Automated Debugging using Path-Based Weakest Preconditions

Haifeng He and Neelam Gupta

hehf@cs.arizona.edu ngupta@cs.arizona.edu
Dept. of Computer Science, The University of Arizona, Tucson, Arizona 85721

Abstract. Software debugging is the activity of locating and correcting erroneous statements in programs. Automated tools to locate and correct the erroneous statements in a program can significantly reduce the cost of software development. In this paper, we present a new approach to locate and correct an erroneous statement in a function. We assume the correct specification of the erroneous function is available in the form of preconditions and postconditions of the function. Our approach combines ideas from software testing and weakest preconditions used in correctness proof methods to locate a likely erroneous statement. We have implemented our approach and conducted experiments with several small programs. In our experiments, our approach was able to locate the erroneous statements in a large number of cases. Our preliminary experimental results show that our approach has potential for development of an automated bug location and correction tool.
Keywords: Fault location, software testing, weakest precondition, postcondition.

1 Introduction

Software debugging is the process of locating and correcting erroneous statements in a faulty program. It is an expensive and challenging activity requiring understanding of the program and is often done manually by the programmers. Automated tools that help the programmers in locating the erroneous statements can signifi cantly reduce the cost of software development.

The program slicing based approaches [1, 11, 15] extract a subset of program statements that can effect the values of variables at the point where a fault is manifested. A novel approach to automatically isolate cause-effect chains, that have higher precision than static or dynamic slices, has been developed in [19, 9]. Approaches based on dynamic invariant detection [8, 16] to give warnings about program anomalies have also been developed. All these approaches assist the programmers by narrowing down the search for erroneous statements to a subset of program statements. However, they do not generate the exact modifi cations to be made to the program to automatically correct the errors. To determine the exact nature of an error and check whether it lies in the localized program statements, the programmers have to modify the program and re-execute the program until they obtain correct output.

In this paper, we develop an approach to automatically *localize and correct* an erroneous statement in a faulty function. Our approach assumes that the precondition and the postcondition of the function are available as fi rst order theory formulas (FOT) [6] formulas over a fi nite domain. We also assume that a test suite for a given test adequacy criteria for structural testing of the given function is available. In our current work, we also assume that there is at most one error statement in the faulty function to avoid interaction among multiple errors. Our method takes as input an error trace generated by executing some failed test case in the test suite of the function. The notion of weakest precondition [5, 7] of a statement in a program for a given postcondition of the program has been used for proving program correctness. In this paper, we defi ne a notion of *path-based weakest precondition* for statements along a path in a program. Using this, we also defi ne the notions of a *hypothesized program state* and an *actual program state* at every point along the error trace. Our algorithm traverses the statements along the trace in reverse order of execution and compares these states at each point along the trace to detect an *evidence* for a likely faulty statement. It then generates modifi cations to the function to remove this evidence. The algorithm terminates if a modifi ed function successfully executes all the test cases in the test suite. If all the statements along the current error trace have been processed and the algorithm fails to correct the error, another error trace (corresponding to another failed test case in the test suite) is tried until all the error traces have been attempted.

We have implemented our algorithm and conducted experiments with several small programs by introducing one error at a time. In our experiments, our technique was able to correct errors such as a wrong relational operator used in a branch predicate, wrong variable used in a branch predicate, wrong variable used in an assignment statement, incorrect constant used in an assignment statement and some cases of incorrect number of loop iterations. Our approach requires normal *termination* of the program execution for the input that generated the error trace so that the postcondition can be evaluated for this input. Therefore, it is not able to correct errors that result in a non-terminating loop or result in segmentation faults such as due to illegal memory access.

The organization of this paper is as follows. The terminology used and the details of our approach are explained in section 2. The steps of our algorithm are described in section 3. The experiments are presented in section 4, the related work is discussed in section 5 and conclusions are mentioned in section 6.

2 Our Approach

The problem addressed in this paper can be stated as follows:

Problem statement: *Given a faulty function F with a single error statement, a test suite T_s for F, an error trace of F generated by T_s, and the precondition and postcondition of F in the first order predicate logic, localize and modify a statement in the error trace so that the modified function is able to pass the test cases in T_s.*

First, we execute F with T_s and identify the set of error traces i.e., the set of execution traces for which the postcondition of F evaluates to *False*. We select any one of these error traces for locating and correcting the error in the function. We also express the postcondition in disjunctive normal form using the input for the trace. Having the postcondition in disjunctive normal form, we only need to guarantee the validity of one conjunction in the postcondition to satisfy the postcondition. We select any one of the conjunctions in the evaluated postcondition for the error trace for locating and correcting the error in the error trace and call it *postcondition conjunction* R. If the algorithm is unable to fi x the error for given trace with the selected postcondition conjunction, another conjunction in the postcondition conjunction is selected. The steps of our algorithm are shown in Figure 2. Next, we describe our representation of an error trace. We illustrate our approach with a faulty program *Max* to compute a maximum element in an unsorted array of integers, shown in Figure 1(a). In this program, incorrect relational operator is used in line 4. An error trace with the input a[]= (-2, 5, 3), n=3 is shown in Figure 1(b).

```
int Max(int a[], int n){
int i, s;
precondition n>0
1:   i = 1;
2:   s = a[0];
3:   while (i < n) {
4:      if (s ≥ a[i])
5:         s = a[i];
6:      i = i + 1;
     }
     return s;
     postcondition
     ∀i:0≤i<n, s≥a[i] ∧
     ∃i:0≤i<n, s=a[i]
}
```

precondition (n>0)

$<i, line\#>$	$Stmt_i$
< 1,1>	`i=1`
<2,2>	`s=a[0]`
<3,3>	`(i-n<0)`
<4,4 >	`(s-a[i=1]<0)`
<5,6>	`i=i+1`
<6,3>	`(i-n<0)`
<7,4>	`(s-a[i=2]<0)`
<8,6>	`i=i+1`
<9,3>	`(i-n≥0)`

postcondition

$$\mathbf{(}(\text{s-a[0]} \geq 0)^T \wedge (\text{s-a[1]} \geq 0)^F \wedge (\text{s-a[2]} \geq 0)^F \wedge (\text{s-a[0]=0}^T)\mathbf{)} \quad \vee$$
$$\mathbf{(}(\text{s-a[0]} \geq 0)^T \wedge (\text{s-a[1]} \geq 0)^F \wedge (\text{s-a[2]} \geq 0)^F \wedge (\text{s-a[1]=0}^F)\mathbf{)} \quad \vee$$
$$\mathbf{(}(\text{s-a[0]} \geq 0)^T \wedge (\text{s-a[1]} \geq 0)^F \wedge (\text{s-a[2]} \geq 0)^F \wedge (\text{s-a[2]=0})^F\mathbf{)}$$

Fig. 1. (a) A Program `Max` 1.(b) A trace of `Max` with the input a[]=(-2, 5, 3), n=3.

2.1 Representation of an error trace

An error trace is the execution history of a failed test case. We defi ne it as a sequence of executed instances of statements (assignments, branch predicates, input/output statements) and evaluated precondition and postcondition of the function. We defi ne an *execution point* i, ($i = 1, n$) in a trace as the entry of the i^{th} statement instance executed in the above sequence. We use *bottom* to denote the exit point of the last (n^{th}) statement in the trace.

We use a tuple $< i, j >$ to indicate an instance of an executed statement in an error trace, where i is the execution point of the statement instance in the error trace and j is the line number of the statement in the program. For simplicity, we assume that there is only one statement at a line in the program. Since the execution point of a statement instance is unique in a trace, we denote a statement instance at $< i, j >$ as $Stmt_i$. An error trace generated by executing the function in Figure 1(a) with the input a[]=$(-2, 5, 3)$, n=3 is shown in Figure 1(b).

Representation of branch predicates. We define an *atomic predicate* as having the form $(expr\ relop\ const)$, where $expr$ is an arithmetic expression without a constant term, $relop$ is a relational operator $(<, \leq, >, \geq, =, \neq)$ and $const$ is a constant term. At present, we have considered the branch predicates that use only real, integer and character data types. The branch predicates along the trace are represented in the above form. For example, the `while` predicate `(i<n)` at line 3 in Figure 1(a) is formalized as $Stmt_3$:`(i-n<0)` in the error trace in Figure 1(b). The compound predicates are represented in *disjunctive normal form* i.e., one that has the form $E = e_0 \vee \cdots \vee e_n$, where each e_i has the form $g_0 \wedge \cdots \wedge g_m$ and each predicate g_i is an atomic predicate represented in the above standard form.

During program execution, the values of the array indices are known. We denote an array element in the trace as `array[idx = const]`, where `array` is the name of the array, `idx` is the expression for the index of the array element in the program, and `const` is the value of `idx` for the input used for the trace. In addition, if a branch predicate evaluates to true, then it is shown in the trace as it is; otherwise, its negation, which must be true, is shown in the trace. For example, let us consider the control statement `if (s` $\geq$ `a[i])` at line 4 in the example program in Figure 1(a). This statement is executed at position 7 in the trace in Figure 1(b). At that point, the value of `(s` $\geq$ `a[i])` is *False*. Thus, its negation is shown in the trace in Figure 1(b). The corresponding statement instance in the trace is $Stmt_7$: `(s-a[i=1]<0)`. Next, we describe our representation of precondition and postcondition of a program.

Representation of precondition and postcondition. Using the program input for the error trace, we transform the precondition and postcondition of the program into the disjunctive normal form. The transformation is done in two steps. First, during the execution, the precondition and postcondition are transformed into *quantifier-free* predicates. Using the program input for the error trace, the universal quantifier $\forall$ is expanded as a conjunction and the existential quantifier $\exists$ is replaced with a disjunction. For example, for $n = 3$, the quantifier $\forall\ i:\ 0 \leq i < n, s \geq a[i]$ in the postcondition of the program in Figure 1(a) is expanded as $(s \geq a[0]) \wedge (s \geq a[1]) \wedge (s \geq a[2])$. An existential quantifier is expanded as a disjunction. For example, for $n = 3$, the quantifier $\exists\ i:\ 0 \leq i < n, s = a[i]$ in the postcondition of the program in Figure 1(a) is expanded as $(s = a[0]) \vee (s = a[1]) \vee (s = a[2])$. The second step is to convert *quantifier-free* precondition or postcondition into the disjunctive normal. For the trace in Figure 1(b), we obtain the postcondition in the disjunctive normal from as below.

$$\begin{aligned}
&((s-a[0] \geq 0) \wedge (s-a[1] \geq 0) \wedge (s-a[2] \geq 0) \wedge (s-a[0] = 0)) \vee \\
&((s-a[0] \geq 0) \wedge (s-a[1] \geq 0) \wedge (s-a[2] \geq 0) \wedge (s-a[1] = 0)) \vee \\
&((s-a[0] \geq 0) \wedge (s-a[1] \geq 0) \wedge (s-a[2] \geq 0) \wedge (s-a[2] = 0))
\end{aligned}$$

We classify a predicate that evaluates to true with the given input as a *positive predicate* and a predicate that evaluates to False with the given input as a *negative predicate*. We use a superscript on each atomic predicate in the postcondition to show the truth value of that predicate for the given input. For example, in the trace in Figure 1(a) $(s - a[0] \geq 0)^T$ means that this predicate evaluates to true and $(s - a[1] \geq 0)^F$ means that it evaluates to False. Note that all the branch predicates in the trace after their representation in the standard from are positive predicates.

2.2 Weakest precondition and Path-oriented weakest precondition

Given a program S and the postcondition R, the **weakest precondition wp(S, R)** represents the set of all states such that the execution of S begun in any of them is guaranteed to terminate in a state satisfying R [5, 7].
In this paper, we define a weakest precondition semantics with respect to a trace, which we call as **path-based weakest precondition**, or **pwp** for abbreviation.

Definition: Given an execution trace T and the postcondition R of a function F, the **path-based weakest precondition** denoted as **pwp(T, R)** is the set of all states such that an execution of F, that follows T, begun in any of them is guaranteed to terminate in a state satisfying R.
The control flow in a trace is fixed so only the data dependences affect the value of output. Assume that evaluation of control statements does not have any side effects, we formally define **pwp** as below.

pwp$(x = a, R) = R_{x \to a}$, where $x \to a$ means substituting every occurrence of x in R with a
pwp$(B, R) = R$, where B is a branch predicate
pwp$(C; D, R) =$ **pwp**$(C,$ **pwp**$(D, R))$

Given a subtrace $T_{<i,n>}$ of T (from execution point i to the end of trace) and a postcondition R, we denote **pwp($T_{<i,n>}, R$)** as R_i. For example, let us consider the trace in Figure 1(b) and let assume that the postcondition conjunction R be
$((s - a[0] \geq 0) \wedge (s - a[1] \geq 0) \wedge (s - a[2] \geq 0) \wedge (s - a[0] = 0))$. The path-based weakest precondition at different execution points along the trace is:
R_3 = R_4 = R_5 = R_6 = R_7 = R_8 = R_9 = R_{bottom} = R
=$((s - a[0] \geq 0) \wedge (s - a[1] \geq 0) \wedge (s - a[2] \geq 0) \wedge (s - a[0] = 0))$ $(s - a[1] \geq 0)$.
R_1 = R_2 = **pwp**(s=a[0], R_3) = $((0 \geq 0) \wedge (a[0] - a[1] \geq 0) \wedge (a[0] - a[2] \geq 0) \wedge (0 = 0))$.

As seen in this example, in order to compute R_i at each point in a trace, we only need to know the set of the assignment statements that are needed for computation of R_i.

Definition: Given a trace T and the postcondition R of a function F, the **pwpSlice** $S(T, R, i)$ is an ordered set of assignment statements from point i to the end of T, upon which the value of R is directly or indirectly data dependent.
In other words, the **pwpSlice** $S(T, R, i)$ consists of all the assignment statements that are needed for computation of R_i. In the above example, $S(T, R, 1)$ is $\{Stmt_2\}$. At each execution point i on an error trace, we compare the atomic predicates in the predicate representing the set of *hypothesized program states* and the predicate representing the set of *actual program states* to look for an evidence for locating the error in the trace.

2.3 Hypothesized program state

The set of *hypothesized program states* at an execution point along the trace is represented by a predicate in disjunctive normal form derived from the postcondition as explained below.

Definition: Given a trace T and a postcondition R of a function F, the set of **hypothesized program states** at an execution point i along the trace is defined as the path-based weakest precondition R_i = **pwp(**$T_{<i,n>}, R$**)**.

The set of hypothesized program states R_i at any execution point i, (i=1, n) along the trace is computed as R_i = **pwp(**$Stmt_i, R_{i+1}$**)** for i=1, n-1 and R_n = **pwp(**$Stmt_n, R$**)**.

2.4 Actual program state

The set of *actual program states* at an execution point along a trace is represented by a predicate in disjunctive normal form that is actually true for the given input. It consists of a set of *forward program states*, Q_i^F, and a set of *backward program states*, Q_i^B. The set of forward program states Q_i^F at an execution point i along a trace T is defined as:

Q_1^F = positive conjunctions in precondition.
Q_i^F = (Q_{i-1}^F - $Kill_{i-1}$) ∪ Gen_{i-1}, i=1,n
Q_{bottom}^F = (Q_n^F - $Kill_n$) ∪ Gen_n, i=1,n

where $Kill_{i-1}$ is the set of predicates killed by statement instance $Stmt_{i-1}$ and Gen_{i-1} is the set of predicates derived from $Stmt_{i-1}$. A predicate p is killed by $Stmt_{i-1}$ if there is a variable in p that is defined at $Stmt_{i-1}$. For example, `(i-n<0)` is killed by statement `i=i+1`. Since i is redefined, after `i=i+1` is executed, `(i-n<0)` may not hold. If $Stmt_{i-1}$ is an assignment statement, then an equivalence is derived from $Stmt_{i-1}$. If $Stmt_{i-1}$ is a branch predicate, then Gen_{i-1} is the set of predicates in $Stmt_i$. The computation of the set of forward program states Q_4^F for the error trace in Figure 1(b) is shown below.

Q_1^F = (n>0)
Q_2^F = (n>0)∧(i=1)
Q_3^F = (n>0)∧(i=1)∧(s=a[0])
Q_4^F = (n>0)∧(i=1)∧(s=a[0])∧(i-n<0)

Given an execution point i, the set of backward program states at i are defined as:

Q_i^B = **pwp**($Stmt_i, Q_{i+1}^B$), if $Stmt_i$ is an assignment statement
Q_i^B = $Stmt_i \wedge Q_{i+1}^B$, if $Stmt_i$ is a branch predicate
Q_{bottom}^B = { }

We illustrate the computation of the set of backward program states for the error trace in Figure 1(b).

Q_1^B = (3-n≥0)∧(a[0]-a[2=2]≤0)∧(2-n<0)∧(a[0]-a[1=1]≤0)∧(1-n<0)
Q_2^B = (i+2-n≥0)∧(a[0]-a[i+1=2]≤0)∧(i+1-n<0)∧(a[0]-a[i=1]≤0)∧(i-n<0)
Q_3^B = (i+2-n≥0)∧(s-a[i+1=2]≤0)∧(i+1-n<0)∧(s-a[i=1]≤0)∧(i-n<0)
Q_4^B = (i+2-n≥0)∧(s-a[i+1=2]≤0)∧(i+1-n<0)∧(s-a[i=1]≤0
Q_5^B = (i+2-n≥0)∧(s-a[i+1=2]≤0)∧(i+1-n<0)
Q_6^B = (i+1-n≥0)∧(s-a[i=2]≤0)∧(i-n<0)
Q_7^B = (i+1-n≥0)∧(s-a[i=2]≤0)
Q_8^B = (i+1-n≥0)
Q_9^B = (i-n≥0)
Q_{bottom}^B = { }

Finally, we define the set of actual program states Q_i as $Q_i = Q_i^F \wedge Q_i^B$.

2.5 Detection of evidence

A predicate A is less restrictive than predicate B if there is some state in A, which is not contained in B, or in other words, $A(x) \Rightarrow B(x)$ is False. An *evidence* at an execution point i indicates that the predicate Q_i representing the set of actual program states is less restrictive than the predicate R_i representing the set of hypothesized program states. We defi ne two types of evidences *explicit* and *implicit*.

Explicit Evidence. An explicit evidence shows that the set of actual program states represented by Q_i and the set of hypothesized program states represented by R_i are disjoint and thus $Q_i \Rightarrow R_i$ is False; or in other words, Q_i is *n*ot stronger than R_i at this program point. Currently, we consider two special cases to detect that the set of states in Q_i and R_i are disjoint. We refer to them as explicit evidence of Type I and explicit evidence of Type II.

Definition: If at an execution point i along a trace, a negative atomic predicate of the form $r : 0\ relop\ const$, i.e., without any variables, appears in the predicate R_i representing the set of hypothesized states, then r constitutes an **explicit evidence $E_{explicit}$(any, r, i) of Type I**.
Let r be a formalized negative predicate in R_i that has the form $(0\ relop\ const)$, i.e., there is no variable involved in the predicate r. For example, `(0 > 2)` is such a *False* predicate. Since Q_i evaluates to *True* and *False* is the strongest predicate, it is obvious that Q_i is less restrictive than r.

Definition: At an execution point i in a trace, let $q : expr_q\ relop\ const_1$ be an atomic predicate in Q_i and $r : expr_r\ relop\ const_2$ be a negative atomic predicate in R_i. Then, q and r form an **explicit evidence** $E_{explicit}(q, r, i)$ **of Type II** iff $expr_q$=$expr_r$.

Since q evaluates to true and r evaluates to False in the given trace, if $expr_q$ = $expr_r$, then for this trace q exercises a state not contained in the set of states represented by r. The symbolic difference between q and r provides us a clue to what modifi cation should be done to the program so as to remove this evidence of $q \Rightarrow r$ being False. For example, for the error trace in Figure 1(b), predicate Q_7^B representing the set of actual program states contains a predicate `(s-a[i=2]`$\leq$ `0)` and the predicate R_7 representing the set of hypothesized program states contains another predicate `(s-a[2]`$\geq$ `0)`. These two predicates form an explicit evidence $E_{explicit}$`((s-a[i=2]`$\leq$ `0), (s-a[2]`$\geq$ `0),` 7). Note that `a[i=2]` and `a[2]` refer to the same variable.

Implicit Evidence. An implicit evidence $E_{implicit}(r)$ is indicated by a negative predicate r in R_1 that is not present in an explicit evidence. For each implicit evidence $E_{implicit}(r)$ in R_1, we consider that the trace is lacking a constraint on r. For example, let consider the R_1 for the postcondition conjunction:

R= $((s - a[0] \geq 0)^T \wedge (s - a[1] \geq 0)^F \wedge (s - a[2] \geq 0)^F \wedge (s - a[0] = 0)^T)$.
The corresponding R_1 = $((a[0] - a[1] \geq 0)^F \wedge (a[0] - a[2] \geq 0)^F)$. And,

Q_1 = (3-n$\geq$0)$\wedge$(a[0]-a[2=2]$\leq$0)$\wedge$(2-n$<$0)$\wedge$(a[0]-a[1=1]$\leq$0)$\wedge$(1-n$<$0)$\wedge$(n$>$0).
However, in this example both the negative predicates in R_1 have corresponding predicates in Q_1 that form explicit evidence of Type II. Therefore, there is no implicit evidence at the top of the trace in this example.

2.6 Location of a likely erroneous statement and generation of modification

After an evidence of the predicate Q_i being less restrictive than R_i is detected at an execution point i, the goal is to locate a statement instance at some point j in the trace such that a modifi cation to $Stmt_j$ will remove the evidence at i. From the predicates involved in an evidence, we determine a *problem predicate* and a *correcting predicate*. These two atomic predicates are treated as character strings and the symbolic differences between these two strings are computed. We then use these differences to generate a modifi cation to a statement along the trace so that the detected evidence is removed from the trace. The modifi ed function is tested with the given test suite to check if the error is removed. Otherwise, if possible another modifi cation to remove the evidence is generated. If the evidence cannot be removed by all attempted modifi cations at the execution point i, the algorithm moves on to process next statement in the trace. Note that our algorithm

```
Input: An error trace $T$, postcondition conjunction $R$ and test suite $T_s$
Output: $(succ/fail, mod)$, where mod is a modified statement in the program.
procedure $AutoDebug\ (T, R, T_s)$
            for each execution position $i \in T$ from $i = n$ to 1 do
step1:         Compute set of actual program states $Q_i^*$ and set of hypothesized program states $R_i^*$.
               for each negative predicate $r \in R_i^*$ do
step2:                if an $E_{explicit}(any, r, i)$ detected then    Type I Explicit Evidence
                              Generate and test modifications that change the form of $r$ at $i$.
                              if testing successful then $return(succ, mod)$ endif
step3:                elseif  an $E_{explicit}(q, r, i)$ detected then    Type II Explicit Evidence
                              Generate and test modifications that either change the form of $q$ at $i$
                                   or change the form of $r$ at $i$ to remove the evidence.
                              if testing successful then $return(succ, mod)$ endif
                       endif
                endfor
             endfor
step4:       for each negative predicate $r \in R_1^*$ not present in any explicit evidence do
                  Consider Implicit Evidence $E_{implicit}(r)$
                  if $E_{implicit}(r)$ indicates a missing loop iteration(s) then
                       Generate modification to add missing loop iteration to actual program state.
                       Test Modified Program.
                       if testing successful then $return(succ = 1, mod)$ endif
                  else
                      Generate and test modifications that either change form of $r$ at the
                           top of the trace or change the form of a predicate $q$ in $Q_1^{B*}$.
                      if testing successful then $return(succ = 1, mod)$ endif
                  endif
             endfor
             return $fail$
endprocedure
```

Fig. 2. The AutoDebug algorithm

will terminate successfully if no error trace is generated when the modifi ed function is executed with the given test suite. However, that does not necessarily mean that the modified program is correct. All it means is that the original function was not able to pass all the test cases in the given test suite whereas the modified function is able to pass all the test cases in the test suite. The correctness of the solution is clearly dependent upon how thoroughly the test suite tests the program. It may also be helpful to take input about whether the modification generated by our algorithm will be acceptable to the developer.

3 Description of the Algorithm

In this section, we discuss the steps of our algorithm shown in Figure 2 for automatically locating and correcting an erroneous statement in a function.

Step 1: Compute the predicates representing the sets of actual and hypothesized program states. At the entry of each instance of an executed statement $Stmt_i$ in the given error trace T, we compute the predicate Q_i representing the set of actual program states and the predicate R_i representing the set of hypothesized program states. We apply two rules of inference: *transitivity* and *equality* to deduce new predicates from other predicates in each of the program states. Deduced predicates are added to respective set of program states until no new atomic predicates can be deduced. In the remaining paper, we use Q_i^* and R_i^* to represent the extended sets of Q_i and R_i respectively after including deduced predicates.

Step 2: Detect and fix explicit evidence of Type I. In this type of evidence, there is a *negative* predicate r that does not contain any variables and is present in the predicate R_i^* representing the set of hypothesized program states. For example, let a predicate is `(10=0)` be present in R_i^* at an execution point i on an error trace. Then, it forms an explicit evidence $E_{explicit}(any$, (10=0), i) of Type I.

Generate Modification. The next step is to generate modifications for the statements that would remove the above evidence by changing the form of r at execution point i where evidence is detected. We change the form of r by matching r to a predicate which is implied by the actual program state so that the atomic predicate in the postcondition R corresponding to r will be satisfied if the same trace is followed. We consider the following two approaches to change the form of r at i.

First, if the relational operator in r is `=`, then we match r to the positive predicate $True$. For relational operator `=`, we define the $True$ predicate to `0=0`. Note that the form of r can be changed only by an assignment statement between execution point i and the end of trace. It is obvious that modifying an assignment statement in the pwp-Slice of r can change the form of r. However, modifying the LHS of an assignment not in the pwpSlice of r can also change the form of r at execution point i. Therefore, we consider each assignment statement between the point i and the $bottom$ of the trace as a possible candidate for modification. Let $Stmt_k$ be the next assignment statement to be considered and let the predicate in R_k^* corresponding to r be r_k. The goal of transforming r to `0=0` can be attained by making $r_k = (0 = 0)$, i.e., **pwp**$(Stmt_k, r_{k+1})$=$(0 = 0)$. It is obvious that if $expr = lhs - rhs$, then **pwp**$(lhs = rhs, (expr = 0))$ will be

$(0 = 0)$. Therefore, we consider r_{k+1} as the correcting predicate c and the equivalence derived from $Stmt_k$ as the problem predicate e.

We consider e and c as a set of strings of characters and compare them to compute the difference d_e between e and c and the difference d_c between c and e. In our current work, we assume that the error is either on LHS or on RHS of an assignment statement but not on both sides of the assignment statement. If d_e appears on RHS of the assignment statement $Stmt_k$, the modification is generated to replace d_e in $Stmt_k$ by d_c. If d_e appears on LHS of $Stmt_k$, the modification is to replace d_e in $Stmt_k$ by d_c only if d_c is a single variable.

If the evidence cannot be removed by the above modifications, or if the relational operator in r is not =, we then try our second approach to generate modifications explained below. We generate additional modifications by matching r to each predicate q in the predicate Q_i^* with same relational operator as that of r. Note that the predicates in Q_i^* are all positive predicates in the trace, so they are consistent with each other. By matching r to a predicate in Q_i^* other than q, r becomes consistent with q. Thus, in this case r is the problem predicate e and a predicate q in Q_i^* is used as a correcting predicate c. As before, we compute d_e and d_c. If modifications generated at execution point i are not successful in removing the evidence, as before we propagate this matching to execution points $k > i$ so that the effect of matching at the execution point k is to remove the above evidence at the execution point i. However, there is a difference. Now a matching at k can be performed only if the predicate q_k corresponding to q is present in Q_k^* i.e., it is not killed by some assignment between execution points i and k. In addition, in order to make sure that the modification to an assignment $Stmt_k$ at execution point k will change the form of r at execution point i, we need to check for the following. If the modification is for RHS of an assignment statement $Stmt_k$, then $Stmt_k$ must belong to the pwpSlice of r. However, if the modification is for the LHS of the assignment statement, we need to make sure that after modification, the assignment will appear in the pwpSlice of r.

Test Modification. Each of the modification generated above is applied to the original program. The modified program is then executed for all the test cases in the given test suite. If the modified program passes all the test cases in the test suite then we consider that the error has been corrected. Each of the above modification is tested until a version of the program passes the test suite. If all the above modifications have been tried and the fault is not fixed, the algorithm moves onto to detect next evidence.

Step 3: Detect and fix explicit evidence of Type II. In this step we detect and fix an explicit evidence, in which a predicate q in Q_i^{B*} and a negative predicate r in R_i^* have the same expression on LHS. This evidence also shows that the set of states in Q_i^* are disjoint from the set of states in R_i^*. To illustrate this, let us assume that Q_i^{F*}, Q_i^{B*} and R_i^* at an execution point i on an error trace are given as below.

Q_i^{F*}:(n>0)∧(i=0)∧(s=0)∧(i-n<0)
Q_i^{B*}:(i-n+1≥0)∧(s-a[i=0]≤0)
R_i^*:(s-a[0]≥0)^F∧ (s-a[0]=0)^F

Two explicit evidences of Type II are detected at execution point i. They are

E_1 = $E_{explicit}$((s-a[i=0]$\leq$0), (s-a[0]$\geq$0), i)
E_2 = $E_{explicit}$((s-a[i=0]$\leq$0), (s-a[0]=0), i)

For an evidence $E_{explicit}(q, r, i)$ of Type II, either q or r could be in error. Therefore, the modifications for changing the form of q to r at i or changing form of r to q at i are generated. The modifications for changing the form of r are generated in the same manner as described for explicit evidence of Type I. To change the form of q to match to r, we can either change the original branch predicate from which q may be derived, or we can change an assignment statement on the trace. Note that a modification to an assignment statement cannot change the relational operator of q. Therefore, if the relational operators of q and r are different, we directly modify the branch predicate from which r may be derived.

Step 4: Detect and fix implicit evidence. Implicit evidences are detected at the top of the error trace. For each negative atomic predicate r in R_1^* that is not present in any explicit evidence, we form an implicit evidence as $E_{implicit}(r)$. Having an implicit evidence $E_{implicit}(r)$, we check whether the cause for the evidence is because some loop iterations are missing from the trace. If there is a loop in the trace, which contributes some constraints on R_1^*, and the missed constraints have similarity with the constraints added by the loop, then our algorithm attempts to derive the possible missing iterations in the loop and generates modification to a statement that would add those iterations into the trace. This modification is then verified by executing the modified program with the test suite.

If the implicit evidence is not the case of a missing loop iteration(s), the algorithm attempts to remove this evidence from R_1 fix the fault by generating modifications as in Steps 2 and 3. Given an implicit evidence of $E_{implicit}(r)$, modifications to the statements along the trace are generated by matching the negative predicate r to atomic predicates q in Q_1^* and vice versa. As in steps 2 and 3, modifications to the assignment statements in the pwp slice of r are also generated by matching them to the component corresponding to r in the hypothesized state at their exit. As before, each modification is tested by executing the modified program.

4 Experiments

We have implemented our technique using C and Python languages. The autodebug algorithm was implemented in C. Postconditions, preconditions and predicate deduction was implemented using Python. The faulty program is expected to be written in a subset of C using real, integer and character variables, arrays, conditionals and loop control constructs. At present, we do not handle faulty programs using pointers. To handle function calls, we assume that the postconditions and preconditions of the called function are given. We also assume that either the called function does not have errors, or the trace of statements through the called function is available. The faulty program is instrumented to generate execution traces in the format described in the paper. We used the following five programs in our experiments.

Sum: It computes the sum of all integers in an array `a[]`. This problem has simple control structures. The postcondition of this program is a single universal quantifi er which is expanded as conjunctions during the execution.
Max: This program (in section 2) searches the maximum element in an unsorted array of integers.
Binary search: It does binary search on a sorted integer array. Its source code, including the preconditions and postconditions was taken from [7].
Array copy: This example is a simple program to copy the contents of an array to another array.
Quicksort: This program, taken from [2], is for Quicksort algorithm on an integer array. The original code does not have preconditions and postcondition so we derived them ourselves.

We introduced an error in a statement at a time into these programs. The types of errors introduced include wrong relational operator used in a branch predicate, wrong variable used in a branch predicate, wrong variable used in an assignment statement, incorrect constant used in an assignment statement, etc. We conducted experiments with our algorithm for locating and correcting the erroneous statements. We also experimented with computing program dices [1] for these faulty programs. We tried to limit the modifi cation only to the statements in the computed dice. If the algorithm is not able to correct the program by modifi cation of statements in the dice, then we ran the algorithm without using dice. The heuristic used in [1] for picking a dice is to fi rst form all possible dices and then randomly choose one of them. Since there is no conclusion about which heuristic is better, we used more conservative method. We chose the dice with the largest number of statements so that it most likely will not miss an erroneous statement.

4.1 Results

We show the results of our experiments in Table 1. The column labeled Line No. shows the line number of the statement in the function in which the error was introduced. The column labeled Orig. Stmt. shows the original statement in the correct program. The column labeled Faulty Stmt. shows the statement after error was introduced. The column labeled Fault Type shows the type of fault introduced such as wrong constant, wrong operator, missing variable etc. The last column shows the output obtained from our implementation of AutoDebug algorithm.

It is interesting to note that in rows Sum/1 and Max/3 in Table 1, the correction generated by our algorithm was in a different statement than the one in which error was introduced. However, modifi cation in a statement different from the one in which error was introduced also corrected the problem. Some of the errors resulted in nonterminating loop and they were not corrected by our algorithm. Also, if an error resulted in a loop executing more iterations than required, our algorithm was not able to fi x it. Other than these, our algorithm was able to fi x most of the errors. Although, we had expected dices to signifi cantly improve the effi ciency of the technique, in our experiments we did not fi nd dices to be useful in many cases. Only for 3 errors among all the errors in Table 1, were the erroneous statements included in largest dice for the faulty program. We also tried to consider the union of statements in all dices. However, we did not find dices to be very useful in our examples. The reason was that in many cases, such as in branch predicate fault and variable initialization fault, the faulty statement was executed by all failed trace as well all correct traces. In some cases, no correct traces were generated and hence dices were not helpful. The programs used in our experiments were small and there may be benefits of incorporating dices in our technique for large programs.

5 Related Work

The program slicing based approaches [1, 11, 15] use *static* or *dynamic dependency analysis* to extract a subset of program statements that can effect the values of variables at the point where a fault is manifested in the program. A novel approach to automatically isolate cause-effect chains, based on the *difference* between the *program states* of a run corresponding to a failed and a successful run, has been recently developed [19, 9]. The cause-effect chains isolated by this approach have higher precision than static or dynamic slices. Approaches based on dynamic invariant detection that give programmers warnings that there are anomalies found in the program [8, 16] have also been developed. All these approaches assist the programmers by narrowing down the search for erroneous statements to a subset of program statements. However, they do not generate the exact modifications to be made to the program to automatically correct the errors. To determine the exact nature of the error and check whether it lies in the localized program statements, the programmers have to modify the program and re-execute the program until they obtain correct output. In contrast, our approach attempts to automatically locate the error statement and generate the correction to be applied to the erroneous statement. In our future work, we would further analyze the type of errors that can be detected by our approach and the types of errors in which other approaches can be more helpful.

6 Conclusions

In this paper, we have presented a new technique that combines ideas from formal analysis of programs and software testing to automatically locate and correct erroneous statements. Our technique is based on matching of character strings which is guided by removal of some symbolic evidences that make actual program state less restrictive than hypothesized program state at some execution point. Our preliminary experiments show that our approach is promising. In the current work, we have assumed that only one program statement is in error. In our future work, we plan to relax this restriction and evaluate our technique for large programs.

References

1. H. Agrawal, J. R. Horgan, S. London and W. E. Wong, "Fault localization using execution slices and dataflow tests", *Proceedings of the Sixth IEEE International Symposium on Software Reliability Engineering*, pages 143-151, Toulouse, France, October 1995.

Table 1. Results for *Sum*, *Max*, Binary Search(Bin), Array Copy(Arr) and Quicksort(QS) functions.

Program/	Line	Orig. Stmt.	Faulty Stmt.	Fault Type	Output
Error No.	No.				(Line No., Stmt.)
Sum/1	1	`i=0`	`i=1`	`const:wrong`	`(2, s=a[0])`
Sum/2	1	`i=0`	`i=2`	`const:wrong`	`(1, i=0)`
Sum/3	2	`s = 0`	`s = 1`	`const:wrong`	`(2, s=0)`
Sum/4	4	`s=s+a[i]`	`s = a[i]`	`var(s):missing`	`(4, s=s+a[i])`
Sum/5	4	`s=s+a[i]`	`s = i+a[i]`	`var(s):wrong`	`(4, s=s+a[i])`
Sum/6	4	`s=s+a[i]`	`s=s+a[0]`	`var(a):wrong`	`(4, s=s+a[i])`
Sum/7	4	`s=s+a[i]`	`s=s-a[i]`	`op:wrong`	`(4, s=s+a[i])`
Sum/8	5	`i=i+1`	`i=i+2`	`const:wrong`	`(5, i=i+1)`
Sum/9	5	`i=i+1`	`i=i`	`const:wrong`	No (infi nite loop)
Sum/10	3	`while(i<n)`	`while(i<n-1)`	`const:wrong`	`(3, while(i<n))`
Sum/11	3	`while(i<n)`	`while(i<n+1)`	`const:wrong`	No (extra loop)
Sum/12	3	`while(i<n)`	`while(i+n)`	`relop:wrong`	No (infi nite loop)
Sum/13	3	`while(i<n)`	`while(i>n)`	`relop:wrong`	No (loop not enter)
Max/1	2	`s=0`	`s=10`	`const:wrong`	`(2, s=a[0])`
Max/2	5	`s=a[i]`	`s=i`	`var(a):wrong`	`(5, s=a[i])`
Max/3	1	`i=0`	`i=1`	`const:wrong`	`(2, s=a[0])`
Max/4	6	`i=i+1`	`i=i-1`	`op:wrong`	No (system error)
Max/5	4	`if(s<a[i])`	`if(s>a[i])`	`relop:wrong`	`(4, if(s<a[i])`
Max/6	4	`if(s<a[i])`	`if(s<a[0])`	`var(a):wrong`	`(4, if(s<a[i]))`
Max/7	4	`if(s<a[i])`	`if(s>=a[i])`	`relop:wrong`	`(4, if(s<a[i]))`
Max/8	3	`while(i<n)`	`while(i<n-1)`	`branch:const`	`(3, while(i<n))`
Bin/1	1	`i=0`	`i=1`	`const:wrong`	`(2, i=0)`
Bin/2	2	`j=n+1`	`j=n`	`const:wrong`	`(2, j=n+1)`
Bin/3	4	`e=(i+j)/2`	`e=i+j`	`op:wrong`	No (infi nite loop)
Bin/4	6	`i = e`	`i = j`	`var(s):wrong`	No (infi nite loop)
Bin/5	6	`i = e`	`j = e`	`def(s):wrong`	`(6, i = e)`
Bin/6	4	`e=(i+j)/2`	`e=(i*j)/2`	`op:wrong`	No (infi nite loop)
Bin/7	3	`while(i+1!=j)`	`while(i+2<j)`	`const:wrong`	`(3, while(i+2<j+1))`
Bin/8	3	`while(i+1!=j)`	`while(i!=j)`	`relop:wrong`	No (infi nite loop)
Bin/9	5	`if(a[e]<=x)`	`if(a[e]<x)`	`relop:wrong`	`(5, if(a[e]<=x))`
Bin/10	5	`if(a[e]<=x)`	`if(a[e]>x)`	`relop:wrong`	`(5, if(a[e]<=x))`
Bin/11	5	`if(a[e]<=x)`	`if(a[0]<=x)`	`var(a):wrong`	`(5, if(a[e]<=x))`
Bin/12	5	`if(a[e]<=x)`	`if(a[i]<=x)`	`var(a):wrong`	`(5, if(a[e]<=x))`
Bin/13	5	`if(a[e]<=x)`	`if(i<x)`	`var(a):wrong`	`(5, if(a[e]<=x))`
Arr/1	3	`s1[i]=s2[i]`	`s1[i]=s2[i+1]`	`var(a):wrong`	`(3, s1[1]=s2[i])`
Arr/2	3	`s1[i]=s2[i]`	`s1[0]=s2[i]`	`var(a):wrong`	`(3, s1[i]=s2[i])`
Arr/3	3	`s1[i]=s2[i]`	`s1[i]=i`	`var(a):wrong`	`(3, s1[i]=s2[i])`
Arr/4	1	`i=0`	`i=1`	`assign:const`	`(1, i=0)`
Arr/5	4	`i=i+1`	`i=i-1`	`assign:arithm`	No (system error)
Arr/6	2	`while(i<=n)`	`while(i<n)`	`branch:relop`	`(2, while(i<n+1))`
Arr/7	2	`while(i<n)`	`while(i>n)`	`branch:relop`	No (loop not entered)
QS/1	3	`last=(left+right)/2`	`last=(left+right)*2`	`assign:arithm`	No (out of array boundary)
QS/2	4	`temp=a[left]`	`temp=a[0]`	`var(a):wrong`	`(4, temp=a[left])`
QS/3	5	`a[left]=a[last]`	`a[left]=temp`	`var(a):wrong`	`(5, a[left]=a[last])`
QS/4	5	`a[left]=a[last]`	`a[last]=a[last]`	`var(a):wrong`	`(5, a[left]=a[last])`
QS/5	8	`if(a[i]<a[left])`	`if(a[i]>=a[left])`	`relop:wrong`	`(8, if(a[i]<a[left]))`
QS/6	10	`if(a[i]<a[left])`	`if(a[i]<a[last])`	`var(a):wrong`	`(10, if(a[i]<a[left]))`
QS/7	11	`a[last]=a[i]`	`a[left]=a[i]`	`var(a):wrong`	`(11, a[last]=a[i])`
QS/8	13	`a[last]=a[i]`	`a[left]=a[i]`	`var(a):wrong`	`(13, a[last]=a[i])`
QS/9	16	`i=i+1`	`i=i`	`assign:arithm`	No (infi nite loop)
QS/10	18	`temp=a[left]`	`temp=a[last]`	`var(a):wrong`	`(18, temp=a[left])`

2. R. S. Boyer and J. S. Moore “A Computational Logic Handbook”, *Academic Press*, Boston.
3. L. A. Clarke and D. J. Richardson, “The application of error-sensitive testing strategies to debugging”, *Proceedings of the Symposium on High-Level Debugging*, pages 45-52, 1983.
4. R. A. DeMillo, H. Pan and E. H. Spafford, “Critical slicing for software fault localization”, *Proceedings of the International Symposium on Software Testing and Analysis*, pages 121–134, San Diego, CA, 1996.
5. E. W. Dijkstra, “A Discipline of Programming”, *Prentice Hall*, Englewood Cliffs, NJ, 1976.
6. C. Ghezzi, M. Jazayeri and D. Mandrioli, “Fundamentals of Software Engineering”, Second Edition, *Prentice Hall*, 2003.
7. D. Gries, “The Science of Programming”, *Springer-Verlag*, 1981.
8. S. Hangal and M. S. Lam, “Tracking down software bugs using automatic anomaly detection”, *Proceedings of the International Conference on Software Engineering*, May, 2002,
9. R. Hildebrandt and A. Zeller, “Simplifying failure-inducing input”, *Proceedings of the International Symposium on Software Testing and Analysis*, pages 135-145, 2000.
10. J. A. Jones, M. J. Harrold and J. Stasko, “Visualization of test information to assist fault localization”, *Proceedings of the 24th International Conference on Software Engineering*, Orlando, Florida, USA, May 2002, pp. 467-477.
11. B. Korel and J. Rilling, “Application of dynamic slicing in program debugging”, *Automated and Algorithmic Debugging*, pages 43-58, 1997.
12. K. R. M. Leino, J. B. Saxe and R. Stata, “Checking Java programs via guarded commands”, *Compaq SRC Technical Note # 1999-002,* Palo Alto, CA, 1999.
13. K. R. M. Leino, T. Millstein, and J. B. Saxe. “Generating error traces from verification-condition counterexamples”. *http://research.microsoft.com/ leino/papers.html*.
14. R. Lencevicius, “On-the-fly query-based debugging with examples”, *Automated and Algorithmic Debugging*, 2000.
15. J. R. Lyle and M. Weiser, “Automatic program bug location by program slicing”, *Proceedings of the 2nd International Conference on Computers and Applications*, pages 877-882, 1987.
16. J. W. Nimmer and M. D. Ernst. “Invariant inference for static checking”, *Proceedings of the ACM/SIGSOFT Symposium on the Foundations of Software Engineering*, pages 11-20, Charleston, SC, November 2002.
17. M. Renieris and S. Reiss, “Fault localization with nearest neighbor queries”, *Proceedings of the 18th IEEE International Conference on Automated Software Engineering*, October 2003.
18. E. Y. Shapiro, “Algorithmic Program Debugging”, *The MIT Press*, 1983.
19. A. Zeller, “Isolating cause-effect chains from computer programs”, *Proceedings of the ACM/SIGSOFT International Symposium on Foundations of Software Engineering*, 2002.