Localizing Program Errors via Slicing and Reasoning

Fei Pu, Yan Zhang
Intelligent Systems Laboratory
School of Computing and Mathematics
University of Western Sydney, Australia
{feipu,yan}@scm.uws.edu.au

Abstract

Model-based program debugging exploits discrepancies between the program behavior anticipated by a programmer and the program’s actual behavior when executed on a set of inputs. From symptoms exhibited by a failing trace, potential culprits in the program can be localized. However, since the cause of the error is nested deeper into the code than the error itself, localizing errors and correcting the errors are most time consuming hard work. The error trace produced by a model checker may contain more information than it appears. Thus, counterexamples can be enough and are indicative for the cause of violation of the property. We present an assumption-based approach to localize the cause of a property violation using reasoning with constraints. In order to reduce the time consuming for error localizing, we first use dynamic program slicing to localize several statements to account for the violation of property. Assumption among these statements is then made to point out which statement(s) is (are) faulty. Some constraints will be introduced from the properties which are model checked for the program. A calculus of reasoning with these constraints is processed under the assumption along a counterexample. If the result may be consistent, the assumption is true (we can localize errors in those statements which the assumption suppose them to be faulty), otherwise, the assumption is wrong and another assumption should be made. Some examples support the applicability and effectiveness of our approach.

1 Introduction

It is a well-known fact that most non-trivial programs contain errors that prohibit the program performing its intended function in certain cases. Properties-based verification such as model checking [5] exploits formal properties of certain aspects of a program’s behavior to detect errors. Program model checking aims at deciding whether a program conforms to a given formal property under all possible executions of a program and at possibly explaining why it may not. One of the major advantages of model checking is the production of a counterexample at the source level when the specification fails to be verified. The counterexample can help in understanding the symptom related to the error and thus is well suited for debugging. Since the cause of the error is nested deeper into the code than the error itself, a significant amount of effort is often required for the programmer to explain it. The counterexample may contain more information than it appears. We can see more than just the obvious runtime information when examining an error trace. So we can use a counterexample trace to reason about the cause of the error from the symptoms exhibited by the error trace.

Model based diagnosis [2] is a theory for diagnosing physical systems. Suppose one is given a description of a physical system, together with an observation of the system’s behavior which conflicts with the way the system is meant to behave. The diagnostic program is to determine those components of the system which, when assumed to be functional abnormally, will explain the discrepancy between the observed and correct system behavior. Reiter et al. [2,3] have developed a general theory based on first principles. Their algorithm computes all diagnoses which explain the differences between the predicted and observed behavior of a given system via minimal hitting set for all conflict sets of the system. We extend this idea to program debugging instead of physical system diagnosis. The observations in model based diagnosis are the inputs and outputs of the physical system. The system description denoted as SD is the set of sentences represented as first-order sentences. The assumption on which component(s) is (are) faulty is derived when the system is being diagnosed. A unary predicate Ab(c) is used in the assumption which means that component c is abnormal.

Intuitively, a diagnosis is a conjecture that certain of the components are faulty (abnormal) and the rest normal. Formally, a diagnosis for (SD, COMPONENTS, OBS) is a mini-
mal set $\Delta \subseteq \text{COMPONENTS}$ such that $SD \cup OBS \cup \{AB(c)|c \in \Delta\} \cup \{\neg AB(c)|c \in \text{COMPONENTS} - \Delta\}$ is consistent.

Overall, a diagnosis is determined by a smallest set of components with the property: The assumption that each of these components is faulty (abnormal), together with the assumption that all other components are behaving correctly (normal), is consistent with the system description and the observation.

We apply this idea to localize errors of programs. However some modifications must be made to meet the features of programs. The program description is the set of propositional logic rules which represent the behavior of statements. The counterexamples returned by a model checker are observations of the program. The constraints which derived from the specifications of the program represent the intended program’s behavior. We therefore make assumptions on the program statements. The assumption states that which statements are faulty and the rest are correct. It is clear that the consequence of our assumption also explains the differences between the intended behavior given by constraints and the detected behavior given by a specific program run (a counterexample trace). If the result of reasoning may be consistent, then the assumption is correct. Those statements which the assumption supposes to be abnormal will be localized for program debugging.

Since the counterexample of a large program might be lengthy, it also consumes much time to find and understand the actual cause in the error trace. In order to make an assumption on statements in the given counterexample more efficiently, we need to use dynamic program slicing for narrowing the range of statements we want to assume to be abnormal.

Program slicing [6,16] is a program analysis and transformation technique that has been widely used for debugging, testing, verification, reverse engineering, and maintenance. Program slicing especially dynamic slicing [7,14,18] can be used to find the set of statements in an error trace that may be relevant to the cause of the error. Given a faulty statement or a variable of interest, a dynamic slicing algorithm identifies a subset of executed statements that influenced the computation of this statement or variable. Different dynamic slicing algorithms use different notions of what they consider as influencing. In [7] an efficient relevant dynamic slicing algorithm is presented for debugging.

We extend this algorithm to meet the feature of a counterexample trace so that the extended algorithm can capture the possible faulty statements more precisely.

## 2 A Motivating Example

To illustrate the idea of our approach, we consider a motivating example as shown in Figure 1. This small program is to find the maximum of three inputs, and assign the maximum value to a variable \( \text{max} \). We use Java PathFinder (JPF) model checker to verify this program. JPF is an explicit state on-the-fly model checker that takes Java bytecode class files and analyzes all the paths through the program for deadlock, assertion violations and linear temporal logic (LTL) properties [13]. The method Verify.random\( (n) \) will nondeterministically return an integer value in the range \([0, n]\). In the program \( x_1 = \text{Verify.random}(4) + 1 \) means that \( x_1 \) will randomly take an integer value in the range \([1, 5]\). From the specification \( (\text{max} >= x_1) \&\& (\text{max} >= x_2) \&\& (\text{max} >= x_3) \), we can get three constraints of the program which are annotated in Figure 1. A certain constraint at a certain position of the program means that when the program executes at this position, the program behavior should satisfy this constraint.

```java
1:  public class FindMax {
2:      public static void main(String[] args) {
3:          int x1 = Verify.random(4) + 1;
4:          int x2 = Verify.random(4) + 1;
5:          int x3 = Verify.random(4) + 1;
6:          int max = x1;
7:          if (max <= x2) {
8:              max = x2;
9:          } else if (max <= x3) {
10:             max = x3;
11:          } // constraint: (max >= x1) && (max >= x2) && (max >= x3)
12:          assert((max >= x1) && (max >= x2) && (max >= x3));
13:      }
14:  }
```

**Figure 1.** A motivating Java program.

The assertion is not true. JPF returns a counterexample \( CE = \{3, 4, 5, 6, 7, 8, 9, 12\} \), where \( x_1 = 1, x_2 = 1 \) and \( x_3 = 2 \) respectively. We use a dynamic slicing algorithm to slice the counterexample \( CE \) for the criterion \( (12^6, U(12)) \) \( (12^6 \) represents the statement 12 at the eighth step in \( CE \) and \( U(12) \) represents the variables used in statement 12), and get the slice \( \{3, 4, 5, 6, 7, 8, 9\} \) which are responsible for the violation of assertion. We will give the detail for dynamic slicing in Section 3. Notice that when reasoning along a counterexample, we discard the last assertion on the counterexample which is violated. To be better understood, we unwind the counterexample \( CE \) as follows (\( p7 \) and \( p9 \) represent predicates \( max <= x2 \) and \( max >= x3 \) respectively):

\[
\begin{align*}
&3^1: x_1 = 1 \rightarrow 4^2: x_2 = 1 \rightarrow 5^3: x_3 = 2 \rightarrow 6^4: max = x_1 \rightarrow 7^5: max <= x_2 (p7 == T) \rightarrow 8^6: max = x_2 \rightarrow 9^7: max >= x_3 (p9 == 1) \rightarrow (max >= x1) \&\& (max >= x2) \&\& (max >= x3)
\end{align*}
\]

**Assumption 1:** \( \text{AB}(7) \) which means that statement 7 is abnormal (not correct).

1. \( 6^4: max = x_1 \rightarrow max = x_1 \); \{assignment rule\}
2. \( 1^1: max = x_1 \rightarrow \neg (max <= x_2)(abnormal) \); \{abnormal rule\}
3. \( 2^1: max = x_1 \rightarrow 8^6: max = x_2 \); \{assignment rule\}
In fact, statement 9 is faulty, we should replace predicate will be presented in Section 4. In this case, assumption 2 is wrong. Formal definition of diagnoses of the program is true; if the result is of more executed information of the program), we have one counterexample during model checking in general (lack be consistent with the intended one. Since we can get only be assigned to another value, however which value should be assigned to x cannot be predicted. Since assumption 1 states that statement 7 is abnormal, we cannot predict any state of statement 7, i.e., cannot predict any values of predicate max <= x2. From the above reasoning, we can get that the behavior of program under assumption 1 is inconsistent with the intended behavior of program. Hence assumption 1 is not correct, and we have ¬AB(7) which means that statement 7 is correct. With the similar calculus we also get that ¬AB(6) and ¬AB(8).

**Assumption 2:** AB(9), that is, statement 9 is abnormal.

We have the following calculus of reasoning:

\[
\begin{align*}
(1) & \quad \text{max} \mapsto x_1 \quad \text{max} \mapsto x_1 \\
(2) & \quad \text{max} \mapsto x_1 \quad \text{max} \mapsto x_1 \\
(3) & \quad \text{max} \mapsto x_1 \quad \text{max} \mapsto x_1 \\
(4) & \quad \text{max} \mapsto x_1 \quad \text{max} \mapsto x_1 \\
(5) & \quad \text{max} \mapsto x_1 \quad \text{max} \mapsto x_1 \\
(6) & \quad \text{max} \mapsto x_1 \quad \text{max} \mapsto x_1 \\
(7) & \quad \text{max} \mapsto x_1 \quad \text{max} \mapsto x_1 \\
\end{align*}
\]

The logic rules for reasoning are shown in Figure 4. Note that we cannot use the input values of variables of the program such as \(x_1 = 1, x_2 = 1\) and \(x_3 = 2\). We can only process the calculus symbolically. Under this way, we can reason the cause from the symptom exhibited by the counterexample. The notion of “abnormal” in this paper means not correct or faulty. For instance, if an assignment statement \(x = 1\) is assumed to be abnormal, then it means that the variable \(x\) is not assigned correctly, \(x\) should be assigned to another value, however which value should be assigned to \(x\) cannot be predicted. Since assumption 1 states that statement 7 is abnormal, we cannot predict any state of statement 7, i.e., cannot predict any values of predicate \(\text{max} \leq x_2\). From the above reasoning, we can get that the behavior of program under assumption 1 is inconsistent with the intended behavior of program. Hence assumption 1 is not correct, and we have \(\neg AB(7)\) which means that statement 7 is correct. With the similar calculus we also get that \(\neg AB(6)\) and \(\neg AB(8)\).

**Assumption 2:** AB(9), that is, statement 9 is abnormal.

We have the following calculus of reasoning:

\[
\begin{align*}
(1) & \quad x_1 = x_1 \\
(2) & \quad x_1 = x_1 \\
(3) & \quad x_1 = x_1 \\
(4) & \quad x_1 = x_1 \\
(5) & \quad x_1 = x_1 \\
(6) & \quad x_1 = x_1 \\
(7) & \quad x_1 = x_1 \\
\end{align*}
\]

3 Dynamic Program Slicing

A slice consists of all statements and predicates that may affect the variables in a set \(V\) at a given program point [6,16]. We say a statement \(y\) influences statement \(x\) (or equivalently, \(x\) is dependent on \(y\)) if either the values computed at \(x\) or the mere execution of \(x\) depends on values computed at \(y\). We write \(I(y,x)\) if \(y\) influences \(x\). Note that \(I(y,x)\) is in general undecidable. Slicing is a conservative approximation of the predicate \(I\). Slicing algorithm can be classified according to whether they only use statically available information (static slicing) or compute those statements which influence the value of a variable for a specific program input (dynamic slice).

Traditional static slices can be computed using a program-dependence graph \(PDG(N, \rightarrow)\), where \(N\) is a set of nodes representing statements, expressions, and control predicates, and \(\rightarrow\) denotes control dependence edges and data dependence edges [8]. In a program dependence graph \(PDG = (N, \rightarrow)\), \(e \in N\) is control dependent on \(y \in N\), if its execution is determined by \(y\) (e.g., in an if- or while-statement). Two nodes are data dependent on each other if a definition at one node might be used at the other node. This is a quite natural representation of the program under investigation since it captures the program elements of interest and the dependences among them that will eventually determine the slices.

The basic dynamic slicing approaches use different concepts. One of the most common approaches to dynamic slicing is based on computing the dynamic dependencies among program elements. T.Gyimothy et al. [7] present an efficient algorithm for computing dynamic slicing. All it need is an execution history or trace and does not need to construct dynamic dependence graph. The execution history is a simple list of instructions occurrence of the program that have been executed for a specific program input. They use a simple definition-use relationship for all instructions of the program, based on which the actually realized dynamic dependence chain can be computed during processing the trace. Furthermore, by considering the predicates in the program as regular variables, both data- and control-dependencies can be handled uniformly.

In [7], an instruction of a program has the D/U representation: \(i.d_i : U_i\), where \(i\) is an instruction serial number. \(d_i = d(i)\) is the defined variable at the \(i\)th instruction, while \(U(i) = U_i\) is used to denoted the use set that is utilized for computing the value of \(d_i\). As in [7], we treat control predicate as a regular variable that can serve both as the defined variables and as used ones. The execution history will be denoted by \(EH\) (in this paper, it refers to a counterexample of a program), and it will contain actions denoted by \(\nu\), where \(i\) is the serial number of an instruction in the program, while \(j\) is the serial number of an execution step in the ex-
execution history. The total number of the steps executed in a given execution of the program is denoted by $J$, while $I$ denotes the total number of instructions in the program. We have: $i(i') = i$, $j(i') = j$, and $EH = \langle i_1', i_2', \ldots, i_k' \rangle$. The last definition position of variable $d$ is denoted by $LP(d)$. That is, $LP(d)$ is the execution position where $d$ is defined last. Similarly, the serial number of the instruction which is executed at position $LP(d)$ is denoted by $LS(d)$.

If we associate a slicing criterion with a set of program locations whose earlier execution affected the value computed at the criterion, we say a backward slicing. On the contrary, a forward slice is a set of program locations whose later execution depends the values computed at the slicing criterion. We use the notation $BC = (x, i', V)$ for the backward slicing criterion, and $FC = (x, i')$ for the forward slicing criterion, where $x$ is a program input corresponding to a specific execution of the program, $i'$ is the action for which the (dynamic) slice needs to be computed and $V$ is a subset of the used variables at the $i$th instruction. Since $EH$ always refers to a counterexample, and our reasoning is processed along a counterexample, we replace $x$ with $CE$ in our slicing criterion. In this paper, we focus on backward slicing. To meet the feature of counterexample the criterion is always setted to be $(CE, i_0', V)$, where $i_0'$ represents the statement $i_0$ where the assertion is violated, and $V$ is the set of variables that used at statement $i_0$ and $J$ is the last step on the counterexample $CE$.

For a counterexample returned by a model checker, we should pay more attentions on such predicate that at its last occurring position, its value is false. For many cases in model checking, due to the false value of the predicate at its last occurring position, the result of verified property leads to be unsatisfied. On the other hand, if the control predicate takes a false value at a certain position, the dynamic slicing algorithm sometimes will not consider the variables used by this predicate at this position. In this case we need to include these predicates when slicing programs for debugging. Assume that we are running a program on a counterexample. After an instruction $i.d : U$ has been executed at position $j$, sets of variable $d$ used in $j$th instruction at $j$th step on counterexample are determined by $DynSlice(i') = DynSlice(d(j)) = \bigcup_{u_k \in U} (DynSlice(u_k) \cup \{LS(x_k)\})$. We present an extended dynamic slicing algorithm as in Figure 2.

To illustrate how this dynamic slicing algorithm works, we slice the program in Figure 1 along trace $CE$ shown in Figure 3. According to the above slicing algorithm, since predicate $p7$ is false at its last occurrence on $CE$, $DynSlice(12^8) = \{3, 4, 5, 6, 7, 8, 9\}$. If we make assumptions among all statements of a large program without slicing the program first, the calculus of reasoning will be a time-consuming work.

![Figure 2. A dynamic slicing algorithm.](image)

![Figure 3. A dynamic slicing algorithm.](image)
from statements like if (p) . . . or while (p) . . . or successfully executing of assert(p).

In order to detect possible faulty statements of the Java program, a description of the logic rules for reasoning is needed. The predicate AB(i) is used to indicate that statement i is being abnormal. A correctly behaving statement i is therefore denoted as ¬AB(i). We now present a part of logic rules for reasoning shown as in Figure 4. The assignment rule indicates that after v = e the predicate v == e holds (e equals to e). Branching rule says that if the value of predicate p of branching statement is true (false) at the j-th step on the execution path, the predicate p (¬p) holds at the j-th step. Transforming rule indicates that if v equals to e and propositional formula φ related to v is true, then φ(e/v) is also true where φ(e/v) denotes the substitution of v with e. For example, if z == x and φ = (x <= y) then φ(z/x) = (z <= y). Abnormal rules indicate that we should not consider the state of abnormal statement i during reasoning. The meanings of the rules 7 and 8 are obvious (note that p and q are both predicates).

Given a program P and the set S of specifications which need to be verified for P. Let CE be a counterexample returned by a model checker. Let R be a set of propositional logic rules that represents the behavior of the individual statement of P. Note that i is an index corresponding to a certain statement of the program.

(1) ¬AB(i) ⇒ i\[i\] (assignment rule)
(2) ¬AB(i) ⇒ i\[i\] assume(p) p→T (branching rule)
(3) ¬AB(i) ⇒ i\[i\] assume(p) p→¬ (branching rule)
(4) v == e d(v/c) (transforming rule)
(5) AB(i) ⇒ i\[i\] (abnormal rule)
(6) AB(i) ⇒ i\[i\] (abnormal rule)
(7) p→p (V_rule) (8) p→¬P (V_rule)

Figure 4. A part of logic rules of R.

**Definition 1** Given a program P, a set of logic rules R, a counterexample CE, some constraints S and a set of indices I where each i ∈ I corresponds to a certain statement of P. A diagnosis for (P, R, CE, S) is a set i ∈ I such that

R ∪ CE ∪ {AB(c) | c ∈ Δ} ∪ {¬AB(c) | c ∈ I − Δ} ⇒ ¬S ...

**Remark 1** The formula (∗) is a first order logic formula whose truth value can be evaluated using the minimal hitting set approach. The formula (**) represents a procedure of reasoning using propositional logic rules. On the other hand, we can get only one counterexample during diagnosing a program in general (model checkers return only one counterexample when the property is not satisfied), as in Section 2, we may get the result ‘Unknown’ for the above formula (**). Then we say that the result may be consistent with S. In this case, we consider that the assumption Δ is correct due to the lack of more executed information of P.

**Theorem 1** A nonempty diagnosis for (P, R, CE, S) exists iff R ∪ CE ∪ {¬AB(c) | c ∈ I} ⇒ ¬S.

Theorem 1 says that a nonempty diagnosis exists if and only if we assume all statements of program are correct then it will be deduced that the result of reasoning will be inconsistent with S.

**Definition 2** For (P, R, CE, S), if R ∪ CE ∪ {¬AB(c) | c ∈ I} ⇒ ¬S, we then say that all statements of the program are correct. Furthermore, if some constraint of S is not satisfiable, we say that there is an absence of function of the program. That is, the program still does not run correctly according to the specifications.

Definition 2 states that the behavior of the program is partially correct, but we still need to add some statements to perform some function such that all specifications of the program are satisfied. More details will be illustrated in section 5. So far there are two types of diagnoses for program P. One is the diagnoses for the statements of program P. The other is the diagnoses for the absence of function.

**Definition 3** A conflict set for (P, R, CE, S) is a set Δ ⊆ I such that R ∪ CE ∪ {¬AB(c) | c ∈ Δ} ∪ {AB(c) | c ∈ I − Δ} ⇒ ¬S.

For the program in Figure 1, (9) is a conflict set. The relationship between diagnoses and conflicts can be explained as follows.

**Theorem 2** Δ ⊆ I is a diagnosis for (P, R, CE, S) iff Δ is a minimal set such that I − Δ is not a conflict set for (P, R, CE, S).

If diagnosis Δ contains only one element, it is called a single fault diagnosis. If it contains more than one element, it is called a multiple fault diagnosis. It can be derived that the consequence of the assumption also explains the differences between the intended behavior given by constraints and the detected behavior given by a counterexample trace.

For the program shown in Figure 1, it has a single fault diagnosis. We now reason about a program which has two-fault diagnosis. Consider the program shown in Figure 5. Note that this program is also illustrated in [9,12,20].
1: public class AcquireReleaseLocks {
2:   public static int LOCK=0;
3:   public static void main(String[] args) {
4:       int got_lock=0;
5:       do {
6:           if (Verify.randomBool()) {
7:               lock();
8:               got_lock += +;
9:           }
10:         if (got_lock!=0) {
11:             unlock();
12:         }
13:         got_lock -= -;
14:       } while (Verify.randomBool());
15:   }
16:   public static void lock() {
17:       assert(LOCK==0);
18:       LOCK=1;
19:   }
20:   public static void unlock() {
21:       assert(LOCK==1);
22:       LOCK=0;
23:   }
24:   }

Figure 5. A Java program which has a two-fault diagnosis.

This program calls method lock() and unlock() in order to acquire and release a lock. The method lock() checks that the lock is available and requests it. Vice versa, unlock() checks that the lock is held and releases it. The lock is represented by variable LOCK which is set to 0 or 1 with respect to the state of the lock. The variable got_lock is used to keep track of the status of the lock. Method Verify.randomBool() returns a random boolean value in JPF. Thus, the if-statement at line 6 causes the lock to be requested nondeterministically, and the while-statement at line 14 causes the loop to be executed an arbitrary number of times. The explicit specifications are two assertions, and an inherent specification is the program should acquire a lock and release a lock in strict alternation along all execution traces. We put got_lock==0 at line 15 as a constraint to check whether the program acquires a lock and releases a lock in strict alternation along all execution traces.

We check the program by JPF. The assertions are violated and we get a counterexample \{2, 4, 6, 10, 13, 14, 6, 10, 21\}. We first slice the program along this counterexample using the algorithm in Section 3 and obtain the dynamic slice \{2, 4, 6, 10, 13, 14\}. Moreover, We unwind the counterexample CE as follows (p6, p10 and p14 represent predicates at line 6, 10, 14 respectively):

\[2^2: \text{LOCK} = 0 \rightarrow 4^2: \text{got_lock} = 0 \rightarrow 6^3: P6(== \top) \rightarrow 10^4: \text{got_lock}! = 0 (p10 == \bot) \rightarrow 13^5: \text{got_lock} =_→ \]

\[14^6: p14 \ (== \top) \ \{\text{got_lock==0} \} \rightarrow 6^7: P6(== \bot) \rightarrow 10^8: \text{got_lock}! = 0 (p10 == \top) \]

**Assumption 1:** \(\text{AB(10)}\) which means that statement 10 is abnormal.

We have the following calculus of reasoning:

1. \(4^2: \text{got_lock}=0\ 10^4: (\text{got_lock}!)=0\) (abnormal);  
2. \(4^2: \text{got_lock}==0\ 13^5: \text{got_lock}→=\);  
3. \(2^2: \text{got_lock}==0\ 10^8: (\text{got_lock})!=0\) (abnormal).

From above, we conclude that assumption 1 is wrong, i.e., statement 10 is correct.

**Assumption 2:** \(\text{AB(13)}\) which means that statement 13 is abnormal.

1. \(4^2: \text{got_lock}=0\ 10^4: (\text{got_lock}!)=0\) (abnormal);  
2. \(4^2: \text{got_lock}==0\ 13^5: \text{got_lock}→=\);  
3. \(2^2: \text{got_lock}==0\ 10^8: (\text{got_lock})!=0\) (abnormal).

Therefore, assumption 2 is also wrong, i.e., statement 13 is correct. This conclusion is different from the one in [9,12,20]. In fact the real fault of the program is that statement 13 should be placed within the scope of if-statement at line 10. Thus for single statement 13, it is not abnormal, and however it should be controlled by statement 10. Furthermore, We make another assumption which states that both statement 10 and 13 are abnormal.

**Assumption 3:** \(\text{AB(10) and AB(13)}\).

1. \(4^2: \text{got_lock}=0\ 10^4: (\text{got_lock}!)=0\) (abnormal);  
2. \(4^2: \text{got_lock}==0\ 13^5: \text{got_lock}→=\);  
3. \(2^2: \text{got_lock}==0\ 10^8: (\text{got_lock})!=0\) (abnormal).

From the above reasoning, we have that assumption 3 is correct, i.e., statement 10 and 13 are both abnormal. Compared our result with the one of [9,10,20], our method is more precise. We make other assumptions such as \(\text{AB(2)}, \text{AB(4)}, \text{AB(6)}\) and \(\text{AB(14)}\). They are all not correct, so the diagnosis of the program according to this counterexample is a two-fault diagnosis.

## 5 Case Studies

In this section we present initial experimental results supporting the applicability of our approach on real life programs. First we illustrate the sorter program which related to the absence of function.
5.1 Sorter

The sorter program shown in Figure 6 is an example program within JPF 4.1 which is to sort four arbitrary numbers in ascending order.

```java
1: class sorter2 {
2:     public static void main(String[] args) {
3:         int a=verify.random(4);
4:         int b=verify.random(4);
5:         int c=verify.random(4);
6:         int d=verify.random(4);
7:         int temp=0;
8:         if (a>b) {
9:             temp=b;
10:         } else {
11:             temp=a;
12:         }
13:         if (b>c) {
14:             temp=c;
15:         } else {
16:             temp=b;
17:         }
18:         if (c>d) {
19:             temp=d;
20:         } else {
21:             temp=c;
22:         }
23:         if (b>e) {
24:             temp=;
25:         } else {
26:             temp=
27:         }
28:         if (a>b) {
29:             temp=b;
30:         } else {
31:             temp=a;
32:         }
33:         assert(a<=b) && (b<=c) && (c<=d);
34:     }
```

Figure 6. The sorter program.

We check this program by JPF. The assertion is violated and we get a counterexample \(\{3, 4, 5, 6, 7, 8, 13, 14, 15, 16, 18, 19, 20, 21, 23, 28, 29, 30, 31, 33\}\) where the input values of \(a, b, c, \) and \(d\) are \(1, 1, 0, 0\) respectively. Note that the values of predicates at line 8 and 23 are false and the values of predicates at line 13, 18 and 28 are true. We make an assumption that all statements of the program are correct, and have the following reasoning:

1. \(a>b\) \(\implies\ \top\)
2. \(a\leq b\) \(\land\ \top\)
3. \(a\neq b\) \(\land\ \top\)
4. \(a\neq b\) \(\land\ \top\)
5. \(a\neq b\) \(\land\ \top\)
6. \(a\neq b\) \(\land\ \top\)
7. \(a\neq b\) \(\land\ \top\)
8. \(a\neq b\) \(\land\ \top\)
9. \(a\neq b\) \(\land\ \top\)
10. \(a\neq b\) \(\land\ \top\)
11. \(a\neq b\) \(\land\ \top\)
12. \(a\neq b\) \(\land\ \top\)
13. \(a\neq b\) \(\land\ \top\)
14. \(a\neq b\) \(\land\ \top\)

From the above reasoning we obtain that the assumption is correct, namely, all statements are correct. However from \(d>c\) \(\implies\ \top\) we cannot conclude that \(b<=c\) which is a constraint from the specification. Hence the program is partially correct, we should add some statements to perform a comparison between \(b\) and \(c\). Thus we need to add statements \(i f (b>c)\) \(\{\text{temp } = c; c = b; b = temp;\}\) at the end of the program, and so far the program is entirely correct.

5.2 TCAS Program

Traffic Alert and Collision Avoidance System (TCAS) is an aircraft conflict detection and resolution system used by all US commercial aircraft. The Georgia Tech version of the Siemens suite [21] constitutes an ANSI C version of the Resolution Advisory (RA) component of the TCAS system (173 lines of C code). The Siemens suite provides a set of C programs, each of which has a number of test cases. TCAS continuously monitors the radar information to check whether there is any neighbor aircraft that could represent a potential threat by getting too close. In such a case, it is said that an intruder aircraft is entering the protected zone. Whenever an intruder aircraft enters the protected zone, TCAS issues a Traffic Advisory (TA) and estimates the time remaining until the two aircraft reach the closest point of approach (and then begin to fly away from each other). Such estimate is used to calculate the vertical separation between the two aircraft assuming that the controlled aircraft either maintains its current trajectory or performs immediately an upward (downward) maneuver. Depending on the results obtained, TCAS may issue a RA suggesting the pilot either to climb or to descend. To be able to use JPF, we translate these C programs into Java programs.

Let \(\text{ASTEn}\) identify the statement where the analyzed subsystem starts the computation for selecting the best escape maneuver and \(\text{ASTUpRA}\) and \(\text{ASTDownRA}\) identify the statements where a climbing RA and a descending RA are selected respectively. We use a faulty version (version 1) of TCAS to check the property \(P_{1,\text{Cond}} \rightarrow \neg \text{PrA}\).
where $P1 _{\text{Cond}}=\text{Up	extunderscore Separation}<\text{Layer	extunderscore Positive RA ALT\textunderscore Thresh)} \&\& (\text{Down	extunderscore Separation }\geq \text{Layer	extunderscore Positive RA ALT\textunderscore Thresh})$ and $\text{PrA}=(\text{ASTEn} \&\& \text{ASTUpRA})$. This property is not satisfied, and we first slice the counterexample dynamically which selects 40 statements from 173 statements to account for the violation of the property. Moreover, as shown in Figure 7 we localize the fault in three statements as candidates for debugging using reasoning with constraints. In fact the real fault in this version is statement (3), and the corresponding correct statement is $\text{result}=!(\text{Own\_Below\_Threat}) || (\text{Own\_Below\_Threat}) \&\& !(\text{Down\_Separation }\geq \text{ALIM}())$.

Figure 7. Localizing errors in TCAS program of version 1.

Moreover, we use version 2 of TCAS to check the property $P2 _{\text{Cond}}=\text{PrB}$, where $P2 _{\text{Cond}}=(\text{Up\_Separation }\geq \text{Layer\_Positive RA ALT\textunderscore Thresh}) \&\& (\text{Own\_Tracked\_Alt }\geq \text{Other\_Tracked\_Alt}) \&\& (\text{Down\_Separation }\geq \text{Up\_Separation})$ and $\text{PrB}=(\text{ASTDownRA})$. This property is not satisfied. Our method localizes the error in two statements as candidates for debugging shown in Figure 8. Actually the real fault in version 2 is statement (1), and the corresponding correct statement is $(\text{Climb\_Inh}\_\text{Up\_Sep}+\text{NOZCROSS}\_\text{Up\_Sep})$.

Figure 8. Localizing errors in TCAS program of version 2.

5.3 Schedule Program

The schedule program of the Siemens test suite is a priority scheduler. The input of the program is a list of commands of the following kinds: $\text{FINISH}$ (this exits the current process), $\text{NEW\_JOB}$ (this adds a new process at specified priority), $\text{BLOCK}$ (this adds the current process to the blocked queue), $\text{QUANTUM\_EXPRIE}$ (this puts the current process at the end of its priority queue), $\text{UNBLOCK}$ (this unblocks a process from the blocked queue), and $\text{UPGRADE}$ (this promotes a process from the small-priority queue to the next higher priority). The output is a list of numbers indicating the order in which processes exit from the system. The program is written in C, and we also translated it into Java code. The program consists of 405 lines of code.

We use a faulty version of the program (version 2) to verify the property $(P1 _{\text{Cond}} \&\& P2 _{\text{Cond}}) \rightarrow \text{PrC}$, where $P1 _{\text{Cond}}=(\text{int})(\text{priority}[2].\text{mem\_count }\times \text{ratio}+1) \leq \text{priority}[2].\text{mem\_count}, P2 _{\text{Cond}}=(\text{int})(\text{block\_queue}.\text{mem\_count }\times \text{ratio}+1) \leq \text{block\_queue}.\text{mem\_count}$ and $\text{PrC}=(\text{priority}[3].\text{mem\_count} \geq (\text{count\_initial}+n\_new\_job+n\_upgrade+n\_unblock-n\_block-n\_finish))$. Note that $P1 _{\text{Cond}}$ and $P2 _{\text{Cond}}$ guarantee that the scheduler can select a process to upgrade and unblock. $\text{priority[3].mem\_count}$ indicates the number of processes at $\text{priority[3]}$, $\text{count\_initial}$ indicates the initial number of the processes at $\text{priority[3]}$, $n\_new\_job$ is the times of adding a new process at priority 3, $n\_upgrade$ is the times of promoting a process of priority 2 to priority 3, $n\_block$ indicates the times of adding a process of priority 3 to the blocked queue, $n\_unblock$ represents the times of unblocking a process from the blocked queue to $\text{priority[3]}$, and $n\_finish$ represents the times of exiting a process from $\text{priority[3]}$. We check this property after executing each command. The JPF shows that the property is not satisfied after executing command $\text{UNBLOCK}$. We slice the program along the counterexample. The dynamic slicing picks up 120 statements from 405 statements to account for the violation of the property. As shown in Figure 9, we localize the faults in five statements as candidates for debugging using reasoning with constraints. In fact the real faults in this version are statement (1) and statement (2), and the corresponding correct statements are $\text{count}=\text{block\_queue}.\text{mem\_count}$ and $n=(\text{int})(\text{count }\times \text{ratio}+1)$.

Figure 9. Localizing errors in Schedule program of version 2.
6 Related Work

The problem of fault localization for software programs has been the attention of recent research. Model checking based method [10] seeks additional execution traces by deploying the same model checker again with additional constraints. The work of Groce [10] uses a SAT based bounded model checker to produce the counterexample, and then uses a pseudo-boolean constraint solver to find a successful execution that is as close as possible to the counterexample. The difference between the successful execution and the counterexample are computed and is considered as the potential cause of failure.

Ball et al. [1] use multiple calls to a model checker and compare the counterexamples to a successful trace. Transitions that do not appear in a correct trace are reported as a possible cause for the fault. The work of Grove and Visser [9] is based on comparing negative and positive program traces. The set cause(neg) is to compute those statements that only appear in negative traces and are necessary for a trace to be negative. They can use this set to localize the errors of a program. However the precision of error localization depends on the positive and negative traces found by a model checker. As shown in [20], for the program in Figure 5 transition analysis shows that cause(neg) = \{6, 13, 14\} for search depth 6 and cause(neg) = \{6\} for search depth 18 respectively. If the search depth is increased to 23, the transition analysis reveals that it is not possible to identify any statements as a cause for the negatives or positives.

Wang et al. [4] present a causal analysis for a counterexample to determine which a particular line in the code is responsible for the fault of the program. They use a path-based syntactic-level weakest precondition algorithm to produce a proof of infeasibility for the given counterexample, which is a minimal set of word-level predicates extracted from the failed execution that explains why the execution fails. Though the causal analysis for a counterexample is good to localize the error of predicates of a program, it may not localize other types of errors of a program precisely. For the program in Figure 5, the causal analysis localizes statement 2 imprecisely to be responsible for the assertion violation for the counterexample \{2, 4, 6, 10, 13, 14, 6, 10, 21\}.

B.Jobstmann et al. [12] studied the program repair problem which is closely related to fault location. They consider the program repair problem as a game. The game is played between the environment, which provides the inputs, and the system, which provides the correct value for the unknown expression. The game is won if for any input sequence the system can provide a sequence of values for the unknown expression such that the specification is satisfied. A winning strategy fixes the proper values for the unknown expression and thus corresponds to a repair. The weakness of that work is that a suspicion of the location of the fault has to be given by the user. It should be integrated with fault localization.

Delta debugging [17] is a automatic testing algorithm that narrows the difference in the program states between a failing and a passing run to isolate the statement constituting the cause of the failure. This method is empirical, which is quite different from approaches based on formal or static analysis.

By forcible switching a predicate’s outcome at runtime and altering the control flow, the program state can not only be inexpensively modified, but in addition it is often possible to bring the program execution to a successful completion [15]. By examining the switched predicate, also called the critical predicate, the cause of the bug can then be identified. This technique is shown to be practical and effective for localizing errors in many practical cases.

M.Chechik et al. [11] present a general framework for generating and exploring witnesses and counterexamples of temporal logic properties. The framework is based on building evidence in the form of a proof and controlling which portions of the proof are expanded. Currently, this framework is to aid in understanding witnesses and counterexamples. It may be used as a basis for integration between model checking and debugging.

Statistical debugging [19] proposes a statistic approach to localize faults without prior knowledge of program semantics. The approach uses the predicated-based dynamic analysis. By exploring detailed statistics about predicate evaluation it ranks predicate according to how abnormally each predicate evaluates in incorrect executions. The more abnormal the evaluations, the more likely the predicate is fault-relevant.

The closely related work to ours is that of D.Kob and F.Wotawa [20]. They use a model-based debugging approach for error localization. The first step for debugging is also making assumptions on statements of a program. The program model is represented as a set of logical rules and some are the rules of Hoare logic. They use Hoare logic to propagate predicates through the program, the predicate states the values of variables. The resolution calculus is proceeded to find the candidate of diagnoses. To get a better precision for fault localization multiple counterexamples is needed. However their work is quite different from ours. We reason the diagnoses without any Hoare logic. By introducing constraints of the program, we can process reasoning along a counterexample symbolically. Comparing the the predicates, assignments and expressions on counterexample with the constraints which the program should obey, we can get whether the assumptions are correct or not and then can obtain the candidates of the diagnoses. Our approach is more precise. For example, to the program in Figure 5, only using the counterexam-
ple \{2, 4, 6, 10, 13, 14, 6, 10, 21\}, they get the diagnosis localizing fault in \{2, 3, 6, 10, 11, 13, 14\}. By getting another counterexample \{2, 4, 6, 10, 13, 14, 6, 17, 18, 8, 10, 13, 14, 6, 10, 21, 22, 13, 14, 6, 10, 21\}, the diagnosis is \{6, 13, 14\}.

7 Conclusion

We presented a formalization of error localization based on reasoning with constraints. The constraints are from the specifications that need to be verified for the program. The formalization of underlying diagnoses allows to assign a precise semantics to the diagnoses given by our approach. This is a difference to other approaches which have no clear semantics for the error which is localized. The cost of computing the diagnosis is relative inexpensive. The reasoning carries out symbolically on a single execution path (a counterexample). Unlike most of other approaches, we do not need additional execution traces other than a buggy one. Making assumptions on statements of a program is not much time consuming work since we use dynamic slicing first to narrow the range of statements which we want to assume. The initial experimental results support the applicability of our approach.

The main contributions of the paper are: Two types of diagnoses are discussed, the diagnosis of the absence of function and multiple fault program diagnosis have been little discussed in other work. Second, an improved dynamic slicing method is given which can select statements more precisely to account for the violation of the property. Third, the procedure of reasoning only uses the rules of propositional logic without any rules of Hoare logic (as in [20]) and first order logic (as in [2]) which will let the reasoning be more complicated. We currently localize errors for sequential Java programs, and do not consider concurrent features of Java programs. Extending our work to concurrent Java programs is the future work.

References


