Formula-based Fault Localization for Imperative Programs with Multiple Faults

Si-Mohamed LAMRAOUI

Doctor of Philosophy

Department of Informatics
School of Multidisciplinary Sciences
SOKENDAI (The Graduate University for Advanced Studies)

Formula-based Fault Localization for Imperative Programs with Multiple Faults

複数欠陥のある命令型プログラムを対象とした論理式ベース欠陥箇所特定方式に関する研究

A dissertation presented

by

Si-Mohamed LAMRAOUI

to

The Department of Informatics

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

in the subject of

Formal Verification & Software Engineering

SOKENDAI (The Graduate University for Advanced Studies)

Tokyo, Japan

2016

Thesis advisor
Shin NAKAJIMA

Author
Si-Mohamed LAMRAOUI

Formula-based Fault Localization for Imperative Programs with Multiple Faults

Abstract

Program debugging is a trial and error process of finding and eliminating bugs or defects in a computer program, and thus making it behave as expected. As software and hardware systems grow in complexity, debugging techniques for ensuring their correctness are increasingly important. Manual debugging is tedious, time-consuming and error-prone. Thus, making debugging automatically has been one of the major research topics in automated software engineering. Automatic formal verification of programs, such as the bounded model-checking (BMC) method, is useful for checking if a program exhibits erroneous behavior or not. Identifying root causes, which are the fundamental reasons for the occurrence of failing program executions, still involves manual inspection and therefore needs a vast amount of human efforts. This calls for a new method that performs fault localization automatically.

Automatic fault localization of imperative programs is a well-known problem and has been studied from various approaches. Back in the early 1980's, program slicing was introduced. A few years later model-based debugging (MBD) was presented. MBD combines the slicing method with the Reiter's model-based diagnosis theory framework. Thereafter, a work proposed to replace in MBD the algorithmic method for calculating program slices by a method following the Boolean satisfiability problem. In hardware debugging, more specifically in very-large-scale integration (VLSI) and system on chip (SoC) designs, an alternative method for localizing fault was shown effective. The method uses a debugging formulation based on maximum satisfiability (MaxSAT), which is a promising approach to the fault localization tool for imperative programs.

This thesis introduces and studies a new automatic fault localization method, which is formula-based fault localization for imperative programs written in `ANSI C`.

The presented method combines the MBD with MaxSAT, specifically with partial maximum satisfiability. In contrast to other work on fault localization of imperative programs, we focus in this thesis on the localization of faults in multi-fault programs. Fault localization of multi-fault programs is a problem of great importance since real-world programs often have more than one fault. We demonstrate in this thesis that the fault localization of multi-fault programs requires further considerations to be successful. Dealing with multi-fault programs implies that faults may be spread in different program execution paths and that fault localization reports contain information from different faults. Therefore, it is required to use many program failing inputs in order to cover faults as much as possible. Since more than one failing execution is considered, it implies that the complexity of the problem increases and thus it is necessary to have an efficient method to localize all faults in an acceptable amount of time. Moreover, generated fault localization reports have to be processed so that the software engineers spend less time in a posteriori root causes inspection. Here are the main contributions of this thesis. First, we reformulate the problem of formula-based fault localization systematically from a theoretical viewpoint. Second, we introduce new methods for encoding imperative programs into trace formulas. The way programs are encoded has a significant impact on the precision and efficiency of the root causes identification procedure. Third, we present an efficient method to calculate and combine root causes obtained from different failing executions. Fourth, all the methods are implemented in a tool, SNIPER. Several experiments are conducted on SNIPER to show the capabilities of the presented approach.

This thesis is organized as follows. Chapter 2 presents backgrounds of the fault localization of imperative programs. Chapter 3 introduces a series of concepts and terminologies that are needed to define the formula-based fault localization problem. These concepts and terminologies are used in the other chapters. Chapter 4 details the architecture of the tool SNIPER. The implementation of SNIPER, which is based on the LLVM compiler infrastructure and the Yices 1 partial maximum satisfiability solver, is detailed in Annex A. SNIPER is a basis on top of which we implement the different trace formulas presented in Chapters 5 and 7 and the algorithm of Chapter 6. In each of these chapters, we use SNIPER to empirically study the presented methods.

Chapter 5 introduces a method for encoding programs, the full flow-sensitive trace formula (FFTF), which is equivalent to the control flow graph of the target program. The FFTF with appropriate algorithms is successful in localizing root causes in multi-fault programs for at least two of the benchmarks we used. However, although the FFTF is expressive, it is not efficient in view of computing time. In the Chapters 6 and 7 we present methods to deal with this problem. Chapter 6 presents a fault localization algorithm, which enumerates minimal correction subsets (MCS) in an incremental fashion. We show on a benchmark that the computing time can be reduced with this algorithm. Chapter 7 introduces an alternative method for encoding programs, the hardened flow-sensitive trace formula (HFTF). We empirically show on two benchmarks that the use of the HFTF, as compared to the FFTF, makes the fault localization algorithm produce less spurious root causes and perform faster. The HFTF is shown to be as expressive as the FFTF for most programs. Finally, Chapter 8 summarizes the contributions of this thesis and presents a list of future work on formula-based fault localization of imperative programs.

Contents

List of Figures

List of Tables

Listings

Acknowledgments

My time as a graduate student at SOKENDAI and NII has been supported by many kind people. This page attempts to recognize those who have been most helpful along the way.

Firstly, I would like to express my sincere gratitude to my advisor Prof. Shin Nakajima for the continuous support of my Ph.D study and related research, for his patience, motivation, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis. I could not have imagined having a better advisor and mentor for my Ph.D study.

Besides my advisor, I would like to thank the rest of my thesis committee: Prof. Hiroshi Hosobe, Prof. Kozo Okano, Prof. Ichiro Satoh, and Prof. Tomohiro Yoneda, for their insightful comments and encouragement, but also for the hard question which incented me to widen my research from various perspectives.

I thank my fellow labmates in for the stimulating discussions and for all the fun we have had in the last three years.

Last but not the least, I would like to thank my family: my parents and to my brothers and sister for supporting me throughout writing this thesis and my life in general.

"Do, or do not. There is no try."
– Yoda

Chapter 1

Introduction

As software-intensive systems constitute the social infrastructure to support our daily life, achieving the required reliability levels is a major concern in software engineering. In 1972, E.W. Dijkstra [27] discussed in his Turing awards lecture the importance of the correct by construction (CbyC) style development of programs. The presentation is a driving force of a wide variety of work in formal methods, especially in the European research community. The idea is to construct correct programs from their initial formal specifications in a stepwise refinement manner. Correctness of the refinement is ensured in each step, and thus the resultant programs are correct. E.W. Dijkstra, actually, had a negative opinion on the effectiveness of a posteriori verification of programs. Although the idea of CbyC is scientifically sound, it is not adapted in practice. Constructing programs or programming is more or less a human process, requiring much experience and insight. The reliability of programs is also a matter of human activities.

Introducing some form of automation to a posteriori verification of programs is an alternative approach for achieving the required reliability levels. These methods adapt mathematical logic as their scientific bases. Software artifacts such as design specifications or programs have their representations encoded as logic formulas. Correctness of programs with respect to the given specifications is a process of mathematical reasoning. This can be automated when formulas are in decidable fragments of first-order theory. In particular, with the advent of the efficient Boolean satisfia-

bility (SAT) technology [74], SAT-based methods occupy one of the central themes in automated software engineering. These are applying SAT to automatic test-case generations or bounded model checking of programs [20, 21, 41, 51, 66]. They are effective in checking whether the target program is faulty or not. Identifying root causes of the faulty behavior still needs a vast amount of human efforts. Localizing root causes automatically is now an important research challenge.

Automatic fault localization is formulated as a problem of mathematical logic. R. Reiter [77] introduced the model based diagnosis (MBD) theory. In the MBD theory, the model of the artifact is encoded in a formula of some suitable logic. The model together with a given specification is unsatisfiable if the artifact is faulty. The problem is to find a set of clauses in the artifact model that are responsible for this unsatisfiability. If the formula is encoded in Boolean, it is exactly an instance of Boolean unsatisfiability problem (UNSAT). As the complexity of UNSAT is coNP-complete, M.H. Liffiton et al. [59] turns it into a dual problem of maximum satisfiability (MaxSAT) whose complexity is NP-complete [12]. R. Reiter's framework and M.H. Liffiton's work are successful in the fault localization of VLSI circuit designs [80] together with efficient MaxSAT algorithms. Similar methods are employed in the fault localization of imperative programs, but they are limited to programs with a single fault in them [47, 95]. There is a need for a new method to identify multiple faults in a program.

This thesis presents a new automatic fault localization method for programs that have multiple faults in them. The method adapts a modest assumption on the failure model of imperative `ANSI C` programs. The key observation is the way to encode all the potential failing execution paths in a logic formula, called trace formula (TF). The TF encodes the failure model as well as those potential failing execution paths, and thus has much impact on the preciseness and efficiency of localizing faults. The proposed method is implemented in a tool, SNIPER. In some experiments on typical benchmark problems, SNIPER successfully identifies all the injected, both single and multiple faults. These experiments show that the assumption of the failure model is adequate at least for the benchmark problems used, and the fault localization is efficient.

The organization of the thesis is as follows. Chapter 2 presents backgrounds of the fault localization of imperative programs. Chapter 3 introduces a series of concepts and terminologies that are needed to define the formula-based fault localization problem. These concepts and terminologies are used in the other chapters. Chapter 4 details the architecture of the tool SNIPER. The implementation of SNIPER, which is based on the LLVM compiler infrastructure and the Yices 1 partial maximum satisfiability solver, is detailed in Annex A. SNIPER is a basis on top of which we implement the different trace formulas presented in Chapter 5 and 7 and the algorithm of Chapter 6. In each of these chapters, we use SNIPER to empirically study the presented methods. Chapter 5 introduces a method for encoding programs, the full flow-sensitive trace formula (FFTF), which is equivalent to the program's control flow graph. The FFTF with appropriated algorithms is successful in localizing root causes in multi-fault programs for at least two of the benchmarks we used. However, because the FFTF is expressive, it is not efficient in view of computing time. In the Chapter 6 and 7 we present methods to deal with this problem. Chapter 6 presents a fault localization algorithm, which enumerates minimal correction subsets (MCS) in an incremental fashion. We show on a benchmark that the computing time can be reduced with this algorithm. Chapter 7 introduces an alternative method for encoding programs, the hardened flow-sensitive trace formula (HFTF). We empirically show on two benchmarks that the use of the HFTF, as compared to the FFTF, makes the fault localization algorithm produce less spurious root causes and perform faster. Finally, Chapter 8 summarizes the contributions of this thesis and presents a list of future work on formula-based fault localization of imperative programs.

Chapter 2

Backgrounds

This chapter introduces backgrounds of the automatic fault localization of imperative programs, and summarizes related work.

2.1 Introduction

In the world of software development, software reliability [73] is of a great importance, however, it is rather immature in general. "Software reliability is the probability of failure-free operation of a computer program for a specified time in a specified environment [71]". E.W. Dijkstra believed that program reliability would be achieved by finding a way to avoid writing programs with bugs from the very beginning [27]. Such a way would make the programming process cost effective since software engineers would not waste their time in debugging programs. Correct by construction (CbyC) style development is one such approach. For example, the verification method [64] helps in assisting the development of software and hardware systems. The program is refined in a stepwise manner, starting from its formal specification, and ending with actual code. Therefore, the correctness at each refinement step is preserved. However, formal specifications are difficult to apply in practice because of their very limited expressiveness compared to the Natural language. It requires a big effort to use and understand the notation for software engineers.

Even though the idea of CbyC was attracting, posteriori verification became a

standard in recent years. The focus changed from product over process and the goal is now to detect faults after the code is developed. Thus was born the activity of program debugging, which aims at improving software reliability.

2.2 Program Debugging

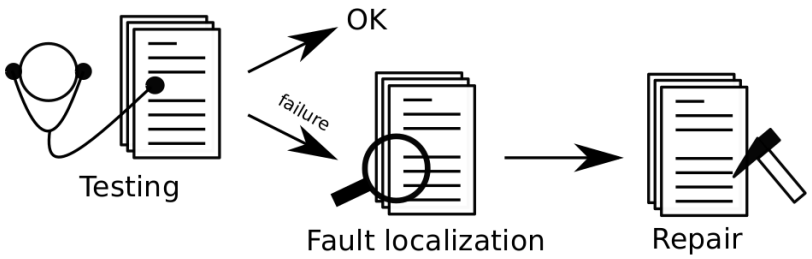


Figure 2.1: A typical debugging process

Program debugging is a complex activity that involves several tasks. As illustrated in Figure 2.1, debugging starts by *testing* a target program to see if it is faulty or not. Testing can be done manually or automatically. There exist various automatic techniques, including program testing or automatic formal verification. Once the testing phase is finished and in the case the target program exhibits inappropriate behavior (i.e., failures), the next task consists of identifying the reasons of the failures. This task is called *fault localization*. Fault localization can be time-consuming and difficult especially when done manually. Once the reasons of the failures are successfully identified and their root causes are localized, they need to be corrected. *Fault repair* aims at repairing the faults previously identified. The process of debugging ends when the target program behaves as expected.

Three of the debugging tasks are closely related and can strongly influence each other. A good testing result is important for making the fault localization task easy to perform. Similarly, fault reports generated by the fault localization task must contain enough information to facilitate and speed up the fault repair task.

In the present thesis, we focus on the task of *automatic fault localization*. In the following sections, we present existing approaches to tackle the problem of fault

localization in imperative programs.

2.3 Program Slicing

Program slicing [88, 89, 90] is a technique for analyzing imperative programs, which can be used in debugging to locate root causes of errors. Program slicing consists of finding the instructions of a program that can affect a variable v at a line l, a slicing criterion. The subset of instructions obtained from the whole program is called a *slice*. A slice is defined with respect to a *slicing criterion*. If the slicing criterion refers to a violation of an assertion, then the obtained slice is a subset of program statements that directly affect the assertion violation. Thus, the slice contains root causes. Further, we sometimes say a failing slicing when a slice of code contains faults. Similarly, a slice of code that does not contain faults is called a successful slice.

Program slicing for localizing faults was empirically shown effective [52]. The average code size reduction (CSR) of program slices is around 30% [13]; such amount of program code needs to be inspected manually to find real root causes.

2.3.1 Static Slicing

According to the original definition of slicing [89], a static slice of a program P is all statements of P that may affect the value taken by a variable v at a point p of the program. This slice is defined for a slicing criterion $C = (s, V)$, where s is a statement of the program P and V is a subset of variables in P. Computing a static slice consists of finding successive sets of statements that are indirectly pertinent, in accordance with data and control dependencies.

2.3.2 Dynamic Slicing

Dynamic slicing is similar to static slicing but instead it uses information obtained at execution time. A dynamic slice contains all statements that actually affect the value of a variable at a program point for a particular program execution. A static

slice would contain all statements that may have affected the value of a variable at a program point for any program execution.

With dynamic slicing the produced slices are usually smaller than with static slicing. This is because static slicing methods do not make any assumptions regarding program inputs. Whereas a dynamic slice contains only the program statements that are dependent on specific program inputs.

2.4 Coverage-based Fault Localization

Coverage-based or spectrum-based debugging [45, 46, 78, 91] calculates ranking orders between program statements or spectrums to show that a particular fragment of code is more suspicious than the others. The method needs many successful and failing executions to calculate the statistical measures. The method basically relies on program testing technique, and needs input-output test data. Generating unbiased test data set is a major challenge.

The method consists of collecting data while executing the program in order to determine the potential root cause locations. The data can be collected at different spectrum, such as: line, block, function, class, or packet.

Example

	Test 1	**Test 2**	**Test 3**	**Test 4**	e_f	e_p	n_f	n_p
Line 1	Executed		Executed	Executed	2	1	0	1
Line 2	Executed	Executed		Executed	1	2	1	0
Line 3		Executed		Executed	1	1	1	1
...								
Test Results	Success	Success	Failed	Failed				

Table 2.1: An example of data collected in coverage-based methods

In the example of Table 2.1, the Line 1 is executed by the Tests 1, 3, and 4. Two of these tests failed (e_f) and one succeeded (e_p). All the failed tests were executed

on this line (n_f) and one successful test does not execute this line (n_p). The Line 2 is executed by Tests 1, 2, and 4. One of these tests failed (e_f) and two succeeded (e_p). One failed test does not execute this line (n_f) and all the successful tests execute this line (n_p).

Metrics

Metrics allow to calculate a Hamming distance between code entities (line, block, function, class) and a bug. There exists many different metrics. The entities can be sorted depending on the distance, the nearest ones from the bug contain the most probable the bug, and the farthest ones are probably not causing the bug. Based on this sorting, software engineers can find faster the source of the error than a manual search.

Tarantula

Tarantula is a metric but also a tool introduced by Jones et al. [45]. Tarantula is representative of the coverage-based approach. It was developed in order to provide a standalone tool for the visualization of program lines of code. The studied lines of code are colored depending on their suspiciousness. In addition to the color, varying from green for not suspected lines to red for suspected lines, a shade shows the number of time the lines where effectively executed by the test suite. These two information are combined in order to produce a display that shows the degree of suspiciousness associated to each line.

Tarantula can be greatly affected by the test data used, and this is also true for all methods following the coverage-based approach. In the experiment sections of Chapter 5 and Chapter 7 we show two cases on which Tarantula is unable to find the faults. The reason is that Tarantula is using statistics to compute the root cause locations. Depending on the provided test data (test-cases) but also the types of faults in the target program, the method may or may not find the faults. We discuss the types of faults in more details in Section 2.8.

2.5 Counterexample Minimization

Model checking methods are used to automatically check whether a target artifact is faulty or not. When the target artifact is shown to be faulty, a model checking methods outputs a counterexample. It requires a great amount of manual efforts to identify root causes by studying the generated counterexamples. Counterexample minimization [6, 16, 37, 38] is introduced to help this task.

Counterexample minimization methods are used together with logic model checking. Fault localization methods based on counterexample minimization attempts to reduce the number of irrelevant part of a counterexample. Informally, the error localization is a problem of finding fragments of the program that appear in the error traces, but not in the successful executions. This observation leads to the automated methods to calculate how the error traces are different from the successful ones [6, 16, 37, 38]. Alternatively, some work views the problem as minimizing the error traces, namely to find a set of minimal lengths in the traces to result in the failing executions [17, 47].

2.6 Model-based Diagnosis

The model-based diagnosis (MBD) theory [36, 77] establishes a logical formalism of the fault localization problem. The *model* is presented as a formula expressed in *suitable* logic. The formula is unsatisfiable if it represents both an artifact and a correctness criteria, with the former violating the latter. The MBD theory distinguishes conflicts and diagnoses. Conflicts are the erroneous situations similar to failing static slicing (see Section 2.3.1), and are represented by minimal unsatisfiable subsets (MUSes) of the unsatisfiable formula. Diagnoses are the fault locations to be identified and are minimal correction subsets (MCSes). The MBD theory states that MUSes and MCSes are connected by the hitting set relationship. Therefore, the problem is to enumerate either all MUSes or all MCSes. Such sets can be calculated automatically if the formula is represented in decidable fragments of first-order theory, for example, as a Boolean formula. If the formula is encoded in Boolean, the problem

becomes a Boolean unsatisfiability problem (UNSAT). Note that the complexity of UNSAT is coNP-complete [12].

The MBD methods, including the model-based debugging [95], first calculate MUSes and then obtain MCSes. There are several ways to calculate the information equivalent to MUSes. An early work [93] used graph-based algorithms to compute a static slice of programs in order to obtain MUSes. Later, MUSes were obtained by calculating irreducible infeasible subsets of constraints [95].

2.7 Formula-based Fault Localization

Formula-based fault localization method combines the SAT-based formal verification techniques [74] with the model-based diagnosis (MBD) theory [77]. This method was first employed in the fault localization of VLSI circuits [80] as an alternative approach to obtaining the minimal correction subset (MCSes). The method reduces the fault localization problem to maximum satisfiability of unsatisfiable formulas in propositional logic and calculates maximal satisfiable subsets (MSSes). An MCS is the complement of MSS [60]. Therefore, the problem is turned into a dual problem of maximum satisfiability (MaxSAT) whose complexity is NP-complete [12].

This idea was applied to the fault localization problem of imperative programs [47] and implemented in a tool, BugAssist. The algorithm of BugAssist, however, does not guarantee the enumeration of all the MSSes. It may miss some faults, especially in programs with multiple-faults.

In summary, the formula-based approach is more systematic than the methods that use program slices (Section 2.3) or the statistical coverages (Section 2.4). It has its logical foundation developed in the MBD theory (Section 2.6).

2.8 Program Fault Types

In this section we explain the notion of faults in imperative programs. We provide insights on the different types of faults and on their characteristics.

In debugging of imperative programs, bugs or defects can be divided into the following categories.

1. Syntax: invalid sequence of characters or tokens,
2. Data Size: arithmetic overflow or underflow,
3. Sanity: buffer overflow, null pointer, division by zero,
4. Application Features .

The formula-based fault localization focuses on faults related to (4) application features. Other categories of defects are not considered here. Application features are usually specified by assertions in the code, or by a design-by-contract (DbC) style specification [41, 67, 68, 69] (pre- and post-conditions, and invariants).

Application feature faults can be further divided into two categories. The first category refers to faults in the calculations. Typically, these faults occur when the software engineer makes a mistake in using a comparison/arithmetic operator. The second category refers to faults in the structure of the program. The structure, also called skeleton, is usually defined by the way program basic blocks are interconnected to each other. In some situations, this skeleton is incorrect and thus making the program not behave as expected. Automatic fault localization methods, introduced so far (Section 2.2 to 2.6), consider *calculation faults* only. Herein we also assume that the structure or skeleton of the target program is correct, and we focus on calculation faults.

Some faults are seeded in existing lines of code that were not properly written. Other faults are caused by missing fragments of code (omissions). The latter is especially difficult to deal with because we have to localize a part of non-existing code fragments. In this thesis we focus on the former and do not consider fault due to omissions.

Now, we will discuss in what kind of programs our method will look for root causes of faults. The classification of imperative programs is made according to the number of faults they are infected with. The following two sections details the types of programs to be considered.

2.8.1 Single-fault Programs

A single-fault program is a program that contains only one fault. The fault is either a wrong comparison operator, for example in an if-statement, or a wrong operator in an arithmetic operation.

Usually, to debug a single-fault program, a single error-inducing input is enough to find out root causes of the fault. This error-inducing input triggers a failing path on which the fault is executed. All the fault localization methods, mentioned in Section 2.3 to 2.7, are effective for such single fault-programs. However, usually, we may have many different error-inducing inputs. Imagine that we have a test suite consisting of many test cases and a target program that has a single fault. There are some cases where some of the test cases fail on the program. The other test cases are passed. This happens if the particular fault can be reached in many different execution paths. The coverage-based fault localization methods, in particular, rely on the existence of such test cases.

Example

We illustrate the problem of fault localization on single-fault programs with an example in Listing 2.1. This program is supposed to compute two values to operate a motor. The first value (`d`) is meant for giving the order to torque, turn right, or turn left to the motor. The second value is a positive or null number representing the rotation degree. However, there is an error in this program, which is located in line 11. The comparison `(d==1)` should be `(d==2)`. Because of this error, when putting as argument to the procedure `rotate` a non-null number, the `assert` in line 16 fails because the value of `r` is negative. Hence, an error-inducing input equal to 1 or -1 can be used to trigger a failing execution.

```
1 void rotate(int degree) {
2   int d;
3   if (degree==0) {
4     d = 0; // torque
5   } else if (degree>0) {
6     d = 1; // rotate right
7   } else {
8     d = 2; // rotate left
9   }
10  int r;
11  if (d==1) {
12    r = degree * -1;
13  } else {
14    r = degree;
15  }
16  assert(0 <= d <= 2 && r >= 0);
17  opMotor(r, d);
18 }
```

Listing 2.1: A single-fault program

2.8.2 Multi-fault Programs

Dealing with programs with multiple faults is one of the important issues in automated fault localization methods. As mentioned before, most of the current fault localization approaches focus on single-fault programs and are not effective for multi-fault programs. We, here, focus on the coverage-based methods. Coverage-based debugging methods are unable to locate multiple faults simultaneously. This was empirically studied by DiGiuseppe et al. [26]. They showed that the presence of multiple faults caused interferences, which inhibits the effectiveness of the method. It is, however, true that at least one fault can be localized. Denmat et al. state that the coverage-based technique Tarantula [46] makes implicit hypotheses requiring independence of multiple faults (every failure is caused exclusively by a single fault) and when these hypotheses do not hold, the technique does not provide "good

result" [25]. Zheng et al. assert that traditional coverage-based technique "cannot distinguish between useful bug predictor and predicates that are secondary manifestations of bugs" [99].

The MBD theory [77] generally considers the multiple fault cases. For simplicity, consider a case where a set M of MUSes is extracted from an unsatisfiable formula and each MUS in M refers to a particular error, a single fault. Then, M may contain, in principle, many conflicts because many elements (clauses) are included. MCSes, calculated using the minimal hitting set of MUSes, contain elements (clauses) representing multiple faults.

In order to study the characteristics of multiple faults in detail, we classify the types of faults that can be found in multi-fault programs in three categories:

- Data flow-dependent faults
- Control-dependent faults
- Independent faults

Figure 2.2 depicts an example of control flow graph (CFG) for each of the above types of faults. Informally, a faulty statement Y is data flow-dependent on a faulty statement X if the result of Y is dependent on X. A faulty statement Y is control-dependent on a faulty branch condition X of a conditional branch statement if the outcome of X determines whether Y should be executed or not. In a special case of the control-dependent faults, the fault X may hide the fault Y, which means that it is impossible to generate a test case that executes the fault Y and detects the existence of Y. We call such special case, *nested faults*. Lastly, a faulty statement Y is independent of a faulty statement X if Y and X are neither control-dependent nor data flow-dependent.

In some cases, multiple faults can lie in one program path. In such a situation, a single test input is enough to localize all the faults. In some cases of data flow-dependent faults or control-dependent faults, the use of full flow-sensitive TF can sometimes locate different faults with a single test input. We will discuss this later in regard to the Bekkouche's Benchmark (Section 5.6.1).

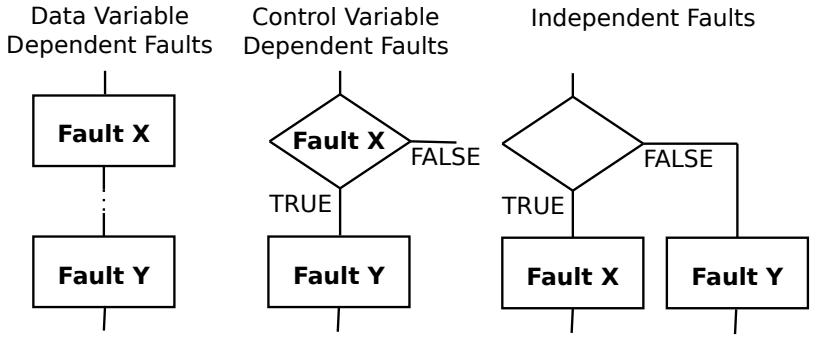


Figure 2.2: Fault types found in imperative programs

When faults are independent, it usually requires to have multiple test inputs in order to successfully identify all faults. Additionally, if faults are localized in different failing paths, a method to combine the results obtained from these paths is needed in order to show to the programmers that there are more than one fault to look at in the faulty programs. A basic MCS enumeration method that uses a single failing test case only is not sufficient when dealing with multi-fault programs whose faults are spread in different program paths or execution paths. In Chapter 4 we introduce an MCS enumeration method and a combination method that are, when used together, suited to tackle this special case of multi-fault programs.

Example

We illustrate the problem of multi-fault program with an example shown in Listing 2.2. This program contains two faults. In line 4 the variable `y` should be set to 42 and in line 6 it should be set to 0. We can find two failing paths, one that goes through the line 4 with the value 1 as argument, and the other that goes through the line 6 with the value 0 as argument. We are faced with two problems with this kind of program. First, we need to take into account both failing paths to localize all the faults. Considering only one path is not sufficient. Second, the quantity of faults in the program may affect the precision of the fault localization because the cause of an erroneous situation can be due to one or many faults acting together. An accurate localization implies an high complexity of analysis.

```
1 void foo1(int x) {
2     int y;
3     if(x>0) {
4         y = 1;  // Fault X
5     } else {
6         y = 42;  // Fault Y
7     }
8     assert((x<=0 && y==0) || (x>0 && y==42));
9 }
```

Listing 2.2: A multi-fault program infected by independent faults

Chapter 3

Preliminaries

This chapter introduces a series of concepts and terminologies, which are mandatory for understanding the remaining part.

3.1 Boolean Satisfiability Problem

Boolean satisfiability problem (SAT problem) is a decision problem defined with logic formulas. It is, given a formula in propositional logic, to decide if this formula has a *model*, meaning that if there exists an assignment of the propositional variables making the formula true.

3.1.1 Basic Definitions

Clause

A clause is a disjunction of literals. In propositional logic, a clause takes the form

$$(l_1 \vee ... \vee l_n)$$

where l_i are literals; an atomic proposition (p) or a negation of an atomic proposition ($\neg p$).

Conjunctive Normal Form

A formula in conjunctive normal form (CNF) is a normalization of a logical expression, which is a conjunction of clauses.

3.1.2 Satisfiability and Unsatisfiability

Satisfiability and validity are elementary concepts of semantics. A formula is *satisfiable* if it is possible to find an interpretation (model) that makes the formula true. A formula is *valid* if for all interpretations the formula is true. Dual concepts are the unsatisfiability and the non-validity. A formula is *unsatisfiable* if none of its interpretations makes the formula true, and *non-valid* if it exists an interpretation that makes the formula false.

The four concepts can be applied to theories. A theory is a set of sentences in a formal language. For example, a first-order theory is a set of first-order sentences. A theory is satisfiable (valid) if one (or all) interpretations make each axiom of the theory true, and the theory is unsatisfiable (non-valid) if all (one) interpretation make each axiom of the theory false.

It is also possible to consider only interpretations that make all of the axioms of another theory true. This generalization is commonly called *satisfiability modulo theories* (see Section 3.2 for details).

The question whether a sentence in propositional logic is satisfiable is a decidable problem, which is a question in some formal system with a yes-or-no answer, depending on the values of some input parameters. In general, the question whether sentences in first-order logic are satisfiable is not decidable.

3.1.3 Maximum Satisfiability

The maximum satisfiability (MaxSAT) problem is the problem of determining the maximum number of clauses, of a given Boolean formula in CNF, that can be made true by an assignment of truth values to the variables of the formula. It is a generalization of the Boolean satisfiability problem, which asks whether there exists

a truth assignment that makes all clauses true.

For example, the CNF formula

$$\varphi = (a \vee b) \wedge (a \vee \neg b) \wedge (\neg a \vee b) \wedge (\neg a \vee \neg b)$$

is not satisfiable, which means that no matter which truth values are assigned to its two propositional variables, at least one of its four clauses will be false. However, it is possible to assign truth values in such a way as to make three out of four clauses true; indeed, every truth assignment will do this. Therefore, if this formula is given as an instance of the MaxSAT problem, the solution to the problem contains three clauses.

3.1.4 Partial Maximum Satisfiability

In the partial maximum satisfiability (pMaxSAT) problem for a CNF formula, some clauses are declared to be *soft*, or relaxable, and the rest are declared to be *hard*, or non-relaxable. The problem is to find an assignment that satisfies all the hard clauses and the maximum number of soft clauses.

For example, consider again the example CNF formula in the previous section, but this time with its clauses assigned as soft or hard:

$$\varphi_{partial} = \underbrace{(a \vee b) \wedge (a \vee \neg b) \wedge (\neg a \vee b)}_{soft} \wedge \underbrace{(\neg a \vee \neg b)}_{hard}$$

One solution is to retract the clause $(a \vee b)$. We then obtain the following assignment: $a = \mathit{true}$ and $b = \mathit{false}$.

3.2 Satisfiability Modulo Theories

The satisfiability modulo theories (SMT) [29] problem is a decision problem for logical formulas, with respect to combinations of background theories expressed in decidable subclass of first-order logic with equality. It is a generalization of a Boolean SAT instance in which various sets of variables are replaced by predicates from a

variety of underlying theories. Examples of theories are the theory of real numbers, the theory of integers, and the theories of various data structures such as lists, arrays, or, bit vectors.

3.3 Bounded Model Checking

Model checking [18, 75] is a set of techniques for automatic verification of systems. The principle is to algorithmically check if a given model, which can be the given system or an abstraction of the system, satisfies a specification often formulated in terms of temporal logic.

Model checking tools face a combinatorial blow up of the state-space, commonly known as the state explosion problem, that must be addressed to solve in cases of real-world problems. Bounded model checking (BMC) [9, 10, 11] is one of the approaches to combatting this problem. BMC algorithms [21, 20, 66] unroll the finite state machine to a fixed number of steps k and check whether a property violation can occur in k or fewer steps. This typically involves encoding the restricted model as an instance of SAT problem. The process can, in principle, be repeated with large values of k until all possible violations have been ruled out.

3.4 Formula-based Fault Localization

This section provides basic definitions on formula-based automatic localization method. Definitions of the basic concepts such as MUS, MCS, MSS, and hitting set, are found in the literature (cf. [60]).

3.4.1 MUS, MCS, Hitting Set, and MSS

In the following definitions, C is a set of clauses, which constitutes a CNF formula φ. We use C and φ interchangeably. Note that in this thesis we sometimes present a set of clauses as a set of line numbers. These line numbers refer to program statements (instructions) in the target source code. Since in our work clauses are

encoded from instructions, for sake of clarity we show the line numbers instead of the associated clauses.

Definition 1 (Minimal Unsatisfiable Subset) *$M \subseteq C$ is a Minimal Unsatisfiable Subset (MUS) iff M is unsatisfiable and $\forall c \in M : M \backslash \{c\}$ is satisfiable.*

Definition 2 (Minimal Correction Subset) *$M \subseteq C$ is a Minimal Correction Subset (MCS) iff $C \backslash M$ is satisfiable and $\forall c \in M : (C \backslash M) \cup \{c\}$ is unsatisfiable.*

An MCS is a set of clauses such that C can be corrected by removing an MCS from C. Therefore, an MCS is considered to represent a root cause.

Definition 3 (Hitting Set) *H is a hitting set of Ω iff $H \subseteq D$ and $\forall S \in \Omega : H \cap S \neq \emptyset$.*

Let Ω be a collection of sets from some finite domain D, a hitting set of Ω is a set of elements from D that covers (*hits*) every set in Ω by having at least one element in common with it. A minimal hitting set is a hitting set from which no element can be removed without losing the hitting set property. There exist many algorithms to compute the hitting set, such as those presented in [36, 77] or in [81, 82]. We show one algorithm for computing the minimal hitting set in Section 4.5.2.

Definition 4 (Maximal Satisfiable Subset) *$M \subseteq C$ is a Maximal Satisfiable Subset (MSS) iff M is satisfiable and $\forall c \in C \backslash M : M \cup \{c\}$ is unsatisfiable.*

By definition, an MCS is the complement of an MSS ($MSS^{\complement}$) [60].

Any solution to MaxSAT problem is also an MSS. However, every MSS is not necessarily a solution to MaxSAT [70].

We illustrate the above definitions with an example taken from [59]. Below, a CNF formula and its MSSes, MCSes and MUSes.

$$\varphi = \overset{W}{(a)} \wedge \overset{X}{(\neg a)} \wedge \overset{Y}{(\neg a \vee b)} \wedge \overset{Z}{(\neg b)}$$

$$\text{MSSes}(\varphi) = \{\{X, Y, Z\}, \{W, Z\}, \{W, Y\}\}$$

$$\text{MCSes}(\varphi) = \{\{W\}, \{X, Y\}, \{X, Z\}\}$$

$$\text{MUSes}(\varphi) = \{\{W, X\}, \{W, Y, Z\}\}$$

To calculate the MSSes of φ_{AL} we use partial maximum satisfiability method.

3.4.2 Root Causes

In fault localization, the faults are identified by localizing their root causes. A root cause is the fundamental reason for the occurrence of a failing program execution. In the MBD theory, a root cause is represented by an MCS, and multiple root causes by MCSes. In practice, root causes help the programmers fix buggy programs. We conceptually distinguish real root causes and spurious root causes. Real root causes are program fragments that correct the program completely or partially when they are appropriately modified. In contrast, spurious root causes are program statements that when modified or removed do not fix the program with regard to the programmer's intention. In this thesis, we sometimes refer to either real root causes or spurious ones, but such distinction is concerned with the so-called high-level design decision of programmers. They are not distinct in view of the fault localization algorithm. In particular, the injected faults in the benchmark problems (Sections 5.5, 5.6, 7.4 and 7.3) are considered *real root causes*.

3.4.3 Code Size Reduction

Code size reduction (CSR) is the ratio of fault locations in an MUS (a program slice) to the total number of lines of code. The CSR is calculated as follows:

$$\text{csr}(MUS) = \frac{|MUS|}{\text{total number of lines}}$$

For example, in Listing 3.1 if we obtain the MUS $M = \{3, 4\}$ with its elements being line numbers. The CSR for M is as follows:

$$\text{csr}(M) = \frac{|M| * 100}{\text{total number of line}} = \frac{2 * 100}{10} = 20\%$$

Note that using static slicing results in CSR to be around 30% [52]. Another common terminology for quantifying the ratio of fault locations is the average CSR (ACSR), which represents the CSR for a set of MUS. For a given set S of MUSes, the ACSR is calculated as follows:

$$\text{acsr}(S) = \frac{\sum_i csr(S_i)}{|S|}$$

```
1 int getAt(int *a, int x) {
2   int y;
3   if (x>=0) {
4     y = a[x];
5   } else {
6     y = 0;
7   }
8   assert(y>=0);
9   return y;
10 }
```

Listing 3.1: A function that returns a positive value at a given index, or zero if the index is negative

3.4.4 Failing Program Paths

Let φ_{AL} be a formula in conjunctive normal form (CNF) such that

$$\varphi_{AL} = \varphi_{TI} \wedge \varphi_{TF} \wedge \varphi_{AS}$$

where φ_{TI} is a formula that encodes the test inputs, φ_{TF} is a trace formula that encodes all the possible program execution paths up to a certain depth, and φ_{AS} is a formula that encodes the assertion that the program must satisfy. The detailed representation of *TF* is irrelevant here, and will be introduced in Chapter 5 and 7. The formula φ_{AS} can be the post-condition or test oracle. When program paths are failing, φ_{TI} represents the input arguments that take certain particular values, making φ_{TF} violate φ_{AS}. We call such φ_{TI} error-inducing (EI). There are several approaches to generating such failing test inputs. Bounded model checking (BMC) [21] is one such approach, which generates a counterexample from which a single failing test input can be extracted. Test case generation methods, such as concolic execution [33, 84], can generate more than one test case for a given program. Generated test data, that result in failing execution, constitute such EI.

Example

We introduce a simple program in Listing 3.2 and its CFG in Figure 3.1. Listing 3.2 shows a function that takes two arguments. This program contains two faults. In line 4 the variable `z` should be set to 42 and in line 6 it should be set to 0. These two statements are called root causes.

```
1 int foo(int x, int y) {
2  int z;
3  if (x>y) {
4   z = 1;
5  } else {
6   z = 42;
7  }
8  // assert (x<=y and z==0)
9  // or (x>y and z==42)
10 return z;
11 }
```

Listing 3.2: A multi-fault program infected by two faults

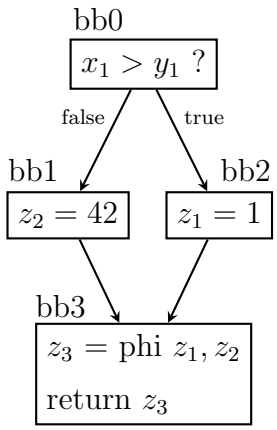


Figure 3.1: Control flow graph of function `foo`

For program of Listing 3.2, we obtained the two failing test cases (error-inducing inputs) below.

$$\varphi^{\text{foo}}_{TI_1} = (x_1 = 1) \wedge (y_1 = 0)$$

$$\varphi^{\text{foo}}_{TI_2} = (x_1 = 0) \wedge (y_1 = 1)$$

The post-condition of the function `foo` is encoded from the assertion of line 8 and takes the following form:

$$\varphi^{\text{foo}}_{AS} = ((x_1 \leq y_1) \wedge (z_3 = 0)) \vee ((x_1 > y_1) \wedge (z_3 = 42))$$

The trace formula is encoded from lines 2 to 7. Its encoding is shown later in Section 5.3.

3.4.5 Fault Localization Problem

Since φ^{AL} encodes failing program paths with EI, the formula φ^{AL} is unsatisfiable. By definition, EI and AS are supposed to be satisfied. The trace formula TF is responsible for the unsatisfiability. It is exactly the situation that the program contains faults. The fault localization problem is to find a set of clauses in TF that are responsible for the unsatisfiability. Such clauses are found in minimal unsatisfiable subsets (MUSes) of φ^{AL}. MUSes of the unsatisfiable formula are erroneous situations (conflicts) similar to failing static slicing (see Section 2.3). However, finding root causes in lengthy slices is difficult because many statements are included. According to the MBD theory [77], such root causes are diagnoses and are represented as MCSes.

An important point in the fault localization problem is the number of test cases being used. Usually, one test case may be enough to identify an erroneous situation caused by a single fault. Lots of test cases are needed to show the existence of all multiple faults. It implies that we check the unsatisfiability of $\varphi^{AL}(EI_i)$ with many different EI_i. A single counterexample approach does not work well for a general case of programs with multiple faults.

The fault localization problem is to find MCSes of φ^{AL}. The formula-based method adapted in herein [53] first calculates MSSes of φ^{AL} and then obtain MCSes by taking the complements of MSSes. Enumerating all the MCSes is mandatory to cover all the root causes.

3.4.6 Example

We explain the above concepts on the program of Listing 3.3. In line 6, there is an error in the computation of the absolute value of `x` in `abs`. The variable `abs` is equal to `x*1` when `x` is negative, which violates the assertion at line 8 which expects `abs` to be greater or equal to zero.

```
1 int absValue(int x) {
2   int abs;
3   if(x>=0) {
4     abs = x;
5   } else {
6     abs = x * 1;    // should be: abs=x*-1;
7   }
8   assert(abs>=0);
9   return abs;
10 }
```

Listing 3.3: A function that computes an absolute value

A failing trace can be obtained with an input value equal to `-1`. The error-inducing input extracted from the failing trace is encoded in EI and takes the following form: $EI = (x_0 = -1)$. The static single assignment (SSA) form of the function body (lines 2 to 7) is encoded in TF, as shown below. For recall, SSA form is a relation on program variables, which requires that each variable is assigned exactly once, and every variable is defined before it is used. Further details on the SSA form can be found in Section 3.5. See Section 5.3.4 concerning the mapping of clauses in TF to the original program statements. The assertion in line 8 is encoded in AS as follows: $AS = (abs_3 \geq 0)$.

$$TF = \underbrace{(guard_0 = (x_0 \geq 0))}_{\text{line 3}} \wedge \underbrace{(abs_1 = x_0)}_{\text{line 4}} \wedge \underbrace{(abs_2 = x_0 \times 1)}_{\text{line 6}} \wedge$$
$$\underbrace{((guard_0 \wedge (abs_3 = abs_1)) \vee (\neg guard_0 \wedge (abs_3 = abs_2)))}_{\text{line 3}}$$

We obtain two MSSes and two MCSes, as shown below. The set elements represent the line numbers of the program in Listing 3.3. We create a set containing the two MCSes. The minimal hitting set of the resulting set gives us a set containing two MUSes, which are the conflicts:

$MSS_0 = \{6\}$ $\quad MCS_0 = MSS_0^{\complement} = \{3, 4\}$
$MSS_1 = \{3, 4\}$ $\quad MCS_1 = MSS_1^{\complement} = \{6\}$

$MCSes = \{MCS_0, MCS_1\} = \{\{3, 4\}, \{6\}\}$
$MUSes = MCSes^{\mathsf{MHS}} = \{\{4, 6\}, \{3, 6\}\}$

We here obtained two diagnoses; one with the line numbers 3 and 4, another with 6. The diagnosis $\{3, 4\}$ indicates that both lines 3 and 4 should be corrected at a time. For example, if we change the statement in line 3 to be `x<0` and the statement in line 4 to be `abs=-x` then, for the input `x=-1`, the program becomes correct. The diagnosis $\{6\}$ indicates that line 6 only has to be modified to correct the program for the input `x=-1`. Software engineers are free to choose between these two diagnoses. From the diagnoses, we obtain two conflicts; one with the line numbers 4 and 6, another with 3 and 6. If we only need a set of potential root causes, we may extract the line numbers from either MCSes or MUSes to have a set, for example, $\{3, 4, 6\}$. The results are the same regardless of using MCSes or MUSes since only the line numbers are significant. It is what BugAssist [47] does to calculate the CSR. Note that with such a combination method, it is difficult, especially in the case of multi-fault programs, to know how many elements of the set have to be considered to fix the entire program.

3.5 Static Single Assignment Form

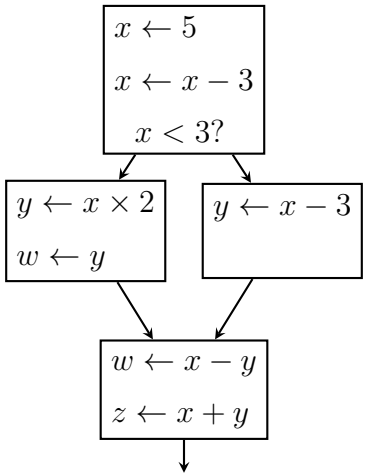
Static single assignment (SSA) form [22, 23] is an intermediate representation (IR) of a program source code, which requires that each variable is assigned once and exactly once, and every variable is defined (namely, assigned a value) before it is used. The variables living in the base representation are divided in "versions", new variables take their original name with an extension version number. The SSA form makes easy the conversion into propositional formulas. For example, the following code:

```
1   y = 1;
2   y = 2;
3   x = y;
```

Humans can see that the first assignment is not necessary, because the value of `y` used in the third line comes from the second assignment of `y`. A program should analyze how to reach variable definitions to determine the latter. But if the program is in SSA form, then this is straightforward; in the next version, this is obvious that `y1` is not used:

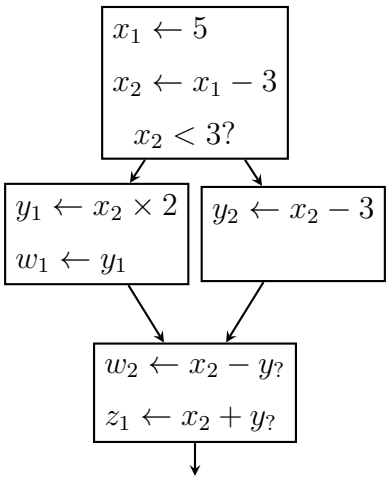
```
1   y1 = 1;
2   y2 = 2;
3   x1 = y2;
```

Converting an ordinary code to an SSA representation is a simple problem. It requires to replace each variable assignment by a new "versioned" variable (the variable `x` is renamed in `x1`, `x2`, `x3`, ... during the successive assignments). Finally, at each usage of a variable, we use the version corresponding to the position in the code. For example, from the following control flow graph:

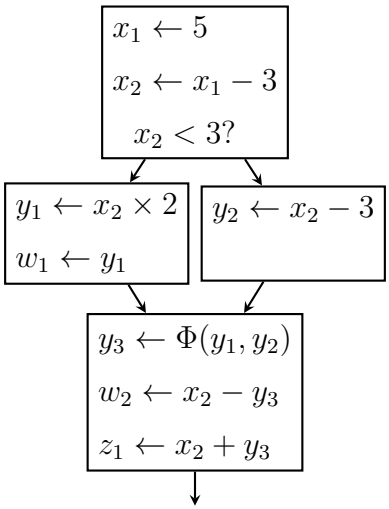


We notice that at the step "$x \leftarrow x - 3$", it is possible to replace in the left part of the expression the variable x by a new variable without changing the program computation. We use this in SSA by making two new variables: x_1 and x_2, each assigned a single time. Equivalently, we treat the other variables, which gives us the following graph:

It is possible to define from which variable versions we are referring to, except for



one case: the variable reference of y in the last block may also refers to y_1 or to y_2 depending on the flow it comes from. In order to know which variable we should use, we use a special declaration, called Φ (Phi), which is written at the beginning of the block. This function will generate a new version of y, y_3 that will have to be chosen between y_1 and y_2, depending on the block it comes from:



Now, using the variable y in the last block means using y_3. Note that it is not necessary to use a Φ function for x because a single version of x, called x_2, reaches this position. Hence, there is no ambiguity.

3.6 Program Dependence Graph

In a control flow graph (CFG), dependences can be represented by a program dependence graph (PDG). "A PDG makes explicit both the data and control dependences for each operation in a program" [31]. We will see in this thesis, particularly in Chapter 5, that dependences within a program must be considered when trying to localize faults, especially multiple faults.

A PDG is a graph whose nodes roughly correspond to program statements and whose edges represent dependences in the program. Assume that the program P is a single function program whose PDG is represented as the graph G_p. There is a directed edge between nodes v_1 and v_2 in G_p if there are dependences between v_1 and v_2. Dependences between program statements can be classified as either control or data dependences. A control edge between nodes v_1 and v_2 is represented as $v_1 \rightarrow_c v_2$. These edges are labeled `true` or `false`. A control dependence exists between nodes v_1 and v_2 iff:

1. The node v_1 is an entry vertex and v_2 is a statement within P that is not nested within any loop or conditional. This edge is labeled `true`.

2. The node v_1 is a control predicate (the test of a if-statement, while-statement, or for-statement block), and v_2 is immediately nested within the block predicated by v_1. If v_1 predicates a while loop, then the edge $v_1 \rightarrow_c v_2$ is labeled `true`. If v_1 is a predicate for a if-statement then the edge $v_1 \rightarrow_c v_2$ is labeled `true` or `false` depending of whether v_2 is on the `then` or `else` path.

For example, in Listing 3.2, s_2 is control dependent on s_1 because the execution of s_2 is conditionally guarded by s_1.

```
1 if (x==0) { // s1
2    y = 42;    // s2
3 }
```

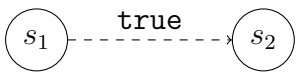


Figure 3.2: The statement s_2 is control-dependent on the statement s_1

A data dependence between statements s_1 and s_2 indicates that the semantics of the program may be changed if the relative ordering of v_1 and v_2 are changed. A PDG contains edges for *flow dependences* and *def-order dependences*. Note that in SSA form these dependences are explicit because all variables are unique. There is a flow dependence edge $v_1 \rightarrow_f v_2$ iff there is a def-use [3] edge from v_1 to v_2. There is a def-order dependence between statements v_1 and v_2 iff:

1. The nodes v_1 and v_2 both define the same variable.
2. The nodes v_1 and v_2 are in the same branch of any conditional statement that encloses both of them.
3. There exists a program component v_3 such that $v_1 \rightarrow_f v_3$ and $v_2 \rightarrow_f v_3$.
4. v_1 occurs to the left of v_2 in the program's abstract syntax tree.

For example, in Listing 3.3, s_2 depends on s_1 because executing s_2 before s_1 would result in s_2 using an incorrect value for `x`. Figure 3.4 shows an example of PDG. Nodes are labeled with line numbers of the function foo's listing.

```
1 x = 21;       // s1
2 y = x * 2;    // s2
```

Figure 3.3: The statement s_2 is data-dependent on the statement s_1

```
1 int foo(int x) {
2     int a = x * 2;
3     int b = x * 4;
4     int c = x * 8
5     int d = 0;
6     if (a < 100) {
7         d = d - b;
8     }
9     if (b >= 42) {
10        d = d + c;
11    }
12    if (c == 8) {
13        d = d * a;
14    } else {
15        d = 0;
16    }
17    return d;
18 }
```

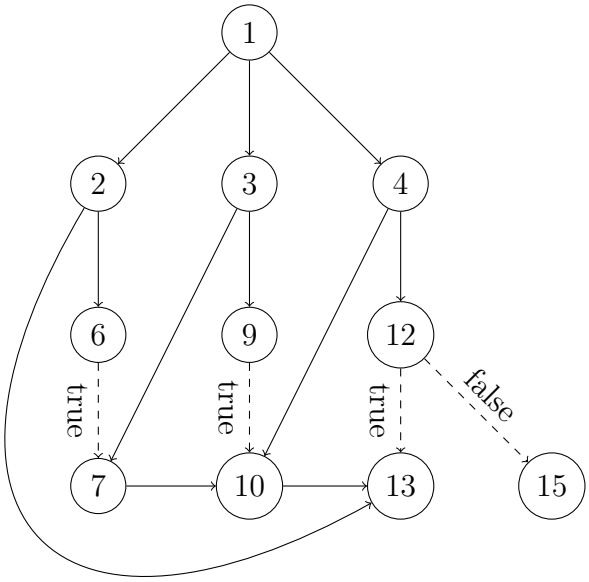


Figure 3.4: A function and its PDG

Chapter 4

Proposed Tool Architecture

4.1 Overview

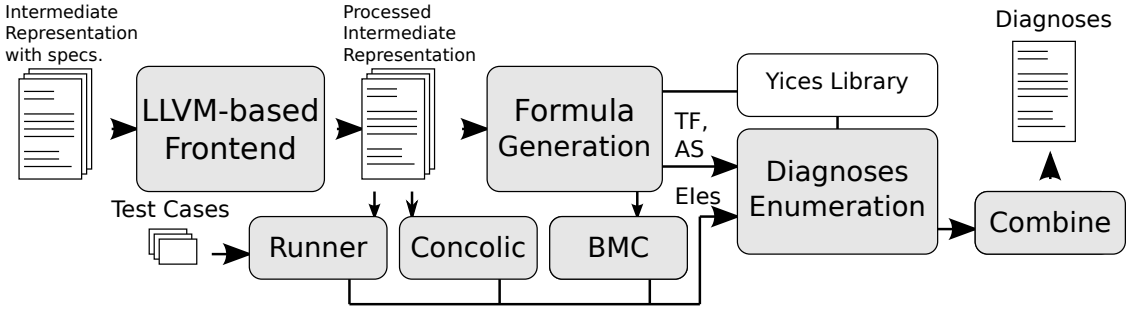


Figure 4.1: SNIPER tool flow

We implemented our formula-based fault localization method in a tool called SNIPER [53, 56] (SNIPER is Not an Imperative Program Errors Repairer). We will present in details the proposed method in Chapter 5 to 7. We now focus on the architecture of SNIPER. Figure 4.1 depicts the tool flow within SNIPER.

The goal of SNIPER is to provide a tool to software engineers for automatically localizing in an efficient manner root causes in faulty imperative programs. The architecture of SNIPER is based on the LLVM compiler infrastructure [57, 58] (see Section A.1 for details) to facilitate the handling of programs to be analyzed. SNIPER uses Yices 1 [28, 29, 30] as an internal solver to implement the different algorithms

that use decision procedures (SAT, MaxSAT, and pMaxtSAT). Yices 1 has some features that enable incremental solving. We exploit these capabilities in some of these algorithms. From a tool development viewpoint, Yices 1 is used flexibly because it provides an API for interacting with the solver. Hence, SNIPER uses Yices 1 as an internal solver rather than as a backend solver in a separate process. Because SNIPER includes in itself the LLVM and Yices 1 libraries, the tool can be considered as standalone. The advantage of a standalone tool is that we have full control at each stage. The control can be on the manipulation of the LLVM's IR (see Section A.3) or on the manipulation of the Yices' context (see Section A.2). This makes it possible for SNIPER to collect several information leading the fault localization to be precise and extensive.

The input of SNIPER is an LLVM intermediate representation (IR) of an imperative program[1]. The version of SNIPER presented herein specifically targets `C` code that follow the `ANSI C`[2] standard, but thanks to the language-agnostic design of LLVM, the basic architecture of SNIPER can, in principle, be used for automatic fault localization of imperative programs written in languages other than `C`. The input program has to be processed so that it can be later represented in a finite number of states. This is important for the next step, which is the encoding of the program in a trace formula (see Section 3.4.4). Most of the processing operations are standard in the bounded model-checking (BMC) of imperative programs (cf. [20, 21, 51, 66]). LLVM provides features to manipulate the intermediate representation. Thanks to this, transforming the IR is doable without much efforts. From the processed IR, SNIPER constructs a trace formula (TF in Figure 4.1) in the language of Yices 1. This formula represents all the potential executions of the input program within the specified scope bound.

In addition to the IR, SNIPER takes as input a correctness specification for the program. The specification describes what the program is expected to do. The

[1]The input IR has to contain debug information (line number), which can be generated by `Clang` (see Section A.1.1 for details.)

[2]The version of SNIPER used herein supports a subset of `ANSI C` only.

specification can be under the form of assertions in the source code, pre- and post-condition, or an oracle on program outputs. This specification is encoded in a formula (AS in Figure 4.1) in the language of Yices 1 and used in the next steps.

As presented in Section 3.4, the fault localization problem requires a formula to represent a pre-state to induce a failing execution. Such formulas are constructed from particular program inputs, the ones that trigger failing executions. SNIPER implements various techniques to automatically generate such program inputs. Even though test case generation is independent of localizing faults, we think that it is important to implement such testing techniques in SNIPER. It is because testing is one practical method to detect the existence of bugs in programs, which is mandatory before applying any fault localization methods. Additionally, this makes SNIPER have a certain control over the number of test cases generated, and their nature (failing test case or successful test case). Once a set of test cases is available, SNIPER encodes them in error-inducing input formulas in the language of Yices 1.

As depicted in Figure 4.1, from the trace formula (TF), the specification formula (AS), and the error-inducing input formulas (EIs), SNIPER computes a set of MCSes, which are diagnoses. The MCSes are obtained from different failing executions and each of them refers to a particular erroneous situation, a particular failing execution. In the case of multi-fault programs, there might be more than one erroneous situation due to the presence of different faults (see Section 2.8). We want to have a total diagnosis that takes into account all the erroneous situations in each of them, so that the software engineers can carefully choose which diagnoses are the most suitable for repairing all the faults in the program. To answer this problem, we introduce a method for combining the diagnoses. After the diagnoses have been combined, SNIPER outputs them to the user as source code lines marked with potential root causes.

In the following, we detail the internal architecture of SNIPER.

4.2 Program Encoding

The processed IR is transformed into an SMT formula, a trace formula. The formula is constructed in SNIPER with the Yices library, which provides an API in `C` language to interact with the solver (see Section A.2).

SNIPER implements two types of encodings: the full flow-sensitive trace formula (FFTF) [53, 55] and the hardened flow-sensitive trace formula (HFTF) [54]. The FFTF enables the systematic localization of single and multiple faults. However, this encoding produces large and complex formulas, which means that the localization algorithm with the FFTF can be computationally expensive. The HFTF is equivalent to the FFTF but some parts of the formula are set as hard (non-relaxable) making the number of soft (relaxable) clauses smaller. The localization algorithm with the HFTF is efficient in term of computation time but may miss some faults in case there are missing test inputs. Both encodings have their advantages and disadvantages. A complete description of these encodings can be found in Chapter 5 for the FFTF and in Chapter 7 for the HFTF.

4.3 Test Cases Generation

As explained in the introduction section of Chapter 2, software engineers start by *testing* the target program to see if it is faulty or not. In case the program is faulty, the engineers usually generate some failing inputs to exhibit incorrect behavior of the faulty program. These failing inputs are mandatory to localize faults.

Although our work focuses on automatic fault localization, we also consider such a testing phase because it has a non-negligible impact on the efficiency of the fault localization method. The user can either provide a set of test cases, or let SNIPER automatically generate one. SNIPER implements three different test case generation methods, which are described in the following sections.

4.3.1 BMC Module

SNIPER implements a classic BMC-based method that can generate a single failing trace. BMC (see Section 3.3 for details) with Boolean satisfiability is performed by the satisfiability checking of the formula φ_{bmc}.

$$\varphi_{\text{bmc}} = \neg(PRE \wedge TF \Rightarrow AS) = PRE \wedge TF \wedge \neg AS$$

If φ_{bmc} is satisfiable, the corresponding assignment represents a counterexample that illustrates a faulty program execution. We extract from the counterexample the values for the input arguments of the procedure that result in the failing execution, namely the error-inducing input values. Let EI be a formula expressing that those arguments take such values, which constitutes the pre-state of the program that will violate the post-condition.

Algorithm 1 describes in detail the BMC-based procedure. The algorithm takes as input a program, a specification the program must satisfy, and a maximal bound k. In line 3, the program is unrolled i times, the unrolled version of the program is encoded in a trace formula in CNF. In line 4, the algorithm checks the satisfiability of the conjunction of the trace formula and the negated assertions. If it is satisfiable, the corresponding assignment is returned. This assignment is a counterexample. In case it is unsatisfiable, the process may be repeated with a greater bound. The algorithm stops when a counterexample is found, or when the number of steps exceeds k.

Algorithm 1 BMCProcedure

Input: a program P, a formula AS that encodes the assertions the program must satisfy, and a bound k.
Output: a counterexample, or $\emptyset$.

```
1: i ← 0
2: while i ≤ k do
3:     TF^i ← encode(P, i)
4:     (st, A) ← SAT(TF^i ∧ ¬AS)        ▷ "A" is a satisfying assignment if st is true
5:     if st = true then
6:         return A
7:     end if
8:     i ← i + 1
9: end while
10: return ∅
```

4.3.2 Concolic Module

SNIPER needs at least one failing test case to enumerate the diagnoses (see Section 4.4 for details). The BMC-based method presented above can provide such a failing test case. However, when dealing with programs that contain multiple faults spread in different execution paths, it is strongly recommended to use more than one failing test case not to miss any faults. SNIPER implements a concolic execution method [84] that can generate many failing test cases. Concolic execution allows us to generate failing executions with a high code coverage.

Standard Concolic Execution

Concolic execution [33, 84] was proposed as a dynamic testing method that simultaneously performs symbolic and concrete execution. This method provides a high code coverage for program testing. We follow the approach of CUTE [84]. The algorithm first executes the program on some random input values. The method keeps track of a path constraint (PC) to collect symbolic predicate expressions from branching points. The conjunction of the predicates in PC holds for the execution path. The CUTE algorithm checks whether the PC with the last constraint negated is satisfiable. If so, new input values are generated allowing each test run to exercise different program paths. The algorithm stops when either no more new inputs can be generated or a given time-out is reached.

This module benefits from the JIT capabilities of LLVM [57] to run the program in an efficient manner. It implies that concolic test generation can be efficient by consuming less time to terminate.

4.3.3 Runner Module

In case the user has already a set of test cases, we implemented a module to parse the test cases in the format used by SNIPER. This module is called the *runner module*. It uses the same implementation of the *concolic module* but with the symbolic execution disabled. The module executes the test cases on the input program and

checks whether or not the obtained output satisfies the post-condition in order to know if the test case is a failing or successful one. As in the concolic module, this module also benefits from the JIT capabilities of LLVM.

4.4 Diagnosis Enumeration

The diagnoses enumeration consists of enumerating all the MCSes (minimal correction subsets) of the input program formula for each failing test input (error-inducing input). In the next section we start by describing a classic MCS enumeration algorithm. Based on this algorithm, we describe a basic algorithm for enumerating MCSes for the case of formula-based fault localization of imperative programs.

4.4.1 Classic MCS Enumeration

Given an unsatisfiable formula in CNF, the problem of enumerating all MCSes is defined in Definition 5.

Definition 5 (AllMinMCS [70]) *Given a constraint system φ, the AllMinMCS problem consists of finding all the minimum size MCSes of φ. The AllMCS problem consists of finding all MCSes of φ (independent of their size).*

We start by explaining the concept of *blocking*, which is necessary for enumerating MCSes. We then explain how a set of minimum size MCSes can be computed with partial maximum satisfiability.

MCS Blocking Technique

A blocking technique is used to prevent a particular MCS from reappearing while enumerating MCSes. This is done by adding a blocking constraint to the working formula. We use a technique introduced by [70] for blocking MCSes. This technique called *blocking by using auxiliary variables* can be efficiently implemented with Yices 1 [28]. This technique has the advantage of not using *relaxation variables* to block MCSes. A relaxation variable is a Boolean variable that is added to clauses

in the formula to block some clauses. Blocked clauses cannot be relaxed anymore, forcing the solver to find alternative solutions. When using relaxation variables it is require to check if the found MCSes are not supersets of previously found MCSes. This check increases the algorithm's complexity. The method of blocking by using auxiliary variables does not need to perform this checking. Blocking by using auxiliary variables consists of initially transforming each soft clause into a hard clause after adding a new Boolean variable called an auxiliary variable. Additionally, a set of unit soft clauses is added that corresponds to the negation of each auxiliary variable. Algorithm 2 shows how this is done.

For example, consider the SMT formula below and suppose that blocking by using auxiliary variables is used.

$$\varphi = \{(x \geq 1), (x < 1), ((x < 1) \vee (y < 1)), (y < 1), (y \geq 1)\}$$

Then the formula given to the pMaxSMT solver is the formula containing the set of soft clauses:

$$\varphi^{soft} = \{(\neg a_1), (\neg a_2), (\neg a_3), (\neg a_4), (\neg a_5)\}$$

and the set of hard clauses:

$$\varphi^{hard} = \{((x \geq 1) \vee a_1), ((x < 1) \vee a_2), ((x < 1) \vee (y < 1) \vee a_3),$$

$$((y < 1) \vee a_4), ((y \geq 1) \vee a_5)\}$$

The input of pMaxSMT is a CNF SMT formula φ, which is a conjunction of clauses. The output of pMaxSMT is an assignment $\mathcal{A}$ (consistent with a background theory $\mathcal{T}$) that minimize the number of falsified soft clauses of φ. When the pMaxSMT solver blocks an MCS, then the blocking constraint to be added to the working formula is as in the formula below, where a_i are auxiliary variables.

$$\varphi_W \leftarrow \varphi_W \cup \{(\bigvee_{\mathcal{A}(a_i)=true} \neg a_i)\}$$

The $\mathcal{A}(a_i) = true$ means that the variable a_i has the value *true* in the assignment $\mathcal{A}$, which was perviously returned by the pMaxSMT solver.

Algorithm 2 AddAuxVars (based on [70])

Input: a CNF formula φ.
Output: a CNF formula φ' with auxiliary variables a_i, or $\emptyset$.

1: $\varphi_W \leftarrow \varphi$ ▷ φ_W is the working formula
2: $AV \leftarrow \emptyset$ ▷ AV is a set of auxiliary variables (Boolean variables)
3: $\varphi_{soft} \leftarrow \emptyset$
4: ▷ Create a set of unit soft clauses
5: **for each** $w \in \varphi_W$, w tagged as *soft* **do**
6: $AV \leftarrow AV \cup \{a_i\}$ ▷ a_i is a new auxiliary variable created
7: $\varphi_{soft} \leftarrow \varphi_{soft} \cup \{(\neg a_i)\}$
8: $w_A \leftarrow (w \vee a_i)$
9: $\varphi_W \leftarrow \varphi_W \setminus \{w\} \cup \{(w_A)^{\mathsf{HARD}}\}$ ▷ Remove w and add w_A as *hard*
10: **end for**
11: **if** $AV = \emptyset$ **then**
12: **return** $\emptyset$ ▷ No possible pMaxSMT solution
13: **end if**
14: $\varphi' \leftarrow \varphi_W \wedge \varphi_{soft}$
15: **return** φ'

Algorithm 3 AllMinMCS (based on [70])

Input: an SMT CNF formula φ with auxiliary variables a_i.
Output: a set M containing all the minimum size MCSes of φ.

1: $\varphi_W \leftarrow \varphi$ ▷ φ_W is the working formula
2: $M \leftarrow \emptyset$
3: **while true do**
4: $(\text{st}, \varphi_{MSS}, \mathcal{A}) \leftarrow$ `pMaxSMT(`φ_W`)` ▷ Solve the working formula
5: ▷ "φ_{MSS}" is an MSS if st is **true**
6: ▷ "$\mathcal{A}$" is a maximal satisfying assignment if st is **true**
7: **if** st = **true then**
8: $\varphi_{MCS} \leftarrow$ `CoMSS(`φ_{MSS}`)` ▷ The complement of an MSS is an MCS
9: $M \leftarrow M \cup \{\varphi_{MCS}\}$
10: $\varphi_W \leftarrow \varphi_W \cup \{(\bigvee_{\mathcal{A}(a_i)=\mathbf{true}} \neg a_i)\}$ ▷ Add the blocking constraint
11: **else**
12: **break** ▷ No more new minimum size MCS solution
13: **end if**
14: **end while**
15: **return** M

AllMinMCS Algorithm

We describe a classic algorithm for enumerating all minimum size MCSes of a given formula. Algorithm 3 takes as input a CNF formula φ with auxiliary variables a_i. An MSS can be computed using a pMaxSAT procedure (line 4). The MCS can be obtained by taking the complement of the MSS (line 8). The algorithm blocks MCSes

by using auxiliary variables (line 10). In the previous section, we explained how we add auxiliary variables to a CNF formula. The algorithm stops when all minimum size MCSes are blocked.

Note that this algorithm fully enumerates MCSes for each error-inducing inputs, as opposed to the algorithm of BugAssist [47] that enumerates MCSes in an incomplete manner.

For program of Listing 3.2 (Section 3.4.4) and for a set of error-inducing inputs $E^{\text{foo}} = \{\varphi^{\text{foo}}_{TI_1}, \varphi^{\text{foo}}_{TI_2}\}$, if we run Algorithm 4 as $AllDiagnoses(\varphi^{\text{foo}}_{FFTF}, E^{\text{foo}}, \varphi^{\text{foo}}_{AS}) = D^{\text{foo}}$ we obtain the following MSSes and MCSes:

$$MSSes(\varphi_1) = \{\{W, X, Z\}, \{Y, Z\}\}$$

$$MCSes(\varphi_1) = \{\{Y\}, \{W, X\}\}$$

$$MSSes(\varphi_2) = \{\{X, Y, Z\}, \{W, Y, Z\}\}$$

$$MCSes(\varphi_2) = \{\{W\}, \{X\}\}$$

$$D^{\text{foo}} = \{MCSes(\varphi_1), MCSes(\varphi_2)\}$$

4.4.2 Basic Diagnosis Enumeration for Imperative Programs

Algorithm 4 describes the computation of the diagnoses for imperative programs. The algorithm takes as input a set of error-inducing inputs, a trace formula, and a formula that encodes the assertions the program must satisfy. For each error-inducing input (lines 3 through 9), a set of MCSes is computed using AllMinMCS (line 5) with the clauses of φ_{TF} set as *soft*, and the clauses of φ_e and φ_{AS} set as *hard*. Finally, a set of diagnoses D is obtained, which contains root causes of the faulty program. The obtained diagnoses in this set must be combined before the user can use them for effectively locating faults in the program. Section 4.5 presents some diagnosis combination techniques.

4.4.3 Discussions

BugAssist [47] uses a MaxSAT-based method for computing MCSes . The method does not enumerate all minimum size MCSes, which means that the method does

Algorithm 4 AllDiagnoses

Input: a set of error-inducing inputs E, a trace formula φ_{TF}, and a formula φ_{AS} that encodes the assertions the program must satisfy.
Output: D a set of diagnoses (MCSes).

```
1: M ← ∅
2: D ← ∅
3: for each e ∈ E do
4:     φ_e ←Encode(e)
5:     M ←AllMinMCS(φ_e^hard ∧ φ_TF^soft ∧ φ_AS^hard)        ▷ Enumerate all minimum size MCSes
6:     if M ≠ ∅ then
7:         D ← D ∪ {M}
8:     end if
9: end for
10: return D
```

not guarantee to cover all the root causes. Wotawa et al. [95] uses a method that enumerates MCSes up to a certain size. The advantage of such method is that the enumeration is faster as compared to enumerating all minimum size MCSes because there are fewer MCSes to enumerate. However, the completeness of the method may not be achieved if the MCSes' sizes are too small. If some MCSes are not covered, root causes may be missed. The latter is especially true in the case of multi-fault programs because MCSes in such programs tend to include many elements, each one corresponding to a particular fault. As opposed to BugAssist and Wotawa's approaches, we enumerate all minimum size MCSes, and hence our method covers all root causes, at least for the error-inducing inputs used.

4.5 Diagnosis Combination

Algorithm 8 provides a function that returns a set of diagnoses (MCSes) for each error-inducing input given as arguments. Each of these sets contains root cause candidates for one failing execution, which is triggered by the error-inducing input associated to the set. The problem of combining diagnoses is to generate sets of fault locations that can be used to potentially fix all the failing executions induced by the provided error-inducing inputs, meaning that the user can use one of the sets to fix all the faults in the program. As discussed in Section 2.8.2, independent multiple faults need more than one failing execution. We call such gathering of sets a *complete*

diagnosis.

Definition 6 (Complete Diagnosis) *Given a formal representation TF of a program P, a formula AS that encodes the assertion the program P must satisfy, and a set of error-inducing inputs E, a complete diagnosis* Δ *is a set of clauses of TF such that* $\forall e \in E \mid (\{e\} \cup (TF \setminus \Delta) \cup AS)$ *is satisfiable.*

SNIPER implements three combination techniques: a flattening-based combination, a hitting-set-based combination and a pair-wise-based combination. The following sections describe these combination techniques.

4.5.1 Flattening-based Combination

The flattening-based combination technique is a simple method to combine diagnosis. The method is described in Algorithm 5. The algorithm takes as input a set of MCSes and output a set of elements, which are root causes (lines of code for example).

This combination method is essentially the same as the one used in the tool BugAssist [47]. This method is not computationally costly as compared with other advanced methods, however, it produces little information regarding the relations between the root causes. Indeed, all the root causes are in a single set and this can be a problem when dealing with multi-fault programs because the engineer does not know if she has to take into account one or more than one root cause of this set to correct the faulty program.

In the following two sections we introduce combination methods being able to face the problem introduced by multi-fault programs.

4.5.2 Hitting-set-based Combination

One way to combine MCSes (diagnoses) is to use the minimal hitting set formulation, which was defined in Section 3.4.1. Suppose that we have a set Ω of MCSes, then if we compute the minimal hitting set of Ω we obtain a set of sets H, which

Algorithm 5 DiagCombineFLA

Input: D a set of diagnoses (MCSes).
Output: C a combined diagnosis (MCS).

```
1: C ← ∅
2: for each x_i ∈ D do                          ▷ x_i = {{...}, ...}
3:     for each y_j ∈ x_i do                    ▷ y_j = {...}
4:         C ← C ∪ z_k
5:     end for
6: end for
7: return C
```

contains *complete diagnoses* because H is a set of elements that covers every set in Ω by having at least one element in common with it.

SNIPER adapts a MaxSAT-based hitting set calculation algorithm. It is an exact algorithm meaning that we do not use any approximation and always obtain sets of minimal cardinality. Algorithm 6 describes our approach for computing minimal hitting sets. The idea behind our algorithm is to iteratively call a MaxSAT solver and use blocking variables (see Section 4.4.1 for details) to block the solution after each call.

We are able to obtain minimal sets because the solver always maximizes the number of variables x_i equal to false, which is equivalent to minimizing the number of x_i equal to true. The x_is equal to true are the resultant hitting set.

4.5.3 Union Pair-wise-based Combination

SNIPER implements a combination technique [53] based on a pair-wise union. This technique ensures that if no fault is missed by the diagnoses generation algorithm of SNIPER, then no fault is missed after the combination.

A set C of complete diagnoses can be calculated using a n-ary pairwise union as defined in Definition 7. Given a n-tuple R we denote R^i the ith component of R. Let us denote $\prod_{j=1}^{n} D_j$ the cartesian product of $D_1, ..., D_n$, which produces the set of all ordered n-tuples $\langle a^1, ..., a^n \rangle$, where $a^i \in D_i$ for all i, $1 \leq i \leq n$. When D is a set of sets of MCSes, each set of C is a complete diagnosis.

Algorithm 6 DiagCombineMHS

Input: Collection Ω of subsets of a finite set D.

Output: A hitting set for Ω, i.e., a subset $H \subseteq D$ such that H contains at least one element from each subset in Ω.

1: $\varphi_{hard} \leftarrow \emptyset$ ▷ Formula for the hard clauses
2: $\varphi_{soft} \leftarrow \emptyset$ ▷ Formula for the soft clauses
3: **for each** $x_i \in D$ **do**
4: $\quad \varphi_{soft} \leftarrow \varphi_{soft} \cup \{(\neg x_i)\}$
5: **end for**
6: **for each** $c_i \in \Omega$ **do**
7: $\quad \varphi_{hard} \leftarrow \varphi_{hard} \cup \{(\bigvee_{\forall x_i \in c_i} x_i)\}$
8: **end for**
9: **while true do**
10: $\quad (\text{st}, \varphi_{MSS}, \mathcal{A}) \leftarrow \texttt{pMaxSMT}(\varphi_{hard} \cup \varphi_{soft})$
11: $\quad \triangleright$ "φ_{MSS}" is a maximal satisfiable subset (MSS) if st is **true**
12: $\quad \triangleright$ "$\mathcal{A}$" is a maximal satisfying assignment if st is **true**
13: $\quad$ **if** st = **true then**
14: $\quad\quad \triangleright$ Add the blocking constraints
15: $\quad\quad \varphi_{hard} \leftarrow \varphi_{hard} \cup \{(\bigvee_{\mathcal{A}(x_i)=\textbf{true}} \neg x_i)\}$
16: $\quad\quad N \leftarrow N \cup \{\bigcup_{\mathcal{A}(x_i)=\textbf{true}} x_i\}$
17: $\quad$ **else**
18: $\quad\quad$ **return** H
19: $\quad$ **end if**
20: $\quad H \leftarrow H \cup N$
21: **end while**

Definition 7 (SetCombine) *Let $D_1, ..., D_n$ be n sets. Then the SetCombine C for D is defined as follows:*

$$C = \left\{ \bigcup_{i=1}^{n} a^i \mid a \in \prod_{j=1}^{n} D_j \right\}$$

The potential effectiveness of complete diagnoses generated from the `SetCombine` operator is demonstrated empirically with the experiments of Section 5.5 and Section 5.6. Nevertheless, we show below a property that directly follows Definition 7. This property aims at showing that the `SetCombine` operator is sound.

Property 1 shows that the `SetCombine` operator does not delete any elements (clauses) from the MCSes output by Algorithm 8, and does not add extra elements (clauses) to the complete diagnoses.

Property 1

$$\bigcup_{C_k \in C} C_k = \bigcup_{D_j \in D} \bigcup_{d_s \in D_j} d_s$$

Proof: For sake of clarity, let us denote $C' = \bigcup_{C_k \in C} C_k$ (the left part of the equation) and $D' = \bigcup_{D_j \in D} \bigcup_{d_s \in D_j} d_s$ (its right part). We must show that $C' = D'$. Let us show that $x \in C'$ iff $x \in D'$. $x \in C'$ iff $x \in \bigcup_{C_k \in C} C_k$ iff there is a $C_k \in C$ such that $x \in C_k$ iff there exist $a \in \prod_{j=1}^{n} D_j$ such that $x \in \bigcup_{i=1}^{n} a^i$ (using Definition 7), i.e., $\exists i$ such that $x \in a^i$, iff there exists $D_j \in D$ such that $d_s \in D_j$ such that $x \in d_s$ iff $x \in \bigcup_{D_j \in D} \bigcup_{d_s \in D_j} d_s$, i.e., $x \in D'$. Therefore, $C' = D'$. □

Algorithm 7 DiagCombinePWU

Input: D a set of diagnoses (MCSes).
Output: C a set of combined diagnoses (MCSes).

```
1: n ← |D|
2: a_i ← 0  ∀i ∈ {0, 1, ..., n − 1}
3: repeat
4:     S ← {∅}
5:     for i ← 0 to i < n do                    ▷ Union for the current indexes in a
6:         j ← a_i
7:         A ← D_i                              ▷ A is a set of MCSes
8:         B ← A_j                              ▷ B is an MCS
9:         S ← S ∪ B
10:    end for
11:    C ← C ∪ {S}
12:    a_0 ← a_0 + 1
13:    for i ← 0 to i < n − 1 do                ▷ Update indexes in a
14:        if a_i ≥ |D_i| then
15:            a_i ← 0
16:            a_{i+1} ← a_{i+1} + 1
17:        end if
18:    end for
19: until a_{n−1} ≥ |D_{n−1}|
20: return C
```

Let us illustrate this with the following example.

Example When running Algorithm 8 on the TF of multi-fault program in Listing 2.2 (see Section 2.8.2) with the following error-inducing inputs: `x=0` and `x=1`, we obtain a set of MCSes and D below.

$$MCSes_a = \{\{3,4\},\{6\}\},\ \ MCSes_b = \{\{3\},\{4\}\},$$
$$D = \{MCSes_a, MCSes_b\}$$

The root cause locations in $MCSes_a$ are related to the failing path triggered by `x=0`, and those in $MCSes_b$ are related to the failing path triggered by `x=1`. The combination of MCSes of D gives us the following *complete diagnoses*:

$$\begin{aligned}\texttt{SetCombine}(D) &= \{\{3,4\}\cup\{3\},\{3,4\}\cup\{4\},\{6\}\cup\{3\},\{6\}\cup\{4\}\}\\ &= \{\{3,4\},\{3,6\},\{4,6\}\}\end{aligned}$$

A set of fault locations to check is needed to fix all faults in the program. For example, the set $\{4,6\}$ provides information to fix the program since it combines root causes from the two failing paths.

In some cases, multiple faults can lie in one program path. In such situation, a single error-inducing input is enough to localize all the faults. In other cases, if two faults are in different paths or triggered by different error-inducing inputs it requires more than one error-inducing inputs that trigger different failing paths. For the program in Listing 2.8.2, we need at least two error-inducing inputs to trigger both failing paths. In summary, a basic MCS enumeration method that only uses a single failing test case is not sufficient when dealing with multi-fault program whose faults are spread in different program paths or execution paths. The association of Algorithm 8 and the combination method of Definition 6 allow the efficient combination of MCSes (root causes), each obtained from different failing paths.

4.5.4 Discussion and Related Work

Wotawa et. al. [95] does not use any combination method. This means that the method returns the original enumerated MCSes, which is the set D in the example of

the previous section. The software engineer has to sort MCSes by hand, and it can be difficult when the number of MCSes is large.

The approach of BugAssist [47] is to combine MCSes by putting all its atomic elements (clauses) in a single set. The clauses (representing fault locations) in this set are later ordered by a ranking mechanism. For example, if we apply the BugAssist's method to the example of the previous section, we obtain the following set of MCSes:

$$\texttt{BugAssistCombine}(D) = \{3, 4, 6\}$$

The resulting set contains elements from different failing paths. With such set, it is difficult to see how many elements we have to consider to fix all faults in the program. As opposed to BugAssist's approach, the pair-wise union method or the minimal hitting set method generates different sets, each set containing potential root causes for fixing all the faults in the program. For example, with the pair-wise union method on the example of the previous section we obtain the following set of MCSes:

$$\texttt{DiagCombinePWU}(D) = \{\{3, 4\}, \{3, 6\}, \{4, 6\}\}$$

Chapter 5

Full Flow-sensitive Trace Formula

5.1 Introduction

In formula-based fault localization methods, potential execution paths of a program are encoded in a trace formula (TF). The efficiency and precision of the fault localization algorithm are highly dependent on the way this formula is encoded. Depending on both the multi-fault type and the way the faulty program is encoded, the faults may or may not be localized. Table 5.1 and 5.2 summarize the encoding methods in existing work and our proposal. The following sections give further details on each encoding method.

Trace Formula Type	**Single Fault Types**	
	Calculation	**Branching Condition**
Flow-insensitive [47]	☑	☐
Flow-sensitive [17]	☑	☑
Full Flow-sensitive	☑	☑
Spectrum-based [45, 46]	☑	☑

Table 5.1: The types of trace formula used in formula-based fault localization and their specificities (1/2)

Trace Formula Type	**Multi-fault Types**		
	Data Variable Dependent	**Control Variable Dependent**	**Independent**
Flow-insensitive [47]	☑	☐	☑
Flow-sensitive [17]	☑	☐	☑
Full Flow-sensitive	☑	☑	☑
Spectrum-based [45, 46]	☐	☐	☐

Table 5.2: The types of trace formula used in formula-based fault localization and their specificities (2/2)

5.2 Trace Formula Encodings

A simple approach [47] consists of encoding a TF from a counterexample obtained using bounded model-checking (BMC) methods. We call this encoding flow-insensitive TF. Since it represents a straight-line program fragment that contains faults, it does not reconstruct information related to the control flow of the original program. Thus, the formula is small, which makes the localization efficient. However, because of this lack of information, potential root causes in the program control flow cannot be localized, which means that control-dependent faults cannot be localized either. For the case of independent faults, it is mandatory to repeat the process of counterexample generation in order to localize all faults because a counterexample represents a single program path containing one of the independent faults only.

In order to overcome the lack of control flow information in flow-insensitive TF, a flow-sensitive TF was proposed [17]. Basically, a flow-sensitive TF is similar to flow-insensitive TF with the exception that flow-sensitive TF includes some control flow information, such as values of variables that determine conditional branches, along the failing program path. This makes possible the localization of faults that lie in the control flow. Since a flow-sensitive TF represents itself more information, it is larger and more complex than a flow-insensitive TF. Furthermore, as in flow-insensitive TF,

it is necessary to construct many TF in order to deal with independent faults because the flow-sensitive TF does not encode all execution paths [83]. Concerning data flow-dependent faults, they can be identified with both flow-insensitive and flow-sensitive TF. This is because these trace formulas encode data flow information of the target failing path, which includes the faults.

For multi-fault programs with nested faults (for recall, a special case of control-dependent faults), both flow-insensitive and flow-sensitive TF cannot be successful to identify them. This is essentially because flow-insensitive and flow-sensitive TF do not encode full control-dependencies.

5.3 Full Flow-sensitive Trace Formula

In this thesis we introduce a new trace formula encoding called full flow-sensitive TF, which is essentially equivalent to the program's CFG. The data flow and control flow of the input program are fully encoded in the formula. In order to faithfully represent all potentially possible executions of the input program, both of these flows must be encoded. When both flows are properly encoded, multi-faults as described in Section 2.8.2, including nested faults, can be localized by our method. The disadvantage is that full flow-sensitive TF is large and complex because it represents the whole program. Hence, we introduce in Chapter 6 an efficient algorithm for computing diagnoses with full flow-sensitive TF.

We will describe below how we translate a pre-processed LLVM IR to a partial SMT formula[1].

We construct the full flow-sensitive trace formula (FFTF) [53] from the control flow graph (CFG) representation of the preprocessed program. The CFG is in static single assignment (SSA) form [22, 23]. A CFG is a directed graph of the form $P = (B, l, e, T)$, where B is the set of all basic blocks (nodes), $l \in B$ is the initial basic block executed on the entrance, $e \in B$ is the exit basic block, and $T \subseteq B \times B$ is the set of all transitions between basic blocks (directed edges). Each basic block consists

[1]For sake of simplicity we omit some details about the IR [57].

of a labeled entry point, a series of phi nodes, a list of instructions, and ends with a terminator instruction, such as a branch or function return.

5.3.1 Encoding of the Data Computation

The arithmetic and comparison instructions in LLVM take two arguments and return one result. We restrict the type of variables to integers and booleans. Let OP be a set of operators. The arithmetic and comparison instructions are encoded in equality constraints as follows:

$$r = (x \ \Delta \ y) \quad \Delta \in OP$$

where r is the result of the computation of the variables x and y. In the case of comparison operators, the result r is a boolean variable, called a guard, that will be used in the representation of the control-flow.

5.3.2 Encoding of the Control-Flow

A function definition contains a list of basic blocks, forming the control flow graph (CFG) of the function body. Each basic block consists of a labeled entry point, a series of ϕ nodes, a list of instructions, and ends with a *terminator instruction* such as a branch or function return.

Let BB be the set of all basic blocks. Let $T \subseteq BB \times BB$ be a subset of all transitions between the basic blocks. For each transition $(bb_i, bb_j) \in T$ with $bb_i, bb_j \in BB$, we have a Boolean variable t_{ij} that is *true* iff the control flow goes from bb_i to bb_j. The set of predecessors of a basic block bb_j is equal to:

$$\text{pred}(bb_j) = \{bb_i \in BB \mid (bb_i, bb_j) \in T\}$$

Let $\text{on}(bb_i)$ with $bb_i \in BB$ be the *enabling condition* that is *true* iff the basic block bb_i is executed. The value of $\text{on}(bb_i)$ is computed as:

$$\text{on}(bb_i) = \bigvee_{bb_j \in \text{pred}(bb_i)} \text{on}(bb_j) \wedge t_{ji}$$

Unconditional branches between basic blocks are encoded by setting the transition variable to the value of the *enabling condition* of the basic block where the branch occurs:

$$\text{on}(bb_i) = t_{ij}$$

Conditional branches make the control flow jump from a basic block bb_i to either a basic block bb_j if the guard g is *true*, or to a basic block bb_k otherwise:

$$(t_{ij} = g) \wedge (t_{ik} = \neg g)$$

As is usual in SSA representation, ϕ nodes join together values from a list of its predecessor basic blocks. Each ϕ node takes a list of (value, label) pairs to indicate the value chosen when the control flow transfers from a predecessor basic block with the associated label. Below, the encoding of a ϕ node, where the new symbol x_i refers to the variable x in bb_i.

$$\bigvee_{x_j \in \text{pred}(bb_i)} (x_i = x_j) \wedge t_{ji}$$

The CFG takes the formula below. The *entry* basic block in a function is immediately executed on entrance to the function and has no predecessor basic blocks. Its enabling condition on(*entry*) is always *true*. φ_{on} is the formula that encodes the *enabling conditions* for all basic blocks, φ_{uncond} is the conjunction of all constraints on unconditional branches, φ_{cond} is the conjunction of all constraints on conditional branches, and φ_{phi} is the conjunction of the constraints encoding the ϕ nodes.

$$\varphi_{\text{CFG}} \equiv \text{on}(entry) \wedge \varphi_{\text{on}} \wedge \varphi_{\text{uncond}} \wedge \varphi_{\text{cond}} \wedge \varphi_{\text{phi}}$$

5.3.3 Whole Trace Formula

The whole trace formula for the IR, TF, takes the form below. φ_{CFG} is the formula that encodes the control flow of the program and $\varphi_{\text{arith/comp}}$ is the conjunction of the constraints encoding the arithmetic and comparison instructions.

$$TF = \underbrace{\varphi_{\text{CFG}}}_{\text{hard}} \wedge \underbrace{\varphi_{\text{arith/comp}}}_{\text{soft}}$$

The clauses that encode the CFG of the program are marked as *hard* because they represent the skeleton of the program and we do not want the solver to relax these clauses. The rest of the clauses are set as *soft* (relaxable) because they contribute to the computations of the program, and are then susceptible to be root cause candidates. Note that with our encoding we can identify root causes related to the control flow. For recall, the control flow of an imperative programs refers to the order in which the individual instructions are executed or evaluated. This order is controlled by branch instructions within the program. The outcome of a branch is made upon its condition's value, which is calculated by a comparison instruction. The trace formula presented herein encodes each comparison instruction as soft. Hence, in situation in which the obtained trace formula encodes a faulty control flow, when enumerating MCSes (see Section 6.2 for details), the solver can relax some clauses related to comparison instructions. When relaxing one of such clauses, the outcome of the branch using the result of this comparison instruction can be inverted (for example, taking the true edge instead of the false edge). In other words, the solver can manipulate the control flow so that it becomes correct in view of the provided program specification.

Example

For example, below a simplified version of the FFTF for the CFG of function `foo` (see Figure 3.1).

$$\varphi^{\text{foo}}_{FFTF} = \underbrace{(\overset{W}{g_1} = (x_1 > y_1)) \wedge (\overset{X}{z_1} = 1) \wedge (\overset{Y}{z_2} = 42)}_{\text{soft}} \wedge \underbrace{(\overset{Z}{z_3} = (\text{ITE } g_1\ z_1\ z_2))}_{\text{hard}}$$

5.3.4 Formula Granularity Level

In the final encoding of the TF shown in the previous section, the formula φ_{cal} contains all the *soft* clauses to be potentially relaxed by the pMaxSAT solver. The

complexity of the problem is related to the way these clauses are grouped together. For the case of our encoding, we study the following three granularity levels:

- Instruction-level
- Line-level
- Block-level

With the instruction level granularity, each LLVM instruction is encoded in a single clause. With the line level granularity all instructions belonging to the same statement line in the original source are grouped in a single clause. With the block level granularity, all instructions belonging to the same LLVM basic block are grouped in a single clause.

A trace formula, with an instruction-level granularity, contains the same number or more soft clauses than a trace formula with a line-level granularity or a block-level granularity. The comparison between a trace formula with a line-level granularity to a formula with a block-level granularity depends on the way the source code is arranged. Usually, however, there are more instructions in a line than in a basic blocks. We show in the experiment section (Section 5.5.4) the difference obtained in term of computing time depending on the granularity level used for constructing the trace formula. Note that the granularity level also has an impact on the precision of the fault localization algorithm.

Example

Let c^i_{l,b_j} be a clause that encodes an LLVM instruction i of a basic block b_j defined at a line l in the original source code. Below, a set of clauses encoded from a program P.

$$c^1_{1,b_1},\quad c^2_{1,b_1},\quad c^3_{2,b_1},\quad c^4_{2,b_1},\quad c^5_{3,b_2},\quad c^6_{3,b_2}$$

Consider below the trace formulas encoding P constructed with different granularity levels.

$$\varphi^{inst}_{TF} = \{(c^1_{1,b_1}),\quad (c^2_{1,b_1}),\quad (c^3_{2,b_1}),\quad (c^4_{2,b_1}),\quad (c^5_{3,b_2}),\quad (c^6_{3,b_2})\}$$

$$\varphi_{TF}^{line} = \{(c_{1,b_1}^1 \wedge c_{1,b_1}^2),\ \ (c_{2,b_1}^3 \wedge c_{2,b_1}^4),\ \ (c_{3,b_2}^5 \wedge c_{3,b_2}^6)\}$$

$$\varphi_{TF}^{block} = \{(c_{1,b_1}^1 \wedge c_{1,b_1}^2 \wedge c_{2,b_1}^3 \wedge c_{2,b_1}^4),\ \ (c_{3,b_2}^5 \wedge c_{3,b_2}^6)\}$$

The trace formula φ_{TF}^{inst} contains 6 groups of clauses, each group made of a single clause. The trace formula φ_{TF}^{line} contains 3 groups of clauses. The trace formula φ_{TF}^{block} contains 2 groups of clauses.

5.4 Fault Localization with FFTF and AllDiagnoses Algorithm

In this section we explain how root causes are localized using the FFTF together with the AllDiagnoses algorithm, which was presented in Section 4.4.

The FFTF is essentially equivalent to the CFG of the target program. In order to explain the modus operandi of the AllDiagnoses algorithm with a FFTF as input we transpose the concept of program dependence graph (PDG) to FFTF. A definition of the PDG can be found in Section 3.6.

Definition 8 transposes the concept of PDG to FFTF. Since a FFTF is equivalent to the target program's CFG, transposing dependences is straightforward. Note that in our case PDG are always constructed from loop-free and function call-free programs.

Definition 8 (Trace Formula Dependence Graph) *Given a full flow-sensitive trace formula TF of a loop-free single-function program P, a trace formula dependence graph (TFDG) G_{TF} is the PDG G_p of P where all node v_i are replaced by clauses c_i with $c_i \in TF$ such that c_i is the encoding of v_i.*

In the FFTF, the control predicates are guard variables appearing in `icmp` instructions. Depending on the values of these guards, a decision is made, which has the effect of *enabling* or *disabling* clauses in the trace formula. We define the concept of clause enabling/disabling in Definition 9.

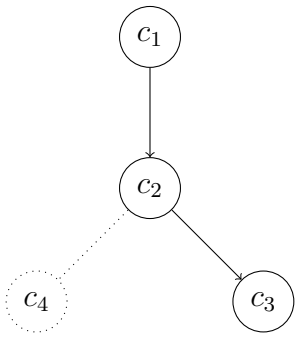


Figure 5.1: Clause Enabling/Disabling

Definition 9 (Clause Enabling/Disabling) *Given a full flow-sensitive trace formula TF of a program P , a TFDG G_{TF} of TF, and an assignment $\mathcal{A}$ of TF, a clause c_1 is enabled iff there exists a path C in G_{TF} between the entry node and c_1, such that all guard variables appearing in the clauses of C are evaluated in $\mathcal{A}$ such that c_1 is reachable. A clause c_1 is disabled iff c_1 is not enabled.*

In other words, a clause is enabled when the formula assignment makes the control flow cover (traverse) the clause. A clause is disabled when the formula assignment makes the control flow not cover (not traverse) the clause. We illustrate clause enabling/disabling in Figure 5.1 where c_1, c_2, c_3 and c_4 are clauses and c_2 is a control predicate. The current assignment of the variables in these clauses makes the control flow traverse c_1, c_2 and c_4 (plain line) but not c_3 (dashed line). We say that c_1, c_2 and c_4 are enabled and c_3 is disabled.

When clauses are disabled they do not need to be satisfied because they do not contribute to the calculation of the program output, and hence have no effect of the program's post condition (specification).

The solver used in AllDiagnoses algorithm can choose outcomes of control predicate by finding appropriate variable assignments. Therefore, the solver can *enable* or *disable* some clauses depending on these assignments. In a way, and only for the case of full flow-sensitive trace formula, the solver can *virtually relax* clauses just by changing control predicate values. The latter is possible because the FFTF encodes all potential program paths. Therefore, the solver can *explore* paths in the trace formula.

When extending the FFTF with an error-inducing input formula and a specification formula, it become unsatisfiable. The path induced by the error-inducing input is enabling clauses, which are together inconsistent with the specification formula. In order to restore the consistency, the solver has to relax clauses of the formula. There are two possible situations when the solver relaxes clauses. (1) Some clauses are relaxed and the path stays the same. (2) Some clauses are relaxed and the path is deviated.

We saw how the solver used in the AllDiagnoses algorithm could enable and disable clauses, and deviate the control flow thanks to the encoding of program dependences in the FFTF. This is the representation of the program dependences in the FFTF that makes possible the localization of the data variable dependent faults, control variable dependent faults, and independent faults.

5.5 Experiments on the TCAS Benchmark

In this section we show the capabilities of SNIPER with the FFTF with some experiments made on the Siemens Test Suite.

5.5.1 TCAS Benchmark

One of the Siemens Test Suite tasks is the TCAS (aircraft collision avoidance system), which is sometimes used in program testing research [43, 79]. The authors of the suite created, in addition to a correct program, 41 versions of the program and in each of these versions one or more faults were injected. The TCAS task comes with a set of 1578 test cases. However, no specification is given.

5.5.2 Experimental Setup

We used the same experimental setup as described in [47]. We first ran the original program on the test cases in order to get the correct output values for each test case. These values constitute the test oracles for the program. As explained in Section 4.5 we use many error-inducing inputs (failing test cases) in order to deal with multi-fault

programs. For the purpose of this experiment on the TCAS benchmark, we ran all test cases on each faulty version to obtain the failing test cases. These are the test cases that give an output different from the correct output.

All the experiments were carried out using an Intel Core 2 Duo 2.4 GHz with 4 GB of RAM on the operating system Mac OS X 10.6 Snow Leopard.

5.5.3 Results for Single and Multiple Faults

Table 5.3 reports the results of running SNIPER with the FFTF on each version of the TCAS. The first column of the table shows the version of the program. The column #Err shows the number of injected fault in this version. The column Error Type shows the type of bug injected, which is explained in Table 5.4, which comes from [47]. The column #FTC shows the number of failing test cases included in the TCAS benchmark set. The right part of Table 5.3 shows the results of SNIPER and BugAssist. The results of BugAssist were taken from [47]. Each column shows the number of time the tools were able to detect at least one of the injected fault locations.

In total, BugAssist pin-pointed 1364 times the injected fault location out of the 1437 runs (73 misses). SNIPER pin-pointed the injected fault location 1435 times out of the 1437 runs (2 misses). The average ACSR (Average Code Size Reduction), which is the percentage of code given by the tool on average to locate the faults, of all the versions is 11.00%. For recall, CSR (Code Size Reduction) is the ratio of fault locations in a MUS (program slice) to the total number of lines of code (Section 3.4.3). We obtain a minimum of 2.31% for the version no. 14 and a maximum of 14.01% for the version no. 10. SNIPER was able to identify the exact bug location of all the single fault programs.

Concerning the multi-fault programs, all the faults that were able to be found with the given test cases were successfully localized. In the version no. 31, the TCAS test cases cover one of the two buggy statements only. Thereby, the uncovered buggy statement cannot be in the root causes. This shows that the coverage of the test input is an important factor in fault localization.

5.5.4 Results with Different Formula Granularity Levels

As explained in Section 5.3.4, depending on the trace formula granularity level, the computing time can greatly vary, as well as the precision of the localization. Figure 5.2 reports the computation times of SNIPER on the TCAS benchmark with the FFTF constructed in different granularity levels. The histograms are separated in two parts for readability. The bars in black represent the times with the block-level granularity, the bars in gray represent the times with the line-level granularity, and the bars in white represent the times with the instruction-level granularity.

We can see that the computation time is the best with the block granularity level. The worst computing time is obtained with the instruction granularity level, which is the most fine-grained. We see in this experiment that a coarse-grained granularity enables the localization algorithm to compute diagnoses fast. Oppositely, with a finer-grain granularity level it takes a longer time to complete. This is because the execution time is more or less dependent on the number of clauses to enumerate. There is a compromise to decide which granularity level is appropriate in view of the execution time and precision. Of course, running the localization algorithm on a trace formula with a fine-grained granularity level results in more precise diagnoses as compared with a trace formula with a coarse-grained granularity level.

To understand why a coarse-grained granularity enables a faster enumeration of MCSes than with a fine-grained granularity, let us consider again the example of Section 5.3.4 and assume that each clause c^i_{l,b_j} is an MCS. Running `AllMinMCS(`φ^{inst}_{TF}`)` requires 7 calls to the pMaxSMT solver. At each of these calls, one clause is relaxed by the solver and then *blocked* until no more clauses can be relaxed (see Section 4.4 for details). Running `AllMinMCS(`φ^{line}_{TF}`)` requires 4 calls to the pMaxSMT solver. And running `AllMinMCS(`φ^{block}_{TF}`)` requires 3 calls to the pMaxSMT solver. Since the total running time of our fault localization method is strongly correlated with the number of calls to the pMaxSMT solver, the more calls to the solver, the more time it takes to localize faults.

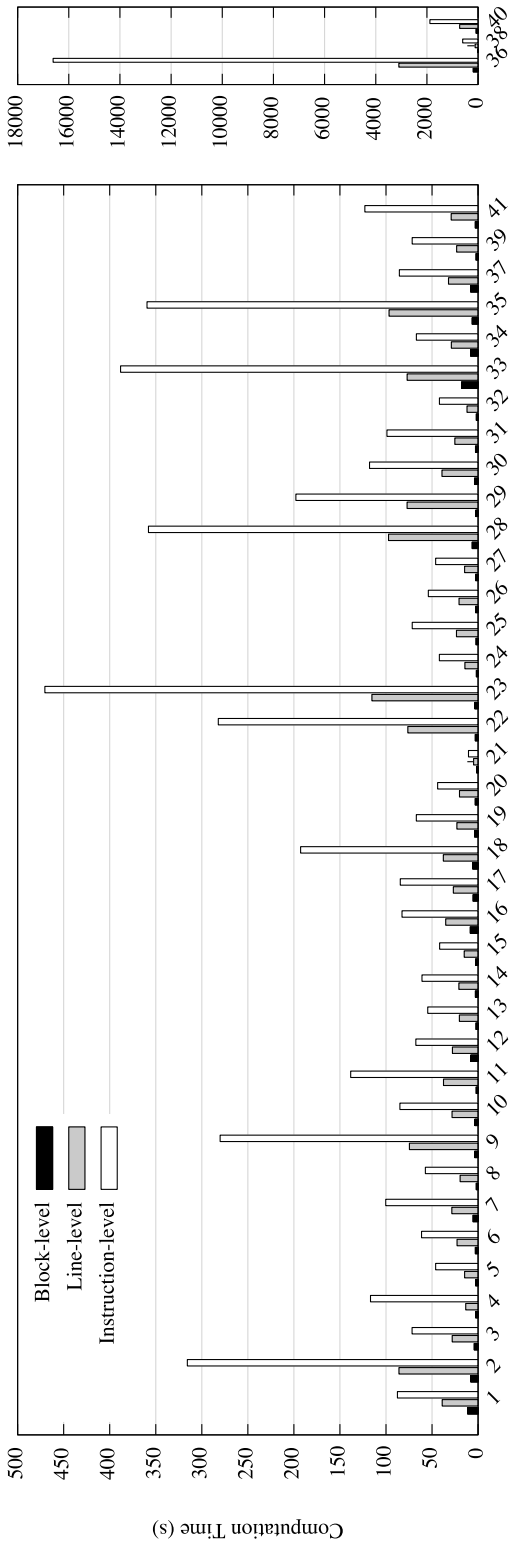


Figure 5.2: Results of running SNIPER on the TCAS benchmark with different granularity levels for the FFTF

5.6 Experiments on the Bekkouche's Benchmark

In the previous section we showed some experiment on the TCAS benchmark. The TCAS program is mostly composed of comparisons. All programs are the same except that the injected faults are different. Since a fault localization performs differently depending on the program natures, we would like to experiment our method on different kind of programs. In this section we show how SNIPER with the FFTF performs on different kind of programs. We made some experiments on a benchmark provided by Bekkouche [7, 8]. We chose the Bekkouche benchmark because it is composed of various kinds and sizes of programs. Furthermore, it includes programs with arithmetics, and the faults in these programs were injected specifically for experimenting automatic fault localization methods, as opposed to TCAS that was originally designed for the research on program testing methods.

The version of SNIPER used herein uses LLVM version 3.3 and Yices version 1.0.39. All the experiments were carried out using an Intel Core 2 Duo 2.4 GHz with 4 GB of RAM on the operating system Mac OS X 10.6 Snow Leopard.

5.6.1 Bekkouche's Benchmark

The Bekkouche's benchmark [7, 8] consists of several C programs of 15 to 180 lines of code. They contained only pre- and post-conditions on inputs and outputs with constant values. We modified the programs by removing these pre- and post-conditions and adding complete specifications under the form of post-conditions on program outputs.

5.6.2 Results for Single and Multiple Faults

Table 5.5 lists the results obtained by running SNIPER with the FFTF on each benchmark program [7, 8]. The first column of the table shows the program name. The column #EI shows the number of error-inducing inputs generated by a concolic unit testing engine implemented in SNIPER. The Found column lists the number of faults that SNIPER was able to localize versus the total number of faults injected in

the program. The Time column lists the total running time of SNIPER in milliseconds, including the bitcode loading time and bitcode preprocessing time of LLVM, the concolic execution time, the formula encoding time, and the solving time.

For all programs, SNIPER with the FFTF was able to localize all faults. The multi-fault program `Maxmin6varKO3` contains a nested fault, which is a fault that is masked by other fault. This type of fault (control-dependent faults in Figure 2.2) are especially difficult to deal with because it is impossible to generate a test case that covers the masked fault. Coverage-based or spectrum-based debugging methods, such as Tarantula [46], are unable to locate the second fault in the program `Maxmin6varKO3`. In addition, the formula-based approach with other encodings than the full flow-sensitive TF cannot identify the masked fault either. Below, we discuss how the full flow-sensitive TF can successfully deal with such faults.

```
1  [...]
2  if ((a>b) && (a>c) && (b>d)  && (a>e) && (a>f)) {
3    max = a;
4    if ((b<c) && (c<d) && (b<e) && (b<f)) {
5      min = b;
6    } [...]
7  } [...]
```

Listing 5.1: Code fragment from program Maxmin6varKO3 showing the two faults (underlined)

Listing 5.1 shows a code fragment of program `Maxmin6varKO3` showing both faults. The underlined condition in line 2 should be `(a>d)` and the underlined condition in line 4 should be `(b<d)`. Because of the fault in line 2, no failing executions can pass through the fault in line 4.

From a practical point of view, the masked fault does not exist (it does not corrupt the program) when the masking fault exists. By fixing the masking fault, we activate the masked fault. In a sense, we introduce indirectly and unintentionally a new fault in the program.

From a view point of MCS enumeration with pMaxSAT (see Chapter 6 for details), the algorithm explores for each test input the search space by relaxing minimal sets

of clauses. Given a test input T, if the algorithm finds and relaxes clauses related to the fault in line 2 for T, it is equivalent to say that the fault in line 2 was temporally removed. Therefore, in the next steps of the exploration the algorithm can find clauses related to the fault in line 4 for the same test input T because it is now not hidden anymore by the fault in line 2.

Note that the search is only possible with full flow-sensitive TF, which is equivalent to the program CFG in our encoding method (see Section 5.3), because it encodes alternative paths for failing executions, which means that the solver (used in Algorithm 8) can relax clauses to force the control flow to take an alternative path that covers both faults.

5.6.3 SNIPER vs. BugAssist

In this experiment, we used the original pre- and post-condition provided in the benchmark [8].

All the experiments were carried out on a virtual machine KVM/QEMU using 4 QEMU Virtual CPUs with 4 cores running at 2.6 GHz with 4 GB of RAM on Linux CentOS release 6.4.

The version of SNIPER used herein uses LLVM version 3.4 and Yices version 1.0.39. We used Clang version 3.4 as a frontend for generating IRs from the C programs. We used BugAssist [47] version 0.1 with MSUnCore2 version 0.5 as a backend solver as suggested on the BugAssist's website.

For this experiment, the error-inducing inputs in SNIPER are computed with the BMC module. Note that both SNIPER and BugAssist here use the same error-inducing inputs since all programs have pre-conditions that require the inputs to be equal to particular constant values (input values that trigger a failing execution).

Table 5.6 lists the results of running SNIPER and BugAssist on the Bekkouche's benchmark [7, 8]. The first column of the table shows the program name. The Found column lists the number of faults that the tools were able to localize versus the total number of faults injected in the program. The Time column lists the total running time in seconds. The times for SNIPER also include the execution time of Clang.

The average computing time is 0.184 seconds with SNIPER and 1.292 seconds with BugAssist. SNIPER pin-pointed all the injected fault location for both single fault and multi-fault programs. BugAssist missed 7 fault locations out of the 22 faults.

Concerning programs with multiple faults, because BugAssist outputs only a flat list of fault locations, it is difficult for the user to find in the program more than one fault. Thanks to a diagnoses combination method, programs with multiple faults are handled by SNIPER in a way that it is easy for the user to read and use the result. For example, in the program `Maxmin6varKO3` there are two faults located in lines 24 and 26. On this program, SNIPER outputs the set {{24, 26}, ...} and BugAssist outputs the set {..., 24, 26, ...}.

5.7 Discussion

We are going to present the characteristics of FFTF in view of the failure model we assume. First, as explained in Section 2.8, we are focusing herein on faults in the program's calculations. Second, the *program structure* is assumed correct.

In our case, the structure of the IR is made of edges connecting all basic blocks together. These edges are in fact unconditional branching instructions, conditional branching instructions, and phi instructions. In the construction of the FFTF, these instructions are all encoded as hard (non-relaxable), which follows our assumption in that these instructions are absolutely correct. As for the calculation part of the IR, it is made of comparison instructions (`icmp`) and binary instructions (`add`, `sub`, ...). These instruction, which calculates new data values, are encoded as soft (relaxable) in the FFTF. These are the locations in which we look for a possible potential fault. SNIPER looks for potential bugs in these locations marked as soft.

In the following, we discuss the ability of the FFTF to localize multiple faults in a program. In formula-based fault localization, root causes are identified by determining the clauses of the formula, which when relaxed, will explain the discrepancy between the observed and correct program behavior.

SNIPER with the FFTF is able to localize multiple faults in a program thanks to

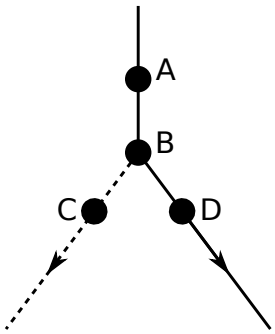


Figure 5.3: A failing execution path

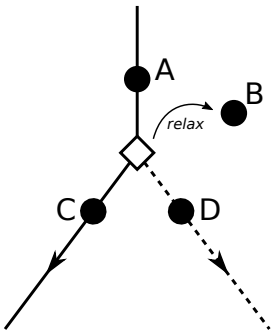


Figure 5.4: Execution path after the clause B was relaxed

the following points:

1. It uses a fault localization algorithm that works on many error-induction inputs, which may exhibit different erroneous situations of the program.

2. SNIPER uses a full diagnoses enumeration algorithm. In some work, for example [47], fault localization is done with a partial enumeration algorithm because of high computation costs. Another approach is to enumerate MCSes up to a certain size [95]. These methods are not exploring all the search process.

3. As explained in Section 5.4, FFTF encodes alternative paths to the failing path. In some situation, the relaxation of some clauses makes the control flow deviate and take those alternative paths. The solver can also relax clauses in these alternative paths. Hence, in case of *control variable dependent faults* or *independent faults*, the solver is able to relax a faulty clause, which is not on the target failing path. Figure 5.3 and 5.4 illustrate the deviation of the control flow when a clause is removed.

In the debugging of VLSI designs, Safarpour et al.. [80] discussed the concept of *error group cardinality*. By grouping clauses related to the same gate, they reduce the complexity of the debugging problem, while maintaining completeness. We apply this concept (Section 5.3.4 and 5.5.4) to reduce the complexity of enumerating root causes. Grouping all clauses derived from the same basic block/line together allows

the pMaxSAT solver to *block* all of those clauses simultaneously. In effect, this gives the solver the ability to treat each basic block/line as a single high-level constraint, leading to solutions (MCSes) found directly in terms of the basic block/line.

The main difference between the formula-based fault localization of VLSI designs and the formula-based fault localization of imperative program is the number of execution paths. We define an execution as a run of a program/circuit with a particular input. An execution path is an execution with a particular series of branching decisions. A VLSI design does not contain any branches, so we can say that there is only a single execution path possible and many possible executions (in term of valuation of variables). A typical imperative program contains many statements that make the control flow jump. Such statements are if-then-else, switch-case, goto, and return. Depending on the program input values, the control flow takes different execution paths. In trace formulas, it is important to encode this behavior, especially in the case of faults in the control variables. When alternative paths to the failing path are not encoded, faults in the control variables cannot be identified.

Characteristics of the different trace formula encodings are summarized in Table 5.7. Despite the fact that flow-insensitive and flow-sensitive trace formulas are small in size and thus can be efficient from a computing time viewpoint, they are not appropriate when dealing with multi-fault programs. We saw in this chapter that for at least two of the benchmarks we used, the full flow-sensitive trace formula could be used for localizing root causes in multi-fault programs. This is because the full flow-sensitive trace formula encoded the CFG in SSA form of the target programs.

5.8 Summary

In this chapter we introduced the full flow-sensitive trace formula whose encoding is equivalent to the control flow graph of the faulty program. This trace formula together with a complete enumeration algorithm can localize root causes in multi-fault programs, at least for the benchmark programs used in our experiments. However, in terms of computation time, the full flow-sensitive trace formula is not appropriate. This is because it is very expressive and therefore, complex. To deal with this, in

the next chapter we propose an optimized algorithm for localizing the root causes. Additionally, in Chapter 7 we propose a new trace formula more adequate in view of computation time.

Ver	#Err	Error Type	#FTC	SNIPER	BugAssist
v1	1	op	131	131	131
v2	1	const	69	69	69
v3	1	op	23	23	13
v4	1	op	25	**24**	25
v5	1	assign	10	10	10
v6	1	op	12	12	12
v7	1	const	36	36	36
v8	1	const	1	1	1
v9	1	op	9	9	9
v10	2	op	14	14	14
v11	2	op	14	14	14
v12	1	op	70	70	48
v13	1	const	4	4	4
v14	1	const	50	50	50
v15	3	const	10	10	10
v16	1	init	70	70	70
v17	1	init	35	35	35
v18	1	init	29	29	29
v19	1	init	19	19	19
v20	1	op	18	18	18
v21	1	op	16	16	16
v22	1	code	11	11	11
v23	1	code	42	42	41
v24	1	op	7	7	7
v25	1	code	3	3	3
v26	1	addcode	11	11	11
v27	1	addcode	10	10	10
v28	1	branch	76	76	58
v29	1	code	18	18	14
v30	1	code	58	58	58
v31	2	addcode	14	14	14
v32	2	addcode	2	2	2
v34	1	op	77	77	77
v35	1	code	76	76	58
v36	1	op	126	126	126
v37	1	index	92	92	92
v39	1	op	3	3	3
v40	2	assign	126	126	126
v41	1	assign	20	**19**	20

Table 5.3: Results of SNIPER with the FFTF and BugAssist on the TCAS. Versions no. 33 and no. 38 are omitted from the table in order to compare the results with BugAssist [47], which does not have entries for them. For versions no. 4 and no. 41, a new option of SNIPER that checks the array index overflow/underflow can detect the missing fault.

Error Type	Explanation of the Error
macro	Wrong value in a macro definition
op	Wrong operator usage.
code	Logical coding bug.
assign	Wrong assignment expression.
addcode	Error due to extra code fragments.
const	Wrong constant value supplied.
init	Wrong value initialization of a variable.
index	Use of Wrong array index.
branch	Error in branching due to negation of branching condition.

Table 5.4: Types of Error in the TCAS programs

Programs	**#EI**	**Found**	**Time (ms)**
MinmaxKO	2	1/1	35
AbsMinusKO	1	1/1	27
AbsMinusKO2	3	1/1	38
AbsMinusKO3	1	1/1	22
TritypeKO	1	1/1	373
TritypeKO2	2	1/1	380
TritypeKO2V2	2	1/1	367
TritypeKO3	2	1/1	542
TritypeKO4	1	1/1	236
TritypeKO5	8	2/2	622
TriPerimetreKO	1	1/1	430
TriPerimetreKOV2	1	1/1	583
TriPerimetreKO2	1	1/1	332
TriPerimetreKO3	2	1/1	656
Maxmin6varKO	37	1/1	30872
Maxmin6varKO2	56	1/1	24365
Maxmin6varKO3	56	2/2	26731
Maxmin6varKO4	61	3/3	32592

Table 5.5: Results of running SNIPER on the Bekkouche's benchmark.

	SNIPER		**BugAssist**	
Programs	**Found**	**Time(s)**	**Found**	**Time(s)**
MinmaxKO	1/1	0.036	1/1	0.146
AbsMinusKO	1/1	0.034	1/1	0.065
AbsMinusKO2	1/1	0.034	**0/1**	0.118
AbsMinusKO3	1/1	0.033	**0/1**	0.109
TritypeKO	1/1	0.122	**0/1**	0.607
TritypeKO2	1/1	0.078	1/1	1.186
TritypeKO2V2	1/1	0.057	1/1	1.184
TritypeKO3	1/1	0.072	1/1	1.118
TritypeKO4	1/1	0.091	1/1	0.533
TritypeKO5	2/2	0.069	**1/2**	0.521
TriPerimetreKO	1/1	0.143	**0/1**	1.349
TriPerimetreKOV2	1/1	0.223	1/1	2.468
TriPerimetreKO2	1/1	0.095	1/1	4.060
TriPerimetreKO3	1/1	0.090	1/1	2.278
Maxmin6varKO	1/1	0.428	1/1	2.115
Maxmin6varKO2	1/1	0.416	1/1	1.379
Maxmin6varKO3	2/2	0.642	2/2	2.457
Maxmin6varKO4	3/3	0.644	**1/3**	1.554

Table 5.6: Results on the Bekkouche's benchmark

Trace Formula Type	**Authors**	**Encoding**	**Formula Size**	**Construction Method**
Flow-insensitive	BugAssist [47] *2011*	Counterexample	Very small	BMC or testing
Flow-sensitive	Christ et al. [17] *2013*	Counterexample + Partial control flow	Small	Static analysis
Full Flow-sensitive	Lamraoui et al. [53] *2014*	CFG / SSA	Large	Static analysis
Spectrum-based	Tarantula [46] *2005*			Testing

Table 5.7: The types of trace formula used in formula-based fault localization and their specificities

Chapter 6

Incremental Diagnosis Enumeration

6.1 Introduction

Diagnoses computation is a key point in the formula-based fault localization method for imperative programs. A diagnosis is an MCS that represents a root cause of the faulty program (see Section 3.4.1 for details). Software engineers use these root causes to repair faulty programs.

A set of diagnoses is computed from a trace formula extended with a formula representing the error-inducing input and a formula representing the program specification. Expressive trace formula, such as the full flow-sensitive trace formula (see Chapter 5 for details), contains a large number of clauses and are complex. This is leading most of the time to large computation times when computing diagnoses. For the case of imperative programs, we have many different trace formulas on which we want to enumerate MCSes. Each of these formulas refers to a particular erroneous situation, which are triggered by an error-inducing input. When dealing with multi-fault programs, the number of formulas can be large because there is a large number of failing executions. Not considering one of the failing execution can be a reason for missing faults. Using many failing test cases is more appropriate. We presented in Section 4.3 some methods to generate such test cases. A classic diagnoses enumer-

ation algorithm, such as the one presented in Section 4.4, is not efficient in view of computation time. This calls for a new method that efficiently enumerates MCSes from many error-inducing inputs.

In this chapter we present a method that computes diagnoses in an incremental fashion. We show how the method performs in an experiment on the TCAS benchmark.

6.2 Proposed Approach

In this section we describe our approach to efficiently compute diagnoses incrementally in the context of formula-based fault localization of imperative programs.

Algorithm 8 AllDiagnoses

Input: a set of error-inducing input formulas E, a trace formula φ_{TF} and a formula φ_{AS} that encodes the assertions the program must satisfy.
Output: D a set of diagnoses (MCSes), or $\emptyset$.

```
1: φ'_TF ← AddAuxVars(φ_TF)
2: if φ'_TF = ∅ then
3:     return ∅                                   ▷ No pMaxSMT solution
4: end if
5: D ← ∅
6: C ← φ'_TF ∪ (φ_AS)^HARD                        ▷ Add the base formulas in the context
7: for each φ_ei ∈ E do
8:     push(C)                                    ▷ Save the context
9:     C ← C ∪ (φ_ei)^HARD                        ▷ Add the error-inducing input formula in the context
10:    M ← AllMinMCS(C)                           ▷ Enumerate all minimum size MCSes for the context C
11:    if M ≠ ∅ then
12:        D ← D ∪ {M}
13:    end if
14:    pop(C)                                     ▷ Restore the context (pushed in line 8)
15: end for
16: return D
```

Algorithm 8 describes the enumeration of all the minimum size MCSes for a given trace formula and a set of error-inducing inputs. The MCSes are to be enumerated for all error-inducing inputs. This algorithm makes use of the *push & pop* mechanism of Yices 1 [28] (see Section A.2). The `push` operation saves the current logical context on the stack. The `pop` operation restores the context from the top of the stack, and pops it off the stack. Any changes to the logical context (adding or retracting

assertions) between the matching push and pop operators are flushed, and the context is completely restored to what it was right before the push operation. This mechanism is useful in our method because we apply many small modifications (lines 19 and 30) to the context C. It does not need to create a completely new context between the calls to the solver. We can just flush the local modifications and reuse the same context basis many times.

6.3 Experiments on the TCAS Benchmark

In this section we show with some experiments on the Siemens Test Suite the performance gain obtained with our incremental diagnosis enumeration method. We used the same experimental setup described in Section 5.5.2.

6.3.1 Results

Figure 6.1 reports the computation times of Algorithm 8 on the TCAS benchmark with and without the *push & pop* optimization, which was explained in Section 6.2. The histograms are separated in two parts for readability. The bars in gray represent the times with the optimization disabled, which correspond to the basic algorithm presented in Section 4.4, and the bars in black represent the times with the optimization activated, which correspond to the algorithm presented in this chapter.

We can see that the computation time is reduced when using the optimization technique. The percentage decrease of the average computation time is 49%. The large difference can be explained by the fact that the same formula is solved many times with only small modifications between the calls to the solver. The benefits of the optimization are particularly noticeable when the number of error-inducing inputs is large. This is explained by the fact that the number of context modifications is directly impacted by to the number of error-inducing inputs used.

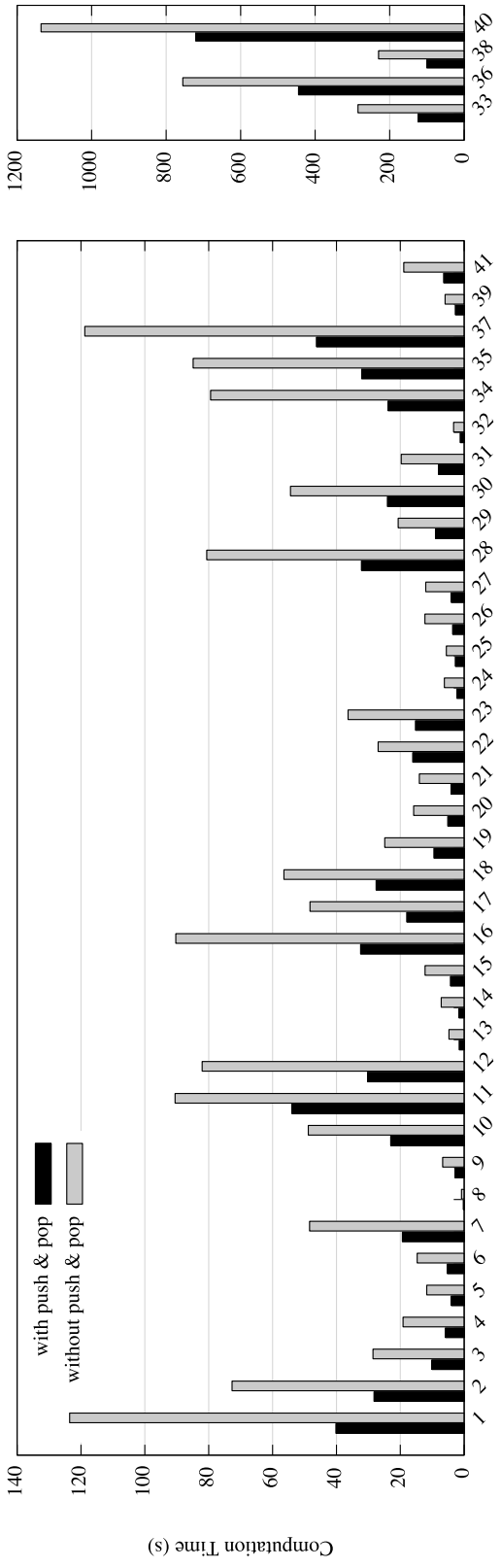


Figure 6.1: Results of running SNIPER on the TCAS benchmark with and without push & pop optimization.

6.4 Summary

This chapter presented an efficient algorithm for enumerating diagnoses in the context of formula-based fault localization of imperative programs. The algorithm uses an incremental solving optimization technique. We implemented the algorithm in SNIPER, and have shown a significant efficiency gain on the TCAS benchmark.

We note that for the case of multi-fault programs, the generated MCSes are corresponding to many faults, and thus have to be combined to have sets that includes root causes of all faults. We, indeed, proposed some combination methods in Section 4.5 to answer this problem.

Another problem we would like to highlight is that even if the algorithm presented in this chapter could reduce the computation time, it is still costly to compute diagnoses. In the next chapter we present a new trace formula that aim at reducing further the computation time of our fault localization method.

Chapter 7

Hardened Flow-sensitive Trace Formula

7.1 Introduction

In Chapter 5, the full flow-sensitive TF was introduced. The full flow-sensitive TF is equivalent to the control flow graph (CFG) of the entire program. It is composed of clauses, which are marked as *soft* (relaxable) or *hard* (non-relaxable). All program statements in the full flow-sensitive TF are encoded as *soft*, which means that all these statements (values computation) are suspicious. As of the *program skeleton*, it is encoded as *hard* because it is assumed to be correct. The full flow-sensitive TF, together with a complete enumeration algorithm of potential root causes, can identify faults in the multi-fault programs of the TCAS benchmark [43, 79]. The full flow-sensitive TF, however, tends to be large and is less adequate in view of efficiency of the localization algorithm.

In this chapter, we propose a new representation of the TF, called a *hardened flow-sensitive trace formula* (HFTF). We introduce a modest assumption that, for each error cause in a multi-fault program, there exists at least one test input (an error-inducing input) to result in such faulty behavior. These test inputs may be obtained efficiently by bounded model-checking or program testing methods. The HFTF is similar to the full flow-sensitive TF, but reduces the number of *soft* clauses

in the formula by marking as *hard* clauses related to instructions that are not in a multi-execution dice. An execution dice [2] is a set difference between a single failing execution slice and a single successful execution slice. A multi-execution dice (MED), proposed in this thesis, is constructed from a set of failing execution slices and successful execution slices. An MED is more adequate for multi-fault programs because it takes into account many failing executions, each of which referring to a particular fault. Since the root causes are considered to lie in the MED, we can still localize such potential root causes. Reducing the number of soft clauses in the formula can improve the efficiency of the localization algorithm without losing the essential CFG information required for localizing faults. Indeed, the less soft clauses to enumerate, the faster is the localization algorithm. The decision procedure for *hard*-marking clauses of the HFTF uses the coverage information obtained using a concolic (CONCrete and symbOLIC) execution method [84].

The remainder of this chapter is organized as follows. Section 7.2 presents our approach. Sections 7.3 and 7.4 report some experiments and discussion. The last section contains a critical analysis of the proposed encoding and a comparison to other approaches.

7.2 Proposed Approach

This section describes in detail the proposed approach to formula-based fault localization using the hardened flow-sensitive trace formula.

7.2.1 Hardened Flow-sensitive Trace Formula

We introduce the hardened flow-sensitive trace formula (HFTF), which is constructed from the CFG $P = (B, l, e, T)$ of a preprocessed program in SSA form [22]. The construction is based on information collected from executions of the target program. Section 7.2.2 details how we obtain these executions.

Definition 10 (Execution Slice) *An execution slice of a program P is a tuple $s = (\pi, F, o)$ where π are program input values that trigger s, F is a finite directed path $F = (b_1, b_2, \ldots, b_n) \in B \times B \times \ldots \times B$ such that $(b_i, b_{i+1}) \in T$ with $b_1 = l$ and $b_n = e$ for $1 \leq i < n$, and $o \in \{\text{PASS}, \text{FAIL}\}$ is the post-condition outcome. If o is* PASS*, s is a successful execution slice. Otherwise, s is a failing execution slice.*

The concept of *execution slice* was originally introduced in [1]. As in Definition 10 an execution slice is a set of basic blocks executed by a test input. We make a distinction between failing execution slices and successful execution slices depending on the post-condition outcome.

Agrawal et al. [2] states that a fault lies in the execution slice of a test case that fails on execution. The rest of the program can thus be ignored while searching for the fault associated to this test case, only the statements in the failing execution slice are worth considering. Then, the work of Agrawal [2] proposes a method to further narrow down the search by focusing on the execution dice. An *execution dice* is referred as a set of basic blocks in one failing execution slice that does not appear in a successful execution slice. In this thesis, we extend the notion of dice by considering many failing executions instead of just one, which we call *multi-execution dice* (MED). The following defines operationally an MED for a given program P and a given set of execution slices.

Given a set $S = \{s_j\}$ of m execution slices with $0 \leq j < m$, we partition executed basic blocks into two sets. F_j refers to a sequence of basic block executed in j-th test run. The set of basic blocks that are executed at least once in a successful execution is

$$SB = \{b \mid b \in F_j \wedge (o_j = \text{PASS}) \wedge \exists j \in \mathbb{Z} : 0 \leq j < m\}$$

The set of basic blocks that are executed at least once in a failing execution is

$$FB = \{b \mid b \in F_j \wedge (o_j = \text{FAIL}) \wedge \exists j \in \mathbb{Z} : 0 \leq j < m\}$$

Basic blocks of P are marked as either *correct basic blocks* (CB) or *potentially infected basic blocks* (PB). The set PB of potentially infected basic blocks is an MED.

A correct basic block is a basic block that is executed only in successful program executions. The set CB is calculated as follows:

$$CB = SB \setminus FB$$

A potentially infected basic block is a basic block that is not a correct basic block, meaning that it is executed at least once in a failing execution or is never executed at all. The set PB is calculated as follows:

$$PB = B \setminus CB$$

We assume that basic blocks in CB are free from faults because none of the failing executions go through these basic blocks. Therefore, the instructions in the basic blocks of CB do not need to be root cause candidates in the HFTF. As explained in Section 3.1.4, in pMaxSAT, the clauses of the formula to be solved are either set as *hard* (non-relaxable) or *soft* (relaxable). For the case of the HFTF, we use the same setting as the FFTF, except for the basic blocks in CB. We set the constraints related to the instructions in the basic blocks of CB as hard because the instructions related to this part are assumed not to be suspicious. As compared with the FFTF, this encoding reduces the number of soft clauses, which means that there are fewer MCSes to enumerate[1]. In an extreme case, however, all instructions are executed in the failing executions, which means that no simplification is possible and that the HFTF is the same as the FFTF.

Given a program P, a set of potentially infected basic blocks PB of P, and a set of correct basic blocks CB of P, the complete HFTF encoding is defined as follows:

$$\varphi_{HFTF} = \underbrace{\varphi_{FFTF(PB)}}_{\text{soft/hard}} \wedge \underbrace{\varphi_{FFTF(CB)}}_{\text{hard}}$$

where $\varphi_{FFTF(PB)}$ is a full flow-sensitive TF of the basic blocks marked as *potentially infected basic blocks*, and $\varphi_{FFTF(CB)}$ is a full flow-sensitive TF of the basic blocks marked as *correct basic blocks*. The formula φ_{HFTF} is equivalent to φ_{FFTF} (Section 5.3) except that the clauses in $\varphi_{FFTF(CB)}$ are now all set to be hard (non-relaxable).

[1]The missing MCSes are believed to be corresponding to spurious root causes.

7.2.2 Calculating Hardened Flow-sensitive Trace Formula

As discussed in Section 7.2, the HFTF is constructed from executions of the target program. These executions are used for deciding which parts of the HFTF should be set as hard. In Section 4.3 we presented different methods to automatically generate test cases. One of them, concolic execution, allows the automatic generation of successful and failing executions with a high code coverage. In this section we explains how we calculate the HFTF using concolic execution.

Concolic Execution for Calculating the HFTF

We can generate both failing and successful executions by augmenting the path constraint PC with the program post-condition (φ_{AS}). If it is satisfiable, the obtained input values will trigger a successful execution. Similarly, failing executions are obtained by augmenting PC with the negation of the program post-condition ($\neg\varphi_{AS}$). Note that it is not always possible to choose the post-condition outcome of the execution in advance. If a variable present in φ_{AS} does not appear as a symbolic variable in PC, then there is no *connection* between the program inputs and the post-condition. In this case, we solve the PC alone, execute the program, and, at the end of the execution, check whether the output value of the program satisfies φ_{AS}.

In order to calculate the *correct basic blocks* and the *potentially infected basic blocks* we need to profile basic blocks, namely monitoring executions of basic blocks. While executing the program (or after the execution) in the concolic execution method, we mark basic blocks as *failing* and/or *successful*. Basic blocks marked as *failing* are the blocks executed at least once in failing executions. Basic blocks marked as *successful* are the blocks executed at least once in successful executions.

Since a program may have a large number of paths, we cannot generate all possible paths in general. In order to tackle this problem, we use the *bounded depth-first search* strategy as introduced in [84]. The strategy prevents an exhaustive search of the entire program by restricting the symbolic execution up to a certain program depth. For cases in which not all of the paths are covered, the number of soft clauses in the resultant HFTF is not minimum because some successful paths are not covered. Even

if only some successful paths are covered, the number of soft clauses in the HFTF can be smaller than the ones in the FFTF because these paths may traverse some basic blocks not covered by failing executions. We discuss in more details the impact of the search strategy on the HFTF in Section 7.5.

7.2.3 Fault Localization with Hardened Flow-sensitive Trace Formula

Algorithm of Fault Localization using HFTF

Algorithm 9 shows the main algorithm for fault localization using the HFTF. The algorithm first preprocesses the program (line 1) as explained in Section A.3. In line 2, the concolic execution method generates a set of error-inducing inputs (E) and a profile of the program (R). The profile holds basic blocks information (marks) for later simplifying the trace formula. From the preprocessed program and the profile, the HFTF is constructed (line 3). In line 4, the diagnosis is computed using Algorithm 4 (Section 4.4.2). Finally, the diagnosis is processed to be displayed to the user.

Algorithm 9 LocFaults

Input: a program P, and a formula that encodes the post-condition φ_{AS}.
1: $P' \leftarrow$ `PreProcess(`P`)`
2: $\{E, R\} \leftarrow$ `ConcolicExecution(`P', φ_{AS}`)`
3: $\varphi_{HFTF} \leftarrow$ `Encode(`P', R`)`
4: $D \leftarrow$ `AllDiagnoses(`E, φ_{HFTF}, φ_{AS}`)`
5: `Display(`D`)`

7.2.4 Example

We illustrate our approach with the failing program presented in Section 2.8.1 (Listing 2.1). We show its CFG in Figure 7.1 (the assert statement and the call to function `opMotor` are not represented in the CFG). An error-inducing equal to -1 can be used to trigger a failing execution. A successful execution can be triggered with an input value equal to 1.

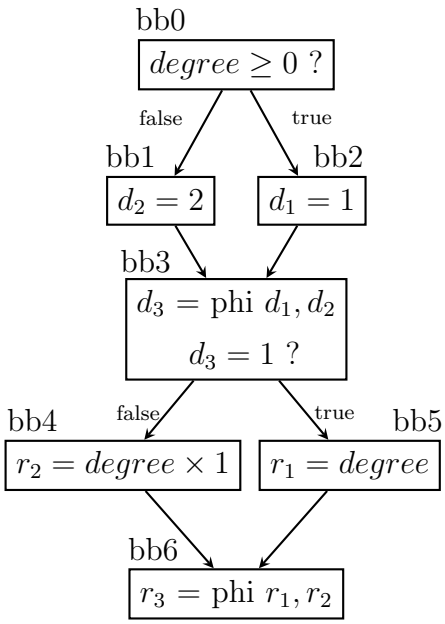


Figure 7.1: Control flow graph of function `rotate` (Listing 2.1)

From the two executions, we obtain the basic block profile below.

$$SB = \{\text{bb0}, \text{bb2}, \text{bb3}, \text{bb5}, \text{bb6}\}$$

$$FB = \{\text{bb0}, \text{bb1}, \text{bb3}, \text{bb4}, \text{bb6}\}$$

From the profile, we calculate an MED (PB):

$$CB = SB \setminus FB = \{\text{bb2}, \text{bb5}\}$$

$$PB = B \setminus CB = \{\text{bb0}, \text{bb1}, \text{bb3}, \text{bb4}, \text{bb6}\}$$

Below, the HFTF for the function `rotate` using the MED calculated above.

$$\varphi^{\text{rotate}}_{HFTF} = \underbrace{(g_1 = (degree \geq 0)) \wedge (d_2 = 2) \wedge (g_2 = (d_3 = 1)) \wedge (r_2 = degree \times 1)}_{\text{soft}}$$

$$\wedge \underbrace{(d_3 = (\text{ITE } g_1\ d_1\ d_2)) \wedge (r_3 = (\text{ITE } g_2\ r_1\ r_2)) \wedge (d_1 = 1) \wedge (r_1 = degree)}_{\text{hard}}$$

We see that thanks to our approach, two out of six soft clauses could be hardened in the trace formula. When running Algorithm 4 on the above formula, we obtain

an MCSes of {{13}}. As a comparison, with the FFTF we would obtain a large MCSes of {{3, 11}, {7, 11}, {10, 11}, {13}}. Therefore, we can obtain more precise MCSes with HFTF than with FFTF. Performance improvement will be discussed with elaborated examples in the next section.

7.3 Experiments on the Bekkouche's Benchmark

In this section we show some experiments made on a benchmark provided by Bekkouche [8]. All of the experiments were carried out using a 2.6 GHz Intel Core i5 with 8 GB of RAM on Mac OS X 10.9 Mavericks. We herein use LLVM version 3.3 and Yices 1 version 1.0.39.

In this experiment, we focus on the comparison of the results obtained with the HFTF and with the FFTF. A detailed study of the FFTF, including a comparison with related work, was presented in Chapter 5, which showed that fault localization with the FFTF was superior to existing tools such as [47] and [95]. Therefore, the HFTF is superior to these related tools as well.

7.3.1 Full Flow-sensitive TF vs. Hardened Flow-sensitive TF

Table 7.2 lists the results on each benchmark program and for both types of TF encodings. The first column of the table shows the program name. The column #EI shows the number of error-inducing inputs found by the SNIPER concolic module. The #H and #S columns list the number of *hard* (non-relaxable) clauses and *soft* (relaxable) clauses in the formula respectively. The Found column lists the number of faults that SNIPER was able to localize versus the total number of faults injected in the program. The ACSR column lists the average code size reduction, which is the percentage of code given by the tool on average to locate the faults. The Time column lists the total running time in milliseconds, including the bitcode loading and preprocessing time of LLVM, the concolic execution time, the formula encoding time, and the solving time.

		Full Flow-sensitive TF					Hardened Flow-sensitive TF				
Programs	**#EI**	**#H**	**#S**	**Found**	**ACSR (%)**	**Time (ms)**	**#H**	**#S**	**Found**	**ACSR (%)**	**Time (ms)**
MinmaxKO	2	15	10	1/1	12.12	23	17	8	1/1	9.09	21
AbsMinusKO	1	23	7	1/1	10.93	19	24	6	1/1	3.12	14
AbsMinusKO2	3	23	8	1/1	13.70	23	23	8	1/1	13.70	22
AbsMinusKO3	1	23	7	1/1	13.79	14	23	7	1/1	13.79	14
TritypeKO	1	128	22	1/1	15.70	209	148	11	1/1	7.79	152
TritypeKO2	2	128	22	1/1	15.72	236	143	14	1/1	11.53	173
TritypeKO2V2	2	128	22	1/1	15.27	211	143	14	1/1	10.38	167
TritypeKO3	2	128	22	1/1	15.15	319	146	13	1/1	9.87	163
TritypeKO4	1	128	23	1/1	11.53	145	149	12	1/1	8.97	132
TritypeKO5	8	128	23	2/2	15.23	482	130	21	2/2	13.37	361
TriPerimetreKO	1	139	26	1/1	14.79	290	165	12	1/1	6.09	162
TriPerimetreKOV2	1	142	29	1/1	16.03	367	168	13	1/1	7.14	183
TriPerimetreKO2	1	139	26	1/1	14.85	192	163	12	1/1	10.84	165
TriPerimetreKO3	2	139	26	1/1	14.58	393	163	14	1/1	9.50	177
Maxmin6varKO	37	194	65	1/1	7.58	47741	287	8	1/1	1.98	6161
Maxmin6varKO2	56	194	65	1/1	5.47	13354	243	43	1/1	4.30	9823
Maxmin6varKO3	56	194	65	2/2	7.10	49276	243	43	**1/2**	4.16	9921
Maxmin6varKO4	61	194	65	3/3	7.35	515211	228	51	3/3	5.90	37650

Figure 7.2: Results on Bekkouche's benchmark with both types of encoding

The average ACSR of all programs is 12.61% with the FFTF and 8.42% with the HFTF. Since the soft part of the HFTF is in average smaller than the soft part of the FFTF, there are less root causes to enumerate, which explains the difference between the ACSRes obtained with each encoding.

The total running time is 628 sec with the FFTF and 65 sec with the HFTF. The total running time with the HFTF is about 10.14% of the case with the FFTF, which shows a good performance gain. For most programs, the computing times could be reduced. This is because some part of the formula could be hardened in the HFTF making the number of soft clauses in the HFTF smaller than the number of soft clauses in the FFTF. The smaller the number of soft clauses, the faster the computing is. For programs `AbsMinusKO2` and `AbsMinusKO3`, the HFTF is the same as with the FFTF, which means nothing was simplified. However, the computing time shows that the overhead of constructing the HFTF is negligible.

All faults could successfully be detected with the HFTF except in the program `Maxmin6varKO3`[2]. The program `Maxmin6varKO3` has one of its faults masking another fault. Listing 5.1 (Section 5.6.2) shows a code fragment of program `Maxmin6varKO3` showing both faults. The underlined condition in line 24 should be `(a>d)` and the underlined condition in line 26 should be `(b<d)`. Because of the fault in line 24, no failing executions can pass through the fault in line 26. Since some successful executions go through the second fault, the related instructions were encoded as hard in the HFTF. Thus, SNIPER with the HFTF cannot localize both faults. In a sense, the masked fault does not exist as long as the first fault is present in the program. After the first fault is corrected, the second fault becomes *active*.

In the case of the FFTF, all arithmetic and comparison instructions are encoded as soft, which means that the solver (used in Algorithm 4) can retract clauses to force the control flow to take a path that covers both faults. Note that the flow-sensitive TF [17] is unable to localize both faults simultaneously because it does not encode all paths [83]. This was discussed in Section 5.6.2.

[2]Section 5.6.2 also discussed this benchmark program.

7.4 Experiments on the TCAS Benchmark

In this section we show some experiments made on on the TCAS task of the Siemens test suite [43, 79].

This experiment aims to show how the proposed method with the HFTF behaves when a set of test suites is available in advance. Since the test suite of TCAS is not complete, these test cases may have an impact on the quality of the fault localization. Furthermore, using the provided test cases makes the comparison with existing tools clear.

7.4.1 Experimental Setup

For the purpose of this experiment on the TCAS task, since we already have a set of test cases, we deactivated the symbolic execution of the concolic execution module of SNIPER. This modified concolic execution module runs concretely the faulty program on each test case and check if the program output is different from the correct output (test oracle). For the case of HFTF, this modified concolic execution module also includes program profiling operations required for the construction of the HFTF, as in the original concolic execution module.

7.4.2 Full Flow-sensitive TF vs. Hardened Flow-sensitive TF

Table 7.4 reports the results on each version of the TCAS and for both encodings. The first column of the table shows the version of the program. The column #STC shows the number of successful test cases used. The column #FTC shows the number of failing test cases used. The #H and #S columns list the number of *hard* (non-relaxable) clauses and *soft* (relaxable) clauses in the formula respectively. The Found column lists the number of faults that SNIPER was able to localize versus the total number of faults injected in the program. The ACSR column lists the average code size reduction. The Time column lists the total running time of SNIPER in seconds, including the bitcode loading and preprocessing time of LLVM, the modified concolic execution module time, the formula encoding time, and the solving time.

			Full Flow-sensitive TF					Hardened Flow-sensitive TF				
Ver	**#STC**	**#FTC**	**#H**	**#S**	**Found**	**ACSR (%)**	**Time (sec)**	**#H**	**#S**	**Found**	**ACSR (%)**	**Time (sec)**
v1	1446	132	108	57	1/1	10.67	39.42	109	56	1/1	10.67	41.75
v2	1507	71	108	57	1/1	10.48	85.95	110	55	1/1	10.10	83.18
v3	1554	24	107	57	1/1	11.27	28.43	107	57	1/1	11.27	28.42
v4	1553	25	107	57	1/1	10.56	13.36	119	49	1/1	10.50	9.05
v5	1567	11	107	57	1/1	11.47	14.69	108	57	1/1	11.47	14.84
v6	1565	13	108	57	1/1	11.48	23.00	110	55	1/1	10.54	21.71
v7	1541	37	108	57	1/1	10.85	28.38	110	55	1/1	10.21	28.61
v8	1576	2	108	57	1/1	10.75	19.29	124	49	1/1	9.44	5.27
v9	1568	10	108	57	1/1	10.69	75.16	123	48	1/1	8.95	13.98
v10	1563	15	108	57	2/2	12.10	28.15	108	57	2/2	12.10	28.24
v11	1563	15	103	55	2/2	12.62	36.89	103	55	2/2	12.62	37.21
v12	1507	71	110	57	1/1	11.77	28.50	110	57	1/1	11.77	28.32
v13	1573	5	108	57	1/1	11.19	20.47	110	57	1/1	10.98	19.16
v14	1527	51	108	57	1/1	9.50	20.64	127	46	1/1	6.35	6.35
v15	1567	11	107	57	3/3	11.47	14.88	111	55	3/3	11.47	15.39
v16	1507	71	108	57	1/1	11.47	35.26	110	55	1/1	10.61	31.76
v17	1542	36	108	57	1/1	10.91	27.12	110	55	1/1	10.18	27.16
v18	1548	30	108	57	1/1	10.53	37.96	110	55	1/1	9.94	37.90
v19	1558	20	108	57	1/1	10.54	23.13	110	55	1/1	10.00	23.37
v20	1559	19	108	57	1/1	10.44	19.95	114	52	1/1	10.44	17.94

Figure 7.3: Results of running SNIPER on the TCAS benchmark with both types of encoding (part 1/2)

			Full Flow-sensitive TF					Hardened Flow-sensitive TF				
Ver	**#STC**	**#FTC**	**#H**	**#S**	**Found**	**ACSR (%)**	**Time (sec)**	**#H**	**#S**	**Found**	**ACSR (%)**	**Time (sec)**
v21	1561	17	104	56	1/1	10.08	4.98	116	47	1/1	10.08	4.59
v22	1566	12	104	56	1/1	10.01	76.18	115	49	1/1	8.25	41.01
v23	1536	42	104	56	1/1	11.06	115.07	125	44	1/1	7.80	6.92
v24	1570	8	104	56	1/1	10.63	14.33	117	49	1/1	9.97	5.06
v25	1574	4	108	57	1/1	9.73	23.23	124	47	1/1	7.51	5.55
v26	1566	12	107	57	1/1	11.40	20.60	107	57	1/1	11.40	20.92
v27	1567	11	107	57	1/1	11.47	14.86	108	57	1/1	11.47	14.82
v28	1502	76	108	57	1/1	11.33	96.43	108	57	1/1	11.33	97.31
v29	1559	19	100	57	1/1	11.09	77.67	113	49	1/1	10.23	57.57
v30	1520	58	100	57	1/1	11.27	39.50	112	48	1/1	9.99	28.86
v31	1563	15	112	58	2/2	9.65	25.08	124	51	2/2	9.00	16.43
v32	1575	3	112	58	2/2	10.40	11.88	129	48	2/2	8.38	4.62
v33	1391	187	108	57	4/4	10.47	76.59	108	57	4/4	10.47	75.67
v34	1500	78	108	57	1/1	10.91	29.06	109	57	1/1	10.91	29.23
v35	1502	76	108	57	1/1	11.33	97.02	108	57	1/1	11.33	97.09
v36	1452	126	108	57	1/1	7.67	3097.12	121	49	1/1	7.10	1078.89
v37	1486	92	104	57	1/1	11.56	32.00	106	56	1/1	10.75	50.04
v38	1572	6	108	57	2/2	10.64	114.08	110	55	2/2	10.50	113.59
v39	1574	4	108	57	1/1	9.73	23.36	124	47	1/1	7.51	5.55
v40	1452	126	98	56	2/2	13.87	721.66	109	48	**1/2**	12.71	247.40
v41	1553	25	106	56	1/1	10.36	28.69	118	48	1/1	10.32	20.07

Figure 7.4: Results of running SNIPER on the TCAS benchmark with both types of encoding (part 2/2)

The average ACSR of all program versions is 10.87% with the FFTF and 10.17% with the HFTF. Since the size of the soft part of the HFTF is in average smaller than the one of the FFTF, there are slightly less root causes to enumerate in the HFTF cases, which explain the small difference between the ACSRes obtained with each encoding.

The total running time is 5360 sec with the FFTF and 2541 sec with the HFTF. The total running time with the HFTF is about 47.4% of the case with the FFTF, which shows a good performance gain. For the versions no. 2, 4, 6-9, 13, 14, 16, 18, 20-25, 27, 29-32, 36 and 38-41 the computing times are better with the HFTF. This is because some part of the formula could be hardened in the HFTF making the number of soft clauses smaller than in the FFTF. For the versions no. 1, 3, 5, 10-12, 15, 17, 19, 26, 28, and 33-35 the number of soft clauses of the HFTF is almost the same as with the FFTF, which means very few or no clauses were hardened. The computing time shows that the overhead of constructing the HFTF is negligible for those cases where the number of soft clauses are almost the same. We see that version no. 37 is an exception in that the ACSR is better, but the computing time is larger for the HFTF than the FFTF. This version shows different behavior than others. Our conjecture is that such difference comes from the way we encode the formula in the Yices 1 solver.

All faults could successfully be detected with the HFTF except in the version no. 40. In this version, one of the two buggy statements is not covered by the failing test cases of the TCAS benchmark. However, it is covered by some of the successful test cases. Thereby, the missed buggy statement was encoded as hard in the HFTF. Namely, the version no. 40 does not satisfy the assumption (see Section 7.1) of the existence of failing test inputs. Apparently, the method using the notion of dice [2] is not able to detect this missing fault.

As shown above, the program coverage of test inputs is an important factor in fault localization. The HFTF as well as coverage-based or spectrum-based methods, or program slicing methods [1, 2] are all affected strongly by the coverage of the test inputs. Note that coverage-based or spectrum-based methods, such as Tarantula [46], are unable to locate the missed fault in the version no. 40 with the provided test cases. With the FFTF it is possible to locate the fault missed in version no. 40 because the

FFTF encodes both buggy statements as soft [53] (See the discussion in Section 5.6).

7.5 Discussion and Related Work

Static program slicing [90] was introduced for localizing faults in programs, and was empirically shown effective [52]. Program dicing is built upon program slicing and was first introduced by Lyle et al. [62]. Thereafter, the concept of execution slice was proposed by Agrawal et al. [1] as an alternative to static program slicing. Agrawal et al. further explored the concept of execution slice with execution dice [2]. Coverage-based or spectrum-based (cf. [46]) calculates ranking orders between program statements or spectrums to show that a particular fragment of code is more suspicious than the others. Coverage-based and slicing-based methods are based on statistical measures. On the other hand, our approach does not rely on such measures to localize the faults, but uses the successful and failing executions to reduce the complexity of the fault localization algorithm. A more detailed comparison between the formula-based method and coverage-based or spectrum-based method, such as Tarantula [46], can be found in Chapter 2.

Concerning the HFTF, its relative effectiveness depends on the MED from which it was constructed. The effectiveness of an MED is dependent on the testing search strategy. Covering too much failing paths makes MEDs large and covering a lot of successful paths makes MEDs small. At the opposite, covering too few failing paths makes MEDs small but increases the probability of missing a bug. Covering too few successful paths leads to large MEDs. For the case in which some paths are not covered at all, MEDs become large and the HFTF may not be different from the full flow-sensitive TF because few or no simplifications are possible, which means that the HFTF is conservative. We assume that a search strategy of choice would:

- cover all bugs with a minimal number of failing paths,
- cover as much as possible successful paths,
- not leave paths uncovered.

We used concolic execution, which is able to effectively generate test cases with a high code coverage [96]. Up to this point, we have assumed that a sufficient number of failing and successful paths could be obtained by concolic execution. However, in practice, we cannot generate all such paths. In our case, we used a *bounded DFS* strategy [84], which fits well with scope-bounded analysis methods, such as the formula-based method. The bound used in the concolic search algorithm has to be carefully chosen so that the formula encodes all paths that the concolic algorithm did found.

Some work employs concolic execution in conjunction with formula-based fault localization. Artzi et al. [4] use a concolic test generation method to generate good test cases for a coverage-based fault localization technique. Konighofer et al. [50] use a concolic test generation method to generate failing program inputs and to collect repair symbols. BugAssist [47] uses a concolic execution method to reduce the size of the formula by concretizing functions and loops. They do not provide further details on their optimization technique.

Formula-based fault localization with formulas that encode several program paths, such as in the present work, produces numerous root causes. The HFTF helps in the sense that it encodes more clauses as *hard* than the FFTF, hence reducing the number of spurious root causes.

7.6 Summary

We proposed a new way of representing programs for formula-based fault localization methods. We showed that in most cases, the proposed encoding reduced the number of soft clauses, which improved the fault localization algorithm performance without decreasing the localization accuracy.

We now have an important open question regarding the use of concolic execution with the HFTF. The heart of concolic execution is in its search strategy. There are many of them (cf. [85]), for example: depth-first search (DFS) (and its variants, such as bounded-DFS), breath-first search (BFS), random search, carfast, CFG-directed search, generational search, and context-guided search. As discussed in Section 7.5,

search strategies may have an impact on the calculation of MEDs, and thus on the shape of HFTF. This calls for a study of these strategies in the context of fault localization with the HFTF.

Chapter 8

Conclusion

8.1 Summary of the Thesis

The belief that formula-based fault localization of VLSI designs could be directly and easily applied to the case of imperative programs was not adequate. The latter requires deep considerations. In this thesis we demonstrated a new clue to amalgamating mathematical logic and software technology to provide a new fault-localization method for imperative programs. Adapting both the computational model and the failure model so that it is suited to imperative programs is the primary foundation of this work. The encoding methods arising therefrom, namely the full flow-sensitive trace formula and the hardened flow-sensitive trace formula, are the keys that enables a better localization of root causes in multi-fault programs.

In this thesis, we note that our new formula-based fault localization method generates many spurious root causes. Furthermore, the computation time of the diagnoses generation remains substantial. Concerning the input test cases, our method requires an appropriate set of test cases for not missing faults. This is especially true in the case of the hardened flow-sensitive trace formula, which is constructed from these test cases. We meet the three central problems with suggestions for future work, which are discussed in the next section.

8.2 Future Work

In the following we provide some possible future work to improve the methods presented in this thesis.

First, our work rises to an important open question regarding the filtering of root causes. Formula-based fault localization with formulas that encode several program paths, such as in the present work, produces numerous spurious root causes and conflictual root causes. The latter are root causes that are mutually inconsistent because they cannot be used together to correct multiple faults. This calls for a new method to segregate spurious root causes from real root causes. This method should also be able to deal with possible conflicts between the generated root causes to help programmers use the results.

We showed a trace formula encoding, HFTF (see Chapter 7), that was able to reduce the number of spurious root causes. Nevertheless, a large number of root causes can be difficult to handle for the software engineers. Semi-automated methods for filtering root causes exist [97].

Another point that deserves to be tackled concerns the number of root causes generated by formula-based methods. A fully automated method to filter and rank root causes depending on their probability of helping the engineers would be highly beneficial.

Second, the enumeration of diagnoses (MCSes) is computationally costly and leads most of the time to a slow fault localization, especially when considering multi-fault programs. In this thesis, we introduced an incremental solving algorithm, which uses the push & pop mechanism of the Yices solver. We showed that this algorithm could reduce the computing time. Another way of enumerating the algorithm faster would be to use a parallel algorithm [14] executing on many cores, CPUes, or even graphical processor units. This algorithm can take advantage of the advancement made in solver runtime prediction, as in portfolio solving [63] or in solver cost estimation [40]. The development of such algorithm would likely significantly improve the fault localization method.

Third, we have an open question about the generation of adequate test suites for

fault localization. It calls for a new test case generation method particularly focusing on exercising paths leading to assertion violations. Also, it is preferable that this set of failing test cases is as small as possible while covering all faults. A large number of failing test cases is not adequate in view of computation time.

Bibliography

[1] H. Agrawal. Toward Automatic Debugging of Computer Programs. PhD thesis, Purdue University, 1991.

[2] H. Agrawal, J.R. Horgan, S. London, and W.E. Wong. Fault Localization using Execution Slices and Dataflow Tests. In Proc. *ISSRE'95*, pages 143–151, 1995.

[3] A. V. Aho, R. Sethi, and J. D. Ullman. Compilers Principles, Techniques, and Tools. *Addison-Wesley*, 1986.

[4] S. Artzi, J. Dolby, F. Tip, and M. Pistoia. Directed Test Generation for Effective Fault Localization. In Proc. *ISSTA'10*, pages 49–60, 2010.

[5] D. Babić and A. J. Hu. Calysto: Scalable and Precise Extended Static Checking. In Proc. *ICSE'08*, pages 211–220, 2008.

[6] T. Ball, M. Naik, and S. K. Rajamani. From Symptom to Cause: Localizing Errors in Counterexample Traces. In Proc. *POPL'03*, pages 97–105, 2003.

[7] M. Bekkouche, H. Collavizza, and M. Rueher. LocFaults: A New Flow-driven and Constraint-based Error Localization Approach. In Proc. *ACM SAC'15*, 2015.

[8] M. Bekkouche. ANSI-C Benchmark. `http://www.i3s.unice.fr/~bekkouch/Benchs_Mohammed.html`

[9] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, Y. Zhu. Bounded Model Checking. *Advances in Computers*, 2003, 58:pages 117–148, 2003.

[10] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In Proc. *Workshop on Tools and Algorithms for the Construction and Analysis of Systems(TACAS'99)*, pages 193–207, 1999.

[11] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In Proc. *Design Automation Conference (DAC'99)*, 1999.

[12] A. Biere, M. Heule, H. Van Maaren, and T. Walsh (eds.). Handbook of Satisfiability: Volume 185. *IOS Press.*, 2009.

[13] D. Binkley, N. Gold, and M. Harman. An Empirical Study of Static Program Slice Size. In *ACM TOSEM*, Vol.16, No.2, Article 8, April 2007.

[14] G. E. Blelloch. Programming Parallel Algorithms. In *Communications of the ACM (CACM)*, 39(3):pages 85–97, 1996.

[15] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In Proc. *OSDI'08*, pages 209–224, 2008.

[16] S. Chaki, A. Groce and O. Strichman. Explaining Abstract Counterexamples. In Proc. *FSE-12*, pages 73–82, 2004.

[17] J. Christ, E. Ermis, M. Schaf, and T. Wies. Flow-Sensitive Fault Localization. In Proc. *VMCAI 2013*, pages 189–208, 2013.

[18] E. M. Clarke and E. A. Emerson. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In Proc. *Logic of Programs*, pages 52–71, 1981.

[19] E. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded Model Checking Using Satisfiability Solving. In *Form. Methods Syst. Des.*, 19(1):pages 7–34, 2001.

[20] E. M. Clarke and D. Kroening. Hardware Verification using ANSI-C Programs as a Reference In Proc. *ASP-DAC 2003*, pages 308–311, 2003.

[21] E. M. Clarke, D. Kroening, and F. Lerda. A Tool for Checking ANSI-C Programs. In Proc. *TACAS 2004*, pages 168–176, 2004.

[22] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An Efficient Method of Computing Static Single Assignment Form. In Proc. *POPL'89*, pages 25–35, 1989.

[23] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman and F. K. Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.*, 13(4):pages 451–490, 1991.

[24] V. Dallmeier, C. Lindig, and A. Zeller. Lightweight Bug Localization with AMPLE. In Proc *AADEBUG'05*, pages 99–104, 2005.

[25] T. Denmat, M. Ducassé and O. Ridoux. Data mining and cross-checking of execution traces: a re-interpretation of Jones, Harrold and Stasko test information visualization (Long version). Research Report RR-5661, INRIA, 2005.

[26] N. DiGiuseppe and J. A. Jones. On the Influence of Multiple Faults on Coverage-based Fault Localization. In Proc. *ISSTA'11*, pages 210–220, 2011.

[27] E.W. Dijkstra. The Humble Programmer. In *Commun. ACM*, 15(10):pages 859–866, 1972.

[28] B. Dutertre and L. de Moura. The Yices SMT Solver. *http://yices.csl.sri.com*

[29] B. Dutertre and L. de Moura. A Fast Linear-Arithmetic Solver for DPLL(T). *Computer-Aided Verification (CAV'2006)*, pages 81–94, 2006.

[30] B. Dutertre and L. de Moura. Integrating Simplex with DPPL(T). Technical report, 2006.

[31] J. Ferrante, K.J. Ottenstein, and J.D. Warren. The Program Dependence Graph and Its Use in Optimization. In *ACM Trans. Program. Lang. Syst.*, 9(3):pages 319–349, 1987.

[32] A. Fijany and F. Vatan. New Approaches for Efficient Solution of Hitting Set Problem. In Proc. *WISICT'04*, pages 1–10, 2004.

[33] P. Godefroid. DART: Directed Automated Random Testing. In Proc. *PLDI'05*, pages 213-223, 2005.

[34] P. Godefroid. Higher-Order Test Generation. In Proc. *PLDI*, 2011.

[35] A. Gonzalez. Automatic Error Detection Techniques Based on Dynamic Invariants. Master's thesis, 2007.

[36] R. Greiner, B. A. Smith, and R. W. Wilkerson. A Correction to the Algorithm in Reiter's Theory of Diagnosis. In *Artif. Intell.*, 41(1):pages 79–88, 1989.

[37] A. Griesmayer, S. Staber and R. Bloem. Fault Localization using a Model Checker. In *STVR*, pages 149–173, 2010.

[38] A. Groce, S. Chaki, D. Kroening, and O. Strichman. Error Explanation with Distance Metrics. In *STTT*, 8(3):pages 229–247, 2006.

[39] A. Groce and W. Visser. What Went Wrong: Explaining Counterexamples. In Proc. *SPIN'03* , 121–136, 2003.

[40] S. Haim and T. Walsh. Online Estimation of SAT Solving Runtime. In Proc. *SAT*, pages 133–138, 2008.

[41] Y. Hashimoto and S. Nakajima. Modular Checking of C Programs Using SAT-Based Bounded Model Checker. In Proc. *APSEC'09*, pages 515–522, 2009.

[42] J.N. Hooker. Solving the Incremental Satisfiability Problem. In *Journal of Logic Programming*,15:pages 177–186, 1993

[43] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments of the Effectiveness of Dataflow- and Controlflow-based Test Adequacy Criteria. In Proc. *ICSE '94*, pages 191–200, 1994.

[44] Y. Jia and m. Harman. An Analysis and Survey of the Development of Mutation Testing. In *Software Engineering, IEEE Transactions on*, 37(5):pages 649–678, 2011.

[45] J.A. Jones, M.A Harrold and J.T. Stasko. Visualization for Fault Localization. In Proc. of the Workshop on Software Visualization, pages 71–75, 2001.

[46] J.A. Jones and M.J. Harrold. Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique. In Proc. *ASE '05*, pages 273–282, 2005.

[47] M. Jose and R. Majumdar. Cause Clue Clauses : Error Localization using Maximum Satisfiability. In Proc. *PLDI 2011*, pages 437–446, 2011.

[48] S. Khoshnood, M. Kusano, and C. Wang. ConcBugAssist: Constraint Solving for Diagnosis and Repair of Concurrency Bugs. In Proc. *ISSTA*, pages 165–176, 2015.

[49] R. Konighofer and R. Bloem. Automated Error Localization and Correction for Imperative Programs. In Proc. *FMCAD*, 2011.

[50] R. Konighofer and R. Bloem. Automated Error Localization and Correction for Imperative Programs. In Proc. *FMCAD'11*, pages 91–100, 2011.

[51] D. Kroening, E. Clarke, and K. Yorav. Behavioral Consistency of C and Verilog Programs Using Bounded Model Checking. In Proc. *DAC 2003*, pages 368–371, 2003.

[52] S. Kusumoto, A. Nishimatsu, K. Nishie, and K. Inoue. Experimental Evaluation of Program Slicing for Fault Localization. In *Empirical Software Engineering*, 7(1):pages 49–76, 2002.

[53] S. Lamraoui and S. Nakajima. A Formula-based Approach for Automatic Fault Localization of Imperative Programs. In Proc. *ICFEM'14*, pages 251–266, 2014.

[54] S. Lamraoui, S. Nakajima, and H. Hosobe. Hardened Flow-sensitive Trace Formula for Fault Localization. In Proc. *ICECCS 2015*, pages 50–59, 2015.

[55] S. Lamraoui and S. Nakajima. A Formula-based Approach for Automatic Fault Localization of Multi-fault Programs. In *Journal of Information Processing*, 24(1):pages 88–98, 2016.

[56] S. Lamraoui, C. Belo Loureno, S. Nakajima, and H. Hosobe. SNIPER (2016), GitHub repository, https://github.com/lamraoui/sniper

[57] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In Proc. *CGO'04*, 2004.

[58] C. Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master Thesis, 2002.

[59] M.H. Liffiton and K.A. Sakallah. On Finding All Minimally Unsatisfiable Subformulas. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 173–186, 2005.

[60] M.H. Liffiton and K.A. Sakallah. Algorithms for Computing Minimal Unsatisfiable Subsets of Constraints. In *Automated Reasoning*, 40(1):pages 1–33, 2008.

[61] L. Lin and Y. Jiang. The Computation of Hitting Sets: Review and New Algorithms. In *Information Processing Letters*, 86(4):pages 177–184, 2003.

[62] J. R. Lyle and M. Weiser. Automatic Bug Localization by Program Slicing. In Proc. *2nd Intl. Conf. on Computers and Applications*, pages 877–883, 1987.

[63] Y. Malitsky, B. O'Sullivan, A. Previti, J. Marques-Silva. A Portfolio Approach to Enumerating Minimal Correction Subsets for Satisfiability Problems. In Proc. *CPAIOR*, pages 368–376, 2014.

[64] Z. Manna, R. J. Waldinger, Fundamentals of Deductive Program Synthesis. In *Transactions on Software Engineering 18*, pages 674–704, 1992.

[65] J. Marques-Silva, F. Heras, M. Janota, A. Previti, and A. Belov. On Computing Minimal Correction Subsets. In Proc. *IJCAI '13*, pages 615–622, 2013.

[66] F. Merz, S. Falke and C. Sinz. LLBMC: Bounded Model Checking of C and C++ Programs Using a Compiler IR. In Proc. *VSTTE'12*, pages 146–161, 2012.

[67] B. Meyer. Design by Contract. *Technical Report TR-EI-12/CO*, Interactive Software Engineering Inc., 1986.

[68] B. Meyer. Design by Contract. In *Advances in Object-Oriented Software Engineering*, eds. D. Mandrioli and B. Meyer, P. Hall, pages 1–50, 1991.

[69] B. Meyer. Applying "Design by Contract". In *Computer (IEEE)*, 25(10):pages 40–51, 1992.

[70] A. Morgado, M. Liffiton, and J. Marques-Silva. MaxSAT-Based MCS Enumeration. In Proc. *HVC-2012*, 2012.

[71] J.D. Musa, A. Iannino, and K. Okumoto. Software Reliability Engineering: Measurement, Prediction, Application. *McGraw-Hill, Inc.*, 1987.

[72] G. C. Necula,S. McPeak, S.P. Rahul, and W. Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In Proc. CC'02, pages 213–228, 2002.

[73] D. A. Peled. Software Reliability Methods. Springer, 2001.

[74] M.R. Prasad, A. Biere, and A. Gupta. A Survey of Recent Advances in SAT-Based Formal Verification. In *STTT*, 7(2):pages 156–173, 2005.

[75] J.P. Queille and J. Sifakis Specification and verification of concurrent systems in CESAR. In Proc. *of the 5th Colloquium on International Symposium on Programming*, pages 337–351, 1982.

[76] S. Rayadurgam and M. P. E. Heimdahl. Coverage Based Test-Case Generation using Model Checkers. In Proc. *ECBS 2001*, pages 83–91, 2001.

[77] R. Reiter. A Theory of Diagnosis from First Principles. In *Artificial Intelligence*, 32(1):pages 57–95, 1987.

[78] M. Renieris and S. P. Reiss. Fault Localization With Nearest Neighbor Queries. In Proc. *ASE 2003*, pages 30–39, 2003.

[79] G. Rothermel and M. Harrold. Empirical studies of a safe regression test selection technique. In *IEEE Trans. Softw. Eng.*, 24(6):pages 401–419, 1990.

[80] S. Safarpour, H. Mangassarian, A. Veneris, M.H. Liffiton, and K.A. Sakallah. Improved Design Debugging using Maximum Satisfiability. In Proc. *FMCAD'07*, 2007.

[81] K. Satoh and T. Uno. Enumerating Minimally Revised Specifications Using Dualization. In *New Frontiers in Artificial Intelligence: Joint JSAI 2005 Workshop Post-Proceedings, LNAI 4012*, T. Washio, A. Sakurai, K. Nakajima, H. Takeda, S. Tojo, M. Yokoo (eds.), pages 182–189, 2006.

[82] K. Satoh, K. Kaneiwa, and T. Uno. Contradiction Finding and Minimal Recovery for UML Class Diagrams. In Proc. *ASE*, pages 277–280, 2006.

[83] M. Schäf, C. D. Schwartz, T. Wies. Explaining Inconsistent Code. In Proc. *ESEC/SIG-SOFT FSE*, pages 521–531, 2013.

[84] K. Sen, D. Marinov, and G. Agha. CUTE: A Concolic Unit Testing Engine for C. In Proc. *ESEC/FSE-13*, 2005.

[85] H. Seo and S. Kim. How We Get There: A Context-Guided Search Strategy in Concolic Testing. In Proc. *FSE 2014*, pages 413–424, 2014.

[86] A. Shay and J. Dolby, F. Tip and M. Pistoia. Directed Test Generation for Effective Fault Localization. In Proc. *ISSTA '10*, pages 49–60, 2010.

[87] S. Thompson and G. Brat. Verification of C++ Flight Software with the MCP Model Checker. In *Aerospace Conference, 2008 IEEE*, pages 1–9, 2008.

[88] F. Tip. A Survey of Program Slicing Techniques. In *Journal of Programming Languages*, 3(3):pages 121–189, 1995.

[89] M. Weiser. Program slicing. Int Proc. of the *5th International Conference on Software Engineering*, pages 439–449, 1981.

[90] M. Weiser. Programmers Use Slices When Debugging. In *Comm. ACM*, 25(7):pages 446–452, 1982.

[91] W.E. Wong, Y. Qi, L. Zhao and K.Y. Cai. Effective Fault Localization Using Code Coverage. In Proc. *COMPSAC'07*, pages 449–456, 2007.

[92] W. E. Wong, S. S. Gokhale, J. R. Horgan, and K. S. Trivedi. Locating Program Features by using Execution Slices. In Proc. *ASSET'99*, pages 194–203, 1999.

[93] F. Wotawa. On the Relationship between Model-based Debugging and Program Slicing. In *Artificial Intelligence*, 135(1):pages 125–143, 2002.

[94] F. Wotawa. A Variant of Reiter's Hitting-Set Algorithm. In *Information Processing Letters*, 79(1):pages 45–51, 2001.

[95] F. Wotawa, M. Nica, and I. Moraru. Automated Debugging based on a Constraint Model of the Program and a Test Case. In *Logic and Algebraic Programming*, 81(4):pages 390-407, 2012.

[96] Q. Xiao and B. Robinson. A Case Study of Concolic Testing Tools and Their Limitations. In Proc. *ESEM*, 2011.

[97] Q. Yi, Z. Yang, J. Liu, C. Zhao, and C. Wang. Explaining Software Failures by Cascade Fault Localization. In *ACM Trans. Design Autom. Electr. Syst.*, 20(3), 41, 2015.

[98] A. Zeller and R. Hildebrandt. Simplifying and Isolating Failure-Inducing Input. In *IEEE Trans. Softw. Eng.*, 28(2):pages 183–200, 2002.

[99] A.X. Zheng, M.I. Jordan, B. Liblit, M. Naik and A. Aiken. Statistical Debugging: Simultaneous Identification of Multiple Bugs. In Proc. *ICML'06*, 2006.

Appendix A

Tool Implementation

In the following sections we start by introducing the LLVM compiler infrastructure on which SNIPER is based. This is followed by an introduction of the Yices 1 SMT solver, which is used as the main solver of SNIPER. The last section details the internal implementation of SNIPER. The source of SNIPER can be found in the Git repository [56].

A.1 LLVM Compiler Infrastructure

Low Level Virtual Machine (LLVM) [57, 58] is a compiler infrastructure that optimizes compilation, link, execution and "idle-time" of programs written in a wide variety of programming languages. What LLVM calls idle-time optimization is an approach that uses profiling information collected at runtime.

The LLVM infrastructure has been built around a dedicated code representation (bitcode). This intermediate representation (IR) is an abstract RISC-like instruction set based on a language-independent type-system. It is composed of high-level instructions while being low-level enough to represent any program. LLVM's code representation has many convenient features such as its SSA (static single assignment) form [22, 23] that facilitates compilation optimization and analysis. LLVM is actually a virtual machine, and it has run-time capabilities that operate on programs during the execution giving several opportunities to improve performances. This last

point is possible thanks to the Just-In-Time (JIT) compilation technique.

Many tools that perform program analysis are built upon LLVM. Tools such as model checkers [66, 87], static checkers [5], and test case generators [15]. SNIPER is a first fault localization tool on top of LLVM, and thus opens a new area of applying LLVM.

A.1.1 Intermediate Representation Compilation

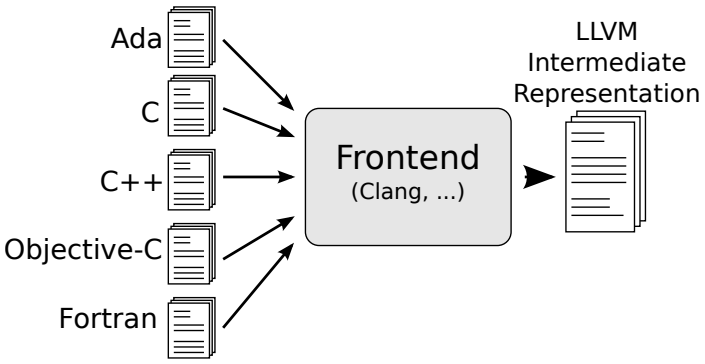


Figure A.1: Frontend for LLVM

As depicted in Figure A.1, IRs are obtained by compiling the source code of a program with an appropriate frontend. There exist a wide variety of frontends for LLVM. For example, it is possible to compile a `C` program into an IR using the command `clang` with the option `-emit-llvm`. For our purpose, we additionally use the option `-O0` not to optimize the code.

Because we work at the bytecode level, we need to go back to the original source code after identifying root causes. Basically, we want to know the line number (in original source code) for a given bytecode instruction. To do that we add debugging options using the `-g` command-line option with the command `clang`. We can now retrieve the corresponding line number of any bytecode instruction. For our purpose, we need only line number information, we then use the more restrictive option `-gline-tables-only`.

A.2 Yices SMT Solver

Yices 1 [28, 29, 30] is an efficient SMT solver that checks the satisfiability of formulas containing uninterpreted function symbols with equality, linear real and integer arithmetic, bitvectors, scalar types, and tuples. Yices 1 also does MaxSMT/p-MaxSMT (and, dually, unsat cores) and is competitive as an ordinary SAT and MaxSAT solver. Below, we explain features of Yices that are relevant to understand SNIPER.

A.2.1 Application Programming Interface

Yices can be used through its library to work with programs that require close interaction with an SMT solver via an application programming interface (API). The Yices API can be used with `C` and `C++` languages. The API provides access to the basic functionalities of Yices.

To construct a formula with the Yices API we first create a *logical context*, which stores a collection of declarations and assertions. We add to this context the data types that will be used. Yices provides the following builtin types: number real, int, nat, and bool. It is also possible to create composed types. Once some types are added to the context, we can add variable declarations. A declaration consists of a name and a type (such as `x::bool`). An instance of the declaration represents a term (`x`).

Using the variable instances, we can make expressions, which are represented by abstract syntax trees in Yices. The expressions are then asserted, meaning that we add constraints into the logical context. The constraints can be *weighted* (relaxable) or not (non-relaxable). All the constraints in the context represents the formula. We can *check* with Yices if the the logical context (formula) is satisfiable. If it is satisfiable, a model can be obtained. A model assigns constant values to variables defined in the context. The context must be consistent for a model to be available. Alternatively to a *check*, we can compute the maximal satisfying assignments for the asserted weighted constraints. If a maximal satisfying assignment is found, then we

can obtain a model.

```
1 #include<stdio.h>
2 #include"yices_c.h"
3
4 int main() {
5   yices_context ctx = yices_mk_context();
6   yices_type int_t  = yices_mk_type(ctx, "int");
7   yices_var_decl x_decl = yices_mk_var_decl(ctx, "x", int_t);
8   yices_var_decl y_decl = yices_mk_var_decl(ctx, "y", int_t);
9   yices_expr x_var = yices_mk_var_from_decl(ctx, x_decl);
10  yices_expr y_var = yices_mk_var_from_decl(ctx, y_decl);
11  yices_expr eq_expr = yices_mk_eq(ctx, x_var, y_var);
12  yices_assert(ctx, eq_expr);
13  switch(yices_check(ctx)) {
14  case l_true:
15    printf("satisfiable\n");
16    yices_model m = yices_get_model(ctx);
17    int x = yices_get_value(m, x_decl);
18    int y = yices_get_value(m, y_decl);
19    printf("x = %d\n", x);
20    printf("y = %d\n", y);
21    break;
22  case l_false:
23    printf("unsatisfiable\n");
24    break;
25  case l_undef:
26    printf("unknown\n");
27    break;
28  }
29  yices_del_context(ctx);
30  return 0;
31 }
```

Listing A.1: Example of using the Yices API

Listing A.1 is an example that illustrates the use of the Yices API. In this example,

we create two variables (lines 7-10) and add a constraint ($x = y$) into the context (lines 11-12). From line 12 through 28, we check if the context is satisfiable. If it is the case, we extract the values of the variables x and y and display them. Finally, we delete the context before leaving the function.

A.2.2 Incremental Assertions

Yices implements incremental solving. The original definition [42] of "incremental solving" is shown in Definition 11.

Definition 11 (Incremental Solving) *Given that a set S of propositional clauses is satisfiable, check whether $S \cup \{C\}$ is satisfiable for a given clause C.*

The logical context of Yices can be viewed as a stack of contexts. The stack of contexts is simulated using trail (undo) stacks. It is possible to create a backtracking point with the procedure `yices_push`. The procedure `yices_pop` restores the context from the top of the stack, and pops it off the stack. Any changes to the logical context (by `yices_assert` or other functions) between the matching `yices_push` and `yices_pop` operators are flushed (popped out), and the context is completely restored to what it was right before the `yices_push`.

A.3 Program Parsing and Pre-processing

Before the IR is encoded into a trace formula (TF), it must be parsed and pre-processed. Most of the transformations involved in this pre-processing stage are common to bounded model-checking (BMC) of imperative programs (cf. [21, 20, 51, 66]). Basically, we have to unroll the program and put it in SSA form. In SNIPER we use the compiler passes provided by LLVM to transform the IR. The following sections details these transformations.

A.3.1 Intermediate Representation Parsing

Listing A.2 shows how SNIPER implements the loading and parsing of the input IR. LLVM library provides the function `ParseIRFile`, which loads and parses a file that contains a bitcode image or LLVM Assembly code and returns a LLVM module for it. All the processing involved in SNIPER is done on the previously returned LLVM module `llvmMod`.

```
1  #include "llvm/IR/LLVMContext.h"
2  #include "llvm/IR/Module.h"
3  #include "llvm/IRReader/IRReader.h"
4  ...
5  LLVMContext &Context = getGlobalContext();
6  SMDiagnostic Err;
7  Module *llvmMod = ParseIRFile(inputIRFilename, Err, Context);
8  if (!llvmMod) {
9    Err.print(argv[0], errs());
10   return 1;
11 }
```

Listing A.2: Loading and parsing of the input LLVM IR bitcode

A.3.2 Function Inlining

In order to expand function calls present in the IR, we use a LLVM pass that implements *inlining*. Inlining or inline expansion is a technique that replaces a function call instruction by the body of the callee function. The LLVM function inlining pass performs a bottom-up inlining of functions into callees.

In SNIPER, we force the inlining of function calls by processing functions before running the LLVM inlining pass. This is done by adding, with the LLVM library, a special attribute `AlwaysInline` to functions in the IR as shown in Listing A.3.

```
1  bool FullFunctionInliningPass::runOnFunction(Function &F) {
2    F.addFnAttr(Attribute::AlwaysInline);
3    processCalls(F);
4    return true;
5  }
```

Listing A.3: Forcing function inlining.

The next required processing step is to prevent the propagation of variables into other instructions while inlining the calls. Listing A.5 and A.6 show some LLVM bitecode to illustrate the problem. In Listing A.5 there is a call to the function `foo`, which is defined in Listing A.4. After inlining (see Listing A.6), the call instruction is replaced by the body of the `foo` function. The variable `a` appearing in function `foo` was propagated in the `mul` instruction, and the information (for example, the line number) about the `ret` instruction of function `foo` is lost. We want to keep this information in case there is a fault in the `ret` instruction.

```
1 define i32 foo(i32 %a) {
2 entry:
3   ret i32 %a
4 }
```

Listing A.4: Function foo

```
1 %ret = call i32 @foo(%arg)
2 %x = mul i32 %ret, 42
```

Listing A.5: Before Inlining

```
1 %x = mul i32 %arg, 42
```

Listing A.6: After Inlining

To prevent the variable propagation, we modify each call instruction in each function of the IR by inserting a dummy `add` instruction after each call. Listing A.7 shows an example with a call before it is processed and Listing A.8 shows the call after being processed.

After the two processing stages explained above, we run the inlining pass of LLVM.

```
1 %ret = call i32 @foo(%arg)
2 %x = mul i32 %ret, 42
```

Listing A.7: Before processing.

```
1 %ret = add i32 %arg, 0
2 %x = mul i32 %ret, 42
```

Listing A.8: After processing.

Listing A.9 shows how the whole inlining process is implemented in SNIPER.

```
1 void Backend::inlineCalls(Module *llvmMod) {
2   DataLayout *DL = new DataLayout(llvmMod);
3   PassManager *PM = new PassManager();
4   PM->add(DL);
5   PM->add(new FullFunctionInliningPass()); // SNIPER pass
6   PM->add(createFunctionInliningPass()); // LLVM pass
7   PM->run(*llvmMod);
8   delete PM;
9 }
```

Listing A.9: SNIPER inlining procedure.

A.3.3 Local Variables Processing

While putting the IR into SSA form, LLVM automatically propagates constant values. This is something we want to avoid because we loose information and may encounter difficulties when mapping back the IR's instructions to the original source code. Listings A.10 and A.11 show what happen after putting the IR into SSA form.

Listing A.10 shows the original IR. It uses registers to handle local variables through `load/store` instructions. The optimization pass that puts the IR in SSA form gets rid of all `load/store` instructions and propagates the constant values of variables if it is possible. The Listing A.11 shows the IR after running the optimization pass. The variables `x` and `y` are now replaced by their actual constant values in the `add` instruction. To avoid this we pre-proccess the IR before putting it in SSA form. For each instruction we save the name of the used variables. Then, in the IR in SSA form, for each instruction containing a constant value, we check if the constant value refers to a variable. If this is the case, we replace the constant value by its variable

```
1  %x = alloca i32
2  %y = alloca i32
3  store 42, %x,
4  store 8, %y
5  %1 = load %x
6  %2 = load %y
7  %tmp = add %1, %2
```

Listing A.10: Original Intermediate Representation

```
1  %tmp1 = add 42,  8
```

Listing A.11: Intermediate Representation in SSA form

name. We also assign these variables with the appropriate values.

A.3.4 Static Single Assignment Form Transformation

SNIPER uses the *memory to register pass* of LLVM to transform the input IR to SSA form. The pass implements a standard SSA construction algorithm to construct "pruned" SSA form (see Section 3.5 for details).

```
1 void Backend::putInSSAForm(Module *llvmMod, Function *targetFun) {
2   FunctionPassManager *FPM = new FunctionPassManager(llvmMod);
3   FPM->add(createPromoteMemoryToRegisterPass());
4   FPM->doInitialization();
5   FPM->run(*targetFun);
6   delete FPM;
7 }
```

Listing A.12: Transform a LLVM function into SSA form

A.3.5 Loops Processing

This stage consists of unrolling all loops to a certain bound. The bound is provided by the user. If the bound is too small, SNIPER stops and return an error: *unwinding assertion failed*. When such situations happen, the user have to run SNIPER again

with a greater bound than the one used in the previous run.

SNIPER uses LLVM compiler passes to unroll loops in the input program. In a classic compilation environment, the goal of the compiler when unrolling loops is to find a compromise between the program execution speed and the size of the program binary. Therefore, the compiler may not always unroll loops. In the case of SNIPER, it is mandatory to systematically unroll program's loops to a given bound. The following sections describe how SNIPER uses the LLVM compiler passes to achieve the systematic unrolling of loops.

```
1 void Backend::unrollLoops(Module *llvmMod, Function *targetFun,
                            int unrollCount) {
2   FunctionPassManager *FPM = new FunctionPassManager(llvmMod);
3   FPM->add(createLoopSimplifyPass());
4   FPM->add(createLoopRotatePass());
5   FPM->add(createLCSSAPass());
6   int threshold = UINT_MAX; // (1)
7   int allowPartial = 1; // true
8   FPM->add(createLoopUnrollPass(threshold,unrollCount,allowPartial)
        );
9   FPM->doInitialization();
10  FPM->run(*targetFun);
11  delete FPM;
12 }
```

Listing A.13: Unroll to a given bound all loops present in the target function

Loop Simplify Pass

In line 3 of Listing A.13, the *loop simplify pass* is run as a preliminary step to *canonicalize* all loops. In this context, canonicalization is the process of putting a natural loop in a *normal form* (simpler form). This makes subsequent analyses and transformations simpler and effective. In general, a natural loop has one entry block (header) and possibly several back edges (latches) leading to the header from the inside of the loop. As shown in Figure A.2, canonicalization ensures that there is a

single *entry point* and only one *back edge*. We take advantage of this fact later in the formula encoding.

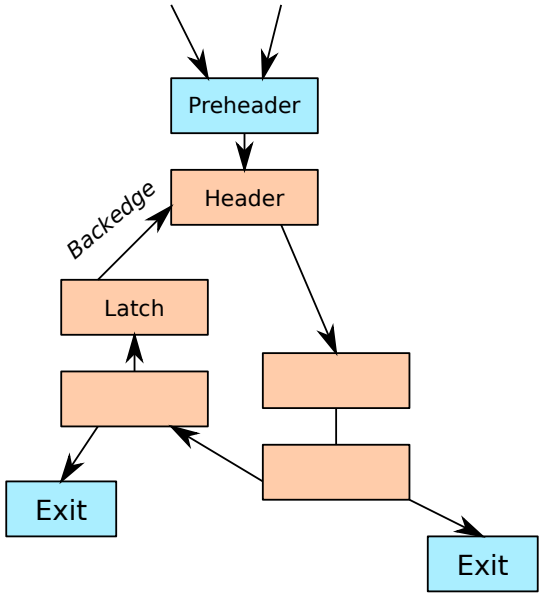


Figure A.2: A loop CFG in normal form

Loop Rotate Pass

In line 4 of Listing A.13, the *loop rotate pass* performs a loop rotation transformation, which moves loop conditionals at the end of the loop. If the transformed loop executes more than once, this optimization eliminates the initial test. This pass is used to facilitates the unrolling of loop in LLVM.

Listings A.14, A.15, and A.16 show the different forms taken by a loop while being rotated by the *loop rotate pass*.

```
1 for (i = 0; i < n; ++i) {
2     ...
3 }
```

Listing A.14: Loop before rotation

```
1 i = 0;
2 while (true) {
3   if (i >= n)
4     break;
5   ...
6   ++i;
7 }
```

Listing A.15: Loop after lowering

```
1 i = n;
2 if (i > 0) {
3   do {
4     ...
5     ++i;
6   } while (i < n);
7 }
```

Listing A.16: Loop after rotation

Loop-Closed SSA Form Pass

In line 5 of Listing A.13, the *LCSSA pass* transforms loops in loop-closed SSA form. Basically, this form requires that all Φ nodes are placed at the end of the loops for all values that are live across the loop boundary. As is usual in SSA representation, a Φ node is an instruction used to select a value depending on the predecessor of the current block. More details about the SSA representation can be found in Section 3.5.

Loop Unroll Pass

Now we have a loop in a *normal form* we can unroll it (line 8 of Listing A.13). In Figure A.3 the three main steps are shown from left to right.
(1) The first step is the duplication of the loop body. This is done by LLVM. The number of times the body is duplicated depends on the number of times the user wants to unroll loops (bound given as argument). When it is possible, this number can also be automatically calculated by LLVM to fully unroll loops.
(2) The second stage replaces the *latch block* (end block when the loop has been fully traversed) by an `assert(false)`. This ensures the program never does more iterations. In the case where this assert is reached, we can conclude that this loop should be unrolled further.
(3) The last step simply removes the *back edge*. As the *back edge* is the only way to go back to the header, its removal completely cuts the loop.

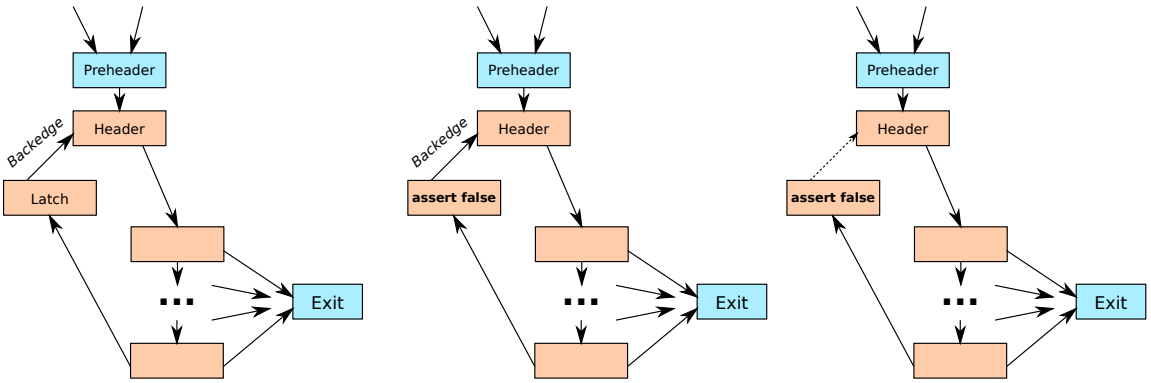


Figure A.3: From left to right, the steps to unroll a loop

A.3.6 Finalization

After applying all the above passes to the IR, we obtain a single function in SSA form with all loops unrolled to a certain bound. At this point the IR contains arithmetic, comparison, ϕ (join), and branching instructions only. Some of these instructions may not have a name assigned to them. Therefore, as shown in Listing A.17, SNIPER additionally run the *instruction namer pass* of LLVM, which simply assigns names to anonymous instructions. This pass is used to assign names to variables while encoding the IR into a formula.

```
1 void Backend::finalize(Module *llvmMod, Function *targetFun) {
2   FunctionPassManager *FPM = new FunctionPassManager(llvmMod);
3   FPM->add(createInstructionNamerPass());
4   FPM->doInitialization();
5   FPM->run(*targetFun);
6   delete FPM;
7 }
```

Listing A.17: Assigning names to anonymous instructions

Appendix B

List of Publications

B.1 Journal Papers (refereed)

1. Si-Mohamed Lamraoui and Shin Nakajima. A Formula-based Approach for Automatic Fault Localization of Imperative Programs. In *Journal of Information Processing*, 24(1):pages 88–98, January 2016.

B.2 International Conference Papers (refereed)

1. Si-Mohamed Lamraoui, Shin Nakajima, and Hiroshi Hosobe. Hardened Flow-sensitive Trace Formula for Fault Localization. In Proc. *ICECCS'15*, pages 50–59, 2015.

2. Si-Mohamed Lamraoui and Shin Nakajima. A Formula-based Approach for Automatic Fault Localization of Imperative Programs. In Proc. *ICFEM'14*, pages 251–266, 2014.

B.3 Domestic Conference and Workshop Papers

1. Shin Nakajima and Si-Mohamed Lamraoui. Fault Localization of Energy Consumption Behavior using Maximum Satisfiability. In Proc. *CyPhy 2015*.

2. Clàudio Belo Loureno, Si-Mohamed Lamraoui, Shin Nakajima, and Jorge Sousa Pinto. Studying Verification Conditions for Imperative Programs. In Proc. *AVoCS 2015*.

3. Shin Nakajima and Si-Mohamed Lamraoui. Fault Localization of Timed Automata using Maximum Satisfiability. In Proc. *WSOFL+MVSL 2015*.

4. Si-Mohamed Lamraoui and Shin Nakajima. SNIPER: A Tool for Automatically Localizing Errors in Imperative Programs. *SES2013*, 2013.

5. Si-Mohamed Lamraoui and Shin Nakajima. Automated Error Localization with Weighted Partial Maximum Satisfiability. *IEICE Technical Report, SS2013-13*, 2013.

6. Si-Mohamed Lamraoui and Shin Nakajima. SNIPER: An LLVM-based Automatic Fault Localization Tool for Imperative Programs. *IEICE Technical Report, SS2015-15*, 2015.

B.4 Awards & Honors

1. IEICE SIGSS 2013 Student Paper Awards. Si-Mohamed Lamraoui and Shin Nakajima. Automated Error Localization with Weighted Partial Maximum Satisfiability. *IEICE Technical Report, SS2013-13*, 2013.