Assumption-based Reasoning with Constraints for Diagnosing Program Errors

Fei Pu[1,2]
[1]College of Computer and Information Engineering
Zhejiang Gongshang University, Hangzhou, China
pufei@mail.zjgsu.edu.cn
[2]State Key Laboratory of Computer Science, Institute of Software
Chinese Academy of Sciences, Beijing, China

***Abstract*—Model checking is an efficient technique for detecting errors of a system. However diagnosing program errors is the most time consuming hard work. One of the major advantages of model checking is the production of a counterexample when a property violation is detected. The error trace produced by a model checker may exhibit the symptom related to the cause of errors. 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. The assumption among the statements in counterexample is made to point out which statement(s) is (are) faulty. Some constraints will be introduced from the specifications of the program. Moreover, we transform the counterexample into a propositional logic formula which is then fed to a SAT solver or a theorem prover together with constraints. A calculus of reasoning with these constraints proceeds under certain assumptions. If the result is satisfiable, the assumption is correct, otherwise, the assumption is wrong and a new assumption should be proposed. Some examples are presented to support the applicability and effectiveness of our approach.**

***Index Terms*—error diagnosing, assumption-based reasoning, model checking, constraints, counterexamples**

I. INTRODUCTION

Diagnosing, i.e., locating and correcting errors in programs, is a difficult task in general, because of the gigantic size of the search space for diagnosing and the complexity of intrinsic links in the system (i.e., the intended behavior of the program), and the process of realization.

Model checking is a popular automated verification technique used to check properties of the finite-state systems [6]. The model checking result is either a counterexample at the source level showing how the system could violate the property or a statement that the system respects the property. The counterexample returned by a model checker can help understanding symptoms related to errors. Since the cause of errors are hidden deeply in the code, A programmer needs to put significant effort to uncover it.

Model based diagnosis[2] is a theory for diagnosing physical systems. The diagnostic program is to determine those components of the system which is assumed to be functional abnormally, and 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. The observations in model based diagnosis are the inputs and outputs of the physical system. The system descriptions denoted as SD are the set of sentences represented as first-order sentences. The assumption where components are assumed faulty is derived after the system is diagnosed. A unary predicate $\mathrm{AB}(c)$ is used in the assumption which means that component c is abnormal.

We apply this idea to allocate errors of programs. However some modifications must be made to meet the features of programs. The program description is the set of transforming 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 determines a particular statement faulty. We then transform the counterexample into a logic formula which is then fed to a theorem prover or a SAT solver together with the constraints. This logic formula represents the detected program behavior and the constraints represent the intended one. The consequence of a certain assumption explains the differences between the intended behavior given by the constraints and the detected behavior given by a specific program (a counterexample trace). If the result of reasoning is satisfied, then the assumption is correct. We then locate errors in those statements where they are assumed faulty for program debugging. Otherwise, the assumption is wrong and another assumption should be made.

The rest of the paper is organized as follows. In Section 2 we give an example to illustrate our approach. Section 3 presents our assumption-based reasoning with constrains for error localizing. In Section 4 we further propose an approach to remove the redundant candidate statements for diagnosing. Three experiments are illustrated in Section 5 to support the applicability and effectiveness of our approach. Related work is presented in Section 6 and the

doi:10.4304/jcp.9.1.1-11

conclusion is given in Section 7.

II. An Example

To illustrate the idea of our approach, we give an example as shown in Figure 1. This segment of code is to find the middle number of three inputs. We use Java PathFinder (*JPF*) model checker[10] to verify this program. The method *Verify.random(n)* will nondeterministically return an integer value in the range $[0, n]$. In the program x=*Verify.random(9)*$+1$ means that x will randomly take an integer value in the range $[1, 10]$. From the specification $(y <= x\&\&x <= z\&\&m == x)||(z <= x\&\&x <= y\&\&m == x)||(z <= y\&\&y <= x\&\&m == y)||(x <= y\&\&y <= z\&\&m == y)||(x <= z\&\&z <= y\&\&m == z)||(y <= z\&\&z <= x\&\&m == z))$, we can get constraints of the program which are annotated in Figure 1. Certain constraint at certain position of the program means that when the program executes at this position, the program behavior should satisfy this constraint.

```
1:  public class MiddleNumber {
2:    public static void main(String[ ] args) {
3:      int x=Verify.random(9)+1;
4:      int y=Verify.random(9)+1;
5:      int z=Verify.random(9)+1;
6:      int m=z;
7:      if (y < z){
8:        if (x < y){
9:          m=y;
10:       } else {
11:         if (x < z) {
12:           m=y;
13:       }
14:      }
15:     }else
16:       if (x > y){
17:         m=y;
18:       } else {
19:         if (x > z) {
20:           m=x;
21:       }
22:     }
23: constraint: (y<=x && x<=z && m==x)|| (z<=x && x<=y &&
m==x)||(z<=y &&y <=x && m==y)||(x<=y && y<=z && m==y)||(x<=z
&& z<=y && m==z)||(y<=z && z<=x && m==z)
24: assert ((y<=x && x<=z && m==x)|| (z<=x && x<=y
&& m==x)|| (z<=y &&y <=x && m==y)|| (x<=y && y<=z
&& m==y)|| (x<=z && z<=y && m==z)|| (y<=z && z<=x
&& m==z))
25:    }
26:  }
```

Figure 1. A segment of code.

Obviously, the assertion is not true. *JPF* returns a counterexample CE=$\{3, 4, 5, 6, 7, 8, 11, 12, 24\}$, where $x = 2$, $y = 1$ and $z = 3$ respectively. The notation i^j represents the statement i at the j-th step in CE. 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($p7$, $p8$ and $p11$ represent predicates $y < z$, $x < y$ and $x < z$ respectively and $\top$ and $\bot$ represent true and false respectively):

$3^1 : x = 2 \rightarrow 4^2 : y = 1 \rightarrow 5^3 : z = 3 \rightarrow 6^4 : m = z \rightarrow 7^5 : y < z(p7 == \top) \rightarrow 8^6 : x < y(p8 == \bot) \rightarrow 11^7 : x < z\ (p11 == \top) \rightarrow 12^8 : m = y \quad \{(y <= x\&\&x <= z\&\&m == x)||(z <= x\&\&x <= y\&\&m == x)||(z <= y\&\&y <= x\&\&m == y)||(x <= y\&\&y <= z\&\&m == y)||(x <= z\&\&z <= y\&\&m == z)||(y <= z\&\&z <= x\&\&m == z)\}$

Assumption 1: AB(7) which means that statement 7 is abnormal or faulty.

We first transform the counterexample into a corresponding trace formula (for its definition see in Section 3).

$(1).\frac{6^4:m=z}{TR(m=z)=(m==z)}$;

$(2).\frac{7^5:p7=(y<z)\text{(abnormal)}}{TR(m=z;assume(y<z))=(m==z)}$;

$(3).\frac{8^6:p8=(x<y)==\bot}{TR(m=z;assume(y<z);assume(x<y))=(m==z)\wedge(x>=y)}$;

$(4).\frac{9^7:p9=(x<z)==\top}{TR(m=z;...;assume(x<z))=(m==z)\wedge(x>=y)\wedge(x<z)}$;

$(5).\frac{12^8:m=y}{TR(m=z;...;assume(x<z);m=y)=(x>=y)\wedge(x<z)\wedge(m==y)}$.

$m == z$ is removed from the trace formula since we can only take the latest value at any time for every variable. Now we get the trace formula $(x >= y) \wedge (x < z) \wedge (m == y)$ for counterexample CE under assumption 1. This trace formula and the constrains are fed to a SAT solver or a theorem prover to be examined if the trace formula is consistent with the constraints.

Note that we cannot use the input values of variables of the program such as $x = 2$, $y = 1$ and $z = 3$. We can only proceed the calculus symbolically. Under this way we reason the cause of property violation symbolically from the symptom exhibited by the counterexample. The notion of "abnormal" in this paper means incorrect or faulty. For instance, if an assignment statement $x = 1$ is assumed abnormal, then it means that the variable x is not assigned correctly, x should be assigned to another value, however it cannot be predicted what value should be assigned to x. 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 $y < z$. From the above reasoning, we get that $x >= y$, $x < z$ and $m == y$ contradict with the constraints $(y < x\&\&x < z\&\&m == x)||(z < x\&\&x < y\&\&m == x)||(z < y\&\&y < x\&\&m == y)||(x < y\&\&y < z\&\&m == y)||(x < z\&\&z < y\&\&m == z)||(y < z\&\&z < x\&\&m == z)$. That is, the behavior of program under assumption 1 is inconsistent with the intended behavior of program which is represented by constraints. This discrepancy implies that 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(11).

Assumption 2: AB(8), statement 8 is abnormal.

We have the trace formula as follows:

$(1).\frac{6^4:m=z}{TR(m=z)=(m==z)}$;

$(2).\frac{7^5:p7=(y<z)==\top}{TR(m=z;assume(y<z))=(m==z)\wedge(y<z)}$;

$(3).\frac{8^6:p8=(x<y)\ (abnormal)}{TR(m=z;assume(y<z);assume(x<y))=(m==z)\wedge(y<z)}$

$(4).\frac{11^7:p11=(x<z)==\top}{TR(m=z;...;assume(x<z))=(m==z)\wedge(y<z)\wedge(x<z)}$

$(5).\frac{12^8:m=y}{TR(m=z;...;assume(x<z);m=y)=(y<z)\wedge(x<z)\wedge(m==y)}$;

Under this assumption, we get the trace formula $(y < z) \wedge (x < z) \wedge (m == y)$ and then feed this trace formula together with constraints to a SAT solver. The result is `satisfied` which means that the observed behavior of the program is consistent with the intended one. We

then know assumption 2 correct, and statement 8 is a candidate for debugging. Formal definition of diagnoses of the program will be presented in Section 3.

Assumption 3: AB(12), statement 12 is abnormal.

We have the trace formula as follows:

$(1).\frac{6^4:m=z}{TR(m=z)=(m==z)}$;

$(2).\frac{7^5:p7=(y<z)==\top}{TR(m=z;assume(y<z))=(m==z)\wedge(y<z)}$;

$(3).\frac{8^6:p8=(x<y)==\bot}{TR(m=z;...;assume(x<y))=(m==z)\wedge(y<z)\wedge(x>=y)}$

$(4).\frac{11^7:p11=(x<z)==\top}{TR(m=z;...;assume(x<z))=(m==z)\wedge(y<z)\wedge(x>=y)\wedge(x<z)}$

$(5).\frac{12^8:m=y(abnormal)}{TR(m=z;...;assume(x<z);m=y)=(y<z)\wedge(x>=y)\wedge(x<z)}$

Note that both $m == z$ and $m == y$ are removed from trace formula since statement $m = y$ is executed later than statement $m = z$. On the other hand, statement $m = y$ is abnormal, then $m == y$ is also removed (since we cannot predict which value variable m should take). Under assumption 3, we get the trace formula $(y < z) \wedge (x >= y) \wedge (x < z)$ and then feed this trace formula together with constraints to a SAT solver. The result is `satisfied`, we then judge assumption 3 is correct and statement 12 is also a candidate for diagnosing. From assumptions 2 and 3 we allocate errors in $\{8, 12\}$. In fact, statement 12 is faulty, we should replace assignment $(m = y)$ with assignment $(m = x)$, then the program has no errors.

Using the assumption-based reasoning with constraints we obtain that the diagnosis for this program according to the counterexample CE is statements $\{8,12\}$.

We now illustrate how we can get whether or not the trace formula under a certain assumption contradicts with the constraints. We can use SAT solving towards this aim. Let $p1 = (y < x)$, $p2 = (x < z)$, $p3 = (m == x)$, $\neg p2 = (z < x)$, $\neg p1 = (x < y)$, $p4 = (z < y)$, $p5 = (m == y)$, $\neg p4 = (y < z)$, and $p6 = (m == z)$. Note that we only consider the case that three variables take different values. Thus $\neg p1 = (x < y)$ provided that $p1 = (y < x)$. Whether or not the trace formula contradicts with the constraints under assumption 1 is depending on the truth value of the propositional formula: $(p1 \wedge p2 \wedge p5) \wedge \{(p1 \wedge p2 \wedge p3) \vee (\neg p1 \wedge \neg p2 \wedge p3) \vee (p1 \wedge p4 \wedge p5) \vee (\neg p1 \wedge \neg p4 \wedge p5) \vee (p2 \wedge p4 \wedge p6) \vee (\neg p2 \wedge \neg p4 \wedge p6)\}$. We still need to present some apriori knowledge/information which is inherent to the program to determine the truth value of the above formula. First, it is easy to see that $p3 \rightarrow \neg p5 \wedge \neg p6$ which indicates that if variable m takes a value x then it cannot take another value such as y and z at the same time. Similarly, we have $p5 \rightarrow \neg p3 \wedge \neg p6$ and $p6 \rightarrow \neg p3 \wedge \neg p5$. Second, we have transitivity of $<$ or $>$. Thus $p1 \wedge p2 \rightarrow \neg p4$ which indicates that if $y < x$ and $x < z$ hold then $y < z$ holds. So we also have $p2 \wedge p4 \rightarrow \neg p1$ and $p1 \wedge p4 \rightarrow \neg p2$.

We use a SAT solver PicoSat [16] to calculate the truth value of the trace formula together with constraints. Under assumption 2, the formula being checked is changed to $(p3 \rightarrow \neg p5 \wedge \neg p6) \wedge (p5 \rightarrow \neg p3 \wedge \neg p6) \wedge (p6 \rightarrow \neg p3 \wedge \neg p5) \wedge (p1 \wedge p2 \rightarrow \neg p4) \wedge (p2 \wedge p4 \rightarrow \neg p1) \wedge (p1 \wedge p4 \rightarrow \neg p2) \wedge (p2 \wedge \neg p4 \wedge p5) \wedge \{(p1 \wedge p2 \wedge p3) \vee (\neg p1 \wedge \neg p2 \wedge p3) \vee (p1 \wedge p4 \wedge p5) \vee (\neg p1 \wedge \neg p4 \wedge p5) \vee (p2 \wedge p4 \wedge p6) \vee (\neg p2 \wedge \neg p4 \wedge p6)\}$. PicoSat returns the `satisfied` result with the true assignments $p1 = false$, $p2 = true$, $p3 = false$, $p4 = false$, $p5 = true$ and $p6 = false$. The true assignments indicate that $x < y$, $x < z$, $y < z$ and $m == y$ must hold at the end of some execution trace of the program. Under assumption 3, we check $(p3 \rightarrow \neg p5 \wedge \neg p6) \wedge (p5 \rightarrow \neg p3 \wedge \neg p6) \wedge (p6 \rightarrow \neg p3 \wedge \neg p5) \wedge (p1 \wedge p2 \rightarrow \neg p4) \wedge (p2 \wedge p4 \rightarrow \neg p1) \wedge (p1 \wedge p4 \rightarrow \neg p2) \wedge (p1 \wedge p2 \wedge \neg p4) \wedge \{(p1 \wedge p2 \wedge p3) \vee (\neg p1 \wedge \neg p2 \wedge p3) \vee (p1 \wedge p4 \wedge p5) \vee (\neg p1 \wedge \neg p4 \wedge p5) \vee (p2 \wedge p4 \wedge p6) \vee (\neg p2 \wedge \neg p4 \wedge p6)\}$, the result is also `satisfied`. The true assignments are $p1 = true$, $p2 = true$, $p3 = true$, $p4 = false$, $p5 = false$ and $p6 = false$ which indicates that $y < x$, $x < z$, $y < z$ and $m == x$ must hold at the end of some execution trace of the program.

III. Assumption-based Reasoning with Constraints

The diagnostic process is to identify faulty components of a functionally abnormal system and explain the discrepancy between the problematic and correct system behavior. In [2], a system is a pair $(SD, \text{COMPONENTS})$ where SD, the system description, is a set of first-sentences, and COMPONENTS is a finite set of constants. The observations of a system are a finite set OBS of first-order sentences of inputs and outputs of the system. A diagnosis for $(SD, \text{COMPONENTS}, \text{OBS})$ is a minimal set $\Delta \subseteq \text{COMPONENTS}$ such that

$$SD \cup \text{OBS} \cup \{\text{AB}(c) | c \in \Delta\} \cup \{\neg\text{AB}(c) | c \in \text{COMPONENTS} - \Delta\} \ldots (*)$$

is consistent. It is easy to derived that a diagnosis exists for $(SD, \text{COMPONENTS}, \text{OBS})$ iff $SD \cup \text{OBS}$ is consistent.

We extend this idea from physical system diagnosing to program debugging. An execution path $\pi = s_1, s_2, \ldots$ is a sequence of program statements. We also use $\pi^{i,j} = s_i, ..., s_j$ to represent the segment between s_i and s_j. As in [4], we use *assume(p)* to denote a branching statement where p is its predicate. That is, branching statement i: *assume(p)*, where p is a predicate. It might come from statements like *if (p)* ... or *while (p)* ... or successfully executing of *assert(p)*.

In order to detect possibly faulty statements of the program, a description of the error trace for reasoning under a certain assumption is needed. The predicate $AB(i)$ is used to indicate that statement i is abnormal. A correct behavior statement i is therefore denoted as $\neg AB(i)$. We now propose a definition of the error tracing formula under a given assumption.

***Definition** 1:* Given $\pi^{1,i-1} = s_1, ..., s_{i-1} (i > 1)$ and a propositional formula ϕ (ϕ is a trace formula which is obtained from the initial statement s_1 to statement s_{i-1}), the trace formula with respect to $\pi^{1,i}$, denoted by $TR(\phi, s_i)$, is defined as follows:

(1) For an assignment statement $s_i : v = e$, there are three subcases:

(1.1) if ϕ contains a formula $v == e_1$ for same variable v, $TR(\phi, s_i) = (\phi \setminus (v == e_1)) \wedge (v == e)$;

(1.2) if ϕ does not contain any formula $v == e_1$ for same variable v, $TR(\phi, s_i) = \phi \wedge (v == e)$;

(1.3) if ϕ is formula related to v, $TR(\phi, s_i) = \phi(e/v)$;

(2.1) For a branch statement $s_i : assume(p)$, if the value of predicate p is true when executing statement s_i, $TR(\phi, s_i) = \phi \wedge p$;

(2.2) For a branch statement $s_i : assume(p)$, if the value of predicate p is false when executing statement s_i, $TR(\phi, s_i) = \phi \wedge \neg p$;

(2.3) For a branch statement $s_i : assume(p)$, if ϕ contains a same predicate p and its value is p_1 ($p_1 = p$ if predicate p is true; otherwise, $p_1 = \neg p$), $TR(\phi, s_i) = (\phi \backslash p_1) \wedge p_2$ where $p_2 = p$ if predicate p is true when executing statement s_i; otherwise $p_2 = \neg p$;

(3.1) For an assignment statement $s_i : v = e$ which is assumed to be abnormal, $TR(\phi, s_i) = \phi$;

(3.2) For a branch statement $s_i : assume(p)$ which is assumed to be abnormal, $TR(\phi, s_i) = \phi$;

(4) For a sequence of statements $s_i; s_{i+1}$, $TR(\phi, s_i; s_{i+1}) = TR(TR(\phi, s_i), s_{i+1})$;

Note that formula $\phi \setminus (v == e_1)$ is a formula obtained by removing $v == e_1$ from ϕ and formula $\phi(e/v)$ denotes the substitution of v with e in ϕ. For instance, if $z == x$ and $\phi = (x <= y)$ then $\phi(z/x) = (z <= y)$.

Case (1.1) means that we can only take the latest value at any time for every variable. Case (2.3) has the same meaning. i.e., the latest value should be taken for the same predicate. Case (1.3) means that the trace formula should reflect the dynamic changes due to the execution of the program. Cases (3.1) and (3.2) represent that we don't need to consider any state of the abnormal statement since its value for the variable cannot be predicted.

We use propositional logic to handle the following program errors: wrong operator usage(e.g.: $<=$ instead of $<$), logical code bug, wrong assignment expression, error due to extra code fragments, wrong constant value supplied(e.g.: off-by-one error), wrong initialization of a variable and array index, predicate error in branching due to negation of branching condition and so on.

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 TR be a trace formula for CE under a certain assumption that represents the detected behavior of P. Note that i is an index corresponding to a certain statement of the program.

***Definition** 2:* Given a program P, a counterexample CE, a trace formula TR, a set of constraints S and a set of indices I where each $i \in I$ corresponds to a certain statement of P. A diagnosis for (P, TR, CE, S) is a minimal set $\Delta \subseteq I$ such that

$$TR \cup CE \cup \{\text{AB}(c)|c \in \Delta\} \cup \{\neg\text{AB}(c)|c \in I - \Delta\} \cup S \ldots (**)$$

is satisfied. On the other hand, if the above formula is unsatisfied, then the assumption $\{\text{AB}(c)|c \in \Delta\}$ is incorrect.

Remark 1: Formula $(*)$ is a first order logic whose truth value can be evaluated using the minimal hitting set approach. Formula (**) represents a procedure of reasoning about the propositional logic formula. Formula $(*)$ uses OBS as the whole observations of the physical system. Since programs may not have outputs, we need the constraints as a part of observations to guarantee that the behavior of the program is correct. On the other hand, the counterexample is also a part of observations representing the detected behavior of the program. Thus, the consequence of the assumption also explains the differences between the intended behavior given by the constraints and the detected behavior given by a trace formula.

***Definition** 3:* Given a diagnosis of (P, TR, CE, S), an abnormal assignment statement $s_1 \in \triangle$ denoted as $s_1 : v = e$, a transitive abnormal statement of s_1 is an assignment s_2 which use the value of v and the state of statement s_1 affects the state of statement s_2 on the execution path of program P.

For example, assume that $s_1 : y = x + 1$ and statement s_1 is abnormal, then the later executed statement $s_2 : z = y + 2$ is the transitive abnormal statement of s_1 since the assignment statement $s_2 : z = y + 2$ use the value of y and the value of y affects the value of z.

The transitive abnormal statements obviously are abnormal during reasoning. Let $\triangle'$ be the set of transitive abnormal statements of assumption Δ, we still denote $\triangle \cup \triangle'$ as $\triangle$ in the rest of this paper.

***Theorem** 1:* A nonempty diagnosis for (P, TR, CE, S) exists iff $TR \cup CE \cup \{\neg\text{AB}(c)|c \in I\} \cup S$ is unsatisfied.

Proof: "$\Rightarrow$" Since a nonempty diagnosis exists for (P, TR, CE, S), from definition 2 there is a nonempty minimal set $\Delta \subseteq I$ such that $TR \cup CE \cup \{\text{AB}(c)|c \in \Delta\} \cup \{\neg\text{AB}(c)|c \in I - \Delta\} \cup S$ is satisfied. This implies that $TR \cup CE \cup \{\neg\text{AB}(c)|c \in I\} \cup S$ is unsatisfied.

"$\Leftarrow$" Since $TR \cup CE \cup \{\neg\text{AB}(c)|c \in I\} \cup S$ is unsatisfied which means that the correct assumption is actually wrong; there is a nonempty minimal set $\Delta \subseteq I$ such that $TR \cup CE \cup \{\text{AB}(c)|c \in \Delta\} \cup \{\neg\text{AB}(c)|c \in I - \Delta\} \cup S$ is satisfied. □

***Definition** 4:* For a diagnosis of (P, TR, CE, S), if $TR \cup CE \cup \{\neg\text{AB}(c)|c \in I\} \cup S$ is satisfied, then we say all statements of the program are correct. On the other hand, some constraint of S is not satisfied, so that we say that there is an absence of function of the program. The program still does not run correctly according to the specifications.

Definition 4 states the fact 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 6. So far, there are two types of diagnoses for program P. One is diagnosis for the

statement of the program P. The other is diagnosis for the absence of the function.

***Definition** 5:* A conflict set for (P, TR, CE, S) is a set $\Delta \subseteq I$ such that $TR \cup CE \cup \{\neg \mathrm{AB}(c) | c \in \Delta\} \cup \{\mathrm{AB}(c) | c \in I - \Delta\} \cup S$ is unsatisfied.

For the program in Figure 1, $\{7\}$ and $\{11\}$ are both conflict sets. The relationship between a diagnosis and a conflict set can be explained as follows.

***Theorem** 2:* $\Delta \subseteq I$ is a diagnosis for (P, TR, CE, S) iff Δ is a minimal set such that $I - \Delta$ is not a conflict set for (P, TR, CE, S).

Proof: "$\Rightarrow$" Let $\Delta \subseteq I$ be a diagnosis for (P, TR, CE, S). Hence, $TR \cup CE \cup \{\mathrm{AB}(c) | c \in \Delta\} \cup \{\neg \mathrm{AB}(c) | c \in I - \Delta\} \cup S$ is satisfied. It can be restated that $TR \cup CE \cup \{\neg \mathrm{AB}(c) | c \in I - \Delta\} \cup \{\mathrm{AB}(c) | c \in \Delta\} \cup S$ is satisfied. Hence, $I - \Delta$ is not a conflict set for (P, TR, CE, S).

"$\Leftarrow$" Suppose that Δ is the minimal set such that $I - \Delta$ is not a conflict set for (P, TR, CE, S). We have that $TR \cup CE \cup \{\neg \mathrm{AB}(c) | c \in I - \Delta\} \cup \{\mathrm{AB}(c) | c \in \Delta\} \cup S$ is satisfied. It is easy to see that $TR \cup CE \cup \{\mathrm{AB}(c) | c \in \Delta\} \cup \{\neg \mathrm{AB}(c) | c \in I - \Delta\} \cup S$ is satisfied and Δ is a diagnosis for (P, TR, 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. We now reason for the program which has two-fault diagnosis. Let's consider the program shown in Figure 2. Note that this program is also illustrated in [7,9,13].

```
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:   }   // constraint: got_lock==0
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 2. A 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 in line 6 causes the lock to be requested nondeterministically, and the $while$-statement in line 14 causes the loop to be executed for an arbitrary number of times. The explicit specifications are two assertions, and an inherent specification is the program which 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 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 in line 6, 10, 14 respectively):

$2^1 : LOCK = 0 \rightarrow 4^2 : got\_lock = 0 \rightarrow 6^3 : P6(== \bot) \rightarrow$
$10^4 : got\_lock! = 0 (p10 == \bot) \rightarrow 13^5 : got\_lock-- \rightarrow$
$14^6 : p14\ (== \top)$ {got_lock==0} $\rightarrow 6^7 : P6(== \bot) \rightarrow$
$10^8 : got\_lock! = 0\ (p10 == \top)$

Assumption 1: AB(10) which means that statement 10 is abnormal.

We get the trace formula under assumption 1 as follows:

$$(1).\frac{4^2:got\_lock=0}{TR(got\_lock=0)=(got\_lock==0)};$$

$$(2).\frac{10^4:(got\_lock!=0)\ \text{(abnormal)}}{TR(got\_lock=0;assume(got\_lock!=0))=(got\_lock==0)};$$

$$(2).\frac{13^5:got\_lock--}{TR(got\_lock=0;...,got\_lock--)=(got\_lock==-1)};$$

So the trace formula is $TR(got\_lock = 0; ..., got\_lock --) = (got\_lock == -1)$ which contradicts with the constraint $got\_lock == 0$.

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

Assumption 2: AB(13) which means that statement 13 is abnormal.

$$(1).\frac{4^2:got\_lock=0}{TR(got\_lock=0)=(got\_lock==0)};$$

$$(2).\frac{10^4:(got\_lock!=0)==\bot}{TR(got\_lock=0;assume(got\_lock!=0))=(got\_lock==0)};$$

$$(3).\frac{13^5:got\_lock--\text{(abnormal)}}{TR(got\_lock=0;...;got\_lock--))=(got\_lock==0)};$$

$$(4).\frac{10^8:(got\_lock!=0)==\top}{TR(...;assume(got\_lock!=0);...;assume(got\_lock!=0))=(got\_lock!=0)}.$$

The trace formula $TR(got\_lock = 0; assume(got\_lock! = 0); ...; assume(got\_lock! = 0)) = (got\_lock! = 0)$ contradicts with the constraint $got\_lock == 0$. Therefore, assumption 2 is also wrong, i.e., statement 13 is correct. Furthermore, We make another assumption which states that both statement 10 and 13 are abnormal.

Assumption 3: AB(10) and AB(13).

$$(1).\frac{4^2:got\_lock=0}{TR(got\_lock=0)=(got\_lock==0)};$$

$$(2).\frac{10^4:(got\_lock!=0)\text{(abnormal)}}{TR(got\_lock=0;assume(got\_lock!=0))=(got\_lock==0)};$$

$$(3).\frac{13^5:got\_lock--\text{(abnormal)}}{TR(got\_lock=0;...;got\_lock--)=(got\_lock==0)};$$

$$(4).\frac{10^8:(got\_lock!=0)\text{(abnormal)}}{TR(got\_lock=0;...;assume(got\_lock!=0))=(got\_lock==0)}.$$

Thus $TR(got\_lock = 0; assume(got\_lock! = 0); got\_lock --) = (got\_lock == 0)$ is consistent with constraint $got\_lock == 0$.

It also shows that $TR(got\_lock = 0; assume(got\_lock! = 0); got\_lock - -; assume(got\_lock! = 0) = (got\_lock == 0)$ is consistent with constraint $got\_lock == 0$. From the above reasoning, assumption 3 is correct, i.e., statements 10 and 13 are both abnormal. 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, however it should be controlled by statement 10. Compared our result with the one of [7,9,13], our method is more precise. We make other assumptions such as AB(2), AB(4), AB(6) and AB(14). They are all incorrect, so the diagnosis of the program according to this counterexample is a two-fault diagnosis which is $\{10,13\}$.

IV. Refinement of Error Localization

Since the given counterexample can exhibit the symptom related to the cause of error, we first analyze the error trace, then the program is diagnosed. However, due to no enough execution information, the diagnosed candidates may be more than what we expected. To identify precise error location, we need more execution information of the program.

If the trace formula does not contradict with the constraints, i.e., $TR \cup CE \cup \{\text{AB}(c)|c \in \Delta\} \cup \{\neg\text{AB}(c)|c \in I - \Delta\} \cup S$ is satisfiable, the SAT solver will return true assignments of the formula which can provide information of correct program behavior. True assignments mean that in order to not contradict with the constraints, conditions of the true assignments must hold at the end of some execution traces. If there is no such execution trace that all conditions of the true assignments hold, then these statements for diagnosing under certain assumption are the most likely faulty statements. On the other hand, if there is such an execution trace that all conditions of the true assignments hold, then these statements for diagnosing under certain assumption are redundant and can be removed from debugged candidates. We can then narrow the range of statements for diagnosing.

The principle behind is that for assumed faulty statements, if there is an execution trace in which all conditions(predicates) of true assignments of formula $TR \cup CE \cup \{\text{AB}(c)|c \in \Delta\} \cup \{\neg\text{AB}(c)|c \in I - \Delta\} \cup S$ hold, then this trace corresponds to a witness of the program. That is, after executing of the witness of program the conditions of true assignment of the formula hold. So, these statements could be correct though they appear spuriously faulty.

On the other hand, if there is no such trace that all conditions of true assignments of formula $TR \cup CE \cup \{\text{AB}(c)|c \in \Delta\} \cup \{\neg\text{AB}(c)|c \in I - \Delta\} \cup S$ hold, then the predicate(s) of statement(s) assumed faulty cannot take any suitable value(s) in the execution of program so that the formula is satisfiable. Then, these statements are the real faulty statements. We illustrate these two cases by the example in Section 2.

As in Section 2, we localize errors in $\{8,12\}$ for the program shown in Figure 1. In order to highlight the more suspicious diagnosed candidates, we apply the above idea to this example. Note that for assumption 2 in Section 2, the corresponding true assignments is $x < y$, $x < z$, $y < z$ and $m == y$. These true assignments imply that in order not to contradict with the constraints, the predicates such as $x < y$, $x < z$, $y < z$ and $m == y$ must hold at the end of some execution traces of the program.

We put the assertion $assert(!((x < y)\&\&(x < z)\&\&(y < z)\&\&(m == y)))$ at line 23 at the end of program in Figure 1 to determine whether or not statement 8 is the real candidate for diagnosing. If there is an execution trace where $(x < y)\&\&(x < z)\&\&(y < z)\&\&(m == y)$ hold, *JPF* will return a counterexample that just represents this trace. We check this assertion and *JPF* returns an execution trace $\{3,4,5,6,7,8,9\}$ with the inputs $x == 1$, $y == 2$ and $z == 3$. Note that trace $\{3,4,5,6,7,8,9\}$ corresponds to a witness of the program and predicate $x < y$ in statement 8 can take value *true* in the witness such that the formula $TR \cup CE \cup \{\text{AB}(c)|c \in \Delta\} \cup \{\neg\text{AB}(c)|c \in I - \Delta\} \cup S$ is satisfiable. Hence, statement 8 is a redundant candidate for debugging.

In Section 2, for assumption 3, the corresponding true assignments are $y < x$, $x < z$, $y < z$ and $m == x$. Similarly, the assertion we put at line 23 is changed to $assert(!((y < x)\&\&(x < z)\&\&(y < z)\&\&(m == x)))$. *JPF* shows that the specification is correct. This means that there is no such execution trace that $(y < x)\&\&(x < z)\&\&(y < z)\&\&(m == x)$ hold at the end of program execution. The real cause is as follows: since $(y < x)\&\&(x < z)\&\&(y < z)\&\&(m == x)$ must hold after running the program, the program should execute the trace $\{3,4,5,6,7,8,11,12\}$. On the other hand, at statement 12 variable m takes a value y instead of x, $m == x$ cannot hold at the end of program execution. This means that statement 12 is the real error of the program. Moreover, the potential correction of the error is also provided. That is, at statement 12 $m = y$ should be replaced with $m = x$.

We present a formal rule for refining the error location as follows:

Refinement Rule: Let $\Delta \subseteq I$ be a diagnosis for (P, TR, CE, S) under a certain assumption. That is,

$$TR \cup CE \cup \{\neg\text{AB}(c)|c \in I - \Delta\} \cup \{\text{AB}(c)|c \in \Delta\} \cup S$$

is satisfiable. If all conditions in the corresponding true assignments of the above formula hold for some execution trace, then the diagnosis Δ is redundant and can be removed from candidates for debugging.

Remark 2: The discrepancies between the true assignments of a counterexample and the true assignments returned by solving the formula $TR \cup CE \cup \{\neg\text{AB}(c)|c \in I - \Delta\} \cup \{\text{AB}(c)|c \in \Delta\} \cup S$ might provide us intrinsic information about how to correct the faulty statement when no such execution trace that all conditions of the true assignments of the formula hold.

V. Case Studies

In this section we present initial experimental results supporting the applicability of our approach on real time programs. We have implemented a prototype program,

called trace formula producer(*TFP*), to get the trace formula for the given counterexample under a certain assumption. The inputs of the trace formula producer are the given counterexample and certain assumptions. First, we illustrate the sorter program related to the absence of function.

A. Sorter

The sorter program shown in Figure 3 is to sort four arbitrary numbers in an ascending order.

```
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:       b=a;
11:       a=temp;
12:     }
13:     if (b>c) {
14:       temp=c;
15:       c=b;
16:       b=temp;
17:     }
18:     if (c>d) {
19:       temp=d;
20:       d=c;
21:       c=temp;
22:     }
23:     if (b>c) {
24:       temp=c;
25:       c=b;
26:       b=temp;
27:     }
28:     if (a>b) {
29:       temp=b;
30:       b=a;
31:       a=temp;
32:     }  // constraint: (a<=b) && (b<=c)&& (c<=d)
33:     assert((a <= b) && (b <= c) && (c <= d));
34:   }
```

Figure 3. 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 in line 8 and 23 are both false and the values of predicates in line 13, 18 and 28 are all true. All assumptions that assume some individual statement is faulty are not true according to our method. We now make an assumption that all statements of the program are correct and have the following trace formula:

$$(1).\frac{8^{6}:(a>b)==\bot\ 13^{7}:(b>c)==\top}{TR(assume(a>b);assume(b>c))=(a<=b)\wedge(b>c)};$$

$$(2).\frac{14^{8}:temp=c}{TR(assume(a>b);...;temp=c)=(a<=b)\wedge(b>temp)};$$

$$(3).\frac{15^{9}:c=b}{TR(assume(a>b);...;c=b)=(c>temp)\wedge(a<=c)};$$

$$(4).\frac{16^{10}:b=temp}{TR(assume(a>b);...;b=temp)=(c>b)\wedge(a<=c)};$$

$$(5).\frac{18^{11}:(c>d)==\top}{TR(assume(a>b);...;assume(c>d))=(c>=a)\wedge(c>b)\wedge(c>d)};$$

$$(6).\frac{19^{12}:temp=d}{TR(assume(a>b);...;temp=d)=(c>=a)\wedge(c>b)\wedge(c>temp)};$$

$$(7).\frac{20^{13}:d=c}{TR(assume(a>b);...;d=c)=(d>=a)\wedge(d>b)\wedge(d>temp)};$$

$$(8).\frac{21^{14}:c=temp}{TR(assume(a>b);...;c=temp)=(d>=a)\wedge(d>b)\wedge(d>c)};$$

$$(9).\frac{23^{15}:(b>c)==\bot}{TR(...;assume(b>c))=(d>=a)\wedge(d>b)\wedge(d>c)\wedge(b<=c)};$$

$$(10).\frac{28^{16}:(a>b)==\top}{TR(...;assume(a>b))=(d>c)\wedge(c>=b)\wedge(d>=a)\wedge(a>b)};$$

$$(11).\frac{29^{17}:temp=b}{TR(...;temp=b)=(d>c)\wedge(c>=temp)\wedge(d>=a)\wedge(a>temp)};$$

$$(12).\frac{30^{18}:b=a}{TR(...;b=a)=(d>c)\wedge(c>=temp)\wedge(d>=b)\wedge(b>temp)};$$

$$(13).\frac{31^{19}:a=temp}{TR(...;a=temp)=(d>c)\wedge(c>=a)\wedge(d>=b)\wedge(b>a)};$$

The trace formula under the given assumption is $TR(assume(a > b); ...; a = temp) = (d > c) \wedge (c >= a) \wedge (d >= b) \wedge (b > a)$ which is consistent with the constraint $\{a <= b, b <= c, c <= d\}$. Hence, the assumption is correct, namely, all statements are correct. However, from $d > c >= a, d >= b > a$ we cannot conclude that $b <= c$ which is a constraint from the specification. So, the program is partially correct, we should add some statements to perform a comparison between b and c. Thus, we need to add statements $if(b > c)\{temp = c; c = b; b = temp;\}$ at the end of the program, and so far, the program is entirely correct. Trace formula producer(*TFP*) constructs the above trace formula and calls Picosat solver[16] to determine whether the trace formula contradicts with the constrains.

B. 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 [15] constitutes an ANSI C version of the Resolution Advisory (RA) component of the *TCAS* system (173 lines of C code). *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 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. Figure 4 shows the code of procedure *alt_sep_test()*.

Let *ASTEn* identify the statement where the analyzed subsystem starts the computation for selecting the best escape maneuver and, *ASTUpRA* and *ASTDownRA* identify the statements where climbing and descending RAs are selected respectively.

We use a faulty version (version 1) of *TCAS* to check the property *P1_Cond* $\rightarrow$ *!PrA*, where *P1_Cond=(Up_Sepration < Layer_Positive_RA_ALT_Thresh) && (Down_Sepration >= Layer_Positive_RA_ALT_Thresh)* and *PrA=(ASTEn && ASTUpRA)*. This property is not satisfied. The prototype *TFP* allocates

the fault in four statements as candidates for diagnosing shown in Figure 4. We further refine the error allocation using the method in Section 4.

```
      ......
public boolean Inhibit_Biased_Climb () {
⟹   return
      (Climb_Inh?Up_Sep+NOZCROSS:Up_Sep)    ... (1)
    }
      ......
public boolean Own_Below_Threat () {
⟹   return
      (Own_Tracked_Alt < Other_Tracked_Alt);    ... (2)
    }
      ......
public boolean Non_Crossing_Biased_Climb(){
      ......
⟹   result=
       !(Own_Below_Threat()) || (Own_Below_Threat())
          && (!(Down_Separation >ALIM());       ...(3)
      ......
public int alt_sep_test(){
      ......
⟹ need_upward_RA=Non_Crossing_Biased_Climb()
                        && Own_Below_Threat();    ...(4)
      ......
    }
      ......
  }
      ......
```

Figure 4. alLocating errors in *TCAS* version 1.

If we assume statement $result = !(Own\_Below\_Threat())||(Own\_Below\_Threat())\&\&(!(Down\_Separation > ALIM())$ is faulty, the true assignments indicate that variable $result$ in statement $result = !(Own\_Below\_Threat())||(Own\_Below\_Threat())\&\&(!(Down\_Separation > ALIM())$ should take false value at the end of some execution traces. We put the assertion $assert(result! = false)$ at the end of the program, *JPF* shows that this assertion is true which means that condition $(result == false)$ of the satisfiable assignments does not hold at the end of all execution traces. For the assumption that supposes any of three other statements is faulty, all conditions of true assignments are satisfied at the end of some execution trace. Thus, the fault to be refined is statement $result = !(Own\_Below\_Threat())||(Own\_Below\_Threat())\&\&(!(Down\_Separation > ALIM())$ and the other three statements are redundant candidates for diagnosing. In fact the real fault in this version is the above statement, and the corresponding correct statement is $result = !(Own\_Below\_Threat())||(Own\_Below\_Threat())\&\&(!(Down\_Separation >= ALIM())$.

Moreover, we use *TCAS* version 2 to check the property *P2_Cond* $\rightarrow$ *PrB*, where *P2_Cond=(Up_Sepration > Layer_Positive_RA_ALT_Thresh) && (Own_Tracked_Alt > Other_Tracked_ALT) && (Down_Separation >Up_Separation)* and *PrB=ASTDownRA*. This property is not satisfied. Our method allocates errors in four statements as candidates for debugging shown in Figure 5. We further refine the error allocation. For the assumption which assumes statement (3) is faulty, the truth assignments suggest that variable $result$ in method *Non_Crossing_Biased_Descend()* should take true value at the end of some execution trace. We put the assertion $assert(result! = true)$ at the end of the program, *JPF* shows that this assertion is true which means that condition $(result == true)$ of the satisfiable assignments does not hold at the end of all execution traces. Thus, statement (3) is the high suspicious candidate for debugging. Similarly, we show that the statements (1), (2) and (4) are redundant candidates for debugging. Actually, the real fault in version 2 is statement (1), and the corresponding correct statement is *(Climb_Inh?Up_Sep+NOZCROSS:Up_Sep)*.

```
      ......
public boolean Inhibit_Biased_Climb () {
⟹ return
    (Climb_Inh?Up_Sep+MINSEP:Up_Sep)          ... (1)
    }
      ......
public boolean Non_Crossing_Biased_Descend(){
      ......
⟹ upward_preferred=Inhibit_Biased_Climb()>
                              Down_Separation;     ... (2)
    if (upward_preferred){
⟹   result=
   !(Own_Below_Threat()) && (Cur_Vertical_Sep
  >=MINISEP) && (Down_Separation >ALIM());
                                                   ... (3)
      ......
    }
      ......
public int alt_sep_test(){
      ......
⟹ need_downward_RA=Non_Crossing_Biased
     _Descend() && Own_Above_Threat();          ... (4)
      ......
    }
      ......
  }
```

Figure 5. Allocating errors in *TCAS* version 2.

C. 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: *FINISH* (The current process exits by this execution), *NEW_JOB* (This adds a new process at specified priority), *BLOCK* (This adds the current process to the blocked queue), *QUANTUM_EXPIRE* (This puts the current process at the end of its prio_queue), *UNBLOCK* (This unblocks a process from the blocked queue), and *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 consists of 405 lines of C code.

We use a faulty version of the program (version 2) to verify the property *(P1_Cond && P2_Cond)* $\rightarrow$ *PrC*, where *P1_Cond=((int)(prio_queue[2].mem_count *ratio+1)* $\leq$ *prio_queue[2].mem_count*, *P2_Cond=((int)(block_queue.mem_count *ratio+1))* $\leq$ *block_queue.mem_count* and *PrC=(prio_queue[3].mem_count)==(count_initial+ n_newjob+n_upgrade+n_unblock-n_block-n_finish))*. *P1_Cond* and *P2_Cond* guarantee that the scheduler can select a process to upgrade and unblock.

prio_queue[3].mem_count indicates the number of processes at *prio_queue[3]*; *count_initial* indicates the initial number of the processes at *prio_queue[3]*; *n_newjob* 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 *prio_queue[3]*, and *n_finish* represents the times of exiting a process from *prio_queue[3]*.

We check this property after executing each command. The *JPF* shows that the property is not satisfied after executing command *UNBLOCK*. The prototype *TFP* allocates the faults in five statements as candidates for diagnosing shown in Figure 6. Similarly, using the refined method the statements $count = block\_queue.mem\_count + 1$ and $n = (int)count * ratio$ are allocated as high suspicious errors for diagnosing. In fact the real faults in this version are the above statements, and the corresponding correct statements are $count = block\_queue.mem\_count$ and $n = (int)(count * ratio + 1)$.

```
    ......
public void unblock_process(float ratio) {
    ......
⟹   count=block_queue.mem_count+1;      ..... (1)
⟹   n=(int) count*ratio;                ..... (2)
    ......
 ⟹ if (proc!=null) {                    ..... (3)
    ......
    }
  }
    ......
public Ele find_nth(List f_list, int n) {
    ......
⟹ if(n<=0)                             ..... (4)
⟹ return null;                         ..... (5)
  }
    ......
```

Figure 6. Allocating errors in Schedule program of version 2.

We also check the property *(P1_Cond && P2_Cond)* $\rightarrow$ *PrC* of Schedule programs in faulty version 1. *JPF* shows that the property is also not satisfied after executing command *FINISH*. The prototype *TFP* allocates the faults in three statements as candidates for diagnosing shown in Figure 7. Furthermore, If the refined method is used, the statements *if (count > 1)* is allocated as high suspicious error for diagnosing. In fact the real faults in this version are the above statement, and the corresponding correct statement is *if (count > 0)*.

```
    ......
public void upgrade_process_prio(int prio, float ratio) {
    ......
⟹   if (count > 1) {                    ..... (1)
      n = (int) (count * ratio + 1);
    ......
    }
  }
    ......
public void unblock_process(float ratio) {
   if (!block_queue.isEmpty()) {
⟹ count = block_queue.mem_count;        ..... (2)
⟹ n=(int)(count*ratio+1);               ..... (3)
    ......
    }
  }
    ......
```

Figure 7. Allocating errors in Schedule program of version 1.

VI. Related Work

The software fault diagnosing problem has caught attention of recent research. Ilan [1] uses causality concepts to explain a counterexample. Case studies show that the method can substantially speed up time needed for understanding of a counterexample. Groce [8] 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 is computed and is considered as the potential cause of failure.

Ball et. al.[2] 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 of the fault. The work of Grove and Visser [7] 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 allocate the errors of a program. However the precision of error allocation depends on the positive and negative traces found by a model checker.

Wang et. al.[5] present a causal analysis for a counterexample to determine what 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.

B.Jobstmann et. al.[9] 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.

Delta debugging [12] 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 [11]. 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 allocating errors in many practical cases.

Statistical debugging [14] proposes a statistic approach to localize faults without prior knowledge of program semantics. The approach uses the predicated-based dynamic analysis. By exploring detailed statics 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.

Some static and hybrid model-based approaches are proposed for software diagnosing [18]. One is a generalized value-based model where the component partitioning of a program as well as test drivers can be chosen freely. To handle programs with unstructured control flow, the static single information approach is presented. Moreover, a hybrid modeling approach is introduced where static analysis and abstract test execution are combined into a more flexible model by customizing a models structure to a given test specification.

Manu Jose et. al.[17] present an algorithm for error cause localization based on a reduction to the maximal satisfiability problem (*MAX-SAT*). They encode the semantics of a bounded unrolling of the program as a Boolean trace formula. For a failing program execution (e.g., one that violates an assertion or a post-condition), They construct an unsatisfiable formula by taking the formula and additionally asserting that the input is the failing test and that the assertion condition does hold at the end. Finally, using *MAX-SAT*, they find a maximal set of clauses in this formula that can be satisfied together, and output the complement set as a potential cause of the error. As stated in [17], their method is limited by the existing code, they cannot localize errors that can only be fixed by adding additional codes.

R.Ceballos, et al, [19] present a method for software diagnosis which is based on the combination of design by contract, model-based diagnosis and constraint programming. The main idea is to transform the contracts and source code into an abstract model based on constraints in a Max-CSP (Maximal Constraint Satisfaction Problem). This model enables the detection of errors in contracts and/or in a source code.

Lei Zhao, et al, [21] propose a context-aware fault localization approach via path analysis. They use the program control flow graph to organize the coverage and calculate edge suspiciousness. Given a failed execution, they use the fault proneness to assess how each block covered by this execution contributes to the failure by contrasting the coverage statistics of different edges covering the block. The sum of all fault proneness for every failed execution is defined as suspiciousness, and finally a ranked list can be synthesized to facilitate fault localization.

The closely related work to ours is that of D.Kob and F.Wotawa [13]. They use a model-based debugging approach for error localization. The program model is represented as a set of logical rules and some of rules 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 proceed reasoning along a counterexample symbolically. Comparing the predicates, assignments and expressions on counterexample with the constraints which the program should obey, we can get whether the assumptions are correct and can then obtain the candidates of the diagnoses. Our approach is more precise. For example, to the program in Figure 2, only using the counterexample {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 {2,4,6,10,13,14,6,17,18,8,10,13,14,6,17}, they localize fault in {6,10,13,14}. If getting the third 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}.

VII. Conclusion

We presented a formalization of fault diagnosing based on reasoning with constraints. The approach consists of three steps. First, the trace formula is constructed for a given counterexample under a certain assumption. This trace formula represents the detected behavior of the program under the given assumption. Second, the discrepancy between the trace formula and the constraints is determined by a SAT solver or a theorem prover. If the trace formula contradicts with the constraints, the given assumption is wrong. Otherwise, the statement under a certain assumption is a candidate for diagnosing. Finally, a refined method is applied to improve the precision of error localization.

Unlike most of other approaches, we do not need additional execution traces other than a buggy one. The cost of diagnosis is relative inexpensive. The procedure of reasoning only use propositional logic formulas without any Hoare logic and first order logic formulas which will let the reasoning more complicated.

Two types of diagnoses are discussed. One is the diagnosis of the absence of function and the other is multiple fault diagnosis which has little discussions on other work. The initial experimental results support the applicability of our approach.

Acknowledgment

This work is supported by Natural Science Foundation of Zhejiang Province under grant No.LY13F020009 and State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences under Grant No.SYSKF1011 and this article is an extended version of HASE 2008 conference paper[20].

References

[1] Ilan Beer, Shoham Ben-David, Hana Chockler1, Avigail Orni, and Richard Trefler. Explaining Counterexamples Using Causality. In *Proc. International Conference on Computer Aided Verification(CAV'09), LNCS 5643*, Springer, 2009, pp.94-108.

[2] T.Ball, M.Naik, S.K.Rajmani. From symptom to cause: Localizing errors in counterexample trace. In *Proc. ACM Symposium on Principles of Programming Languages (POPL'03)*, ACM, 2003, pp.97-105.

[3] R.Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 1987, 32:57-95.
[4] R.Greiner, B.A.Smith, and R.W.Wilkerson. A correction to the algorithm in Reiter's theory of diagnosis. *Artificial Intelligence*, 1989, 41:79-88.
[5] C.Wang, Z.Yang, F.Ivancic, and A.Gupta. Whodunit? Causal Analysis for Counterexamples. In *Proc. Symposium on Automated Technology on Verification and Analysis(ATVA'06), LNCS 4218*, Springer, 2006, pp.82-95.
[6] E.M.Clark, O.Grumberg, and D.Peled. Model Checking. The MIT Press, 1999.
[7] A.Groce and W.Visser. What went wrong: Explaining counterexamples. In *Proc. SPIN Workshop on Model Checking of Software (SPIN'03), LNCS 2648*, Springer, 2003, pp.121-135.
[8] A.Groce, S.Chaki, D.Kroening, and O.Strichman. Error Explanation with Distance Metrics. *International Journal on Software Tools for Technology Transfer*. 2006, 8(3): 229-247.
[9] B.Jobstmann, A.Griesmayer, and R.Bloem. Program repair as a game. In *Proc. International Conference on Computer Aided Verification(CAV'05), LNCS 3576*, Springer, 2005, pp.226-238.
[10] W.Visser, K.Havelund, G.Brat, and S.Park. Model checking programs. In *Proc. IEEE International Conference on Automated Software Engineering (ASE'00)*, IEEE Computer Society, 2000, pp. 3-11.
[11] X.Zhang, N.Gupta, and R.Gupta. Locating faults through automated predicate switching. In *Proc. International Conference on Software Engineering (ICSE'06)*, ACM, 2006, pp. 272-281.
[12] H.Cleve and A.Zeller. Locating causes of program failures. In *Proc. International Conference on Software Engineering (ICSE'05)*, ACM, 2005, pp.342-351
[13] G.Kob and F.Wotawa. Fundamentals of debugging using a resolution calculus. In *Proc. International Conference on Fundamental Approaches to Software Engineering (FASE'06), LNCS 3922*, Springer, 2006, pp.278-292.
[14] C.Liu, L.Fei, X.Yan, J.Han, and S.P.Midkiff. Statistical debugging: A hypothesis testing-based approach. *IEEE Transactions on Software Engineering*. 32(10): 831-848, 2006.
[15] G.Rothermel and M.J.Harrold. Empirical studies of a safe regression test selection technique. *IEEE Transactions on Software Engineering*. 1998, 24:401-419.
[16] http://fmv.jku.at/picosat/
[17] Manu Jose, Rupak Majumdar. Cause Clue Clauses- Error Localization Using Maximum Satisfiablity, In *Proc. ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI'11)*, ACM Press, 2011, pp.437-446.
[18] Wolfgang Mayer. Static and Hybrid Analysis in Model-based Debugging. PhD thesis, University of South Australia, 2007.
[19] R.Ceballos, R.M.Gasca, C.Del Valle, and D.Borrego. Diagnosing Errors in DbC Programs Using Constraint Programming. In *Proc. 11th Conference of the Spanish Association for Artificial Intelligence (CAEPIA'05) LNCS 4177*, Springer, 2006, pp.200-210.
[20] Fei Pu, Yan Zhang. Localizing Program Errors via Slicing and Reasoning, In *Proc. 11th IEEE Symposium on High Assurance Systems Engineering(HASE 2008)*, IEEE Computer Society, 2008, pp.187-196.
[21] Lei Zhao, Lina Wang, Xiaodan Yin. Context-Aware Fault Localization via Control Flow Analysis, *Journal of Software*, 6(10):1977-1984, 2011.

Fei Pu is currently an associate professor at Department of Software Engineering, School of Computer and Information Engineering, Zhejiang Gongshang University. He received the M.S. degree in Computational Mathematics from Hunan University in 1997, and the Ph.D. degree in Computer Science from Academy of Mathematics and System Science, Chinese Academy of Sciences in 2004. He was a post-doctor at Institute of Software, Chinese Academy of Science during 2004-2006 and a research fellow at University of Western Sydney, Australia from 2007 to 2009. He has published over 20 papers on formal methods and software engineering. His current research interests include formal methods, software verification and artificial intelligence.