the convergence of model checking and program analysis. In W. Damm and H. Hermanns, editors, *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, volume 4590 of *Lecture Notes in Computer Science*, pages 504–518. Springer, 2007.

[4] D. Beyer and M. E. Keremoglu. Cpachecker: A tool for configurable software verification. In G. Gopalakrishnan and S. Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 184–190. Springer, 2011.

[5] S. Chaki, A. Groce, and O. Strichman. Explaining abstract counterexamples. In R. N. Taylor and M. B. Dwyer, editors, *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2004, Newport Beach, CA, USA, October 31 - November 6, 2004*, pages 73–82. ACM, 2004.

[6] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In E. A. Emerson and A. P. Sistla, editors, *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2000.

[7] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, June 22-25, 1997, Proceedings*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer, 1997.

[8] A. Groce. Error explanation with distance metrics. In K. Jensen and A. Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 10th*

International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings, volume 2988 of *Lecture Notes in Computer Science*, pages 108–122. Springer, 2004.

[9] L. Guo, A. Roychoudhury, and T. Wang. Accurately choosing execution runs for software fault localization. In A. Mycroft and A. Zeller, editors, *Compiler Construction, 15th International Conference, CC 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 30-31, 2006, Proceedings*, volume 3923 of *Lecture Notes in Computer Science*, pages 80–95. Springer, 2006.

[10] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In D. F. Redmiles, T. Ellman, and A. Zisman, editors, *20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005), November 7-11, 2005, Long Beach, CA, USA*, pages 273–282. ACM, 2005.

[11] S. Löwe and P. Wendler. Cpachecker with adjustable predicate analysis - (competition contribution). In C. Flanagan and B. König, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 18th International Conference, TACAS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, volume 7214 of *Lecture Notes in Computer Science*, pages 528–530. Springer, 2012.

[12] T. Margaria and B. Steffen, editors. *Leveraging Applications of Formal Methods, Verification and Validation. Verification - 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part II*, volume 11245 of *Lecture Notes in Computer Science*. Springer, 2018.

[13] L. Mariani and X. Zhang, editors. *Proceedings of the International Workshop on Dynamic Analysis: held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2011), WODA 2011, Toronto, ON, Canada, July 18, 2011*. ACM, 2011.

[14] R. E. Overill. Book review: "time warps, string edits, and macromolecules: the theory and practice of sequence comparison" by david sankoff and joseph kruskal. *J. Log. Comput.*, 11(2):356, 2001.

[15] T. Wang and A. Roychoudhury. Automated path generation for software fault localization. In D. F. Redmiles, T. Ellman, and A. Zisman, editors, *20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005), November 7-11, 2005, Long Beach, CA, USA*, pages 347–351. ACM, 2005.

[16] A. Zakari, S. P. Lee, and I. A. T. Hashem. A single fault localization technique based on failed test input. volume 3-4, page 100008, 2019.