Electronic Preprint for Journal of Information Processing Vol.24 No.1

Regular Paper

A Formula-based Approach for Automatic Fault Localization of Multi-fault Programs

Si-Mohamed Lamraoui[1,2,a)] Shin Nakajima[2,1,b)]

Received: January 15, 2015, Accepted: October 2, 2015

Abstract: Formula-based fault localization approach is an algorithmic method that is able to provide fine-grained information account for identified root causes. The method combines the SAT-based formal verification techniques with the Reiter's model-based diagnosis theory. This paper adapts the formula-based fault localization method, and introduces a new program encoding, called full flow-sensitive trace formula. This encoding is particularly useful for programs with multiple faults. Furthermore, we improve the efficiency of computing the potential root causes by using the push & pop mechanism of the Yices solver. We implemented the method in a tool, SNIPER, which was applied to some benchmarks. All single and multiple faults were successfully identified and discriminated.

Keywords: model-based diagnosis theory, multiple faults, partial maximum satisfiability, LLVM, Yices

1. Introduction

Debugging is one of the most expensive tasks of software development. A challenging activity in debugging is fault localization, which consists of identifying root cause locations of a program that shows faulty behavior. Automatic fault localization was introduced to help software engineers tackle this task. Automatic fault localization of imperative programs is a well-known problem, and has been studied from various approaches (cf., Refs. [8], [9], [24], [27]). Among these, coverage-based or spectrum-based debugging [11] is considered a promising method. It is an empirical method that calculates ranking orders between the program statements or spectrums to show that a particular fragment of code is more suspicious than the others. The method, however, needs a lot of both successful and failing executions to calculate the statistical measures. Generating an unbiased input test data set is a major challenge. In addition, the causal explanation of results is not clear in regard to program semantics.

The formula-based fault localization method uses only failing executions, and is more systematic than the coverage-based approach. This is because it has a logical foundation developed in the model-based diagnosis (MBD) theory [20]. However, existing tools following the formula-based method, such as BugAssist [12] or Wotawa's tool [26] mainly consider single-fault programs. Nevertheless, in practice it is common to have more than one fault in a program. Automatic fault localization of multi-fault programs is not an easy task. DiGiuseppe et al. [6] empirically studied the coverage-based fault localization on multi-fault programs and concluded that at least one of the faults could be effectively localized. However, the method is not effective for localizing simultaneously all the faults. Contrarily, the MBD theory considers the case of artifacts with multiple faults, but it needs more work for the case of imperative programs.

Furthermore, existing formula-based methods do not guarantee to cover all the root causes. It is partly because the complete enumeration of root causes requires a high computational effort. Its complexity grows exponentially with the size of the program, the number of test cases used and the number of faults in the program.

This paper reports a new formula-based automatic fault localization method, which follows the maximum satisfiability (MaxSAT) approach as in Refs. [12], [22]. Our method uses a full flow-sensitive trace formula in order to consider control-oriented faults and programs with multiple faults. We adapt a new enumeration algorithm [18] and ensure obtaining all the root causes efficiently by using the Yices SMT solver [7] in an incremental fashion with its push & pop mechanism. Furthermore, our approach uses a fault localization algorithm that can work on a set of failing test cases. In most cases it is not enough to use a single failing program path to reproduce behavior caused by the multiple faults.

This paper makes three contributions. First, we reformulate systematically the problem of automatic fault localization for imperative programs from a view point of formula-based approach. Second, we present an efficient method for identifying root causes of faults in multi-fault programs. The key point of this method is the way we encode programs. Our encoding, called full flow-sensitive trace formula, is essentially equivalent to the program's control flow graph (CFG). In addition, our method uses a full MCS enumeration algorithm working on different failing test

[1] The Graduate University for Advanced Studies (SOKENDAI), Hayama, Kanagawa 240–0193, Japan
[2] National Institute of Informatics, Chiyoda, Tokyo 101–8430, Japan
[a)] simo@nii.ac.jp
[b)] nkjm@nii.ac.jp

This paper is a revision of the conference presentation [14].

cases. The results, which are potential root causes collected from different failing test cases, are combined in order to facilitate the identification of root causes by the software engineers. Third, we experimentally show that multiple faults in a program are successfully detected by our method.

This paper is organized as follows. Section 2 presents the backgrounds of the work. Section 3 discusses issues on the fault localization of multi-fault programs. Section 4 provides basic definitions. Section 5 presents our approach for localizing faults automatically. Section 6 reports experiments on two benchmarks, which is followed by Section 7 for the summary and directions for future work.

2. Automatic Fault Localization Methods

This section introduces technical backgrounds of the automatic fault localization of imperative programs.

Program slicing [24] was introduced for localizing faults, and was empirically shown effective [13]. The average code size reduction (CSR) of program slices is around 30% [2]; such amount of program code needs to be inspected to find real root causes. Coverage-based or spectrum-based debugging (cf., Ref. [11]) 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. Generating unbiased input test data set is a major challenge.

The formula-based automatic fault localization method essentially combines the SAT-based formal verification techniques [19] with the model-based diagnosis (MBD) theory. The MBD theory establishes a logical formalism of the fault localization problem [20]. The *model* is presented as a formula expressed in *suitable* logic. The formula is unsatisfiable as it represents an artifact containing faults. The MBD theory distinguishes conflicts and diagnoses. Conflicts are the erroneous situations represented by minimal unsatisfiable subsets (MUSes) of the unsatisfiable formula. Conflicts constitute a set of program fragments containing faults, and thus are to be compared with the identified slices in the program slicing approach. 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.

The MBD methods, including the model-based debugging [26], first calculate MUSes and then obtain MCSes. An early work [25] 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 [26]. Both methods resulted in rather large MUSes for the TCAS benchmark [10], [21].

An alternative approach to obtain MCSes was employed in the fault localization of VLSI circuits [22]. The method reduces the fault localization problem to maximum satisfiability of the unsatisfiable formula in propositional logic and calculates maximal satisfiable subsets (MSSes). An MCS is the complement of an MSS [16]. This idea was applied to the fault localization problem of imperative programs and implemented in a tool, BugAssist [12]. It uses an iterative localization algorithm to obtain the MCSes, from which the CSR is calculated. The CSR is smaller than the case of program slicing approaches. The algorithm, however, does not guarantee the enumeration of all the MSSes, which means that it may miss some faults. Furthermore, when multiple runs of the fault localization are needed, BugAssist combines MCSes by putting all its atomic elements (clauses) in a single set. For example the set of MCSes $\{\{1,2,3\},\{4,5,6\}\}$ becomes $\{1,2,3,4,5,6\}$. Such combination looses information contained in the calculated MCSes because MCSes obtained for the same error-inducing inputs are all merged together. This makes the debugging of multi-fault programs difficult for the software engineers.

3. Multi-fault Programs

3.1 Problem

Dealing with programs with multiple faults is one of the important issues in automated fault localization methods. Most of the current fault localization approaches focus on single-fault programs and are not effective for multi-fault programs. For example, coverage-based debugging methods are unable to locate multiple faults simultaneously. This was empirically studied by DiGiuseppe et al. [6]. 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 [11] 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" [5]. Zheng et al. assert that traditional coverage-based technique "cannot distinguish between useful bug predictor and predicates that are secondary manifestations of bugs" [28].

The MBD theory [20] 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 1 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 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

Table 1 The types of trace formula used in formula-based fault localization and their specificities.

Trace Formula Type	Authors	Encoding	Formula Size	Construction Method
Flow-insensitive	BugAssist [12] *2011*	Counterexample	Very small	BMC or testing
Flow-sensitive	Christ et al. [3] *2013*	Counterexample + Partial control flow	Small	Static analysis
Full Flow-sensitive	Lamraoui et al. [14] *2014*	CFG / SSA	Large	Static analysis
Spectrum-based	Tarantula [11] *2005*			Testing

Trace Formula Type	Single Fault Types		Multi-fault Types		
	Data Flow	Control Flow	Data-dependent	Control-dependent	Independent
Flow-insensitive	☑	☐	☑	☐	☑
Flow-sensitive	☑	☑	☑	☐	☑
Full Flow-sensitive	☑	☑	☑	☑	☑
Spectrum-based	☑	☑	☐	☐	☐

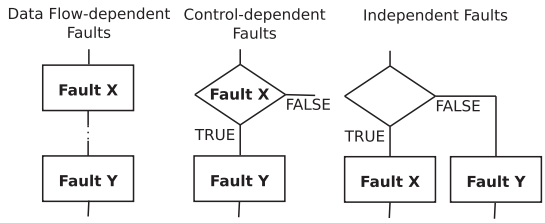

Fig. 1 The different types of multiple faults.

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.

3.2 Importance of Trace Formulas

In formula-based fault localization methods, a program under test is 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 1** summarizes the encoding methods in existing work and our proposal.

A simple approach [12] 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 [3]. Basically, a flow-sensitive TF is similar to flow-insensitive TF with the exception that flow-sensitive TF includes some control flow information 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 [23]. 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.

In this work we introduce a new trace formula encoding called full flow-sensitive TF, which is essentially equivalent to the program's CFG (see Section 5.3 for details). 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 above, 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 Section 5.4 an efficient algorithm for computing diagnoses with full flow-sensitive TF.

3.3 Failing Test Inputs

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 6.2).

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. The association of Algorithm 1 and the combination method of Definition 5 of this paper allow the efficient combination of root causes (MCSes), each obtained from different failing paths.

4. Preliminaries

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., Ref. [16]).

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 paper, 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 (Section 6) are considered *real root causes*.

Failing Program Paths Let φ^{AL} be a formula $EI \wedge TF \wedge AS$ in conjunctive normal form (CNF) where EI is a formula that encodes the error-inducing inputs, TF is a trace formula that encodes all the possible program paths, and AS is a formula that encodes the assertion the program must satisfy. The formula AS can be the post-condition or the test oracle. The formula EI represents the input arguments, which take some particular values that make TF violate AS. The detailed representation of TF is irrelevant here, and will be introduced in Section 5.3.

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. A slice is a set of instructions of a program that can affect a variable v at a line l, a slicing criterion. It is a subset of instructions obtained from the whole program. 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. However, finding root causes in lengthy slices is difficult because many statements are included.

In the following definitions C is a set of clauses, which constitutes a CNF formula φ. We use C and φ interchangeably.

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.

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 ($MSS^{\complement}$) [16].

Fault Localization Problem Revisited The fault localization problem is to find MCSes of φ^{AL}. Two approaches are possible. A classical model-based debugging method first calculates MUSes of φ^{AL} and then obtains MCSes using the hitting set of MUSes. The formula-based method adapted in this paper first calculates MSSes of φ^{AL} and then obtain MCSes by taking the complement of MSSes. In both approaches, enumerating all MCSes is mandatory to cover all the root causes.

Example Using the example in Listing 1, we explain the above concepts. In line 6, there is an error in the computation of the absolute value. The absolute value of `x` is equal to `x*1` (with `x` negative), which violates the assertion at line 8 which expects `abs` to be greater or equal to zero.

Listing 1: A function that computes an absolute value.

```
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 }
```

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. See Section 5.1 concerning the mapping of clauses in TF to the original line numbers.

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

The assertion in line 8 is encoded in AS as follows: $AS = (abs_3 \geq 0)$.

We obtain two MSSes and two MCSes below. The set elements represent the line numbers of the program in Listing 1. 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:

$MSS_0 = \{6\}$ $MCS_0 = MSS_0^{\complement} = \{3, 4\}$
$MSS_1 = \{3, 4\}$ $MCS_1 = MSS_1^{\complement} = \{6\}$
$MCSes = \{MCS_0, MCS_1\} = \{\{3, 4\}, \{6\}\}$
$MUSes = MCSes^{\mathsf{MHS}} = \{\{4, 6\}, \{3, 6\}\}$

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 [12] 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.

Partial Maximum Satisfiability The maximum satisfiability (MaxSAT) problem for a CNF formula is to find an assignment that maximizes the number of satisfied clauses. Any solution to MaxSAT problem is also an MSS. However, every MSS is not necessarily a solution to MaxSAT [18].

In the partial MaxSAT (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.

5. SNIPER

5.1 Tool Overview

We implemented our method in a tool called SNIPER [14] (SNIPER is Not an Imperative Program Errors Repairer). **Figure 2** depicts an overview of SNIPER. Since SNIPER includes in itself the LLVM [15] and Yices [7] libraries, the tool can be considered as standalone. The advantage of a standalone tool is that we have full control at each stage. This makes it possible for SNIPER to collect several information leading the error localization to be precise and extensive.

SNIPER takes as input a source program with some specifications (cf., pre- and post-condition). This input program is translated to an intermediate representation (IR) with `clang` and then pre-processed as explained in Section 5.2. From this IR, we construct the *TF* formula and the *AS* formula. Because we work at the bytecode level (IR), 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 LLVM instruction. To do this we add debugging options using the `-g` command-line option with the command `clang`. We can now retrieve the corresponding line number of any LLVM instruction. When encoding LLVM instructions, we tag each clause of the trace formula with the line number of the corresponding LLVM instruction.

Regarding the formulas that represent the error-inducing inputs (*EIes*), they are generated from failing test cases given as input. Then, using the *TF* formula, the *AS* formula and the set of error-inducing input formulas *EIes* we compute a set of diagnoses (MCSes) (see Section 5.4 for details) and combine them to obtain a set of complete diagnoses (see Section 5.5 for details). The complete diagnoses, which is a concept introduced in this paper, are output to the user under the form of source code lines marked with potential root causes.

5.2 Program Pre-processing

Before the program is encoded into a trace formula, it must be pre-processed. The input is an ANSI C program [*1] that is sequential. Pre-processing a program starts by translating it to an LLVM [15] intermediate representation (IR) with `clang`. The resulting IR is then transformed into a loop-free IR that contains a single function. Most of these operations are standard in the bounded model-checking (BMC) of imperative programs (cf., Refs. [4], [17]). First, all function calls are inline-expanded, meaning that the call instructions are replaced by the callee function bodies. The second step consists of unrolling all loops to a specified bound, and unfolding all recursive functions a certain number of time. Finally, the IR is put in static single assignment (SSA) form. At this point the IR contains a single local function with arithmetic, comparison, ϕ (join), and branching instructions only.

5.3 Full Flow-sensitive Trace Formula

We describe how we translate a pre-processed LLVM IR [*2] to a partial SMT formula. Our encoding takes into account both the control and data flow of programs. We can produce full flow-sensitive trace formulas.

5.3.1 Data Flow

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:

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

[*1] The version of SNIPER used for this paper supports a subset of ANSI C only.
[*2] For sake of simplicity we omit some details about the IR [15].

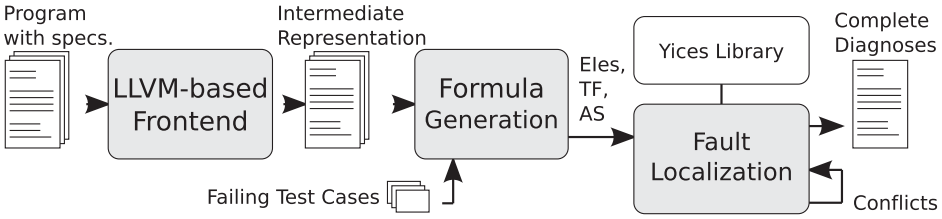

Fig. 2 SNIPER tool flow.

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 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 $\text{on}(entry)$ 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.

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

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 5.4 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.

5.4 Computing Diagnoses

Algorithm 1 implements by using pMaxSMT the AllMCS function, which finds all the MCSes (diagnoses) of TF. This algorithm makes use of the *push & pop* mechanism of Yices [7]. 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 predicates) between the matching push and pop operators are flushed, and the context is completely restored to what it was right before the push. This mechanism is very 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 modifications and reuse the same context basis many times.

Algorithm 1 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 sets of diagnoses (MCSes)

1: $\varphi_W \leftarrow \varphi_{TF}$ ▷ φ_W is the working formula
2: $AV \leftarrow \emptyset$
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 var. created
7: $\varphi_{soft} \leftarrow \varphi_{soft} \cup \{(\neg a_i)\}$
8: $w_A \leftarrow (w \vee a_i)$
9: ▷ Remove w and add w_A as hard
10: $\varphi_W \leftarrow \varphi_W \setminus \{w\} \cup \{(w_A)^{\mathsf{HARD}}\}$
11: **end for**
12: **if** $AV = \emptyset$ **then**
13: **return** $\emptyset$ ▷ No MaxSMT solution
14: **end if**
15: $D \leftarrow \emptyset$
16: $C \leftarrow \varphi_W \cup \varphi_{soft} \cup \varphi_{AS}$ ▷ Add the formulas in the context
17: **for each** $e_i \in E$ **do**
18: `push(`C`)` ▷ Save the context
19: $C \leftarrow C \cup e_i$ ▷ Add the error-inducing input in the context
20: $M \leftarrow \emptyset$
21: **while true do**
22: (st, $\varphi_{MSS}, \mathcal{A}$) $\leftarrow$ `pMaxSMT(`C`)` ▷ Solve the context
23: ▷ "φ_{MSS}" is an MSS if st is **true**
24: ▷ "$\mathcal{A}$" is a maximal satisfying assignment if st is **true**
25: **if** st = **true then**
26: ▷ The complement of an MSS is an MCS
27: $\varphi_{MCS} \leftarrow$ `CoMSS(`φ_{MSS}`)`
28: $M \leftarrow M \cup \{\varphi_{MCS}\}$
29: ▷ Add the blocking constraint
30: $C \leftarrow C \cup \{(\bigvee_{\mathcal{A}(a_i)=\textbf{true}} \neg a_i)\}$
31: **else**
32: **break**
33: **end if**
34: **end while**
35: **if** $M \neq \emptyset$ **then**
36: $D \leftarrow D \cup \{M\}$
37: **end if**
38: `pop(`C`)` ▷ Restore the context (pushed in line 18)
39: **end for**
40: **return** D

The MCSes are enumerated for all error-inducing inputs. A set of MCS (MCSes) can be computed using MaxSAT. Algorithm 1 is based on the MaxSAT-based MCS enumeration algorithm of Morgado et al. [18]. Morgado's algorithm enumerates all MCSes of a given formula. We implement the algorithm with Yices [7].

5.5 Combination of MCSes

Algorithm 1 provides a function that returns MCSes for each error-inducing inputs given as arguments. Each of these sets are root cause candidates for one failing execution, which are triggered by the error-inducing inputs associated to the set. The problem of combining MCSes 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. As discussed in Section 3, independent multiple faults need more than one failing execution. We call such gathering of sets a *complete diagnosis*.

Definition 5 (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.

A set C of complete diagnoses can be calculated using a n-ary pairwise union as defined in Definition 6. 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.

Definition 6 (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 6. Nevertheless, we show below a property that directly follows Definition 6. 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 1, 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 6), 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, $C' = D'$. □

Let us illustrate this with an example:

Listing 2: A multi-fault program.

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

Example When running Algorithm 1 on the TF of program in Listing 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*:

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

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.

6. Experiments

In this section we show the capabilities of SNIPER with some experiments made on the TCAS benchmark of the Siemens Test Suite [10], [21] and on a benchmark provided by Bekkouche [1].

Since a fault localization method performs differently depending on the program nature, we would like to experiment our method on different kinds of programs. The TCAS program is mostly composed of comparisons. All program versions in the TCAS benchmark are the same except that the injected faults are different. The injected multiple faults are all of type *data flow-dependent faults* and *independent faults*. The benchmark provided by Bekkouche is composed of various kinds and sizes of programs. 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 testing community. The injected multiple faults are all of type *control-dependent faults*. Finally, the TCAS comes with a set of test cases, which is not the case of the Bekkouche's benchmark. For the latter, we will generate a set of test cases by ourselves.

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.

6.1 TCAS Benchmark

One of the Siemens Test Suite tasks is the TCAS, which is sometimes used in program testing research [10], [21]. The authors of the suite created 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 1,578 test cases. However, no specification is given.

6.1.1 Experimental Setup

We used the same experimental setup as described in Ref. [12] because we compare the experiment results with BugAssist. 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 5.5 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, which are the test cases that give an output different from the correct output.

6.1.2 Results for Single and Multiple Faults

Table 2 [*3] reports the results of running SNIPER 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 #FTC shows the number of failing test cases included in the TCAS benchmark set. The right part of Table 2 shows the results of SNIPER and BugAssist. The results of BugAssist were taken from Ref. [12]. Each column shows the number of time the tools were able to detect at least one of the injected fault locations.

Table 2 Results of SNIPER and BugAssist on the TCAS.

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

* A new option of SNIPER that checks the array index overflow/underflow can detect the missing one.

In total, BugAssist pin-pointed 1,364 times the injected fault location out of the 1,437 runs (73 misses). SNIPER pin-pointed the injected fault location 1,435 times out of the 1,437 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 SNIPER and 8.00% for BugAssist [12]. 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. 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 can be found with the given test cases were successfully localized. Particularly, the data flow-dependent faults in the version no.32

[*3] Versions no.33 and no.38 are omitted from Table 2 in order to compare the results with BugAssist [12], which does not have entries for them.

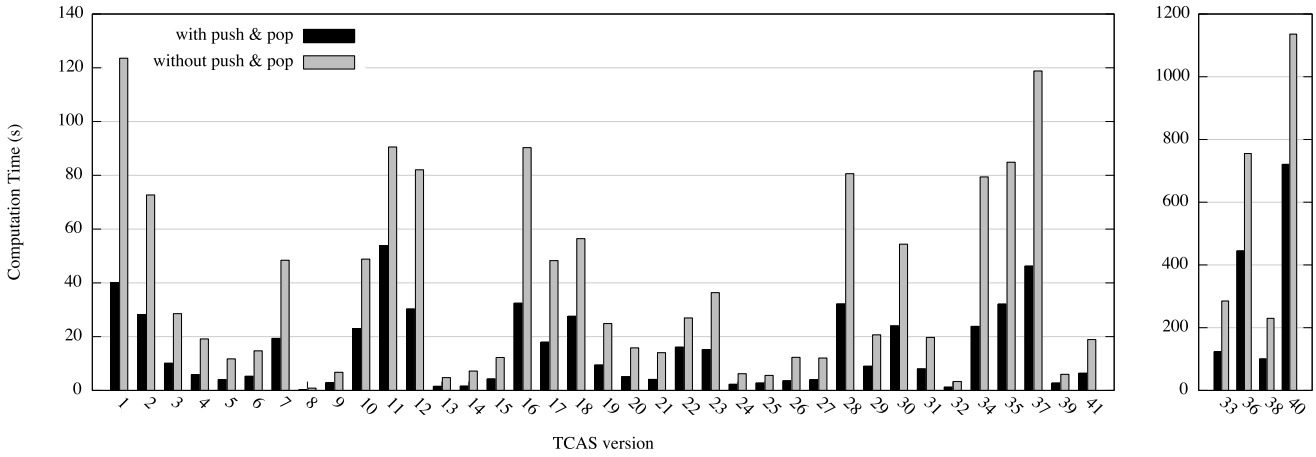

Fig. 3 Results of running SNIPER on the TCAS benchmark with and without push & pop optimization.

could be successfully identified. In the version no.31, the failed test cases provided in TCAS test suits only cover one of the two buggy statements. 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.

In addition to identifying all the single and multiple faults with the given test cases, the CSR is almost the same as BugAssist although SNIPER adapts a complete enumeration algorithm.

6.1.3 Push & Pop Optimization Results

Figure 3 reports the computation times of Algorithm 1 on the TCAS benchmark with and without the *push & pop* optimization, which was explained in Section 5.4. The histograms are separated in two parts for readability. The bars in gray represent the times with the optimization disabled and the bars in black represent the times with the optimization activated.

We can see that the computation time is reduced when using the optimization. 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 some small modifications between the calls to the solver.

6.2 Bekkouche's Benchmark

The Bekkouche's benchmark [1] consists of several C programs of 15 to 100 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.

6.2.1 Experimental Setup

As explained in Section 5.5 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 Bekkouche's benchmark, we use a concolic unit testing engine implemented in SNIPER for generating such inputs. This concolic engine has been run without any timeout because the programs are not big. In this experiment, the engine could visit all program paths before terminating.

6.2.2 Results for Single and Multiple Faults

Table 3 lists the results obtained by running SNIPER on each benchmark program [1]. The first column of the table shows the program name. The column #SI shows the number of successful inputs. The column #EI shows the number of error-inducing inputs. See Section 6.2.1 for details concerning the input generation method. 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.

Table 3 Results of running SNIPER on the Bekkouche's benchmark.

Programs	#SI	#EI	Found	Time (ms)
MinmaxKO	7	2	1/1	35
AbsMinusKO	2	1	1/1	27
AbsMinusKO2	0	3	1/1	38
AbsMinusKO3	2	1	1/1	22
TritypeKO	14	1	1/1	373
TritypeKO2	14	2	1/1	380
TritypeKO2V2	15	2	1/1	367
TritypeKO3	15	2	1/1	542
TritypeKO4	13	1	1/1	236
TritypeKO5	6	8	2/2	622
TriPerimetreKO	13	1	1/1	430
TriPerimetreKOV2	13	1	1/1	583
TriPerimetreKO2	16	1	1/1	332
TriPerimetreKO3	15	2	1/1	656
Maxmin6varKO	419	37	1/1	30,872
Maxmin6varKO2	404	56	1/1	24,365
Maxmin6varKO3	404	56	2/2	26,731
Maxmin6varKO4	403	61	3/3	32,592

For all programs, SNIPER was able to localize all faults. The multi-fault program `Maxmin6varKO3` contains a nested fault, which is a fault that is masked by another fault. These types of faults (control-dependent faults in Fig. 1) are especially difficult to deal with because it is impossible to generate a test case that covers the masked fault. As explained in Section 3, coverage-based or spectrum-based debugging methods, such as Tarantula [11], are unable to locate the second fault in the program `Maxmin6varKO3`.

Listing 3: Code fragment from program Maxmin6varKO3 showing the two faults (underlined).

```
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 3 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 view point of MCS enumeration with pMaxSAT (see Section 5.4 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.

Let us consider a concrete scenario of SNIPER on program `Maxmin6varKO3`. For an input of $(a = 1, b = -2, c = 0, d = 0, e = -1, f = -1)$ we obtain the following diagnosis:

$$\{..., \{(tmp38 = (b > d)),\ (tmp46 = (c < d))\}, ...\}.$$

Since $(b = -2)$ and $(d = 0)$, the variable $tmp38$ is false. The solver relaxes the clause $(tmp38 = (b > d))$, and the variable $tmp38$ is assigned to true, then the control flow goes in the "then" branch. The variable `max` is assigned to the value 1 (a). Since $(c = 0)$ and $(d = 0)$, the variable $tmp46$ is false. The solver relaxes the clause $(tmp46 = (c < d))$, and the variable $tmp46$ is assigned to true, then control flow goes in the "then" branch. The variable `min` is assigned to the value -2 (b). At this point, `min` and `max` are correctly assigned and the post-condition is satisfied.

Note that the search is only possible with full flow-sensitive TF, which is equivalent to the program's CFG (see Section 5.3), because it encodes alternative paths to the failing path, which means that the solver (used in Algorithm 1) can relax clauses to force the control flow to take an alternative path that covers both faults.

7. Summary and Future Work

We presented a new formula-based method for automatic fault localization, which combined the SAT-based formal verification techniques with the model-based diagnosis theory. It has two core algorithms, computing all the diagnoses and combining them. In addition, we introduced a new way of encoding programs, namely the full flow-sensitive trace formula. The association of the two core algorithms and the full flow-sensitive trace formula enables the localization of root causes of multi-fault programs. SNIPER adapts partial maximum satisfiability to implement the algorithms efficiently, which makes use of the *push & pop* mechanism of Yices. Furthermore, the multiple faults in the TCAS benchmark programs and Bekkouche's benchmark programs could successfully be detected by combining a set of the results obtained from multiple failing program paths.

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. Note that we need only a set of failing traces, which is different from the coverage-based methods in which unbiased test suits are needed for both successful and failing traces.

Acknowledgments This research was partially supported by JSPS KAKENHI Grant Number 24300010 and the Kayamori Foundation of Informational Science Advancement.

List of Acronyms

ACSR	Average Code Size Reduction
BMC	Bounded Model-Checking
CFG	Control Flow Graph
CNF	Conjunctive Normal Form
CSR	Code Size Reduction
IR	Intermediate Representation
MaxSAT	Maximum Satisfiability
MBD	Model-Based Diagnosis
MCS	Minimal Correction Subset
MSS	Maximal Satisfiable Subset
MUS	Minimal Unsatisfiable Subset
pMaxSAT	Partial Maximum Satisfiability
SAT	Satisfiability Problem
SMT	Satisfiability Modulo Theories
SSA	Static Single Assignment
TCAS	Traffic Collision Avoidance System
TF	Trace Formula

References

[1] Bekkouche, M.: ANSI-C Benchmark, available from ⟨http://www.i3s.unice.fr/~bekkouch/Benchs_Mohammed.html⟩.
[2] Binkley, D., Gold, N. and Harman, M.: An Empirical Study of Static Program Slice Size, *ACM TOSEM*, Vol.16, No.2, Article 8 (2007).
[3] Christ, J., Ermis, E., Schaf, M. and Wies, T.: Flow-Sensitive Fault Localization, *Proc. VMCAI 2013*, pp.189–208 (2013).
[4] Clarke, E., Kroening, D. and Lerda, F.: A Tool for Checking ANSI-C Programs, *Proc. TACAS 2004*, pp.168–176 (2004).
[5] Denmat, T., Ducass, M. and Ridoux, O.: 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).
[6] N. DiGiuseppe and Jones, J.A.: On the Influence of Multiple Faults on Coverage-based Fault Localization, *Proc. ISSTA'11*, pp.210–220 (2011).
[7] Dutertre, B. and de Moura, L.: The Yices SMT Solver, available from ⟨http://yices.csl.sri.com⟩.
[8] Griesmayer, A., Staber, S. and Bloem, R.: Fault Localization using a Model Checker, *STVR*, pp.149–173 (2010).
[9] Groce, A., Chaki, S., Kroening, D. and Strichman, O.: Error Explanation with Distance Metrics, *STTT*, Vol.8, No.3, pp.229–247 (2006).
[10] Hutchins, M., Foster, H., Goradia, T. and Ostrand, T.: Experiments of the Effectiveness of Dataflow- and Controlflow-based Test Adequacy Criteria, *Proc. ICSE '94*, pp.191–200 (1994).
[11] Jones, J.A. and Harrold, M.J.: Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique, *Proc. ASE '05*, pp.273–282 (2005).
[12] Jose, M. and Majumdar, R.: Cause Clue Clauses: Error Localization using Maximum Satisfiability, *Proc. PLDI 2011*, pp.437–446 (2011).
[13] Kusumoto, S., Nishimatsu, A., Nishie, K. and Inoue, K.: Experimental Evaluation of Program Slicing for Fault Localization, *Empirical Software Engineering*, Vol.7, No.1, pp.49–76 (2002).
[14] Lamraoui, S. and Nakajima, S.: A Formula-based Approach for Automatic Fault Localization of Imperative Programs, *Proc. ICFEM'14*, pp.251–266 (2014).

[15] Lattner, C. and Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation, *Proc. CGO'04*, pp.75–86 (2004).
[16] Liffiton, M.H. and Sakallah, K.A.: Algorithms for Computing Minimal Unsatisfiable Subsets of Constraints, *Automated Reasoning*, Vol.40, No.1, pp.1–33 (2008).
[17] Merz, F., Falke, S. and Sinz, C.: LLBMC: Bounded Model Checking of C and C++ Programs Using a Compiler IR, *Proc. VSTTE'12*, pp.146–161 (2012).
[18] Morgado, A., Liffiton, M. and Marques-Silva, J.: MaxSAT-Based MCS Enumeration, *Proc. HVC-2012*, pp.86–101 (2012).
[19] Prasad, M.R., Biere, A. and Gupta, A.: A Survey of Recent Advances in SAT-Based Formal Verification, *STTT*, Vol.7, No.2, pp.156–173 (2005).
[20] Reiter, R.: A Theory of Diagnosis from First Principles, *Artificial Intelligence*, Vol.32, No.1, pp.57–95 (1987).
[21] Rothermel, G. and Harrold, M.: Empirical studies of a safe regression test selection technique, *IEEE Trans. Softw. Eng.*, Vol.24, No.6, pp.401–419 (1990).
[22] Safarpour, S., Mangassarian, H., Veneris, A., Liffiton, M.H. and Sakallah, K.A.: Improved Design Debugging using Maximum Satisfiability, *Proc. FMCAD'07*, pp.13–19 (2007).
[23] Schäf, M., Schwartz, C.D. and Wies, T.: Explaining Inconsistent Code, *Proc. ESEC/SIG-SOFT FSE*, pp.521–531 (2013).
[24] Weiser, M.: Programmers Use Slices When Debugging, *Comm. ACM*, Vol.25, No.7, pp.446–452 (1982).
[25] Wotawa, F.: On the Relationship between Model-based Debugging and Program Slicing, *Artificial Intelligence*, Vol.135, No.1, pp.125–143 (2002).
[26] Wotawa, F., Nica, M. and Moraru, I.: Automated Debugging based on a Constraint Model of the Program and a Test Case, *Logic and Algebraic Programming*, Vol.81, No.4, pp.390–407 (2012).
[27] Zeller, A. and Hildebrandt, R.: Simplifying and Isolating Failure-Inducing Input, *IEEE Trans. Softw. Eng.*, Vol.28, No.2, pp.183–200 (2002).
[28] Zheng, A.X., Jordan, M.I., Liblit, B., Naik, M. and Aiken, A.: Statistical Debugging: Simultaneous Identification of Multiple Bugs, *Proc. ICML'06*, pp.1105–1112 (2006).

Si-Mohamed Lamraoui was born in 1987. He received his M.S. degree from Joseph Fourier University in 2012 and he is currently a Ph.D. candidate at SOKENDAI, Japan. His research interest are program verification and model checking.

Shin Nakajima received his B.S. and M.S. degrees from the University of Tokyo 1979 and 1981 respectively, and Ph.D. degree in 2000. He is currently a professor at National Institute of Informatics, Tokyo, Japan. His current interests include formal methods, automated verification, and software modeling. He is a member of the FME, ACM, JSSST, and IPSJ.