Research Article

Combining Slicing and Constraint Solving for Better Debugging: The CONBAS Approach

Birgit Hofer and Franz Wotawa

Institute for Software Technology, Graz University of Technology, 8010 Graz, Austria

Correspondence should be addressed to Franz Wotawa, wotawa@ist.tugraz.at

Received 1 June 2012; Accepted 17 August 2012

Academic Editor: Zhenyu Zhang

Copyright © 2012 B. Hofer and F. Wotawa. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Although slices provide a good basis for analyzing programs during debugging, they lack in their capabilities providing precise information regarding the most likely root causes of faults. Hence, a lot of work is left to the programmer during fault localization. In this paper, we present an approach that combines an advanced dynamic slicing method with constraint solving in order to reduce the number of delivered fault candidates. The approach is called Constraints Based Slicing (CONBAS). The idea behind CONBAS is to convert an execution trace of a failing test case into its constraint representation and to check if it is possible to find values for all variables in the execution trace so that there is no contradiction with the test case. For doing so, we make use of the correctness and incorrectness assumptions behind a diagnosis, the given failing test case. Beside the theoretical foundations and the algorithm, we present empirical results and discuss future research. The obtained empirical results indicate an improvement of about 28% for the single fault and 50% for the double-fault case compared to dynamic slicing approaches.

1. Introduction

Debugging, that is, locating a fault in a program and correcting it, is a tedious and very time-consuming task that is mainly performed manually. There have been several approaches published that aid the debugging process. However, these approaches are hardly used by programmers except for tools allowing to set breakpoints and to observe the computation of variable values during execution. There are many reasons that justify this observation. In particular, most of the debugging tools do not smoothly integrate with the software development tools. In addition, debuggers fail to identify unique root causes and still leave a lot of work for the programmer. Moreover, the approaches can also be computationally demanding, which prevents them from being used in an interactive manner. In this paper, we do not claim to solve all of the mentioned problems. We discuss a method that improves debugging results when using dependence-based approaches like dynamic slicing. Our method makes use of execution traces and the dependencies between the executed statements. In contrast to slicing, we also use symbolic

execution to further reduce the number of potential root causes.

In order to introduce our method, we make use of the program `numfun` that implements a numeric function. The program and a test case are given in Figure 1. When executing the program using the test case, `numfun` returns a value of 10 for variable `f`, which contradicts the expectations. Since the value for variable `g` is computed correctly, we only need to consider the dependencies for variable `f` at position 6. Tracing back these dependencies, we are able to collect statements 6, 4, 3, 2, and 1 as possible fault candidates. Lines 7 and 5 can be excluded because both lines do not contribute to the computation of the value for `f`. The question now is can we do better? In the case of `numfun`, we are able to further exclude statements from the list of candidates. For example, if we assume that the statement in Line 4 is faulty, the value of `y` has to be computed in a different way. However, from Statement 7, the value of `y` = 6 can be derived using the value of `z` = 6 and the expected outcome `g` = 12. Knowing the value of `y`, we are immediately able to derive a value for `f` = 6 + 6 - 2 = 10 and again we obtain a contradiction with the expected value. As a consequence

```

// TEST INPUT: a=2, b=2, C=3, d=3, e=2
(1) cond = a > 0 & b > 0 & c > 0 & d > 0 & e > 0;
(2) if cond {
(3)   x = a * c;
(4)   y = b * d;
(5)   z = c * e;
(6)   f = x + y - a;
(7)   g = y + z;
(8) } else {
(9)   f = 0;
(10)  g = 0; }
// TEST OUTPUT: f = 12, g = 12

```

FIGURE 1: The program `numfun` computing a numeric function, the test input, and the expected test output.

the assumption that Line 4 is faulty alone cannot be true and must be retracted.

Using the approach of assuming correctness and incorrectness of statements and proving consistency with the expected values, we are able to reduce the diagnosis candidates to Lines 1, 2, 3, and 6. It is also worth to mention that we would also be able to remove statements 1 and 2 from the list of candidates. For this purpose we only have to check whether a different value of `cond` would lead to a different outcome or not. For this example, assuming `cond` to be false also leads to an inconsistency. However, when using such an approach possible alternative paths have to be considered. This extension makes such an approach computationally more demanding.

From the example we are able to summarize the following findings. (1) Using data and control dependences reduces the number of potential root causes. Statements that have no influence on faulty variables can be ignored. (2) During debugging, we make assumptions about the correctness or incorrectness of statements. From these assumptions, we try to predict a certain behavior, which should not be in contradiction with the expectations. (3) Backward reasoning, that is, deriving values for intermediate variables from output variables and other variables, is essential to further reduce the number of fault candidates. In this case, statements are not interpreted as functions that change the state of the program but as equations. The interpretation of statements as equations allows us to compute the input value from the output value. (4) Further reductions of fault candidates can be obtained when choosing alternative execution paths, which is computationally more demanding than considering the execution trace of a failing test case only.

In this paper, we introduce an approach that formalizes findings (1)–(3). However, finding (4) is not taken into consideration, because of the resulting computational overhead. In particular, we focus on providing a methodology that allows for reducing the size of dynamic slices. Reducing the size of dynamic slices improves the debugging capabilities of dynamic slicing. We introduce the basic definitions and algorithms and present empirical results. We gain a reduction of more than 28% compared to results obtained from pure

slicing. Although the approach increases the time needed for identifying potential root causes, the overhead can be neglected at least for smaller programs. It is also worth noting that we do not claim that the proposed approach is superior compared to all other debugging approaches. We believe that a combination of approaches is necessary in practice. Our contribution to the field of debugging is in improving dynamic slicing with respect to the computed number of bug candidates.

This paper is based on previous work [1], where the general idea is explained. Now, we focus on the theoretical foundations and an extended empirical evaluation. In the reminder of this paper, we discuss related work in Section 2. We introduce the basic definitions, that is, the used language, execution traces, relevant slicing, model-based debugging, and the conversion into a constraints system in Section 3. We formalize our approach, named constraints based slicing (CONBAS), in Section 4. In Section 5, we apply CONBAS to several example programs. We show that CONBAS is able to reduce the size of slices for programs with single and double faults without losing the fault localization capabilities. In addition, we apply CONBAS to circuits. In Section 6 we discuss the benefits and limitations of the approach as well as future work. Finally, we conclude the paper in Section 7.

2. Related Research

Software debugging techniques can be divided into fault localization and fault correction techniques. Fault localization techniques focus on narrowing down possible fault locations. They comprise spectrum-based fault localization, delta debugging, program slicing, and model-based software debugging.

- (i) Spectrum-based fault localization techniques are based on an observation matrix. Observation matrices comprise the program spectra of both passing and failing test cases. Harrold et al. [2] give an overview of different types of program spectra. A high similarity of a statement to the error vector indicates a high probability that the statement is responsible for the error [3]. There exist several similarity coefficients to numerically express the degree of similarity, for example, Zoetewij et al. [4] and Jones and Harrold [5]. Empirical studies [3, 6] have shown that the Ochiai coefficient performs best. Several techniques have been developed that combine spectrum-based fault localization with other debugging techniques, for example, BARINEL [7] and DEPUTO [8].
- (ii) Delta debugging [9] is a technique that can be used to systematically minimize a failure-inducing input. The basic idea of delta debugging is that the smaller the failure-inducing input, the less program code is covered. Zeller et al. [10, 11] adopted delta debugging to directly use it for debugging.
- (iii) Program slicing [12] narrows down the search range of potentially faulty statements by means of data and control dependencies. A slice is a subset of program

statements that directly or indirectly influence the values of a given set of variables at a certain program line. A slice behaves like the original program for the variables of interest. Slices will be discussed in detail in Section 2.1.

- (iv) Model-based software debugging derives from model-based diagnosis, which is used for locating faults in physical systems. Mayer and Stumptner [13] give an overview of existing model-based software debugging techniques. Some of these techniques will be discussed in detail in Section 2.2.

Fault correction techniques focus on finding solutions to eliminate an observed misbehavior. They comprise, for instance, genetic programming techniques.

Genetic programming deals with the automatic repair of programs by means of mutations on the program code. Arcuri [14] and Weimer et al. [15, 16] deal with this debugging technique. Genetic programming often uses fault localization techniques as a preprocessing step.

In the following, we will discuss techniques that are related to our approach, that is, slicing and model-based debugging, in detail.

2.1. Slicing. Weiser [12] introduced static program slicing as a formalization of reasoning backwards. The basic idea here is to start from the failure and to use the control and data flow of the program in the backward direction in order to reach the faulty location. Static program slices tend to be rather large. For this reason, Korel and Laski [17] introduced dynamic program slicing which relies on a concrete program execution. Dynamic slicing significantly reduces the size of slices. Occasionally, statements which are responsible for a fault are absent in the slice. This happens when the fault causes the nonexecution of some parts of the program. Relevant slicing [18] is a variant of dynamic slicing, which eliminates this problem.

A mentionable alternative to relevant slicing is the method published by Zhang et al. [19]. This method introduces the concept of implicit dependencies. Implicit dependencies are obtained by predicate switching. They are the analog to potential data dependencies in relevant slicing. The obtained slices are smaller since the use of implicit dependencies avoids a large number of potential dependencies.

Sridharan et al. [20] identify data structures as the main reason for too large slices. They argue that data structures provided by standard libraries are welltested and thus they are uncommonly responsible for observed misbehavior. Their approach, called Thin Slicing, removes such statements from slices.

Gupta et al. [21] present a technique that combines delta debugging with forward and backward slices. Delta debugging is used to find the minimal failure-inducing input. Forward and backward slices are computed for the failure-inducing input. The intersection of the forward and backward slices results in the failure-inducing chop. This technique requires a test oracle in contrast to CONBAS.

Zhang et al. [22] introduced a technique that reduces the size of dynamic slices via confidence values. These confidence values represent the likelihood that the corresponding statement computed the correct value. Statements with a high confidence value are excluded from the dynamic slice. Similar to CONBAS, this approach requires only one failing execution trace. However, it requires one output variable with a wrong value and several output variables where the computed output is correct. In contrast, CONBAS requires at least one output variable with a wrong value, but no correctly computed output variables.

Other mentionable work of Zhang et al. includes [23–25]. In [24], they discuss how to reduce the time and space required for saving dynamic slices. In [23], they evaluate the effectiveness of dynamic slicing for locating real bugs and found out that most of the faults could be captured by considering only data dependencies. In [25], they deal with the problem of handling dynamic slices of long running programs.

Jeffrey et al. [26] identify potential faulty statements via value replacement. They systematically replace the values used in statements so that the computed output becomes correct. The original value and the new value are stored as an interesting value mapping pair (IVMP). They state that IVMPs typically occur at faulty statements or statements that are directly linked via data dependences to faulty statements. They limit the search space for the value replacement to values used in other test cases. We do not limit the search space to values used in other test cases. Instead, our constraint solver determines if there exist any values for the variables in an abnormal statement so that the correct values for the test cases can be computed. On the one hand, our approach is computationally more expensive, but on the other hand it does not depend on the quality of other test cases. As Jeffrey et al. stated, the presence of multiple faults can diminish the effectiveness of the value replacement approach. In contrast, the CONBAS approach is designed for handling multiple faults.

Many other slicing techniques have been published. For a deeper analysis on slicing techniques the reader is referred to Tip [27] for slicing techniques in general and to Korel and Rilling [28] for dynamic slicing techniques.

2.2. Model-Based Software Debugging. Our work builds on the work of Reiter [29] and Wotawa [30]. Reiter describes the combination of slicing with model-based diagnosis. Wotawa proves that conflicts used in model-based diagnosis for computing diagnoses are equivalent to slices for variables where the expected output value is not equivalent to the computed one.

Nica et al. [31] suggest an approach that reduces the number of diagnoses by means of program mutations and the generation of distinguishing test cases. Our approach and [31] differ in two major aspects. First, our approach uses the execution trace instead of the source code. Thus, we do not have to explicitly unroll loops. Second, we use a constraint solver to check whether a solution can be found. The diagnosis candidates are previously computed via

the hitting set algorithm. In contrast, Nica et al. [31] use a constraint solver to obtain the diagnoses directly.

Wotawa et al. [32] present a model-based debugging approach which relies on a constraint solver as well. They show how to formulate a debugging problem as a constraint satisfaction problem. Similar to [31], they use source code instead of execution traces. Other related work includes research in applying model-based diagnosis for debugging directly, for example, [33, 34] and research in applying constraints for the same purpose, for example, [35].

3. Basic Definitions

In this chapter, we introduce the basic definitions that are necessary for formalizing our approach. Without restricting generality we make some simplifying assumptions like reducing the program language to a C-like language and ignoring arrays and pointers. However, this language is still Turing-complete. The reason for the simplification is to focus on the underlying ideas instead of solving purely technical details. We start this chapter with a brief introduction of the underlying programming language $\mathcal{L}$. We define execution traces and dynamic slices formally. Afterwards, we define test cases and the debugging problem. Finally, we introduce model-based software debugging and the conversion of execution traces into their constraint representation.

3.1. Language. The syntax definition of $\mathcal{L}$ is given in Figure 2. The start symbol of the grammar in Backus-Naur form (BNF) is P . A program comprises a sequence of statements B . In $\mathcal{L}$, we distinguish three different types of statements: (1) the assignment statement, (2) the if-then-else statement, and (3) the while statement. In the following, we will refer to if-then-else statements and while statements as conditional statements (or conditionals) and to the conditions in conditional statements as test elements. The right side of an assignment has to be a variable (id). The name of the variable can be any word except a keyword. An expression is either an integer (num), a truth value (true, false), a variable, or two expressions concatenated with an operator. An integer optionally starts with a “-” (if a negative integer is represented) followed by a sequence of digits (0, 1, ..., 9). We do not introduce data types, but we assume that the use of Boolean and integer values follow the usual expected type rules. We further assume that comments start with `//` and go to the end of a line. The program `numfun` (Figure 1) gives an example for a program written in $\mathcal{L}$.

After defining the syntax of $\mathcal{L}$, we have to define its semantics. In this section, we rely on an operational definition. For this purpose, we introduce an interpretation function $\llbracket \cdot \rrbracket : \mathcal{L} \times \Sigma \mapsto \Sigma \cup \{\perp\}$, which maps programs and states to new states or the undefined value $\perp$. In this definition, Σ represents the set of all states. A concrete state $\omega \in \Sigma$ specifies values for the variables used in the program. We call a state also a variable environment. Hence, ω itself is a function $\omega : \text{VARS} \rightarrow \text{DOM}$, where VARS denote the set of variables and DOM its domain comprising all possible values. Note that we also represent $\omega \in \Sigma$ as a set

$P \leftarrow B$
$B \leftarrow SB \mid \epsilon$
$S \leftarrow \text{id} \equiv E; \text{if } E\{B\} \text{OE} \mid \text{while } E\{B\}$
$\text{OE} \leftarrow \text{else } \{B\} \mid \epsilon$
$E \leftarrow \text{id} \mid \text{num} \mid \text{true} \mid \text{false} \mid \text{EOPE} \mid (E)$
$\text{OP} \leftarrow \pm \mid - \mid * \mid / \mid \leq \mid \geq \mid == \mid \leq \mid \geq \mid \& \mid \mid$

FIGURE 2: The semantics of the language $\mathcal{L}$.

$\{\dots, x_i v_i, \dots\}$, where $x_i \in \text{VARS}$ is a variable and $v_i \in \text{DOM}$ is its value. Further note that in our case $\text{DOM} = \mathbb{Z}^M \cup \mathbb{B}$, where $\mathbb{Z}^M = \{x \mid x \in \mathbb{Z}, \min \leq x \leq \max\}$ is an integer number between predefined minimum and maximum values, and $\mathbb{B} = \{\text{T}, \text{F}\}$ is the Boolean domain.

The definition of the semantics of $\mathcal{L}$ is given in Figure 3. We first discuss the semantics for conditions and expressions. For this purpose, we assume that `num` (`id`) represents the lexical value of the token `num` (`id`) used in the definition of the grammar. An integer `num` is evaluated to its corresponding value `num*` $\in \mathbb{Z}^M$ and the truth values are evaluated to their corresponding values in $\mathbb{B}$. A variable `id` is evaluated to its value specified by the current variable environment ω . Expressions with operators are evaluated according to the semantics of the used operator. After defining the semantics of the expressions, we define the semantics of the statements in $\mathcal{L}$ in a similar manner. A sequence of statements, that is, the program itself or a sub-block of a conditional or while statement, $S_1 \dots S_n$ is evaluated by executing the statements S_1 to S_n in the given order. Each statement might change the current state of the program. An assignment statement changes the state for a given variable. All other variables remain unchanged. An if-then-else statement allows for selecting a certain path (via block B_1 or B_2) based on the execution of the condition. A while-statement executes its block B until the condition evaluates to false. Therefore, the formal definition of the semantics is very similar to the semantics definition of an if-then-else statement without else-branch. In order to finalize the definition of the semantics of $\mathcal{L}$, we assume that if a program does not terminate or in case of a division by zero, the semantics function returns $\perp$. Moreover, we further assume that all variable values necessary to execute the program are known and defined in $\omega \in \Sigma$.

When executing the program `numfun` (Figure 1) on the state

$$\omega_I = \{(a, 2), (b, 2), (c, 3), (d, 3), (e, 2)\}, \quad (1)$$

the semantics function $\llbracket \cdot \rrbracket$ on I returns the state

$$\omega_O = \{(a, 2), (b, 2), (c, 3), (d, 3), (e, 2), \\ (\text{cond}, \text{T}), (f, 10), (g, 12)\}, \quad (2)$$

where the value for f contradicts the expected value for the same variable.

When obtaining a result that is in contradiction with the expectations someone is interested in finding and fixing

FIGURE 3: The semantics of the language $\mathcal{L}$.

the fault, that is, locating the statements that are responsible for the faulty computation of a value and correcting them. Weiser [12] introduced the idea to support this process by using the dependence information represented in the program. Weiser's approach identifies those parts of the program that contribute to faulty computations. Weiser called these parts a slice of the program. In this paper, we use extensions of Weiser's static slicing approach and consider the dynamic case where only statements, which are considered in a particular test run, are executed. In order to define dynamic slices [17] and further on relevant slices [18] we first introduce execution traces.

Definition 1 (execution trace). An execution trace of a program $\Pi \in \mathcal{L}$ and an input state $\omega \in \Sigma$ are a sequence $\langle s_1, \dots, s_k \rangle$, where $s_i \in \Pi$ is a statement that has been executed when running Π on test input ω , that is, calling $\llbracket \Pi \rrbracket \omega$.

For our running example `numfun`, the execution trace of the input ω_I is illustrated in Figure 4 and comprises the statements 1–7.

We now define dependence relations more formally. For this purpose, we introduce the functions `DEF` and `REF`, where `DEF` returns a set of variables defined in a statement and `REF` returns a set of variables referenced (or used) in the statement. Note that `DEF` returns the empty set for conditional statements and a set representing the variable on the left side of an assignment statement. Using these functions we define data dependencies as follows.

Definition 2 (data dependence). Given an execution trace $\langle s_1, \dots, s_k \rangle$ for a program Π and an input state $\omega \in \Sigma$,

```
(1) cond = a > 0 & b > 0 & c > 0 & d > 0 & e > 0;
(2) if cond
(3) x = a * c;
(4) y = b * d;
(5) z = c * e;
(6) f = x + y - a;
(7) g = y + z;
```

FIGURE 4: The execution trace of `numfun` (Figure 1).

an element of the execution trace s_j is data dependent on another element s_i where $i < j$, that is, $s_i \rightarrow_D s_j$, if and only if there exists a variable x that is element of `DEF`(s_i) and `REF`(s_j) and there exists no element s_k , $i < k < j$, in the execution trace where $x \in \text{DEF}(s_k)$.

Beside data dependences, we have to deal with control dependences representing the control flow of a given execution trace. In $\mathcal{L}$, there are only if-then-else and while statements that are responsible for the control flow. Therefore, we only have to consider these two types of statements.

Definition 3 (control dependence). Given an execution trace $\langle s_1, \dots, s_k \rangle$ for a program Π and an input state $\omega \in \Sigma$, an element of the execution trace s_j is control dependent on a conditional statement s_i with $i < j$, that is, $s_i \rightarrow_C s_j$, if and only if the execution of s_i causes the execution of s_j .

In the previous definition the term *cause* has to be interpreted very rigorously. If the condition of the while statement executes to TRUE, then all statements of the outermost sub-block of the while-statement are control dependent. If the condition evaluates to FALSE, no statement is control dependent because the first statement after the while-statement is always executed regardless of the evaluation of the condition. Please note, that we do not consider infinite loops. They are not in the scope of this paper. For if-then-else statements the interpretation is similar. If the condition of an if-then-else statement evaluates to TRUE, the statements of the then-block are control dependent on the conditional statement. If it evaluates to FALSE, the statements of the else-block are control dependent on the conditional statement. Note that in case of nested while-statements or if-then-else statements, the control dependencies are not automatically assigned for the blocks of the inner while-statements or if-then-else statements.

Figure 5 shows the execution trace for our running example where the data and control dependencies have been added. Alternatively, the execution trace including the dependencies can be represented as directed acyclic graph, the corresponding execution trace graph (ETG).

In addition to data and control dependencies, we make use of potential data dependencies in relevant slicing [18]. In brief, a potential data dependency occurs whenever the evaluation of a conditional statement causes that some statements which potentially change the value of a variable

FIGURE 5: The execution trace of `numfun` (Figure 1) enhanced with data and control dependencies.

are not executed. Ignoring such potential data dependencies might lead to slices where the faulty statements are missing.

Definition 4 (potential relevant variables). Given a conditional (while or if-then-else) statement n , the potential relevant variables are a function PR that maps the conditional statement and a Boolean value to the set of all defined variables in the block of n that is not executed because the corresponding condition of n evaluates to TRUE or FALSE.

The previous definition requires all defined variables to be element of the set of potential relevant variables under a certain condition. This means that if there are other while-statements or if-then-else statements in a sub-block, the defined variables of all their sub-blocks must be considered as well. For the sake of clarity Table 1 summarizes the definition of potential relevant variables.

Based on the definition of the potential data dependence set, we define potential data dependences straightforward.

Definition 5 (potential data dependence). Given an execution trace $\langle s_1, \dots, s_k \rangle$ for a program Π and an input state $\omega \in \Sigma$, an element of the execution trace s_j is potentially data dependent on a test element s_i with $i < j$, which evaluates to TRUE (FALSE), that is, $s_i \rightarrow_p s_j$, if and only if there is a variable $x \in PR(s_i, \text{TRUE})$ ($x \in PR(s_i, \text{FALSE})$) that is referenced in s_j and not redefined between i and j .

After defining the dependence relations of a program that is executed on a given input state, we are able to formalize relevant slices, which are used later in our approach.

Definition 6 (relevant slice). A relevant slice S of a program $\Pi \in \mathcal{L}$ for a slicing criterion (ω, x, n) , where $\omega \in \Sigma$ is an input state, x is a variable, and n is a line number in the execution trace that comprises those parts of Π , which contribute to the computation of the value for x at the given line number n .

We assume that a statement contributes to the computation of a variable value if there is a dependence relation. Hence, computing slices can be done by following the dependence relations in the ETG. Algorithm 1 `RELEVANTSLICE` (ET, Π, x, n) computes the relevant slice for a given execution trace and a given variable at the execution trace position n . The program Π is required for determining the potential data dependences.

The relevant slice is likely smaller than the execution trace, where a statement might be executed more often. In our approach, we use relevant slices for restricting the search space for root cause identification.

3.2. The Debugging Problem. Using the definition of $\mathcal{L}$ together with the definition of test cases and test suites, we are able to formally state the debugging problem. Hence, first we have to define test cases and test suites. We do not discuss testing in general. Instead we refer the interested reader to the standard text books on testing, for example, [36]. In the context of our paper, a test case comprises information about the values of input variables and some information regarding the expected output. In principle, it is possible to define expected values for variables at arbitrary positions in the code. For reasons of simplicity, we do not make use of an extended definition.

Definition 7 (test case). A test case t is a tuple (I, O) , where $I \in \Sigma$ is the input and $O \in \Sigma$ is the expected output.

A given program $\Pi \in \mathcal{L}$ passes a test case $t = (I, O)$ if and only if $\llbracket \Pi \rrbracket I \supseteq O$. Otherwise, we say that the program fails. Because of the use of the $\supseteq$ operator, partial test cases are allowed, which do not specify values for all output variables. If a program passes a test case t , then t is called a passing test case. Otherwise, the test case is said to be a failing test case. Note that we do not consider inconclusive test cases explicitly. In cases where inconclusive test cases exist, we treat them like passing test cases. Since we are only considering failing test cases for fault localization, this assumption has no influence on the final result.

Definition 8 (test suite). A test suite TS for a program $\Pi \in \mathcal{L}$ is a set of test cases.

When using the definition of passing and failing test cases, we are able to partition a test suite into two disjoint sets comprising only positive (PASS), respectively, failing (FAIL) test cases, that is, $TS = \text{PASS} \cup \text{FAIL}$ and $\text{PASS} \cap \text{FAIL} = \emptyset$. Formally, we define these two subsets as follows:

$$\begin{aligned} \text{PASS} &= \{(I, O) \mid (I, O) \in TS, \llbracket \Pi \rrbracket I \supseteq O\} \\ \text{FAIL} &= TS / \text{PASS}, \end{aligned} \tag{3}$$

TABLE 1

Statement n	Cond. E	Potential relevant variables PR
while E { B }	TRUE	$PR(n, \text{TRUE}) = \emptyset$
	FALSE	$PR(n, \text{FALSE}) = \{m \mid m \text{ defined in } B\}$
if E { B_1 } else { B_2 }	TRUE	$PR(n, \text{TRUE}) = \{m \mid m \text{ defined in } B_2\}$
	FALSE	$PR(n, \text{FALSE}) = \{m \mid m \text{ defined in } B_1\}$

Require: An execution trace $ET = \langle s_1, \dots, s_m \rangle$, a variable of interest x , and a certain line number of the execution trace n .

Ensure: A relevant slice.

- (1) Compute the execution trace ETG using the dependence relations $\rightarrow_C$, $\rightarrow_D$, and $\rightarrow_P$.
- (2) Mark the node s_k in the ETG, where $x \in \text{DEF}(s_k)$ and there is no other statement s_i , $k < i \leq n$, $x \in \text{DEF}(s_i)$ in the ETG.
- (3) Mark all test nodes between s_k and s_n , which evaluate to the boolean value B and where $x \in \text{PR}(s_k, B)$.
- (4) Traverse the ETG from the marked nodes in the reverse direction of the arcs until no new nodes can be marked.
- (5) Let S be the set of all marked nodes.
- (6) Return the set $\{s_i \mid s_i \in S\}$ as result.

ALGORITHM 1: RELEVANTSLICE (ET, Π, x, n).

For a negative test case $t = (I, O) \in \text{FAIL}$, we know that there must be some variables $CV_t = \{x_1, \dots, x_k\}$, where for all $i \in \{1, \dots, k\}$ $x_i = v_i \in O$ and $x_i = w_i \in \llbracket \Pi \rrbracket I$ follows that $v_i \neq w_i$. We call such variables $x_1, \dots, x_k$ conflicting variables. The set of conflicting variables for a test case t is denoted by $CV(t)$. If the test case t is a positive test case, the set $CV(t)$ is defined to be empty. Using these definitions, we define the debugging problem.

Definition 9 (debugging problem). Given a program $\Pi \in \mathcal{L}$ and a test suite TS, the problem of identifying the root cause for a failing test case $t \in \text{TS}$ in Π is called the debugging problem.

A solution for the debugging problem is a set of statements in a program Π that are responsible for the conflicting variables $CV(t)$. The identified statements in a solution have to be changed in order to turn all failing test cases into passing test cases for the corrected program.

3.3. Model-Based Debugging. In the introduction, we mentioned that correctness assumptions are the key for fault localization. Therefore, a technique for diagnosis that is based on such assumptions would be a good starting point for debugging. Indeed, such methodology can be found in artificial intelligence. Reiter [29] introduced the theoretical foundations of model-based diagnosis (MBD) where a model that captures the correct behavior of components is used together with observations for diagnosis. The underlying idea of MBD is to formalize the behavior of each

component C in the form $\neg \text{AB}(C) \rightarrow \text{BEHAV}(C)$. The predicate AB stands for abnormal and is used to state the incorrectness of a component. Hence, when C is correct, $\neg \text{AB}(C)$ has to be true and the behavior of C has to be valid. In debugging, we make use of the same underlying idea. Instead of dealing with components, we now have statements, and the behavior of a statement is given by a formal representation of the statement's source code. We use constraints as a representation language for this purpose.

In the following we adapt Reiter's definition of diagnosis [29] for representing bug candidates in the context of debugging.

Definition 10 (diagnosis). Given a formal representation SD of a program $\Pi \in \mathcal{L}$, where the behavior of each statement s_i is represented as $\neg \text{AB}(s_i) \rightarrow \text{BEHAV}(s_i)$ and a failing test case (I, O) , a diagnosis Δ (or bug candidate) is a subset of the set of statements of Π such that $\text{SD} \cup \{I, O\} \cup \{\neg \text{AB}(s) \mid s \in \Pi \setminus \Delta\} \cup \{\text{AB}(s) \mid s \in \Delta\}$ is satisfiable.

In this definition of a diagnosis, the representation of programs (or execution traces) and failing test cases is not included. Furthermore, a formalism that allows for checking satisfiability is premised. However, the definition exactly states that we have to find a set of correctness assumptions that does not lead to a contradiction with respect to the given test case. We do not want to discuss all the consequences of this definition and refer the interested reader to [29, 37, 38]. In the following, we explain how to obtain a model for a particular execution trace and how to represent failing test cases.

The representation of programs for our model-based approach is motivated by previous work [31, 32]. In [31, 32] all possible execution paths up to a specified size are represented as set of constraints. In contrast, we now only represent the current execution path. In this case, the representation becomes smaller and the modeling itself is much easier since only testing actions and assignments are part of an execution trace. On the contrary, we lose information and we are not able to eliminate candidates that belong to testing actions. Hence, in the proposed approach, we expect improvements of debugging results compared to slicing. Even though we cannot match obtained with model-based debugging approaches like Wotawa et al. [32], our approach requires less runtime.

Modeling for model-based debugging in the context of this paper comprises two steps. In the first step, we convert an execution trace of a program for a given test case to its static single assignment form (SSA) [39]. In the second step, we use the SSA representation and map it to a set of constraints. When using a constraint representation, checking for consistency becomes a constraint satisfaction problem (CSP). A constraint satisfaction problem is a tuple (V, D, CO) where V is a set of variables defined over a set of domains D connected to each other by a set of arithmetic and Boolean relations, called constraints CO . A solution for a CSP represents a valid instantiation of the variables V with values from D such that none of the constraints from CO is violated. We refer to Dechter [40] for more information on constraints and the constraint satisfaction problem.

Now, we explain the mapping of program execution traces into their constraint representations in detail. We start with the conversion into SSA form. The SSA form is an intermediate representation of a program with the property that no two left-side variables share the same name. The SSA form can be easily obtained from an execution trace by adding an index to each variable. Every time a variable is re-defined, the value of the index gets incremented such that the SSA form property holds. Every time a variable is referenced, the current index is used. Note that we always start with the index 0. Algorithm 2 formalizes the conversion of execution traces ET into their SSA form.

The application of the SSA algorithm on the execution trace of our running example `numfun` delivers the following execution trace:

- (1) `cond_1 = a_0 > 0 & b_0 > 0 & c_0 > 0 & d_0 > 0 & e_0 > 0,`
- (2) `if cond_1 {,`
- (3) `x_1 = a_0 * c_0,`
- (4) `y_1 = b_0 * d_0,`
- (5) `z_1 = c_0 * e_0,`
- (6) `f_1 = x_1 + y_1 - a_0,`
- (7) `g_1 = y_1 + z_1,`
- (8) `}`.

In the second step, the SSA form of the execution trace is converted into constraints. Instead of using a specific

language of a constraint solver, we make use of mathematical equations. In order to distinguish equations from statements, we use `==` to represent the equivalence relation. Algorithm 3 formalizes this conversion. In the algorithm, we make use of a global function index that maps each element of ET to a unique identifier representing its corresponding statement. Such a unique identifier might be the line number where the statement starts. Note that in Algorithm 3 we represent each statement of the execution trace using the logical formula of the form $AB(\dots) \vee \text{Constraint}$, which is logically equivalent to $\neg AB(\dots) \rightarrow \text{Constraint}$. Moreover, `CONSTRAINTS(ET, (I, O), ssa)` also converts the given test case.

Applying Constraints $(ET, (I, O), ssa)$ on the SSA form of the execution trace of the `numfun` program extracts the following constraints:

$$\begin{aligned}
 & a_0 == 2, \\
 & b_0 == 2, \\
 & c_0 == 3, \\
 & d_0 == 3, \\
 & e_0 == 2, \\
 & f_1 == 12, \\
 & g_1 == 12, \\
 & AB(1) \vee (\text{cond_1} == a_0 > 0 \wedge b_0 > 0 \wedge c_0 > 0 \wedge d_0 > 0 \wedge e_0 > 0), \\
 & AB(2) \vee (\text{cond_1}), \\
 & AB(3) \vee (x_1 == a_0 * c_0), \\
 & AB(4) \vee (y_1 == b_0 * d_0), \\
 & AB(5) \vee (z_1 == c_0 * e_0), \\
 & AB(6) \vee (f_1 == x_1 + y_1 - a_0), \\
 & AB(7) \vee (g_1 == y_1 + z_1).
 \end{aligned} \tag{4}$$

4. The CONBAS Algorithm

In this section, we present our approach, CONBAS. The basic idea of CONBAS is to reduce the size of summary slices by computing minimal diagnoses. Minimal diagnoses are computed by combining the statements of the slices of the faulty variables of a single test case as follows. (1) Each diagnosis must contain at least one element of every slice. (2) If there exists a diagnosis that is a proper subset of the diagnosis, the superset diagnosis is skipped. The remaining diagnoses are further reduced with the aid of a constraint solver. For doing so, the execution trace of a failing test case is converted into constraints. The constraint solver checks for satisfiability of the converted execution trace assuming that the statement of the diagnosis is incorrect. Figure 6 gives an overview of the CONBAS approach.

Algorithm 4 explains the CONBAS approach in detail. The function `RUN( $\Pi, t$ )` executes a test case t on a program Π . It returns the resulting execution trace ET and the set of conflicting variables CV . The relevant slices are computed for all conflicting variables CV by means of the function `RELEVANTSLICE( $ET, \Pi, x, n$ )` (Algorithm 1), at which $|ET|$ is established for n . For all conditional statements c in ET the relevant slices SC_c are computed. This is done by computing the relevant slices for all variables contained in the conditions. The function `POSITIONINEXECUTIONTRACE`

Require: An execution trace $ET = \langle s_1, \dots, s_m \rangle$.
Ensure: The execution trace in SSA form and a function that maps each variable to its maximum index value used in the SSA form.

- (1) Let $index$ be a function mapping variables to integers. The initial integer value for each variable is 0.
- (2) Let ET' be the empty sequence.
- (3) **for** $j = 1$ to m **do**
- (4) **if** s_j is an assignment statement of the form $x = E$ **then**
- (5) Let E' be E where all variables $y \in E$ are replaced with $y_index(y)$.
- (6) Let $index(x)$ be $index(x) + 1$.
- (7) Add $x_index(x) = E'$ to the end of the sequence ET' .
- (8) **else**
- (9) Let s' be the statement s_j where all variables $y \in E$ are replaced with $y_index(y)$.
- (10) Add s' to the end of the sequence ET' .
- (11) **end if**
- (12) **end for**
- (13) Return $(ET', index)$.

ALGORITHM 2: SSA(ET).

Require: An execution trace $ET = \langle s_1, \dots, s_m \rangle$ in SSA form, a test case (I, O) , and a function ssa returning the final index value for each variable.
Ensure: The constraint representation of ET and the test case.

- (1) Let CO be the empty set.
- (2) **for all** $(x, v) \in I$ **do**
- (3) Add " $x_0 == v$ " to CO .
- (4) **end for**
- (5) **for all** $(y, w) \in O$ **do**
- (6) Add " $y_ssa(y) == w$ " to CO .
- (7) **end for**
- (8) **for** $j = 1$ to m **do**
- (9) **if** s_j is an assignment statement of the form $x = E$ **then**
- (10) Add " $(AB(index(s_j)) \vee (x == E))$ " to CO .
- (11) **else**
- (12) Let c be the condition of s_j where $\&$ is replaced with $\wedge$ and $_$ with $\vee$.
- (13) Add " $(AB(index(s_j)) \vee (c))$ " to CO .
- (14) **end if**
- (15) **end for**
- (16) Return CO .

ALGORITHM 3: CONSTRAINTS (ET, (I, O), ssa).

(c, ET) is used to obtain the line number of c in the execution trace.

The function $MINHITTINGSETS$ (SET OF ALL S_x) returns the set of minimal hitting sets HS of the set of slices S_x . A set d is a hitting set for a set of sets CO if it contains at least one element of every set of CO :

$$d \text{ is hitting set} \iff \forall s \in CO : s \cap d \neq \{\}. \quad (5)$$

A hitting set d w.r.t. CO is minimal if there exists no subset of d , which is a valid hitting set w.r.t. CO . Minimal hitting sets can be computed by means of the corrected Reiter algorithm [29, 41].

Bugs causing a wrong evaluation of a condition lead to the wrong (non) execution of statements. In order to handle

such bugs, the function $EXTENDCONTROLSTATEMENTS(ET, \Pi)$ adds a small overhead to each control statement c in the execution trace ET . For each variable v that could be redefined in any branch of c , the statement $v=v$ is added to the execution trace ET . These additional statements are inserted after all statements that are control dependent on c . The inserted statements will be referenced by the line number of c when calling the function $index$ in Algorithm 3. The returned execution trace is assigned to ET_C . This extension can be compared with potential data dependencies in relevant slicing.

$SSA(ET_C)$ (Algorithm 2) transforms the execution trace ET_C into its single static assignment form and also delivers the largest index value for each variable used in

FIGURE 6: The CONBAS approach.

the SSA form. CONSTRAINTS (ET_{ssa}, t, ssa) (Algorithm 3) converts each statement into its equivalent constraint representation. The statements added in the function `EXTENDCONTROLSTATEMENTS` (ET, Π) ($v=v$, in SSA form: $v_{i+1} = v_i$) are concatenated with the predicate $AB(j)$: $AB(j) \vee v_{i+1} = v_i$. The reason for this is that we cannot reason over the variables if the execution path alters. Please note that the required test oracle information can be fully automated extracted from an existing test case.

The result set S represents the set of possible faulty statements. At first, the result set is the empty set. For all minimal hitting sets d in the set of minimal hitting sets HS , we check if the constraint solver is able to find a solution. For this purpose, we set all $AB(i)$ to false except those where the corresponding statements are contained in d . For all conditional statements c where SC_c and d have at least one common element, we set $AB(c)$ to true. The function `CONSTRAINTSOLVER`($CS \cup AB$) calls a constraint solver and returns true if the constraint solver is able to find a solution. If a solution is found, we add all elements of d to the result set S .

We illustrate the application of the algorithm by means of our running example. The function `RUN`(Π, t) computes the execution trace illustrated in Figure 4 and $\{f\}$ as the set of conflicting variables. The set of relevant slices of the conflicting variables is $\{\{1, 2, 3, 4, 6\}\}$. The union of the slices for the variables in the condition in Line 2 is

$\{1\}$. The function `MINHITTINGSETS`($\{\{1, 2, 3, 4, 6\}\}$) delivers $\{\{1\}, \{2\}, \{3\}, \{4\}, \{6\}\}$ as minimal hitting sets. The resulting constraints are

$$\begin{aligned}
 & a_0 == 2, \\
 & b_0 == 2, \\
 & c_0 == 3, \\
 & d_0 == 3, \\
 & e_0 == 2, \\
 & f_2 == 12, \\
 & g_2 == 12, \\
 & AB(1) \vee (cond_1 == a_0 > 0 \wedge b_0 > 0 \wedge c_0 > 0 \wedge d_0 > 0 \wedge e_0 > 0), \\
 & AB(2) \vee (cond_1), \\
 & AB(3) \vee (x_1 == a_0 * c_0), \\
 & AB(4) \vee (y_1 == b_0 * d_0), \\
 & AB(5) \vee (z_1 == c_0 * e_0), \\
 & AB(6) \vee (f_1 == x_1 + y_1 - a_0), \\
 & AB(7) \vee (g_1 == y_1 + z_1), \\
 & AB(1) \vee (x_2 == x_1), \\
 & AB(1) \vee (y_1 == x_1), \\
 & AB(1) \vee (z_2 == z_1), \\
 & AB(1) \vee (f_2 == f_1), \\
 & AB(1) \vee (g_2 == g_1).
 \end{aligned} \tag{6}$$

There are five different configurations for the values of AB :

$$\begin{aligned}
 & AB(1) == \text{true} \wedge AB(2) == \text{true}, \\
 & AB(2) == \text{true}, \\
 & AB(3) == \text{true}, \\
 & AB(4) == \text{true}, \\
 & AB(6) == \text{true}.
 \end{aligned} \tag{7}$$

We only indicate the AB values that are set to true. All other AB variables are set to false. Note that setting $AB(1) == \text{true}$ implies $AB(2) == \text{true}$ since we cannot reason over the correctness of the condition if the computed values used in the condition are wrong. The constraint solver is able to find solutions for all configurations, except for $AB(4)$. Since our approach does only reason on the execution trace and not on all possible paths, $AB(1) == \text{true}$ and $AB(2) == \text{true}$ are satisfiable even though taking the alternative path does not compute the correct value for f .

Algorithm CONBAS terminates if the program Π terminates when executing T . The computational complexity of CONBAS is determined by the computation of the relevant slices, the hitting sets, and the constraint solver. Computing relevant slices only adds a small overhead compared to the execution of the program. Hitting set computation and constraint solving are exponential in the worst case (finite case). In order to reduce the computation time, the computation of hitting sets can be simplified. We only compute hitting sets of the size 1 or 2; that is, we only compute single and double fault diagnoses. Faults with more involved faulty statements are unlikely in practice. Only in cases where the single and double fault diagnoses cannot

```

Require: Program  $\Pi$  and failing test case  $t$ 
Ensure: Dynamic slice  $S$ 
(1)  $[ET, CV] = \text{RUN}(\Pi, t)$ 
(2) for all  $x \in CV$  do
(3)    $S_x = \text{RELEVANTSLICE}(ET, \Pi, x, |ET|)$ 
(4) end for
(5) for all  $c \in ET$  do
(6)    $n = \text{POSITIONINEXECUTIONTRACE}(c, ET)$ 
(7)   for all variables  $x$  in  $c$  do
(8)      $SC_x = \text{RELEVANTSLICE}(ET, \Pi, x, n)$ 
(9)   end for
(10)   $SC_c = \cup SC_x$ 
(11) end for
(12)  $HS = \text{MINHITTINGSETS}(\text{SETOFALL } S_x)$ 
(13)  $ET_C = \text{EXTENDCONTROLSTATEMENTS}(ET, \Pi)$ 
(14)  $[ET_{SSA}, ssa] = \text{SSA}(ET_C)$ 
(15)  $CS = \text{CONSTRAINTS}(ET_{SSA}, t, ssa)$ 
(16)  $S = \{\}$ 
(17) for all  $d \in HS$  do
(18)   $\forall i \in d : AB(i) = \text{true}$ 
(19)   $\forall i \notin d : AB(i) = \text{false}$ 
(20)   $\forall c \text{ where } SC_c \cap d \neq \{\} : AB(c) = \text{true}$ 
(21)  if  $\text{CONSTRAINTSOLVER}(CS \cup AB)$  has solution then
(22)     $S = S \cup d$ 
(23)  end if
(24) end for
(25) return  $S$ 

```

ALGORITHM 4: CONBAS (Π, t).

explain an observed misbehavior, the size of the hitting sets is increased.

5. Empirical Results

This empirical evaluation consists of two main parts. First, we show that CONBAS is able to reduce the size of slices without losing the fault localization capabilities of slicing. We show this for single faults as well as for multiple faults. Second, we investigate the influence of the number of output variables on the reduction result.

We conducted this empirical evaluation using a proof of concept implementation of CONBAS. This implementation accepts programs written in the language $\mathcal{L}$ (see Figure 2). In order to test existing example programs, we have extended this implementation to accept simple Java programs, that is, Java programs with integer and Boolean data types only and without method calls and object orientation. The implementation itself is written in Java and comprises a relevant slicer and an interface to the Minion constraint solver [42]. The evaluation was performed on an Intel Core2 Duo processor (2.67 GHz) with 4 GB RAM and Windows XP as operating system. Because of the used constraint solver, only programs comprising Boolean and integer data types (including arrays of integers) could be handled directly. Note that the restriction to Boolean and integer domains is not a limitation of CONBAS.

For this empirical evaluation, we have computed all minimal hitting sets. We did not restrict the size of the hitting sets. Since we only deal with single and double faults, hitting sets of the sizes 1 and 2 would be sufficient. This reduction would improve our results concerning the number of final diagnoses and the computation time.

For the first part of the empirical evaluation, we use the 10 example programs listed in Table 2. Most of the programs implement numerical functions using conditional statements. The programs *IfExample*, *SumPower*, *TrafficLight*, and *WhileLoops* are borrowed from the JADE project (<http://www.dbai.tuwien.ac.at/proj/Jade/>). The program *Taste* is borrowed from the Unravel project (<http://hissa.nist.gov/unravel/>). Table 2 depicts the obtained results. In the table, we present the following data:

- (i) the name of the program (Program),
- (ii) the fault version (V),
- (iii) the number of lines of code (LOC),
- (iv) the size of the execution trace (Exec. trace),
- (v) the number of constraints (Con.),
- (vi) the number of intern variables in the CSP (Int. var.),
- (vii) the number of minimal diagnoses that are computed by the hitting set algorithm (Total diag.),
- (viii) the number of minimal diagnoses that are satisfiable by the constraint solver (Valid diag.),

TABLE 2: Results for single faults.

Program	V	LOC	Exec. trace	Con.	Int. var.	Total diag.	Valid diag.	Sum. Slice	Red. Slice	Time
AKSWT	1	12	14	29	8	6	5	6	5	656
	2	12	17	39	11	3	3	6	3	203
	3	12	14	29	8	6	5	6	5	500
	4	12	4	3	2	2	1	2	1	141
	5	12	20	45	12	3	3	6	3	219
ProdSum	1	14	14	29	10	6	6	6	6	469
	2	14	14	31	11	7	6	8	7	469
	3	14	11	22	9	7	5	8	6	453
	4	14	14	31	11	3	3	8	3	203
	5	14	11	22	9	3	2	8	2	188
PowerFunction	1	15	15	25	9	9	8	9	9	625
	2	15	9	9	5	6	5	6	6	359
	3	15	12	16	7	9	7	9	8	578
	4	15	15	23	9	4	4	9	5	250
	5	15	15	23	9	4	4	9	5	266
Multiplication	1	16	13	26	12	10	6	10	7	672
	2	16	14	25	9	8	8	8	8	500
	3	16	10	17	9	4	1	8	2	422
	4	16	11	19	7	6	3	8	3	375
	5	16	16	33	13	4	4	10	4	344
Divide	1	15	24	55	17	10	10	10	10	735
	2	15	26	62	22	12	12	12	12	906
	3	15	13	26	8	8	8	8	8	500
	4	15	10	18	6	4	3	8	3	250
	5	15	7	10	4	6	5	6	5	359
IfExamples	1	7	4	4	2	3	3	3	3	172
	2	8	3	3	2	2	1	2	1	110
	3	14	4	2	1	2	1	3	2	125
SumPowers	1	22	43	89	25	12	7	13	8	1141
	2	22	7	8	4	2	1	6	3	125
	3	22	7	8	4	2	1	6	3	109
	4	22	28	51	15	11	11	12	12	907
	5	22	28	58	17	12	12	13	13	890
TrafficLight	1	17	61	80	12	11	1	11	4	1078
	2	17	44	63	12	6	1	6	2	500
WhileLoops	1	14	52	99	14	7	7	7	7	719
	2	14	12	19	6	6	3	6	3	406
	3	14	52	99	14	7	7	7	7	719
Taste	1	15	10	21	11	4	4	7	5	266
	2	15	10	21	11	4	4	7	5	250
	3	15	28	69	23	6	6	9	7	469
	4	15	25	61	21	6	6	9	7	453
	5	15	19	45	17	6	6	9	7	438
Average		15.1	18.1	34.1	10.4	6.0	4.9	7.7	5.5	454

(ix) the number of statements contained in the union of the relevant slices of all faulty variables (Sum. Slice),

(x) the number of statements in the reduced slice (Red. Slice),

(xi) the time (in milliseconds) required for reducing the slice (Time).

On average, the size of the slice is reduced by more than 28% compared to the size of the corresponding summary slice. Figure 7 illustrates the relation of the program size, the

FIGURE 7: Comparison of the number of statements in total (LOC), in the summary slice and in the reduced slice. The used program variants deal with single faults.

FIGURE 8: Comparison of the number of minimal diagnoses and the number of valid minimal diagnoses. The used program variants deal with single faults.

summary slice size, and the reduced slice size for the data presented in Table 2.

Figure 8 illustrates the proportion of the number of minimal diagnoses (total diag.) and the number of valid minimal diagnoses (valid diag.) for the data presented in Table 2. The constraint solver reduces about 20% of the number of diagnoses.

In order to estimate the computation time for larger programs, we have investigated if there exists a correlation between the time (in milliseconds) required for CONBAS and (1) the LOC, (2) the size of the execution trace (exec. trace), (3) the number of constraints (con.), or (4) the number of diagnoses to be tested for satisfiability (total diag.). We found out that the strongest correlation is between the execution

time and (4). Figure 9 illustrates this correlation. The blue data points represent the data from Table 2. The red line represents the least squares fit as an approximation of the data.

One advantage of CONBAS is that it is able to reduce slices of programs that contain two or more faults. In order to demonstrate this, we have performed a small evaluation on double faults. For this, we combined some faults used in the single fault evaluation. The faults were not combined according a particular schema (i.e., masking of faults or avoiding masking of faults). We only made the following restriction: faulty program versions were not combined, where the faults were in the same program line. The reason for this is that two faults in the same line can be seen as one single fault.

TABLE 3: Results for double faults.

Program	V	Faults in Red. Slice	LOC	Sum. Slice	Red. Slice
Taste	6d	2	15	9	6
	7d	2	15	9	6
	8d	2	15	7	4
	9d	2	15	9	5
	10d	2	15	7	4
	11d	2	15	7	4
	12d	2	15	9	6
	13d	2	15	9	6
	14d	2	15	9	6
	15d	2	15	9	2
SumPowers	6d	1	22	6	1
	7d	1	22	12	7
	8d	2	22	13	7
	9d	1	22	6	1
	10d	1	22	6	1
	11d	1	22	12	7
WhileLoops	4d	1	14	6	3
	5d	1	14	7	7
	6d	1	14	6	3
ProdSum	6d	1	14	8	3
	7d	1	14	8	3
	8d	1	14	8	3
	9d	1	14	8	2
	10d	2	14	8	2
	11d	1	14	8	2
Average			16.3	8.2	4.0

FIGURE 9: The correlation of the number of diagnoses to be tested (Total diag.) and the computation time (Time, in milliseconds).

Table 3 shows the results obtained when executing CONBAS on these new program versions. The table contains the following data:

- (i) the name of the program (Program),
- (ii) the fault version (V),
- (iii) the number of faults contained in the reduced slice (Faults in Red. Slice),
- (iv) the number of lines of code (LOC),

- (v) the number of statements contained in the union of the relevant slices of all faulty variables (Sum. Slice),
- (vi) the number of statements in the reduced slice (Red. Slice).

It can be seen that sometimes only one of the two faults is contained in the reduced slice. The reason for this is that one fault can be masked by the other fault. CONBAS guarantees that at least one of the faults is contained in the reduced slice. This is not a limitation since a programmer can fix the first bug and then apply CONBAS again on the corrected program. Figure 10 shows the relation of the program size, the summary slice size, and the reduced slice size for the investigated double faults. On average, the summary slice can be reduced by 50%.

In the second part of the empirical evaluation, we investigate if more than one faulty output variable allows for a higher reduction of the summary slice. For this purpose, we use the circuits C17 and C432 of the ISCAS 85 [43] benchmark. The ISCAS 85 circuits describe combinational networks. We have chosen ISCAS 85, because the different circuits of ISCAS 85 have many input and output variables. The circuit C17 has 5 input variables and 2 output variables. The circuit C432 has 36 input variables and 7 output variables. For the evaluation, we have used test cases with different input and output combinations. We used 3 as

FIGURE 10: Comparison of the number of statements in total (LOC), in the summary slice and in the reduced slice. The used program variants deal with double faults.

TABLE 4: Average results for the Iscas 85 benchmark.

Circuit	LOC	Exec. trace	Con.	Int. var.	Total diag.	Valid diag.	Sum. Slice	Red. Slice
C17	31	14	26	25	12	5.9	9.3	5.1
C432	832	398	646	662	874.5	373.7	328.6	108.3
Average	339.1	161.7	264.5	270	343.8	147.3	132.1	44.8

the upper bound for the number of faulty output variables. In total, we created more than 150 program variants. Table 4 presents the obtained average results for the two circuits of the Iscas 85 benchmark. The column headings are similar to those used in Table 2. An explanation of the column headings can be found as previously mentioned. CONBAS is able to reduce the size of the summary slice by 66%.

Now, we want to answer the question if we could yield a higher reduction of the summary slice, when there are more faulty output variables. In order to answer this question, we make use of the REDUCTION metric, which is defined as

$$\text{REDUCTION} = \left(1 - \frac{\text{Reduced Slice}}{\text{Summary Slice}}\right) \times 100\%. \quad (8)$$

We group the tested program variants by the number of faulty output variables and compute the REDUCTION metric for the program variants. Figure 11 shows the box plots for the different numbers of output variables. It can be seen that two and three faulty output variables yield a better reduction of the slice size than only one output variable. The reason for this is that it is more difficult for the constraint solver to find configurations which meet all of the specified output variables.

6. Discussion and Future Work

Although, CONBAS substantially reduces the number of diagnosis candidates with a reduction of about 28% in

FIGURE 11: Comparison of the REDUCTION quality for a different number of faulty output variables for the Iscas 85 benchmark. The central lines of the boxes indicate the median of the reduction. The diamonds indicate the arithmetic mean of the reduction. The whiskers extend to the most extreme data points not considered as outliers. Outliers are plotted individually as crosses.

the single fault and 50% in the double fault case, there is still room for improvements. In particular, the current implementation is not optimized both in terms of handling different kinds of program language constructs and time required for performing the analysis. It would benefit from a relevant slicer for Java programs without restrictions on

the language's syntax. Currently, only dynamic slicers are available, which might cause root causes to be ignored during the debugging process. Moreover, the combination of slices and constraint solving that is currently used might be improved. Especially, in cases where there are many possible faults, the calls to the external constraint solver slow down the computation, which could be improved by a closely integrated constraint solver.

Apart from these technical issues, there are some open research questions. We start with discussing possible improvements of CONBAS that make use of the same underlying ideas but change the way of computing the final results. Instead of computing the hitting sets of the slices, the constraint solver can be directly used to compute all solvable diagnoses of a particular size. Such an approach would restrict the number of constraint solver calls and also the time required for computing the hitting sets for the slices. Such an approach would be very similar to the approaches of Nica and colleagues [31, 32], but it works on execution traces instead of the whole program representation. The expectation is that such an approach would be more efficient. However, there have been no publications on this topic so far.

Another research challenge is to improve CONBAS by using information about the evaluation of conditions. We have to analyze if taking the alternative execution path of a condition (e.g., the else path if the condition evaluates to true) could satisfy the test case. If the change leads to a consistent program behavior, a root cause is identified. Otherwise, the condition can be assumed to be correct and removed from the list of fault candidates. The underlying challenge is to make such tests only in cases where infinite loops or infeasible behaviors can be avoided. For example, executing a conditional or a recursive function not as intended might cause a non-terminating behavior. Moreover, it is also important that the computational requirements are not significantly increased.

The empirical evaluation of CONBAS, especially in comparison with other approaches, has to be improved. The used programs are rather small. Larger programs that belong to different application domains have to be used for evaluation. The currently used programs implement a variety of functions from state machines to numeric algorithms. Therefore, we believe that the obtained comparison with a pure slicing approach would not change even when using larger programs assuming that the underlying constraint problems can be solved within a reasonable amount of time. However, an empirical study that compares different approaches such as spectrum-based debugging with CONBAS would be highly required in order to structure the general research field of automated debugging.

The integration of debugging tools into integrated development environments (IDEs) like Eclipse is another hot topic. Technically, the integration seems to be easy. However, the challenge lies in effectively integrating these tools in an interactive environment such that the time needed for the overall debugging process is reduced. For this purpose, research on human-computer interaction in the context of debugging and program development has to

be done. Moreover, user studies with the aim of proving that automated debugging tools really support humans are required. Such studies should go beyond the usual student-based studies that are carried out as part of the course program. Instead, the studies should be carried out using real programmers in their industrial environment. Unfortunately, there are only few user studies in the case of automated debugging available, where [44] is the most recent.

Furthermore, the relationship between testing and debugging has not been sufficiently explored. There is work on this topic that deals with answering the question about the influence on the used test cases for debugging and how to construct test cases to further support debugging. An in-depth analysis of this topic and a well established methodology are still not available. Work in the direction of combining testing and debugging includes [45] and [31]. The latter discusses an approach for actively constructing test cases that allow for distinguishing diagnosis candidates. However, a method for test case construction that optimizes the whole debugging process is not available to the best of our knowledge. Moreover, the impact of such a method on other metrics such as mutation score or coverage criteria is not known and worth being researched.

7. Conclusion

Dynamic program slices are a valuable aid for programmers because they provide an overview of possibly faulty statements when debugging. They are used in many automated debugging techniques as a preprocessing step. However, they are often still too large to be a valuable help.

In this paper, we have introduced the theoretical foundations for an approach which reduces the size of dynamic slices by means of constraint solving. We have formalized the approach for the reduction of slices, named constraint based slicing (CONBAS). In an empirical evaluation, we have shown that the size of dynamic slices can be reduced by 28% on average for single faults and by 50% for double faults with the aid of constraint solving. Furthermore, our approach can be used even if there exist multiple faults. We have applied CONBAS on circuits of the IsCAS 85 benchmark. These circuits contain many data dependencies but lack control dependencies. For these types of programs, CONBAS yields a reduction of 66% on average compared to the union of all slices.

The objective behind CONBAS is to improve relevant slicing for debugging. Even though other approaches outperform CONBAS in certain cases, we point out two application areas where CONBAS should be the preferred method to use. First, in case of software maintenance where the root cause for one failing test case has to be identified. In this case, mostly limited knowledge about the program is available. Moreover, the programs themselves are usually large, which makes debugging a very hard task. In such a case, low-cost approaches that require a set of test cases might not be applicable and the application of heavy-weighted approaches might be infeasible because of computational requirements.

Second, in case of programs with a low number of control statements that need a more detailed analysis of data dependences and relationships between variables. In such a case, CONBAS provides the right means for analysis because of handling data dependences and constraints between program variables, which originate from the program statements.

Even though CONBAS cannot solve all debugging problems, we are convinced that CONBAS is a valuable technique for improving the debugging process. Moreover, a combination with other debugging techniques may even increase its fault localization capabilities.

Acknowledgments

The research herein is partially conducted within the competence network Softnet Austria II (<http://www.soft-net.at/>, COMET K-Projekt) and funded by the Austrian Federal Ministry of Economy, Family and Youth (bmwf), the province of Styria, the Steirische Wirtschaftsförderungsgesellschaft mbH. (SFG), and the city of Vienna in terms of the center for innovation and technology (ZIT).

References

- [1] B. Hofer and F. Wotawa, "Reducing the size of dynamic slicing with constraint solving," in *Proceedings of the 12th International Conference on Quality Software*, 2012.
- [2] M. J. Harrold, G. Rothermel, R. Wu, and L. Yi, "An empirical investigation of program spectra," in *Proceedings of the Workshop on Program Analysis for Software Tools and Engineering*, pp. 83–90, ACM, New York, NY, USA, 1998.
- [3] R. Abreu, P. Zoetewij, R. Golsteijn, and A. J. C. van Gemund, "A practical evaluation of spectrum-based fault localization," *Journal of Systems and Software*, vol. 82, no. 11, pp. 1780–1792, 2009.
- [4] P. Zoetewij, R. Abreu, R. Golsteijn, and A. J. C. Van Gemund, "Diagnosis of embedded software using program spectra," in *Proceedings of the 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS '07)*, pp. 213–220, March 2007.
- [5] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE '05)*, pp. 273–282, ACM Press, November 2005.
- [6] R. Abreu, P. Zoetewij, and A. J. C. Van Gemund, "An evaluation of similarity coefficients for software fault localization," in *Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing (PRDC '06)*, pp. 39–46, Washington, DC, USA, December 2006.
- [7] R. Abreu and A. J. C. Van Gemund, "Diagnosing multiple intermittent failures using maximum likelihood estimation," *Artificial Intelligence*, vol. 174, no. 18, pp. 1481–1497, 2010.
- [8] R. Abreu, W. Mayer, M. Stumptner, and A. J. C. Van Gemund, "Refining spectrum-based fault localization rankings," in *Proceedings of the 24th Annual ACM Symposium on Applied Computing (SAC '09)*, pp. 409–414, Honolulu, Hawaii, USA, March 2009.
- [9] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 183–200, 2002.
- [10] H. Cleve and A. Zeller, "Locating causes of program failures," in *Proceedings of the 27th International Conference on Software Engineering (ICSE '05)*, pp. 342–351, ACM Press, New York, NY, USA, May 2005.
- [11] A. Zeller, "Isolating cause-effect chains from computer programs," in *Proceedings of the 10th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 1–10, November 2002.
- [12] M. Weiser, "Programmers use slices when debugging," *Communications of the ACM*, vol. 25, no. 7, pp. 446–452, 1982.
- [13] W. Mayer and M. Stumptner, "Model-based debugging—state of the art and future challenges," *Electronic Notes in Theoretical Computer Science*, vol. 174, no. 4, pp. 61–82, 2007.
- [14] A. Arcuri, "On the automation of fixing software bugs," in *Proceedings of the 30th International Conference on Software Engineering (ICSE '08)*, pp. 1003–1006, New York, NY, USA, May 2008.
- [15] W. Weimer, S. Forrest, C. Le Goues, and T. Nguyen, "Automatic program repair with evolutionary computation," *Communications of the ACM*, vol. 53, no. 5, pp. 109–116, 2010.
- [16] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*, pp. 364–374, May 2009.
- [17] B. Korel and J. Laski, "Dynamic program slicing," *Information Processing Letters*, vol. 29, no. 3, pp. 155–163, 1988.
- [18] X. Zhang, H. He, N. Gupta, and R. Gupta, "Experimental evaluation of using dynamic slices for fault location," in *Proceedings of the 6th International Symposium on Automated and Analysis-Driven Debugging (AADEBUG '05)*, pp. 33–42, September 2005.
- [19] X. Zhang, S. Tallam, N. Gupta, and R. Gupta, "Towards locating execution omission errors," *ACM SIGPLAN Notices*, vol. 42, no. 6, pp. 415–424, 2007.
- [20] M. Sridharan, S. J. Fink, and R. Bodik, "Thin slicing," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*, pp. 112–122, ACM, San Diego, Calif, USA, June 2007.
- [21] N. Gupta, H. He, X. Zhang, and R. Gupta, "Locating faulty code using failure-inducing chops," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE '05)*, pp. 263–272, New York, NY, USA, November 2005.
- [22] X. Zhang, N. Gupta, and R. Gupta, "Pruning dynamic slices with confidence," *ACM SIGPLAN Notices*, vol. 41, no. 6, pp. 169–180, 2006.
- [23] X. Zhang, N. Gupta, and R. Gupta, "A study of effectiveness of dynamic slicing in locating real faults," *Empirical Software Engineering*, vol. 12, no. 2, pp. 143–160, 2007.
- [24] X. Zhang, R. Gupta, and Y. Zhang, "Efficient forward computation of dynamic slices using reduced ordered binary decision diagrams," in *Proceedings of the 26th International Conference on Software Engineering (ICSE '04)*, pp. 502–511, Los Alamitos, Calif, USA, May 2004.
- [25] X. Zhang, S. Tallam, and R. Gupta, "Dynamic slicing long running programs through execution fast forwarding," in *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT '06)*, pp. 81–91, New York, NY, USA, November 2006.
- [26] D. Jeffrey, N. Gupta, and R. Gupta, "Fault localization using value replacement," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA '08)*, pp. 167–177, ACM, New York, NY, USA, July 2008.

- [27] F. Tip, "A survey of program slicing techniques," *Journal of Programming Languages*, vol. 3, no. 3, pp. 121–189, 1995.
- [28] B. Korel and J. Rilling, "Dynamic program slicing methods," *Information and Software Technology*, vol. 40, no. 11-12, pp. 647–659, 1998.
- [29] R. Reiter, "A theory of diagnosis from first principles," *Artificial Intelligence*, vol. 32, no. 1, pp. 57–95, 1987.
- [30] F. Wotawa, "On the relationship between model-based debugging and program slicing," *Artificial Intelligence*, vol. 135, no. 1-2, pp. 125–143, 2002.
- [31] M. Nica, S. Nica, and F. Wotawa, "On the use of mutations and testing for debugging," *Software—Practice and Experience*. In press.
- [32] F. Wotawa, M. Nica, and I. Moraru, "Automated debugging based on a constraint model of the program and a test case," *Journal of Logic and Algebraic Programming*, vol. 81, no. 4, pp. 390–407, 2012.
- [33] W. Mayer, *Static and hybrid analysis in model-based debugging [Ph.D. thesis]*, School of Computer and Information Science, University of South Australia, 2007.
- [34] W. Mayer, R. Abreu, M. Stumptner, and J. C. A. van Gemund, "Prioritising model-based debugging diagnostic reports (DX)," in *Proceedings of the International Workshop on Principles of Diagnosis*, pp. 127–134, 2008.
- [35] F. Wotawa and M. Nica, "On the compilation of programs into their equivalent constraint representation," *Informatica*, vol. 32, no. 4, pp. 359–371, 2008.
- [36] B. Beizer, *Software Testing Techniques*, Van Nostrand Reinhold, New York, NY, USA, 2nd edition, 1990.
- [37] L. Console, D. T. Dupre, and P. Torasso, "On the relationship between abduction and deduction," *Journal of Logic and Computation*, vol. 1, no. 5, pp. 661–690, 1991.
- [38] J. de Kleer, A. K. Mackworth, and R. Reiter, "Characterizing diagnoses and systems," *Artificial Intelligence*, vol. 56, no. 2-3, pp. 197–222, 1992.
- [39] M. M. Brandis and H. Mossenbock, "Single-pass generation of static single-assignment form for structured languages," *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 6, pp. 1684–1698, 1994.
- [40] R. Dechter, *Constraint Processing*, Elsevier Morgan Kaufmann, 2003.
- [41] R. Greiner, B. A. Smith, and R. W. Wilkerson, "A correction to the algorithm in reiter's theory of diagnosis," *Artificial Intelligence*, vol. 41, no. 1, pp. 79–88, 1989.
- [42] I. P. Gent, C. Jefferson, and I. Miguel, "Minion: a fast, scalable, constraint solver," in *Proceedings of the 17th European Conference on Artificial Intelligence (ECAI '06)*, pp. 98–102, Riva del Garda, Italy, August 2006.
- [43] F. Brglez and H. Fujiwara, "A neutral netlist of 10 combinational benchmark circuits and a target translator in fortran," in *Proceedings of the IEEE International Symposium on Circuits and Systems*, pp. 663–698, June 1985.
- [44] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?" in *Proceedings of the 20th International Symposium on Software Testing and Analysis (ISSTA '11)*, pp. 199–209, ACM, New York, NY, USA, 2011.
- [45] a. González-Sánchez, R. Abreu, H.-G. Hans-Gerhard Gross, and A. van Gemund, "Prioritizing tests for fault localization through ambiguity group reduction," in *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE '11)*, pp. 83–92, IEEE Computer Society, Washington, DC, USA.