4

Verification Based Model Localizes Faults from Procedural Programs

Safeeullah Soomro
Department of Computer Science,
National University of Computer Sciences and Emerging Sciences
ssoomro@ist.tugraz.at, safeeullah.soomro@nu.edu.pk
NU-FAST, Karachi
Pakistan

1. Overview

Detecting and locating faults is one of the most important issues of current research community. Many efforts have been taken to improve software development and to prevent faults. But still software faults pose the most challenging problems to software engineers. Fault localization is a most challengeable task during the debugging process. Fault localization is the next step after detecting faults in programs. Use of fault localization in control engineering where engineers often employ the procedural programming paradigm. Often controls software is safety-critical and thus detection but also localization of bugs is uttermost important.

This chpater makes use of abstract dependences between program variables for locating and localizing faults in procedural programs. The so called verification-based Model (VBM) for debugging is an extension of dependence model from Jackson's Aspect System, which has been used for verification of C programs. The Aspect system analyzes the dependences between variables of a given program and compares with specified dependences. Otherwise, the program fulfills the specification. In case of mismatch the program is said to violate the specification. Unfortunately, the Aspect system does not allow locating the source of mismatch. The VBM extends Jackson's idea towards not only detecting behavior but also localizing malfunctioning real cause. The VBM performs fix-point computation for recursive invocation (in all cases where we obtain a cyclic call graph). We presented algorithm and proof for fix point computation which ensures that no dependences loss during iteration and we always reached fix-point after finite number of iterations. Furthermore we present novel results obtained from our most recent case studies. Notably, whenever our novel model detects a structural fault, it also appears to be capable of localizing the detected misbehavior's real cause.

Key Words
Model Based Reasoning, Software Verification, Software Debugging, Fault Detection and Localization.

2. Introduction

Software verification is an important phase of software development. In the last decade the software verification and the debugging communities have made considerable progress. In this chapter we focus on fault localization which is based on the abstract dependencies that are used by the Aspect system (Jackson 1995) for detecting faults. The verification-based model for debugging is an extension of the dependence model from Jackson's Aspect system (Jackson 1995) which has been used for dependency based verification of C programs. The Aspect system analysis the dependences between variables of a given program and compares them with the specified dependences. In case of a mismatch the program is said to violate the specification. Otherwise, the program fulfils the specification. Unfortunately, the Aspect system does not allow locating the source of a mismatch. In the following we extend Jackson's idea towards not only detecting misbehaviour but also localizing the malfunctioning real cause.

Although program slicing, as a lightweight technique, has seen successful application in fault localization (Agrawal et al., 1993, Fritzson et al., 1999, Lyle and M. Weiser 1987, its discrimination like MBSD (Wotawa 2002). In (Kuper 1989, Wieland 2001) the authors employ the notion of dependences for fault localization. In contrast to latter approach we do not employ detected difference in variable values at a certain line in code but use of differences between specified and computed dependencies and thus also incorporate the structural properties of program and specification. Thus, the models introduced in (Kuper 1989, Wieland 2001) can not deal with assertions or pre- and post conditions in a straightforward way.

In this chapter we focus on localizing faults in procedural programs and dealing with global variables. Procedural programs are generally more computationally efficient that object oriented programs, because there is less overhead to handle abstractions and the data structures more closely resemble the hardware that must manipulate them.

The chapter is organized as follows. In Section 3 we introduce our verification based model by using motivating example. The results and discussion given in Section 4 reveal the verification based model provides a useful means for detecting and localizing common errors for procedural programs and particular in contest of global variables. In Section 5 we present related research. Further more we resent limitation of our model in section 6. Finally we summarize the chapter.

3. Motivating Example

In this section we explain the basic idea of localizing the fault by checking whether the post condition is satisfying or not using the verification based model.

In this chapter we focus on fault localization which is based on abstract dependencies that are used by the Aspect system (Jackson 1995) for detecting faults. Abstract dependencies are relations between variables of a program. We say that a variable x depends on a variable $y$iff

a new value for *y* may causes a new value for *x*. For example, the assignment statement *x* = *y* + 1 implies such a dependency relation. Every time we change the value of *y* the value of *x* is changed after executing the statement. Another example which leads to the same dependency is the following program fragment (Fig. 1):

```
If (y < 10) then
    X = 1;
Else
    X = 0;
```

Fig. 1. Simple conditional block

In this fragment not all changes applied to *y* cause a change on the value of *x*, although *x* definitely depends on *y*. The Aspect system now takes a program, computes the dependencies and compares them with the specified dependencies. If there is a mismatch the system detects a bug and notifies the user. However, the Aspect system does not pinpoint the root-cause of the detected misbehavior to the user.

In the following program fragment (Fig. 2) we explain the basic ideas using the following small program which implements the computation of the circumference and area of a circle. The program contains one fault in line 2 where a multiplication by π is missing.

```
// pre true
1. d = r * 2;
2. c = d; // BUG! a = d * pi;
3. c = r * r * pi;
// post c = r². π ^ a = 2. r. π
```

Fig. 2. Calculation of circumference and area

These dependences solely are given by a statement whenever we assume that the statement is correct (w.r.t. the dependences). If a statement is assumed to be incorrect, the dependences are not known. We express the latter fact by introducing a new type of variable, the so called model variables. Model variables are variables that work as place-holder for program variables. For example, if we assume statement 2 to be incorrect, we introduce a model that says that program variable *a* depends on model variable ε^2 (where ε^2 is unique).

The idea behind our approach is to find assumptions about the correctness and incorrectness of statements which do not contradict a given specification. In our running example, the specification is given in terms of a post-condition. From this post-condition we derive that *c* has to depend on *r* and *pi*. However, when assuming statement 1 and 2 to be correct, we derive that *a* depends on *d* and *d* in turn depends on *r* which leads to *c* depends on *r* but not on *pi*. Hence, the computed dependence contradicts the specified one.

To get rid of this inconsistency, we might assume line 2 to be faulty. Hence, we can compute that *c* depends on model variable ε^2. When now comparing the specification with the

computed dependence we substitute ε^2 by *r* and *pi* and we can not derive an inconsistency anymore.

The authors of (Wotawa and Soomro 2005, Peischl and Soomro 2006, Soomro 2007) present a detailed formalization of this idea and also present rules for most important language artifacts like an assignment statement, the if-then-else statement, while loop and procedures. In this chapter we focus on method invocation, parameter substitution, return statement and global variables.

```
0.   // Pre conditions of class: true
1.    public class sumpowers {
2.      int i, start, sum;
3.      int stop, f;
4.      …
5.      …
6.      i = start;
7.      while (i < stop)
8.      {
9.        sum = sum + power(x,f);
9.        // post (sum, x), (sum,f), (sum, power)
10.       i = i + 1;
11.     }
12.
13.     //Pre-condition of method: true
15.     int power(xf,ef) {
1.        int power = 1;
2.        while (ef > 0)
2.        {
3.            power = power * 10;
3.            //instead of power = power * xf
4.            ef = ef - 1;
4.        }
5.        return power;
16        // post {(power,power),(power,xf), (power, ef)}
16.    }
17.      //Equation power = xf^ef
17.   }
```

Fig. 3. Sum and power of integer values

In Fig. 3 we show an example that use a method to compute the sum and power of integer numbers.The program contains one method which computes the n^{th} power of an integer number.

In computing the dependences for procedures and their invocations we first compute the dependences of the procedure being invoked. Afterwards we substitute the procedure's formal parameters by the actual ones. We capture recursive invocations by computing the transitive and reflexive closure of the procedure's body and subsequently get rid of

dependences induced by local variables. Finally, we add those dependences caused by the procedure's return values.
In Fig. 3., line number 9 we call the method *power* with some parameters. The specification of method is ***{ (power, power), (power, ef), (power, xf) }***. When computing the dependences from the method we derive these dependences ***{ (power, power), (power, ef), (ef, ef) }.***

In these dependences the variable pair *(ef, ef)* are not impacting overall dependences.
So we have final dependences of method are ***{ (power, power), (power, ef) }***.
We have to map formal to actual parameters from these derived dependences.

Definition 1 (Parameter Substitution)
Let *d* be the dependences of the a method *m* and let *f* be a formal parameter and *a* the corresponding actual parameter. The dependences after method invocation are given by
***d'*= { (a, y) | (f, y) ∈ d } ∪ { (x, a) | (x, f) ∈ d }.**

After mapping parameters we derived dependences ***(sum, power), (sum, f)*** , here we find a contradiction with the post conditions ***{ (sum, x), (sum, f), (sum, power) }***. If post conditions are consistent with computed ones then we introduce model variables. We used unique model variable for every assumption. The Definition 2 states that how to establish the relationship between the return variables and the target variable of calling context.

Definition 2 (Return Values of a Method)

$$\begin{aligned} \neg Ab(x = proc(a_1, a_2, \ldots, a_n)) \rightarrow \\ D(t = proc(a_1, a_2, \ldots, a_n)) = \\ \{t\} \times \{v \mid (x, v) \in D(proc(a_1, a_2, \ldots, a_n)), x \in return(proc)\} \end{aligned} \tag{1}$$

$$\begin{aligned} Ab(t = proc(a_1, a_2, \ldots, a_n)) \rightarrow \\ D(t = proc(a_1, a_2, \ldots, a_n)) = \{(t, \xi_l), (g, \xi'_l) \mid g \in G\} \end{aligned} \tag{2}$$

where t denotes the target variable, g global variable in proc(body) and return(proc) is a function returning the return values of the procedure proc. Where Ab shows abnormal and ¬Ab shows not abnormal conditions.

The definition 3 states how to establish dependences of the two consecutive statements of the program.

Definition 3 (Composition)
Given two dependency relations R_1, $R_2 \in D$ on V and M. The composition of R_1 and R_2 is defined as follows:

$$\begin{aligned} R_1 \circ R_2 = \quad & \{ (x,y) \mid \exists(x,z) \in R_2 \,\&\, \exists(z,y) \in R_1 \} \cup \\ & \{ (x,y) \mid \exists(x,y) \in R_1 \,\&\, \sim\exists(x,z) \in R_2 \} \cup \\ & \{ (x,y) \mid \exists(x,y) \in R_2 \,\&\, \sim\exists(y,z) \in R_1 \} \end{aligned} \tag{3}$$

This definition ensures that no information is lost during computing the overall dependency relation for a procedure or method. Hence, the first line of the definition of composition handles the case where there is a transitive dependency. The second line states that all dependencies that are not re-defined in R_2 are still valid. In the third line all dependencies that are defined in R_2 are in the new dependency set provided that there is no transitivity relation.

For combining the dependencies of two consecutive statements we define the following composition operator as given in definition 3 for dependency relations to obtain the following dependences ***D (sum = power(x , f)) = { (power, power), (power, ef) }***. After substituting formal to actual parameter derived dependences of line number 9 are ***{ (sum, power), (sum, f) }*** but the post conditions are ***{ (sum, x), (sum, f), (sum, power) }***. Here we find contradiction between both dependences, derived ones and specified ones.

In order to compare a computed dependence set with the specification we have to find a substitution that makes the computed dependence set equivalent to the specified one. If there is no such substitution the sets are said to be inconsistent.

A substitution σ is a function which maps model variables to a set of program variables, i.e., σ: M → 2^V. The result of the application of the substitution σ on a dependence relation R is a dependence relation where all model variables x in R have been replaced by $\sigma(x)$.

We assume that statement 9 is abnormal, we take the target variable from the assignment statement and introduce a model variable (sum, ξ_9) . In order to compute dependences
we derive ***{ (sum, ξ_9), (i, i), (i, stop) }*** and the substitution variables are ***{ (x, f, power) }***. We now compare the specification with the computed dependences obtained by substituting ξ_9 with ***{ (x, f, power) }***. Since, we can not derive an inconsistency anymore, so line number **9** is a bug candidate.
Some assumptions for other lines of method call:
If we assume statement **3** to be incorrect then we can take the left variable from the assignment statement and introduce a model variable to arrive at ***(power, ξ_3)***. After computing dependences with this model variable we derive ***{(power, ξ_3), (ef, ef)}*** and substitution variables are ***{ (power, xf, ef) }***. If we now compare the specification with the computed dependences obtained by substituting ξ_3 by ***{ (power, xf, ef) }***, we can no longer derive inconsistency. So line number **3** from method is a bug candidate.

Assuming line **2** to be incorrect, the dependences derived with model variable are ***{((power, power) , (power, ξ_2), (ef , ef)) }***. Now we substitute the model variable from set of program variables, i.e, ξ_2 ***= { ef, xf, power }***. After substituting we derived ***{ (power, power) ,(power, xf), (power, ef), (ef , ef) }***. In order to allow the direct comparison of specified dependences with the computed ones, we introduce a projection operation which deletes all dependences for variables which have no impact on the overall dependences, like an internal variable pair ***(ef, ef)*** from Fig. 3.

A projection is defined on dependence relations $R \epsilon D$ and a set of variables $A \rightarrow M \cup V$. The projection of R on A written as $\prod_A (R)$ is defined as follows in an equation 3:

$$\prod_A (R) = \{ (x,y) \mid (x,y) \in R \wedge x \in A \tag{5}$$

However when assuming statement numbers from method *1,2,3,4* we obtain three diagnoses. Line numbers *1,2,3* are said to be faulty, but line number 4 did get substitution so line number ***4*** is not faulty.

Definition 4 (Treatment of Global Variables)
To obtain dependences form global variables we are dealing with the following features

1. Global variables impact global variables

If a global variable depends upon global variable in a program then we use similar rules to derive dependences from simple statements.

For an assignment statement $g = a + g$ the dependences are (g, a), (g, g).

2. Local variable impact global variables

If a global variable depends on a local variable of a method and the return variable also depends upon the same local variable then we can compute dependences as: Let d be the dependences of the method m and let l be a local variable, g the corresponding global variable and x be the returning variable . The dependences after method invocation is given by $\{ (g, x) \mid (g,l) \}$, where x is $(x \in return)$.

3. Formal variables impact global variables dependences

Let d be the dependences of a method m and let f be a formal parameter, a be an actual parameter and g the corresponding global variable. The dependences after method invocation is given by $\{ (g, a) \mid (g, f) \in d \}$.

If we assume an invocation to be abnormal we introduce a single variable for every occurrence of a certain procedure. For recursive invocations (in all cases where we obtain an cyclic call graph) we have to perform a fix-point computation. In order to guarantee that the computed dependences increase monotonically w.r.t. the subset relation like $d_n = d_n + 1$. Computing fixed point we add these dependences to overall dependences.

Fig. 4 shows the fixed point computation.

```
INPUT : while C do {B} end
OUTPUT: D (Dependences)
initialize d = d0
initialize d_Prev = null
do
{
    d_Prev = d
    d = d_Prev ∪ (d_Prev ∘ d_0)
}
while (!d.equals(d_Prev))
alldep = (alldep ∘ d)
```

Fig. 4. Algorithm of Fixed Point Computation

In

Fig. 4 d_0, d, d_{Prev} are variables which are storing pairs of dependences, where d computes new dependences, d_{Prev} stores previous dependences. d comprises of block dependencies from while loop **while C do { B } end**. The return statement stores overall dependences into d after finding fix-point. Function *union* adds both dependences $d_{Prev} \cup (d_{Prev} \circ d_0)$. The composition operator $\circ$ ensures that no information is lost during computing the overall dependency relation. Union operator in the algorithm shows that dependencies are adding after every iteration of loop. The condition of the *do {} while* loop ensures that whenever previous and new dependences are same we reached fix-point with finite number of iterations. After reaching fix-point we have to terminate loop and add computed dependence pairs into overall dependences d.

Theorem 1 (Fixed Point Computation Theorem)
The fixed-point computation algorithm computes a fix-point from repetitive invocation within a finite number of iterations.

Proof. We prove this theorem in two steps: First we prove that the dependencies are increasing monotonically. Second these dependencies should become equal within finite number of iterations at one point which is a fix-point.

- Dependencies are increasing monotonically i.e $\forall i\ d_{i+1} \supseteq d_i$.
  From the above algorithm, we know that $d_{i+1} = d_i \cup \{d_i \circ d\}$. The computed dependences of block statements are stored in d. Because of the union operator it is obvious that $d_{i+1} \supseteq d_i$. This $(d_i \circ d)$ are the new dependences which are added to the old dependency set. This leads to a monotonically increasing amount of dependences.
- Fix Point Computation i.e. $\exists i\ d_{i+1} = d_i$.
  We know that set of variable v is finite. Hence, $d_i = V \times V$ is finite and the upper bound of the dependency computation. From this follows that iteration an i exists when $d_{i+1} = d_i$.

(Wieland's 2001 (Fix-point Computation)) Complexity of computing dependences from while loop to reached fix-point in a finite number of iterations.

Theorem 1 shows the complexity of while loop for computing the dependences using the above algorithm. In [Wieland 2001], the author proves the theorem depending upon the number of variables used in dependences set. This example represents the worst case.

In Fig. 5 we call method ***foo*** recursively. Here we show that the dependences of recursively methods are in this fashion with using fixed point computation. The dependences of calling method ***foo*** has following dependences ***{ (y,x), (z, y), (z, x), (res, z), (res, y), (res, x) }***. The definition 4 ensures that we substitution of local, global variables are derived correctly.We use fixed point algorithm to find $d_n = d_n + 1$. In the *fooexample* method line number 5 has an assignment statement *t = foo(a, b)* that calls a method. Now we have to subsitute formal into actual parameters from computed dependences of calling method. After substitution we derived following dependences *(t, a),(t, b),(t, res)*.

```
1.    public int fooexample {
2.           int t, a, b;
3.           …
4.           …
5.           t = foo(a, b)
5.           //{(t,a),(t,b),(t,res)}
      private int foo(int x, int y) {
6.           int res=0;
7.           int z=1;
8.           if ( x < 0 )
9.              y = x;
             else
10.             z = foo(x-1, y);
11.          res = z + y;
12.          return res;
12.    {// (y, x), (z, y), (z, x), (res, z), (res, y), (res, x)}
12.  }
```

Fig. 5. Recursively call foo function

4. Example applying on fixed point computation

We show the small example to find fixed point over transitive relations. In an example program in Fig.6 we compute dependences step by step fashion to show that how we reach fixed point by using the algorithm in

Fig. 4. The body of while loop iterates i times.

In particular we are interested to compute dependences from the body of loop and summarized the loop dependences. Obviously all variables are used in the the body of loop *(a, b, c, d and i)*, which depends on variable *i*, because *i* appears in the condition of while loop. We did not show the computed dependences of i variable in Fig.6 and the graph's Fig. 7. The dependences of i are explicitly computable and do not make any changes during iteration of loop. This is the reason we are not presenting these dependences. The dependences of other variables are presented in the below. The algorithm presented presented earlier outlines the method of computing dependences from recursive invocation.

The dependences of the variables a; b; c and d are depicted in the following cases:

- (i = 0): and d_0 = {(a, b), (b, c), (c, d), (d, e)}
- (i = 1): and d_1 = {(a, b), (b, c), (c, d), (d, e), (c, e), (a, c), (b, d)}
- (i = 2): and d_2 = {(a, b), (b, c), (c, d), (d, e), (c, e), (a, c), (b, d), (a, d), (b, e)}
- (i = 3): and d_3 = {(a, b), (b, c), (c, d), (d, e), (c, e), (a, c), (b, d), (a, d), (b, e), (a, e)}
- (i = 4): and d_4 = {(a, b), (b, c), (c, d), (d, e), (c, e), (a, c), (b, d), (a, d), (b, e), (a, e)}

- The given specification from Fig. 6 and computed dependences are equal. So we reached fix point with finite number of iteration. We find $d_3 = d_4$ is a fix-point where both dependences are equal. Which ensures $d_n = d_{n+1}$.

All variables of assignment statements are depend upon variable i $\leftarrow$ (a, b, c, d).

SPEC {(a, b),(b, c),(c, d), (d, e), (c, e), (a, c),(b, d), (a, d), (b, e),(a, e)}

```
1. public int WhileExample {
2.     int a, b, c, d, e;
3.     int i = 0;
4.     ....
5.     ....
6.     while (i ≤ 5)
7.     {
8.            a = b;
9.            b = c;
10.           c = d;
11.           d = e;
12.           i = i + 1;
       }
    }
```

Fig. 6. Example program showing transitive dependencies in a loop

The graph shows the dependences of the while loop. There are five nodes in Graph including incoming and outgoing edges. Based on Theorem 1 we can prove that after a finite number of iterations we reached fixed point. we explain as follows:

Dependences are increasing monotonically i.e. $\forall i\ d_{i+1} \supseteq d_i$. We present the algorithm in

Fig. 4 for computing dependences of block statements. Because of the union operator it is obvious that $d_{i+1} \supseteq d_i$. This leads to a monotonically increasing amount of dependences. which is proven in Theorem 1. After calculating n we find fix-point within four iterationsn = 4. Therefore, we reached at fixed point after a finite number of iterations which is described in the algorithm. The graph shows the graphically representation of finding fixed point.

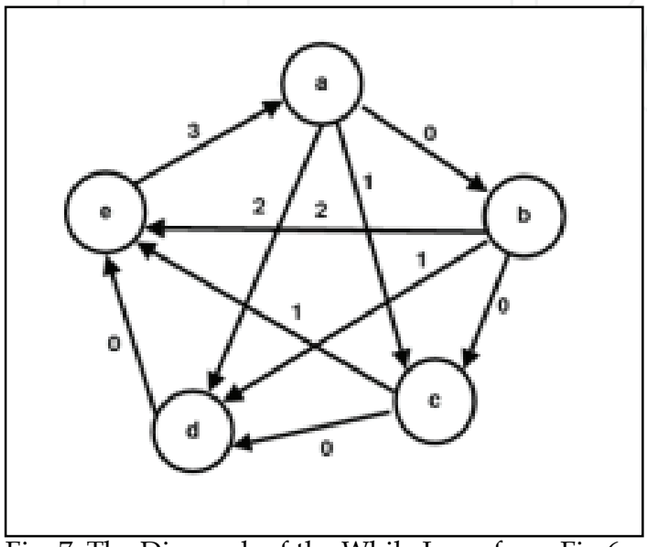

Fig. 7: The Diagraph of the While Loop from Fig.6

5. Experimental Results and Discussions

The proposed model has been implemented in Java using the Eclipse platform. In this section, we present the experiments that evaluate the result using dependencies in Java programs without using object-oriented features. Experiments were performed on a Intel Pentium 4 Workstation (3 GHZ,512 MB Memory) running Gentoo Linux (Gentoo Base System Version 1.4.9, Kernel version 2.6.5). The results are reported in Table 1.

For various examples programs, we introduced a single fault, and afterwards computed all single-fault diagnoses. Table 1 presents empirical results of programs with methods. We considered medium sized programs. The second column shows the lines of code from 26 to 509. The third column counts the number methods in the programs. The fourth column reports the number of diagnosis candidates. The 5th column gives the number of input variables and the last column shows the number of output variables.

In Table 1 the tested programs consisting of simple and multiple statements, loops, methods and global variables. Program specification consists of all variables i.e., input variables and output variables. In Fig. we show all these programs with minimum, maximum and average of diagnoses.

In Fig. 8 we presented a graph of programs with the number of faults depending on output variables (presented in Table 1). We used hundred iteration for every possible combination of output variables. It shows the minimum, maximum and mean of diagnoses in respect to

the output variables and the faults. In all graphs full line represent the minimum, dash line the maximum and dotted line the number of diagnoses candidates.

Programs	LOC	Methods	Diag.No	Input-VAR	Output-VAR
Adder	51	3	6	18	15
AddulseTime	378	21	10	98	54
Equation	26	4	4	13	5
MathFunctions	509	22	3	80	57
MethodTest1	42	3	12	14	11
MethodTest2	75	5	2	22	16
MethodTest3	46	5	3	20	14
Programs	**LOC**	**Methods**	**Diag.No**	**Input-VAR**	**Output-VAR**
MethodTest4	218	15	3	53	44

Table 1. Diagnosing candidates obtained by an introducing a single fault

Note, in the graph we consider only those diagnoses which has contradiction. This means that we never pick values which lead to no contradiction. Full line indicates that when we increase the number of output variables used in the specification, and then the number of diagnosis increases. The results indicate that our approach is feasible for detecting and localizing real cause of misbehaviour. The results presented there solely stem from procedural programs.

6. Limitation

Unlike previous approaches (Wieland 2001, Mayer. W & Stumptner M 2003) , the debugging approach introduced in this chapter specially intends localizing structural faults. In an account of this, we focus this discussion our model's weaknesses in detecting and localizing these faults.

First, the proposed model is not localizing faults caused by a erroneous target variable. For example, the following code snippet assigns the value of variable z to two different variables, namely, x and y.

1. **y ← z // should be x ← z**
2. **y ← z**

Due to (structural) error in line 1, we obtain *(y, z)* as a single dependence, but our specification enumerates *{ (x, z), (y, z) }* as dependences. Since *{(y, z)}* is not super set or equal to *{(x, z), (y, z) }*, we know some thing must be wrong. Thus, obviously, we can detect this bug. In localizing the detected misbehaviour's cause, we assume statement 1 to be abnormal and obtain R_1 = *{ (x, ε_1)}.* R_2 = *{(x, z)}, and* $R_1 \circ R_2$ =*{(x, z)}.* This results shows that we can not find any substitution σ that removes the given contradiction. Virtually, we can not correct our program by modifying solely the right-hand side of statement 1.

Allowing for model variable left hand side we can get rid of this deficiency: R1 = { (ε_1, z)} R_2, = { (y, z) }, which yields to $R_1 \circ R_2$ = { (ε_1, z), (x, z) }. Apparently, the substitution σ(ε_1) = x fulfils {(ε_1, z), (y, z)} is super set or equal to *{(x, z), (y, z)}* and thus we can localize this fault. However, this is approved solely in the rather are case that the substituted variable does not appear in any of the statements (on the right hand side) subsequent to variables substitution. Thus, this approach is not applicable in a practical setting.

Furthermore our model is only applicable to alias-free programs. This issue is further discussed in (Jackson 1995). In summary we conclude that, provided the fault appears on the right hand side of an assignment, our model allows for correcting these kind of structural faults as long as we can also detect this fault in terms of the specified dependences.

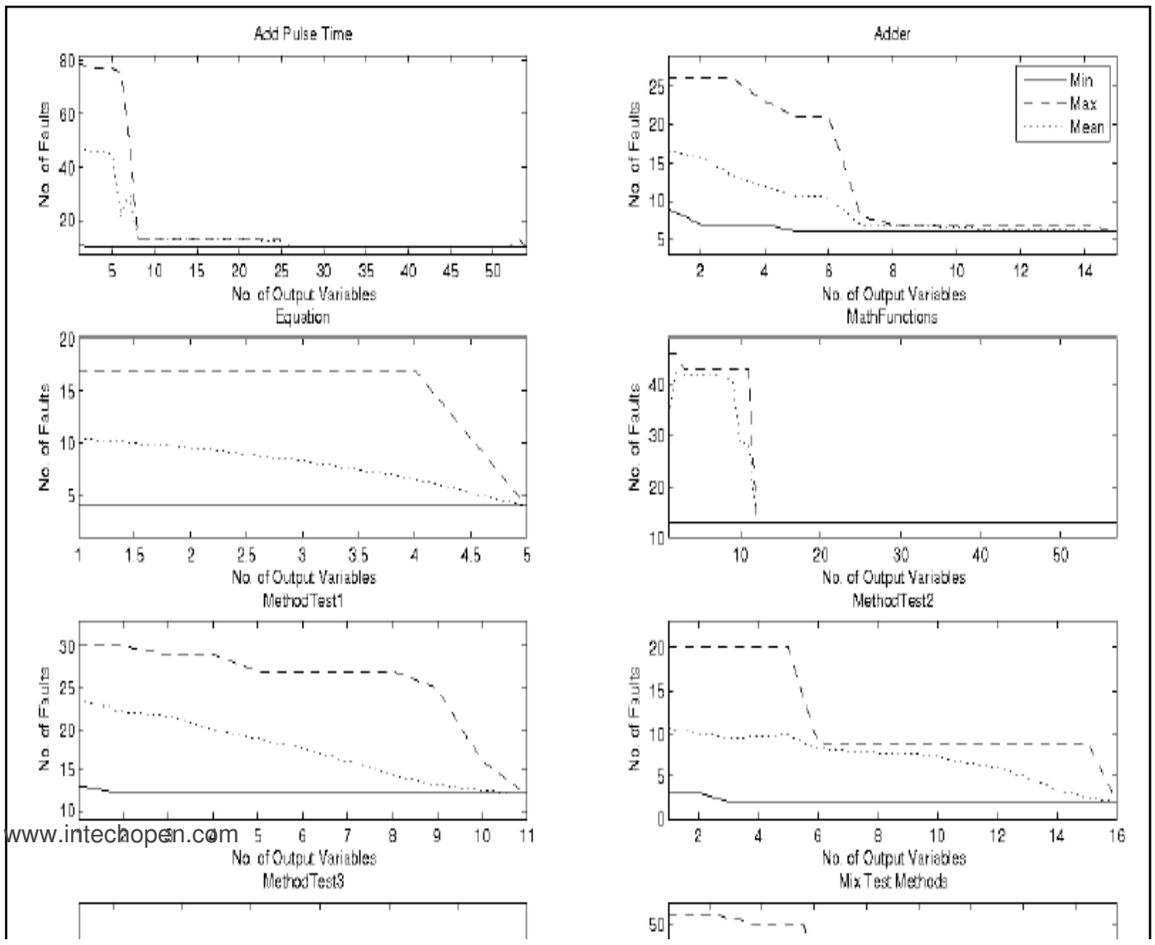

Fig. 8. Sensitivity Analysis of All Programs of Table 1.

7. Related Research

The author of (Jackson 1995) presents work which is closest to the work presented herein. This work employs abstract dependences for detecting rather than for localizing a fault. Furthermore in (Kuper 1989, Wieland 2001) the authors employ the notion of dependences for fault localization. In contrast to the latter approach we do not employ detected differences in variable values at a certain line in code but make use of differences between specified and computed dependences and thus also incorporate the structural properties of program and specification.

In the recent past the authors (Wotawa 2000, Wotawa et, al. 1999, Koeb and Wotawa 2004, Mayer and Stumptner 2003) developed models for different languages at various abstraction levels in the model-based context. In general, abstract modelling approaches sacrifice detail in favour of computational complexity whereas more detailed value-level models (Wotawa 2002, Peischl and Wotawa 2003) provide accurate fault localization capabilities but on the other hand require considerable computational resources in terms of space and computing power.

Although program slicing, as a lightweight technique, has seen successful application in fault localization (Agrawal et, al., 1993, Fritzson et, al., 1999, Lyle and Weiser 1987) its discrimination like MBSD (Wotawa 2000). In (Kuper 1989, Wieland 2001) the authors employ the notion of dependences for fault localization. In contrast to latter approach we do not employ detected difference in variable values at a certain line in code but use of differences between specified and computed dependencies and thus also incorporate the structural properties of program and specification. Thus, the models introduced in (Kuper 1989, Wieland 2001) can not deal with assertions or pre- and post conditions in a straightforward way.

The authors of (Wotawa 2000, Friedrich et, al., 1999, Wieland 2001) solely make use of concrete values in incorporating correctness information. These models do not allow taking advantage of arbitrary relationships between several variables or variables and constants. The author (Stumptner 2001) shows that localizing structural faults requires exploiting design information like assertions, and pre and post conditions.

Other approaches like (Johnson 1986) focus on novice programmers and make use of methods that help to find faults in the code by comparing the code with pre-specified problem formulations.

7. Conclusion

In this chapter we extend and present the novel model which detect and localize real faults from programs, comprising methods invocations and global variables. The computation of dependences from recursive invocation we found that every iteration has added new dependences and the number of dependences increasing monotonically. In order to guarantee that the computed dependences are increasing monotonically w.r.t. iterations, we find fixed-point where all dependences are equal with finite number of iterations. We presented an algorithm and proof for fixed point computation which ensures that no dependences loss during iteration and we always reaches fix-point after a finite number of iterations. Moreover, the approach is different to other available dependency-models and provides better results for medium sized programs.

A future research challenge is the formal and empirical evaluation of the modelling approaches when apply it to real object-oriented programs.

8. Acknowledgments

This work was supported by the Higher Education Commission, Islamabad, Pakistan under its research and development funding. The author would like to thank Prof. Franz Wotawa and Bernhard Peischl for their valuable comments and advice during my stay at Graz University of Technology, Austria.

8. References

Peischl. B & Wotawa F. (2003). Model-Based Diagnosis or Reasoning from First Principles, *IEEE Intelligent Systems*, 18(3), (May-June 2003) page numbers (32—37), 1541-1672.

Peischl B.; Soomro S. & Wotawa F. (2006). Abstract Dependence Model in Software Debugging. *Proceedings of the 17th International Workshop on Principles of Diagnosis (DX-06)* , France.

Jackson D. (1995). Aspect: Detecting Bugs with Abstract Dependences, *ACM Transactions on Software Engineering and Methodology*, 4(2), (April 1995) page numbers (109-145), 1049-331x.

Wieland D. (2001). Model Based Debugging of Java Programs Using Dependences. *PhD*

Thesis, Vienna University of Technology, Computer Science Department, Institute of Information Systems (184)}, Database and Artificial Intelligence Group (184/2), Vienna, Austria.

Koeb D. & Wotawa F. (2004). Introducing Alias Information into Model-Based Debugging. *Proceedings of 16th European Conference on Artificial Intelligence (ECAI)*, IOS Press, pp 833--837, Valencia, Spain.

Wotawa F.(2000). On the Relationship between Model-Based Debugging and Program Slicing, *Artificial Intelligence*, 135(1-2) (February 2002), page numbers (124—143), 0004-3702.

Wotawa. F(2002). Debugging Hardware Designs using a Value-based Model, *Applied Intelligence*, 16(1) , (January-Fabruary 2002) page numbers (71—92), 0924-669x.

Wotawa F. & Soomro S (2005). Using abstract dependencies in debugging. *Proceedings of 19th International Workshop on Qualitative Reasoning QR-05*, pp. 23--28, Austria.

Wotawa F. & Soomro S (2005). Fault Localization Based on Abstract Dependencies. *Proceedings of the 18th Conference on International Conference on Industrial, Engineering & Other Applications of Applied Intelligent Systems (IEA/AIE 2005)*, Lecture Notes in Artificial Intelligence (LNAI),pp. 357-359, Springer Verlag, Italy.

Friedrich G.; Stumptner M. & Wotawa F.(1999). Model-based Diagnosis of Hardware Designs, *Artificial Intelligence*, 111(1-2), page numbers (3—39), 0004-3702.

Agrawal, H.; A. Demillo; Richard & H. Spafford; Eugene (1993). Debugging with dynamic slicing and backtracking Softw. *Practice and Experience*, 23(6), (June 1993), page numbers (589—616) , 0038-0644.

Lyle J.R. & Weiser. M (1987). Automatic Program Bug Location by Program Slicing. *Proceedings of the 2nd International Conference on Computers and Applications*, pp. 877--882, Beiging (Peking), China.

Johnson W. Lewis (1986). Intention-Based Diagnosis of Novice Programming Errors, In: *Morgan Koffman*,(Ed.), page numbers (333) Los Altos.

Weiser M. (1984). Program Slicing, *IEEE Transactions on Software Engineering*, 10(4), (July 1984) , (439-449) 0-89791-146-6.

Stumptner M. (2001). Using Design Information to Identify Structural Software Faults, *Proceedings of the 14th Australian Joint Conference on Artificial Intelligence*, pp. 473-486, London, Springer-Verlag, UK.

Fritzson P.; Gyimothy T.; Kamkar M. & Shahmehri N. (1999). Generalized Algorithmic Debugging and Testing, *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation,* pp 317—326, Toronto, Canada,

Kuper Ron. I. (1989). Dependency Directed localization of software bugs. *Technical Report* AI-TR 1053, MIT AI Lab, May USA.

Mayer W. & Stumptner M. (2003). Extending Diagnosis to Debug Programs with Exceptions. *Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE), Montreal, IEEE Conferences on Automated Software Engineering*, pp 240--244, Montreal, Canada.

Soomro. S. (2007). Using Abstract Dependences to Localize Faults from Procedural Programs. *Proceedings of the 25th conference on Proceedings of the 25th IASTED International Multi-Conference: artificial intelligence and applications (AIA),* pp-180-185, Innsbruck, Austria.

Frontiers in Robotics, Automation and Control

Edited by Alexander Zemliak

ISBN 978-953-7619-17-6
Hard cover, 450 pages
Publisher InTech
Published online 01, October, 2008
Published in print edition October, 2008

This book includes 23 chapters introducing basic research, advanced developments and applications. The book covers topics such us modeling and practical realization of robotic control for different applications, researching of the problems of stability and robustness, automation in algorithm and program developments with application in speech signal processing and linguistic research, system's applied control, computations, and control theory application in mechanics and electronics.

INTECH
open science | open minds

InTech Europe
University Campus STeP Ri
Slavka Krautzeka 83/A
51000 Rijeka, Croatia
Phone: +385 (51) 770 447
Fax: +385 (51) 686 166
www.intechopen.com

InTech China
Unit 405, Office Block, Hotel Equatorial Shanghai
No.65, Yan An Road (West), Shanghai, 200040, China
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元
Phone: +86-21-62489820
Fax: +86-21-62489821

IntechOpen