USING ABSTRACT DEPENDENCES TO LOCALIZE FAULTS FROM PROCEDURAL PROGRAMS

Safeeullah Soomro
Technische Universität Graz
Institute for Software Technology
8010 Graz, Inffeldgasse 16b/2, Austria
email: ssoomro@ist.tugraz.at

ABSTRACT
Use of fault localization in control engineering where engineers often employ the procedural programming paradigm. Often control software is safety-critical and thus detection but also localization of bugs is uttermost important. Detecting and locating faults is one of the most important phase of Software Engineering. Many efforts have been taken to improve software development and to prevent faults. But still software faults pose that most challenging problems to software engineers. This article makes use of abstract dependencies between program variables for locating faults in programs. We motivate the underline theory and give experimental results. Our fault localization model is based on previous work that uses the abstract dependencies for fault detection. In this paper we introduce a model for method calls including parameter substitution, global variables and the return statement of methods. The results show that our model is practically applicable in the field of software debugging.

KEY WORDS
Model Based Reasoning, Software Verification, Software Debugging, Fault Localization, Fault Detection

1 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 paper we focus on fault localization which is based on the abstract dependencies that are used by the Aspect system [5] for detecting faults. The verification-based model for debugging is an extension of the dependence model from Jackson’s Aspect system [5] 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 fulfills the specification. Unfortunately, the Aspect system does not allow to locate the source of a mismatch. In the following we extend Jackson’s idea towards not only detecting misbehavior but also localizing the malfunctioning’s real cause.

In this article 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 paper is organized as follows. In Section 2 we introduce our verification based model by using motivating example. The results and discussion given in Section 3 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 4 we present related research. Finally we summarize the paper.

2 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 paper we focus on fault localization which is based on abstract dependencies that are used by the Aspect system [5] 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:

\[
\text{if } ( y < 10) \text{ then} \\
\quad x = 1; \\
\text{else} \\
\quad x = 0;
\]

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 systems does not pinpoint the root-cause of the detected misbehavior to the user.
In the following 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 \( \pi \) is missing.

0. // pre true
1. \( d = r \ast 2; \)
2. \( c = d; // \text{BUG!} \ a = d \ast \pi; \)
3. \( c = r \ast r \ast \pi; \)
4. // post \( c = r^2 \cdot \pi \wedge a = 2 \cdot r \cdot \pi \)

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 \( \xi_2 \) (where \( \xi_2 \) 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 \( \xi_2 \). When now comparing the specification with the computed dependence we substitute \( \xi_2 \) by \( r \) and \( \pi \) and we can not derive an inconsistency anymore.

The authors of [10, 11] 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 paper we focus on method invocation, parameter substitution, return statement and global variables.

In Figure 1 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 Figure 1 line number 9 we call the method \textit{power} with some parameters. The specification of method is

0. // Pre Conditions of Class true
1. public int sumpowers {
2. int i, start, sum;
3. int stop, f, start;
4. ....
5. ....
6. i = start;
7. while (i < stop)
8. {
9. sum = sum + power(x,f);
10. } // post (sum,x),(sum,f),(sum,power)
11. i = i + 1;
12. }
13. // Pre Condition of Method true
14. int power(xf, ef) {
15. int power = 1;
16. while(ef>0)
17. {
18. power=power \ast 10;
19. // instead of power = power \ast xf;
20. ef = ef - 1;
21. }
22. return power;
23. // post (power,power),(power,xf),(power,ef)
24. }
25. // Equation power = xf^{ef}
26. }

Figure 1. Sum and power of integer values

\((power,power),(power,ef),(power,xf)\). 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.

\textbf{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 \( d' = \{(a,y)|(f,y) \in d\} \cup \{(x,a)|(x,f) \in d\} \).

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)
\neg \text{Ab}(x = \text{proc}(a_1, a_2, \ldots, a_n)) \rightarrow \text{D}(t = \text{proc}(a_1, a_2, \ldots, a_n)) = \{t \times \{v(x, v) \in \text{D(proc(a_1, \ldots, a_n)), } x \in \text{return(proc)}\}.
\neg \text{Ab}(t = \text{proc}(a_1, a_2, \ldots, a_n)) \rightarrow \text{D}(t = \text{proc}(a_1, a_2, \ldots, a_n)) = \{(t, \xi_1), (g, \xi'_1) | g \in G\},
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 \( \text{Ab} \) shows abnormal and \( \neg \text{Ab} \) 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 \( R_1, R_2 \in \mathcal{D} \) on \( V \) and \( M \). The composition of \( R_1 \) and \( R_2 \) is defined as follows:
\[ R_1 \circ R_2 = \{(x, y) | \exists(x, z) \in R_2 & \exists(z, y) \in R_1\} \]
\[ \cup \{(x, y) | \exists(x, y) \in R_1 & \bigwedge(x, z) \in R_2\} \]
\[ \cup \{(x, y) | \exists(x, y) \in R_2 & \bigwedge(y, z) \in R_1\} \]

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 dependences that are not re-defined in \( R_2 \) are still valid. In the third line all dependences 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 3 for dependency relations to obtain the following dependences \( \text{D(sum = power(x,f))} = \{(\text{power, power}), (\text{power, ef})\} \). 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 \( \sigma \) is a function which maps model variables to a set of program variables, i.e., \( \sigma : M \rightarrow 2^V \). The result of the application of the substitution \( \sigma \) 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 \( \{\text{sum, } \xi_9\} \). In order to compute dependences we derive \{\{sum, \xi_9\}, \{i, i\}, \{i, \text{stop}\}\} and the substitution variables are \{\{x, f, \text{power}\}\}. When now compare the specification with the computed dependences by substituting \( \xi_9 \) with \{\{x, f, \text{power}\}\}. Hence we can no derive an inconsistency more, So line number 9 is a bug candidate.

Here are assumptions of lines for method call. If we assume statement 3 is incorrect then we take the left variable from the assignment statement with introduce a model variable \( \{\text{power, } \xi_3\} \). After computing dependences with the model we derive \{\{\text{power, } \xi_3\}, \{\text{ef, } \text{ef}\}\} and substitution variables are \{\{\text{power, } \text{xf, } \text{ef}\}\}. When now comparing the specification with the computed dependences we substitute \( \xi_3 \) by \{\{\text{power, } \text{xf, } \text{ef}\}\} and we can no longer derive inconsistency. So line number 3 from method is a bug candidate.

Assumption of line 2 from method we derived dependences with model variable are \{\{\text{power, } \text{power}\}, \{\text{power, } \xi_2\}, \{\text{ef, } \text{ef}\}\}. Now we substitute the model variable from set of program variables, i.e, \( \xi_2 = \{\text{ef, } \text{xf, } \text{power}\} \). After substituting we derived \{\{\text{power, } \text{power}\}, \{\text{power, } \text{xf}\}, \{\text{power, } \text{ef}\}, \{\text{ef, } \text{ef}\}\}. In order to allow the direct comparison of specified dependences with the computed ones, we introduce a projection which deletes all dependences for variables which are not impact on overall dependences, like an internal variable pair \{\text{ef, } \text{ef}\} from Figure 1. A projection is defined on dependence relations \( R \in \mathcal{D} \) and a set of variables \( A \subseteq M \cup V \). The projection of \( R \) on \( A \) written as \( \Pi_A(R) \) is defined as follows:
\[ \Pi_A(R) = \{(x, y) | (x, y) \in R \wedge x \in A\} \]

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 global variable is depending upon global variable in a program then we use similar rules for derive dependences from simple statements. An assignment statement \( g = a + g \) we derived dependences \( (g, a), (g, g) \).

2. **local variable impact global variables**
   If global variable is depending on local variable of the method and the return variable is depend upon the same local variable then we compute dependences.Let \( d \) be the dependences of the a method \( m \) and let \( l \) be a local variable, \( g \) the corresponding global variable and \( x \) is returning variable . The dependences after method invocation is given by \{\{(g, x)|(g, l)\}, \) where \( x \) is \( (x \in \text{return}) \).

3. **formal variables impact global variables dependences**
   Let \( d \) be the dependences of the a method \( m \) and let \( f \) be a formal parameter, \( a \) be a actual parameter and \( g \) the corresponding global variable. The dependences after method invocation is given by \{\{(g, a)|(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 \( dn = dn + 1 \). Computing fixed point we add these dependences to overall dependences. Algorithm 1 shows the fixed point computation.

**Algorithm 1 Algorithm of Fixed Point Computation**

initialize \( d = d_0 \)
initialize \( d_{prev} = null \)
do
\( d_{prev} = d \)
\( d = d_{prev} \cup (d_{prev} \circ d_0) \)
while (\(!d.equals(d_{prev})\))
\( \text{alldep} = (\text{alldep} \circ d) \)

**Ensure:** \( d_n = d_{n+1} \)

In Algorithm 1 \( d_0, d, d_{prev} \) are variables which are storing pairs of dependences, where \( d \) computes new dependences, \( d_{prev} \) stores previous dependences. The \( \text{alldep} \) is a global variable which stores overall dependences. Composite operator ensures that no information is lost during computing the overall dependency relation between all statements of program. Function \( \text{union} \) 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. The Condition of the while loop ensures that whenever previous and new dependences are same then loop terminate. In a given algorithm the last assignment statement stores the set of pairs to overall dependences in \( \text{alldep} \).

In Figure 2 we call method \( \text{foo} \) recursively. Here we show that the dependences of recursively methods are in this fashion with using fixed point computation. The dependences of calling method \( \text{foo} \) 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 \( dn = dn + 1 \). In the \( \text{fooexample} \) method line number 5 has an assignment statement \( t = \text{foo}(a,b) \) call the method. Now we have to substitute formal into actual parameters from computed dependences of calling method. After substitution we derived following dependences \( (t,a),(t,b),(t,res) \).

### 3 Experimental Results and Discussion

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.

All programs consisting of statements, loops, methods and global variables. Program specification consists of all variables i.e., input variables and output variables.

<table>
<thead>
<tr>
<th>Programs</th>
<th>LOC</th>
<th>Methods</th>
<th>Diag. no.</th>
<th>Input-VAR</th>
<th>Output-VAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddpulseTime</td>
<td>378</td>
<td>21</td>
<td>10</td>
<td>98</td>
<td>54</td>
</tr>
<tr>
<td>Equation</td>
<td>26</td>
<td>4</td>
<td>4</td>
<td>13</td>
<td>5</td>
</tr>
<tr>
<td>Math-Functions</td>
<td>509</td>
<td>22</td>
<td>3</td>
<td>80</td>
<td>57</td>
</tr>
<tr>
<td>MethodTest1</td>
<td>42</td>
<td>3</td>
<td>12</td>
<td>14</td>
<td>11</td>
</tr>
<tr>
<td>MethodTest2</td>
<td>75</td>
<td>5</td>
<td>2</td>
<td>22</td>
<td>16</td>
</tr>
<tr>
<td>MethodTest3</td>
<td>46</td>
<td>5</td>
<td>3</td>
<td>20</td>
<td>14</td>
</tr>
<tr>
<td>MixMethods</td>
<td>218</td>
<td>15</td>
<td>3</td>
<td>53</td>
<td>44</td>
</tr>
</tbody>
</table>

Table 1. Diagnosis candidates obtained by introducing a single fault.

In Graph 3 we show all 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.

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 indicate that when we increase the number of output variables used in the specification, then the number of diagnosis increases. The results indicate that our approach is feasible for detecting and localizing real cause of misbehavior. The results presented there solely stem from procedural programs.

4 Related Research

The author of [5] 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 [17, 18] 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 of [1, 6, 15, 16] developed models for different languages at various abstraction levels in the model-based context. In general, abstract modeling approaches sacrifice detail in favor of computational complexity whereas more detailed value-level models [1, 3] 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 [2, 12, 13], its discrimination like MBSD [7]. In [17, 18] 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 [17, 18] can not deal with assertions or pre- and post conditions in a straightforward way.

The authors of [1, 6, 18] solely make use of concrete values in incorporating correctness information. These models do not allow to take advantage of arbitrary relationships between several variables or variables and constants. The author [9] shows that localizing structural faults requires exploiting design information like assertions, and pre and post conditions.

Other approaches like [4] 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.

5 Conclusion and Future Research

In this paper we presented the model which detect and localize real faults in programs, comprising methods invocations and global variables. We present experimental results from medium sized programs to indicate that this model is able to localize the misbehavior’s real cause.

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

6 Acknowledgments

The Austrian Science Fund (FWF) supports this work under project grant P17963-N04. The Higher Education Commission (HEC), Pakistan, supports this work under its scholarship program.

References

Figure 3. Sensitivity Analysis of All Programs


