Fault Localization Models Using Dependences

Safeeullah Soomro[1], Abdul Hameed Memon[2],
Asif Ali Shah[3, 4] and Wajiha Shah[3, 4]
Emails: safee@ieee.org, memon@yic.edu.sa &
{asif.shah,wajiha.shah}@tuwien.ac.at
[1]*Yanbu University College, KSA*
[2]*Yanbu Industrial College, KSA*
[3]*Vienna University of Technology, Austria*
[4]*Mehran University of Engineering & Technology Jamshoro, Pakistan*

Abstract

In recent years Mode-Based Diagnosis has an acheived a tremendous recognition and has been applied to variety of disgnosis problems, mainly software debugging. Many efforts have been taken to improve software development and prevent faults. Still software faults pose challenging problems to software designers. Fault localization is next step after detecting faults in programs. This chapter makes use of dependences between program variables to detecting and localization faults from strucural programs. Further more we discuss the relationship between the FDM (functional-dependences model) and VBM (verification-based model) under presence of partial specifications artifacts like assertions are pre and post conditions by exemplifying specific scenarios in software debugging. Moreover, we discuss the relationship between VBM model and the well-known functional-dependence model particularly under presence of partial specification artifacts like assertions or pre- and post conditions. In the last we present the summary regarding dependences models that helps us to choose which model is detecting and locating errors from different type of data structures. Finally we discuss the case studies between FDM and VBM with some test programs.

Keywords: *Model Based Software Debugging, Software Debugging, Model-Based Diagnosis, Fault Detection and Localization.*

1. Introduction

Within the last decades several techniques for debugging, i.e., detecting, locating, and repairing faults in programs, have been proposed. Some of the techniques make use of models of the program's structure and behavior like algorithmic debugging or model-based debugging. Other techniques try to find bugs by examining code for suspects, i.e., code fragments that do not obey given rules, by finding differences between different program runs, or by finding (structural) differences between a program and another implementation.

The latter technique is used mainly for tutoring systems where the desired implementation is known in advance.

We introduce the basic idea of automated fault localization in software by means of a small Java program example. The program given in Figure 1. multiplies two integer variables x and y and returns the product and the product's sign. In line 11 we assign 0-y to the loop counter i, thus our program is buggy. For the test case x=-2 and y=+4 we obtain that result is 0, which contradicts our expectation result = -8.

In localizing the misbehavior's cause we start with static code analysis techniques. For example, we compute a static slice in line 18 with respect to the variable result. This rules out irrelevant statements by considering solely statements influencing the variable result. Our static slice contains the statements [2,6,7,8,9,11,12,14,15,16] - that is, we can exclude the statements 5 and 10 from causing the result's wrong value.

In addition we know that our loop counter *i* is always bigger or equal to 0 thus we might add the assertion in line 13. When executing the program with the test case x=-2 and y=+4 the variable i becomes -4 which causes the assertion to fail. At this point we again compute the static slice obtaining the lines [2,7,9,11]. Thus at this point we definitely know that one of the statements given in this slice causes the misbehavior with regard to variable i. Moreover, the second slice is a subset of the first one, thus we suspect the same error being responsible for both assertion violations, line 13 and 17.

In the next step we incorporate the specific test case's outcome. Notably, dynamic slices cannot localize the fault's real cause, thus the authors (Gyimothy et al, 1999 ; Agrawal et al, 1993) extend this notion to a so called relevant slice. A relevant slice is a dynamic slice augmented with potentially affecting conditions and their data dependences. We employ this notion of a slice for the given test case and the slicing criterion *(13,{result =0})*. When taking into account this kind of slice we can further rule out line 7. Since, for the given test case, the condition in line 8 is true, line 11 rather than 7 determines the value of i in line 13.

In our example, slicing techniques alleviate localizing the misbehavior's cause. In general, fault localization is a difficult task and requires the incorporation of various models (the spectrum ranges from abstract models relying on dependences (Jackson, 1995) to concrete value-level models) and techniques (e.g. algorithmic debugging (Shapiro, 1983), model-based debugging (Wotawa, 2000 ; Kleer & Williams, 1987) into a debugging environment.

```
1         // pre true
2         public int mult (int x,y) {
3         int add,i,result;
4
5         sign = 1;
6         add = y;
7         i = x;
8         result = 0;
9         if (x < 0) {
10                sign = -1;
11                 i = 0 - y;  // should be 0 - x
12                add = 0 - y; }
13        ASSERT (i >= 0)
                          14         while ( 0 < i ) {
15                i  = i - 1;
16                 result = result + add;}
17         ASSERT (result = xy);
18         output result, sign;
19        }
20        // post result = xy
21        // post sign = sign xy
```

Fig. 1. Multiplication of two integer values.

The chapter is organized as follows. In Section 2 we present our Verification Based Model (VBM). The comparison between VBM and (Functional Dependence Model) FDM is given in Section 3. In Section 4 we present case studies regarding debugging model. In section five we present the related research work of dependences based models. Finally we summarize the chapter including furure research.

2. Verification Based Model

Our novel debugging model allows one for reasoning about functions over dependence relations under given assumptions. Therefore the notion of a dependence relation is fundamental to our approach:

Definition 1 ***(Dependence Relation)*** *Given a program with variables V, and a set of model variables $M=\{\xi_1\ldots\}$. A dependence relation is a subset of the set $D = 2^V X ((M v V)$.*
The key idea is to abstract over the actual state transitions of the program and instead view its behavior as a dependence relation between inputs and outputs. Any pair *(x,y)* Є D says that the value of x after depends on the value of y before statement execution.

We obtain the dependence relation of compound statement by putting together the dependence relation of its constituent parts. Given two statement S_1 and S_2, the dependence relation of the compound statement $S_1 ; S_2$ is given by $D(S_1 ; S_2)=D(S_1) \bullet D(S_2)$. To reflect this we define two composition operators:

Definition 2 ***(Composition1)*** *Given two dependence relations R_1, $R_2 \in$ Don V and M. The composition of R_1 and R_2 is defined as follows:*

$$R_1 \bullet R_2 = \begin{array}{l} \{ (x,y) \mid \exists \text{ exists } (z,y) \text{ in } R_1 \ \& \ \exists \ (x,z) \in R_2 \} \cup \\ \{ (x,y) \mid \exists \text{ exists } (z,y) \text{ in } R_1 \ \& \ \nexists \ (x,z) \in R_2 \} \cup \\ \{ (x,y) \mid \nexists \text{ exists } (z,y) \text{ in } R_1 \ \& \ \exists \ (x,z) \in R_2 \} \end{array}$$

This definition ensures that no information is lost during computing the overall dependence relation for a procedure or method. Hence, the first line of the definition of composition handles the case where there is a transitive dependence. 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 dependence set provided that there is no transitivity relation.

Note that functional composition is not a commutative operation and that { } is the identity element of composition. For example, the combined dependences of our below examples are:

$$r^1 \bullet r^2 = \{(c,r),(d,r)\} = r' \text{ and } r' \bullet r_3 = \{(a,r), (a, pi), (c,r),(d,r)\} = r''$$

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 **Π (pi)** is missing.

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

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).

To point out intricacies with the previous definition we further need to define relation composition as follows:

Definition 3 ***(Relational Composition)*** *Given two dependence relations R_1, $R_2 \in D$ on V and M. Given two statement S_1 and S_2, the dependences of the compound statement S_1; S_2 in terms of* *relational composition is given by $D(S1; S2) = D(S2) \circ D(S1)$. The relational composition of R_1 and R_2 is defined as follows:*

$$R_1 \circ R_2 = \{ (x,y) \mid (x,y) \in R_1 \;\&\; (z,y) \in R_2 \}$$

Note, that as long as we rely on the simple dependence definition given previously, this operator suffers from information loss since it solely preserves transitive relationships. For our running example we obtain $r_1 \circ r_2 = \{(c,r)\} = r'$ and $r_3 \circ r' = \{ \} = r''$.

In order to allow the direct comparison of specified dependences with the computed ones we introduce a projection operator which deletes all dependences for variables that are not of interest like the internal variable d.

Definition 4 ***(Projection)*** *Given a dependence relations $R \in D$ and a set of ariables $A \subseteq M \cup V$. The projection of R on A written as $\prod A(R)$ is defined as follows:*

$$\prod{}_A(R) = \{ (x,y) \mid (x,y) \in R \;\&\; x \in A\}$$

For example, $\prod_{\{r,c,a,pi\}} (r'')$ *is {a,r), (c, r),(c, pi)}* which is equivalent to the specification.

From here on we assume that the computed dependence relation is always projected onto the variables used within the specification before comparing it with the specification - that is, *A= {x | (x,y) ∈ SPEC* denotes the specified dependences.

Definition 5 ***(Grounded dependence relation)*** *A dependence relation is said to be grounded (or variable-free) if it contains no model variables.*

We assume that all specification are grounded dependencerelations. Thus, we have to compare dependence relations containing model variables with grounded dependence relations. Wepropose a similar solution to that employed in the resolution calculus of first-order logic, namely substitution and finding the most general unifier. However, in contrast to variable substitution in first-order logic, we do not only replace one variable by one term but one model variable by a set of program variables.

Definition 6 ***(Substitution)*** A substitution σ 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 σ on a dependence relation R is a dependence relation where all model-variables x from R have been replaced by σ (x).

For the purpose of finding an efficient algorithm for computing asubstitution that makes a dependence set equivalent to its specification we first map the problem to an equivalent constraint satisfaction problem (CSP). A CSP (Dechter, 1992 & Dechter 2003) comprises variables Vars, their domains *Dom*, and a set of constraints *Cons* that have to be fulfilled when assigning values to the variables. A value assignment that fulfills all constraints is said to be a σ solution of the CSP. Every solution to the corresponding CSP is a valid substitution. For more details about how make use of standard CSP algorithms for computing substitutions. we refer the reader to (Jackson, 1995).

Finally, we are now able to define the equivalence of a dependence set and its grounded specification.

1. Assignmnts:
   $\neg Ab(x = e) \rightarrow D(x = e) = \{(x,v) \mid v \in vars(e)$ where vars is assumed to return all variables which are used in exression *e.*
   $M(x = e)= \{x\}$
   $Ab(x = e) \rightarrow D(x =e) =\{(x,\xi_i)\}$
2. Conditionals:
   $\neg Ab(if\ e\ then\ S_1\ else\ S_2) \rightarrow D(if\ e\ then\ S_1\ else\ S_2) = D(S_1)\ \upsilon\ D(S_2)\ \upsilon\ \{(x,x) \mid x \in (M(S_1)\ \upsilon\ M(S_2))\}\ \upsilon\ ((M(S_1)\ \upsilon\ M(S_2))\ x\ vars(e))$
   $M(if\ e\ then\ S_1\ else\ S_2) = M(S_1)\ \upsilon\ M(S_2)$
   $Ab(if\ e\ then\ S_1\ else\ S_2)\ \ D(if\ e\ then\ S_1\ else\ S_2)= D(S_1)\ \upsilon\ D(S_2)\ \upsilon\ \{(x,x) | x \in (M(S_1)\ \upsilon\ M(S_2))\}\ \upsilon\ ((M(S_1)\ \upsilon\ M(S_2))\ x\ \{\xi_i\})$
3. Loops:
   $\neg Ab(while\ e\ \{S\}) \rightarrow D(while\ e\ \{ S \}) = D(S)^+\ \upsilon\ (M(S)\ x\ vars(e)))\ \bullet\ D(S)^+$
   $M\ (while\ e\ \{S\}) = M(S)$
   $Ab(while\ e\ \{ S \}) \rightarrow D(while\ e\ \{ S \}) = D\ (S)^+\ \upsilon\ \ (M(S)\ X\ \{\xi_i\})\ \bullet\ D(S)^+$
   In the above rules $D(S)^+$ *is the transiive closure of* $D(S)$.
4. No-operation (NOP):
   $D((nop)= I\ where\ I =\{(x,x) \mid x \in V\}$
   $M(nop) = \{ \}$
5. Sequence of statements:
   $D(S_{1;} S_2) = D(S_1) \bullet D(S_2)$
   $M(S_1 ;S_2) = M(S_1)\ \upsilon\ M(S_2)$

Fig. 2. The Verification-Based Model.

Definition 7 ***(Equivalence)*** *A dependence set R is equivalent to its grounded specification S iff there exists a* $\sigma = findSubstitution\ (R,S) \neq \perp$ *and* $\sigma\ (R) = S$.

Formally, it remains to introduce how to extract dependence information from the source code. Figure 2 shows the appropriate rules. In the figure function *D* returns the dependences for a given statement and function M returns the variables employed within a given statement. Moreover, function var returns a given expression's variables. For more details about how to extract dependences we refer the reader to (Jackson, 1995). In Figure 2 we presented rules of of extracting dependences from multiples statements and more details about how to extract dependences from procedures we refer to reader (Soomro., S. 2007).

3. Comparison Between VBM and FDM

In order to compare different models of programs for fault detection and localization, we first introduce the debugging problem formally. Similar to Reiter's definition of a diagnosis problem (Wotawa, 2002) a debugging problem is characterized by the given program and its expected behavior. In contrast to Reiter we assume the existence of a specification that captures the whole expected behavior and not only behavioral instances like given by the set of observations *OBS* in Reiter's original definition.

Definition 8 ***(Debugging Problem)*** *A debugging problem is characterized by a tuple ($\prod$, SPEC) where $\prod$ is a program written in a certain programming language and SPEC is a (formal) specification of the program's intended behavior. The debugging problem now can be separated into three parts:*

1. **Fault Detection**: *Answer the question: Does $\prod$ fulfill SPEC?. In case a program fulfills (does not fulfill) its specifications we write $\prod \upsilon$ SPEC $\nvDash \perp$ ($\prod \upsilon$ SPEC $\models \perp$ respectively).*
2. **Fault Localozation** : *Find the root cause in $\prod$ which explains a behavior not given in SPEC.*
3. **Fault Correction** : *Change the program such that $\prod$ fulfills SPEC.*

Note that SPEC is not required to be a formal specification. It might represent an oracle, i.e., a uman, which is able to give an answer to all questions regarding program $\prod$. In this section we focus on the first two tasks of the debugging problem. Because fault localization and correction can only be performed when identifying a faulty behavior, from here on we assume only situations where ($\prod$, *SPEC*) $\models \perp$. The question now is how such situations can be detected in practice.

The availability of a specification that is able to answer all questions is an assumption which is hardly (and not to say impossible) to fulfill. What we have in practice is a partial specification. Therefore, we are only able to detect a faulty behavior and not to prove correctness. Obviously different kind of specifications may lead to different results to the first task of the debugging problem, i.e., identifying a faulty behavior. In the context of this chapter the question about the satisfiability of $\prod$ *υ SPEC* $\models$ is reduced to checking the satisfiability of two entences, i.e., *FDM*($\prod$) *υ* $SPEC_{FDM} \models \perp$ and *VBM*($\prod$)*υ* $SPEC_{VBM} \models \perp$ where $SPEC_{VBM}$ and $SPEC_{FDM}$ are the partial specification which belong to the FDM and VBM respectively.

```
1.   proc (a,b) {.....
2.   x= a+b;
3.   y = a / b; //instead of y= a * b
4.   assert (y = = a * b)
5.   }...
```

$\neg AB(2) \wedge ok(a) \wedge ok(b) \rightarrow ok(x)$
$\neg AB(3) \wedge ok(a) \wedge ok(b) \rightarrow ok(y)$

$\rightarrow ok(a), \rightarrow ok(b), \rightarrow \neg ok(y)$
$DIAG = \{\{AB(3)\}\}$

$SPEC(proc) = \{(y,a), (y,b)\}$
$dep(proc) = \{(y,a), (y,b)$
$dep(proc) \supseteq SPEC(proc)$
$DIAG = \{ \}$

Fig. 3. Code Snippet, FD model, Specified and Computed Dependences.

The model comparison we present in the following relies on a couple of (reasonable) assumptions. First, for the FDM we need to have a test case judging the correctness of specific variables. In general, finding an appropriate test case revealing a misbehavior w.r.t. specific variables is a difficult task, however, the presence of such a single test case is a requirement for the applicability of the FDM. For the VBM, we assume an underlying assertion language, and a mechanism for deducing dependence specifications from this language. Dependences are further oriented according to last-assigned variables and specified in terms of inputs or input parameters rather than intermediate variables. For simplicity, we further assume that there are no disjunctive post conditions.

In the following we illustrate the introduced models' strength and weaknesses in terms of simple scenarios. In the figures the left hand side is a summary of the FDM model including the observations obtained from running the test case and the left hand side outlines the VBM. For both columns we summarize the obtained diagnosis candidates in terms of the set *DIAG*. Note that we only focus on single-fault diagnosis throughout the following discussion.

Figure 3 outlines a code snippet together with the assertion checking a certain property, the FDM, and the specified computed dependences. Obviously, the VBM is unable to detect and thus localize this specific (functional) fault. In contrast to this, the FDM is able to localize this specific fault. Due to the failed assertion we can conclude that there is something wrong with variable y, thus ¬ ok(y) holds. We also can assume that inputs *a* and *b* are correct, thus the assumptions *ok(a)* and *ok(b)* directly deliver line 3 (AB(3)) as the sole single-fault diagnosis.

```
1.  proc (a,b,x,y) {.....
2.  x= a+b;
3.  x = +2; //instead of  y= x+2
4.  assert (y = = x +2, x = = a+b)
5.  }...
```

$\neg AB(2) \land ok(a) \land ok(b) \rightarrow ok(x')$	$SPEC = \{(y,a), (y,b)(x,a)(x,b)\}$
$\neg AB(3) \land ok(a) \land ok(b) \rightarrow ok(y)$	$dep(proc) = \{(x,a), (x,b)$
$\rightarrow ok(x), \rightarrow ok(a); \rightarrow ok(b)$	$dep(proc) \subseteq SPEC(proc)$
	$\sigma(\xi_2) = \{\}; \ \sigma(\xi_3) = \{\}$
$\rightarrow \neg ok(x''); \rightarrow ok(a), \rightarrow ok(b)$	
$DIAG = \{\{AB(3)\}, \{AB(3)\}\}$	$DIAG = \{\}$

Fig. 4. The misplaced Left-hand Side Variable.

Moreover, as Figure 4 illustrates, although the FDM allows for detecting misplaced left-hand side variables, the VBM cannot localize these kind of faults. Assume that *a=1,b=1,x=2* thus *y=4*. Our assertion suggests to assume the dependences *{(y,a),(y,b),(x,a),(x,b)}*. Both models allow for detecting the fault. When employing the FDM, from the raised assertion we know that ¬ *ok(x)* holds. In order to conclude that the outcome of statement 3 is correct, we need to know that *x* is correct prior to this statement's execution. Thus, to obtain the contradiction we have to assume that both statements are correct.

By reverting the correctness assumption about statement 2 we obviously can remove the

contradiction. Moreover, reverting the assumption about statement 3 also resolves the contradiction. Thus, we obtain two single-fault diagnosis *AB(2)* and *AB(3)*. In contrast to this, since *y* never appears as target variable, we cannot obtain dependences for variable *y* and thus the VBM cannot localize these kind of (structural) faults.

```
1.  proc (a,b,x,y) {.....
2.  x= a+b;
3.  y = x+c+d; //instead of y= x+c
4.  assert (y = = x +c)
5.  }...
```

$\neg AB(2) \land ok\ (a)\ C) \rightarrow ok(x)$
$\neg AB(3) \land ok\ (x) \ \rightarrow ok(c) \land ok(d) \rightarrow ok(y)$
$\rightarrow \neg\ ok(y), \rightarrow ok(a),\ \rightarrow ok(b)$
DIAG = {{AB(2)}, {AB(3)}}

SPEC(proc) = {Y,a),(y,b)(y,c)}
Dep(proc) = {(y, a), (y, b), (y, c), (x, a), (x, b)}
$Dep(proc\,) \supseteq SPEC(proc)$
DIAG = { }

Fig. 5. A Typical (Structure) Fault Inducing Additional Dependences.

The next example points out that the VBM fails in case the fault introduces additional dependences. In Figure 5 we assign $x + c + d$ instead of $x + c$ to the variable *y*. Our assertion indicates that *y* depends upon *x* and *c*, thus *SPEC(proc) = {(y,a),(y,b),(y,c)}*. Computing the program's actual dependences dep(proc), however, yields to {(y,a),(y,b),(y,c),(y,d)} *{(y,a),(y,b),(y,c)}* and thus VBM cannot detect this specific malfunctioning nor locate the misbehavior's cause. By employing the FDM under the assumption $\neg\, ok(y)$ we obtain two single-fault diagnosis *AB(2)* and *AB(3)*.

Figure 6 illustrates an example where the fault manifests itself in inducing less dependences than specified. Our specification is *SPEC(proc) = {(y,a),(y,b),(y,c)}*. Obviously, the computed dependences *{(y,a),(y,b)}* $\not\trianglerighteq$ *SPEC(proc)*. As the figure outlines, we obtain two single-fault diagnosis candidates, *AB(2)* and *AB(3)*. In this case, the FDM is also capable of delivering the misbehavior's real cause, it returns two single-fault diagnosis candidates: *AB(2)* and *AB(3)*.

The Author (Stumptner, 2001) shows that localizing structural faults requires exploiting design information like assertions, and pre- and post conditions. Again, we outline this in terms of a few small examples. Although the previous examples show that the VBM cannot detect neither locate certain types of faults, it may provide reasonable results in capturing structural faults.

Our final example in Figure 7 illustrates that both approaches might deliver reasonable but different results. We assume *a=1,b=1,e=0*, thus we expect *z=2* and *d=0*. However, due to the introduced fault, we obtain *z=1 and d=0*. Since the value of *z* is incorrect, but *d=0,* we conclude that ┐*ok(z)* and *ok(d)* holds. Thus, we btain *AB(2)* and *AB(4)* as diagnosis candidates. Note that this result primarily roots in the coincidental correctness of variable *d*.

```
1.  proc (a,b,c) {.....
2.  x= a+b;
3.  y = x;  //instead of  y= x+c
4.  assert (y = = a+b +c)
5.  }...
6.
```

$\neg AB(2) \wedge ok(a) \wedge ok(b) \rightarrow ok(x)$	$SPEC = \{(y,a), (y,b)(y,c)\}$
$\neg AB(3) \wedge ok(a) \rightarrow ok(y)$	$dep(proc) = \{(y,a), (y,b)\}$
$\rightarrow \neg ok(y), \rightarrow ok(a), \rightarrow ok(b)$	$dep(proc) \not\supseteq SPEC(proc)$
	$\sigma(\xi_2) = \{a,b,c\}, \sigma(\xi_3) = \{a,b,c\}$
$DIAG = \{\{AB(2)\}, \{AB(3)\}\}$	$DIAG = \{\{AB(2)\}, \{AB(3)\}\}$

Fig. 6. A Typical (Structural) Fault Inducing Fewer Dependences than Specified

Given the assertion in Figure 7 we are aware of the dependences *{(d,a),(d,b),(d,e),(z,a) ,(z,b), (z,e)}*. As the figure outlines, we obtain two single-fault diagnosis *AB(2)* and *AB(3)*. As is also indicated in the figure, when solely employing a single assertion requiring *z == c + d*, we obtain *SPEC'(proc)*={ (z,a),(z,b),(z,e)*}* and *dep'(proc)* ⊇ *SPEC'(proc)*. Consequently, we obtain 3 diagnoses *(AB(2), AB(3)* and *AB(4))* in this case. However, even when employing the FDM we cannot exclude a single statement, thus, in this specific case, both models deliver the same accuracy.

The examples outlined above should have made clear that a comparison of both models in terms of their diagnostic capabilities inherently depends on how we deduce observations from violated properties. Note that the FDM itself cannot detect any faults, rather faults are detected by evaluation of the assertions on the values obtained from a concrete test run.

The VBM can reliably detect and localize faults that manifest in missing dependences on the right-hand side of an assignment statement. Due to the over-approximation of dependences we cannot locate faults manifesting in additional dependences as it is impossible to distinguish if (1) the specification is incomplete, (2) the model computes spurious dependences, or (3) an unwanted dependence is present due to a fault.

Table 1 summarizes the illustrated examples by listing the individual models' fault detection and localization capabilities. For those examples, where both models deliver diagnosis candidates, we checked whether the diagnoses provided by the VBM are a subset of those provided by the *FDM*.

In comparing both models, we start by contrasting the well-known artifacts in the area of MBSD. Table 2 summarizes the most notable differences in employing the VBM and FDM for fault localization. In both models we employ a partial specification (e.g. test case, assertion, invariant) for deducing a number of observations. Whereas the VBM encodes observations in terms of dependence relations, the FDM relies on a program's execution and subsequent classification of the observed variables. Variables are merely classified as being correct or incorrect with respect to a given (partial) specification.

```
1.  proc (a,b,c) {.....
2.  c= a;// should be c=a+b
3.  d=c * e;
4.  z =c+d;
5.  assert (z = =c+d, [d = = c * e])
6.  }...
```

$\neg AB(2) \wedge ok(a) \rightarrow ok(c)$
$\neg AB(3) \wedge ok(c) \wedge ok(e) \rightarrow \neg ok(d)$
$[\rightarrow \neg ok(d]), \rightarrow \neg ok(z)$
$DIAG = \{\{AB(2)\}, \{AB(4)\}\}$
$DIAG' = \{\{AB(2)\}, \{AB(3)\},\{AB(4)\}\}$

$SPEC(proc) = \{(z,a), (z,b), (z,e), (d,a), (d,b), (d,e)\}$
$dep(proc) = \{(z,a), (z,e), (d,a), (d,e)\}$
$dep(proc) \not\supseteq SPEC(proc)$
$\sigma(\xi_2) = \{a,b\}, \sigma(\xi_3) = \{a,b,e\}, \sigma(\xi_4) = \{\ \}$
$DIAG = \{\{AB(2)\}, \{AB(3)\}\}$
$SPEC'(proc) = \{(z,a), (z,b), (z,e)\}$
$dep'(proc)=\{(z,a),(z,e)\}$
$dep'(proc) \not\supseteq SPEC'(proc)$
$\sigma(\xi'_2) = \{a,b\}, \sigma(\xi'_3) = \{a,b,e\}, \sigma\ \sigma(\xi_4) = \{a,b,e\}$

Fig. 7. A Degenerated Example (Error Masking), Diags (FDM) (symbol) diags(VBM)

Example	FDM loc	VMB		Diags (FDM) $\supseteq$ diags (VBM)
		det.	loc	
Fig. 3	✓	X	X	-
Fig. 4	✓	X	X	-
Fig. 5	x	✓	✓	-
Fig. 6	✓	✓	✓	✓
Fig. 7	✓	✓	✓	x

Table 1. Summary on the Outline Scenarios

Furthermore, the VBM models the program in terms of functions over ependence relations, the FDM captures the programs behavior by a number of logical sentences, in particular we employ a Horn clause theory. The VBM detects a fault by checking whether the system description fulfills the given specification. In case this relationship does not hold, a fault has been detected. In contrast, we detect a fault with the FDM if the system description together with the specification yields to logical contradiction.

artifact	VBM	FDM
observatuibs	dependence relations	Ok, ¬ (ok)
systemn descr	function over dependences relation VBM (∏)	Horn clauses FDM(∏)
fault detect.	VBM (∏) $\not\supseteq$ SPEC	-
fault localiz.	VBM (∏) $\supseteq$ SPEC	FDM(∏) υ Spec ≠ ⊥
assumptions	Varible substitution ξ = …	¬ AB
theorem prover	CSP solver	Horn clauses theorem prover
structureal faults	detect., localiz,	detect, localiz.
functional faults	no detect., no localiz	detect., localiz.

Table 2. Comparing the Most Common Artifacts

The VBM locates possible causes for detected misbehavior by assuming that specific statements depend on model variables, and checking whether there is a valid substitution. As authros outlined in (Peischl et al, 2006), this process is an efficiently done by solving a CSP. Instead, the FDM employs a Horn clause theorem prover under the assumption of statement abnormality in computing diagnosis candidates. Note, that whereas the FDM does not assume any faulty behavior for specific statements, the VBM assumes specific dependences guided by the specification.

As indicated by the example above, the VBM is tailored towards detection and localization of structural faults, whereas the FDM may capture structural but particularly functional faults. Similar to static slicing capturing control as well as data flow dependences, the FDM must comprise all statements responsible for the computation of an erroneous variable. Thus, the FDM always provides diagnosis candidates under presence of an erroneous variable. The author (Wotawa, 2002) points out that the FDM delivers at least the same results as static slicing. Moreover, we know that the misbehavior's real cause is always among the delivered diagnosis candidates when employing the FDM. This perspective is supported by theoretical foundation (Friedrich et al, 1999) as well as practical evidence in numerous case studies.

Particularly, a comparison w.r.t. the accuracy and completeness of the obtained diagnosis is of interest. Figure 8 summarizes the relationship of the FDM and the VBM regarding their abilities of checking satisfiability. The lines between the nodes building up the lattice denote a subset relationship. As illustrated by the examples, there are debugging problems where the VBM allows for finding a discrepancy but the FDM does not and vice versa.

4. Case Studies

Authors (Peischl et al, 2006) present first experimental results indicating the approaches' applicability. The results presented there solely stem from small programs. In evaluating the model's fault localization capabilities under presence of procedural abstraction, we decompose a program into several procedures in a step by step fashion. This procedure allows for a first evaluation of both, the model for (1) parameter passing and (2) handling of return values.

Table 3 summarizes our most recent results. Specifically, the program *eval* evaluates the arithmetic expression *z ←(r x h)+(c /d) - (d+h) x (e+f)*. The specification says that the left-hand side *z* depends on the variables *r,h,c,d,e*, and *f*. We introduced a single structural fault and decomposed this program by adding procedures computing specific subexpressions in a step by step fashion. A specific subexpression is thus evaluated by a single procedure and replaced by the variable capturing this procedure's evaluation. We refer to the decomposed programs comprising *i* ethods by *eval(i)*. In the remaining programs, which perform simple computations like taxes or evaluate simple arithmetic expressions, we also introduced a single structural fault.

Removing certain dependences from the specification allows for evaluating our odel's capabilities in localizing structural faults under presence of partial knowledge of the dependences of the output variables. Thus, we observed a subset of the output dependences involving up to 5 variables and recorded the minimum and maximum number of diagnosis candidates.

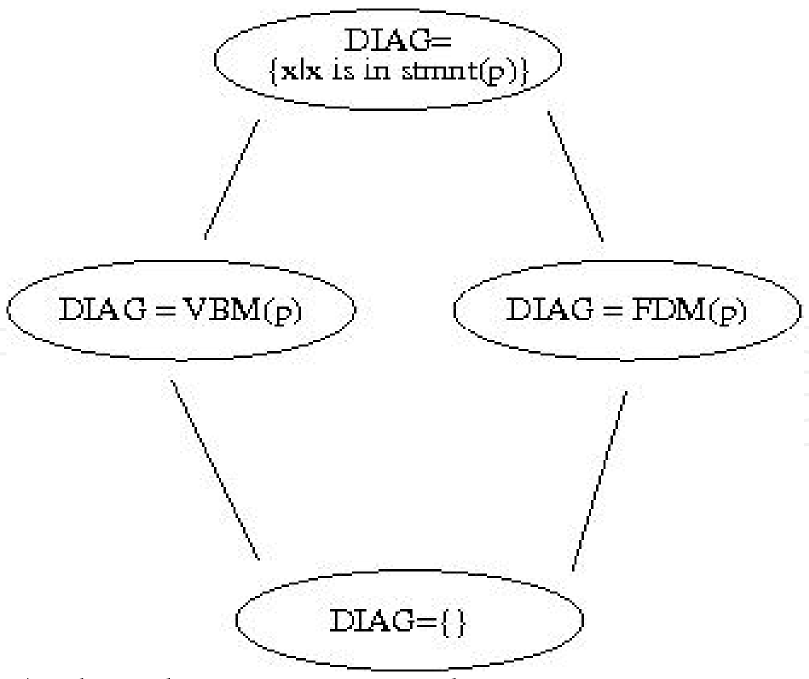

Fig. 8. The (Open) Relationship Between VBM and FDM

Method no	LOC	Total dep.no	Min-Max no. Diagnosis Candidates				
			5	4	3	2	1
eval (1)	10	9	-	-	-	-	4
eval (2)	14	10	-	-	-	4	4-11
eval (3)	18	11	-	-	4	4-13	4-18
eval (4)	22	12	-	4	4-15	4-22	4-22
eval (5)	26	13	4	4-17	4-26	4-26	4-26
sum	22	11	-	-	4	4-13	4-18
arithmetic	26	12	-	4	4-15	4-15	4-22
tax comp.	30	13	4	4-17	4-26	4-26	4-26
calculator	40	12	1-31	1-31	1-33	1-34	1-34

Table 3. Number of single-Faults Diagnosis with Decreasing Number of Specified Output Variables.

For example, regarding the program eval(3) we obtained 4 diagnosis candidates when observing all outputs. Afterwards we selected 2 output variables out of the 3 output variables, and for all possible combinations of selecting 2 out of 3 outputs, we recorded the number of diagnoses. The table specifies the minimal and maximal number of diagnosis candidates obtained in this way (in this specific case of considering 2 output variableswe obtain at least 4 and at most 13 diagnosis andidates). We checked whether or not the introduced faults appear among the delivered diagnosis candidates. Regarding all our experiments, we have been able to locate the misbehavior's real cause.

Furthermore, the table lists the number of total dependences (column 3) and the program's size in terms of the lines of code (column 2). Our experiments indicate an increase in the number of candidates with a decreasing number of outputs being considered. In the table, we did not take into account cases where the reduced output dependences are not capable of detecting the fault. In this case our approach obviously returns *{ }*. In summary, the obtained results, confirm the findings in (Hamscher & Davis 1984): As our problem becomes under-constrained by removing certain output dependences, the number of diagnosis candidates may increase drastically. As our experiments indicate, this also appears to hold for the novel model introduced herein.

5. Related Research

We present related work which provides an overview of related research of our chapter.

5.1 Model-Based Software Debugging

Model-based diagnosis(MBD) is a well-known Artificial Intelligence(AI) Technique for the localization and malfunctioning parts in (mostly physical) systems. The definitions of MBD as given in (Reiter, 1987; Kleer & Williams 1987) and show how this approach can be used to locate faulty components in a given system. First-order logic produced formal model for MBD and covers sound framework. Test cases are uses to examine the specifications and diagnostic engine locates single or multiple faults from derived system. Several authors (Hamscher, W.C 1991, Greiner, R et. al. 1989) provides the well-founded theory underlying

Model-Based Diagnosis (MBD). Traditionally MBD focuses on diagnosing physical systems (Console, L. et. al. 1993, Cascio., F. et. al. 1999, Williams., B.C. 1996 & Malik., A. 1996), however several authors (Console, L. et. al. 1993, Bond., G. W. 1994, Stumptner., M & Wotawa., F. 1999a, Stumptner., M & Wotawa., F. 1999b, Stumptner., M & Wotawa., F.1999c & Stumptner., M & Wotawa., F. 1999d) employing model-based diagnosis techniques in software debugging.

Authors (Stumptner., M & Wotawa., F. 1999a & Stumptner., M & Wotawa., F. 1999b) covers model-based software debugging of functional programs and the other authors of these publications (Mayer., W. & Stumptner., M. 2003, Mayer., W. et. al 2002a & Wieland., D. 2001) focus on software debugging of Java programs, particularly objected-oriented programming features. These authors work together ***JADE*** project to develop functional dependency model and value based model for software debugging. The comparison between two models are presented in (Mayer., W. et. al 2002a, Mayer., W. et. al 2002b, Stumptner., M. et. al. 2001a & Stumptner., M. et. al. 2001b) where they discussed capabilities of both models in respect with advantages and weaknesses.

The work described in this chapter is solely based on diagnosis from first principle (Reiter, 1987). According to Reiter's (Reiter, 1987) a precise theoretical foundation of diagnostic reasoning from first principle will be a essential ingredient in any common theory of diagnostic reasoning.Applying theory of model-based diagnosis to software debugging first requires an adaption of the diagnosis theory. We wrote some articles (Wotawa., F. & Soomro., S. 2005, Peischl., B., et. al. 2005, Soomro., S. 2007., Soomro., S. 2008, ., Soomro., S. et. al. 2008; Soomro., S. & Wotawa., F. 2009) for software debugging to localize faults using model-based diagnosis technique.

5.2 Aspect System: Abstract Dependencies

Aspect is a static analysis technique for detecting bugs in imperative programs, consisting of an annotation language and a checking tool. This system introduced by (Jackson, 1995) which is based on the abstract dependencies that are used by the Aspect system (Jackson, 1995) for detecting faults.The Aspect system analysis the dependences between variables of a given program and compares them with the specified dependences.The computed dependencies are compared to expected ones specified by software engineer. After comparing these dependencies with derived ones this system pinpoint the missing dependencies. This work employs abstract dependences for detecting rather than for localizing a fault.Aspect system (Jackson, 1995) which has been used for dependency based verification of C programs. Aspect system provides a very simple way to specify and verify properties of software system, even though the results of this approach is limited,as the correctness of program is hardly expressable due to terms of variable dependences.

The work described in this thesis 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 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.

5.3 Program Slicing

The author (Weiser, 1982 & 1984) introduces program slicing towards debugging systems. Program slicing is an approach for reduce to complexity of programs by focusing on selected aspects of semantics. The process of slicing deletes those parts of the program which can be determined to have no effect upon the semantics of interest. Slicing has applications in testing and debugging, re-engineering, program comprehension and software measurement. The author (Tip, 1995) scrutinize various approaches of slicing with respect with static and dynamic slicing.

Program slicing is a general, widely-used, and accepted technique applicable to different software engineering tasks including debugging, whereas model-based diagnosis is an AI (Artificial Intelligence) technique originally developed for finding faults in physical systems. During the last years it has been shown that model-based diagnosis can be used for software debugging (Console, L. et. al. 1993, Bond., G. W. 1994, Stumptner., M & Wotawa., F. 1999a, Stumptner., M & Wotawa., F. 1999b, Stumptner., M & Wotawa., F.1999c & Stumptner., M.) The author (Wotawa, 2002) clarifies that model-based diagnosis is related to program slicing. In case of diagnosis it proves that slices of a program in a fault situation are equivalent to conflicts in model-based debugging. This result helps debugging community to compute diagnosis faster and gives more information about possible bugs in certain situations.

6. Conclusion and Future Research

In this chapter we focused on verification-based model (VBM) specifically tailored towards detecting and localizing structural faults. We discussed the relationship between this model and the well-known functional dependence model (FDM) by exemplifying the weaknesses and strengths of both models. Our examples show, that there are debugging problems where the verification-based model (VBM) delivers different diagnoses than the functional-dependence model (FDM) and vice versa. Furthermore, we present case studies conducted recently. Notably, whenever our novel model detects a structural fault, it also appears to be capable of localizing the misbehavior’s real cause.

A future research challenge is the empirical evaluation of the modeling approaches discussed herein. Also include another extensions of the Verifcation Based Model, e.g, to handle object oriented features and to provide an emprical anaylsis. Another open issued which is the connection with program slicing that is also based on abstract dependences.

7. References

Bond., G. W. (1994). Logic Programs for Consistency Based Diagnosis. *Phd. Thesis Dissertation,* Carleton University, Faculty of Engineering, Ottawa, Canada.

Cascio., F., Console., L, Guagliumi., M, Osella., M., Panato., A., Cauvin., S., Cordier. M., Theseider, D & Sottano. S. (1999) . Generating on-board diagnostics of dynamic automotive systems based on qualitative models. *Artificial Intelliegence Communication,* Vol., No. 12, 1999.

Cauvin., S., Cordier. M. & Dousson.C (1998). Monitoring and Alarm Interpretation in Industrial Environments. *Artificial Intelliegence Communication,* Vol., No. 11, 1998.

Console, L., Friedrich, G. & Theseider, D. (1993). Model-Based Diagnosis Meets Error Diagnosis in Logic Programs, *Proceedings of Joint International Conference on Artifical Intelligence,* pp. 1494-1499, Chambery, August and 1993.

Greiner, R., Smith, B & Ralph, W. (1989). A Correction to the Algorithm in Reiter's Theory of Diagnosis. *Artificial Intelliegence,* Vol., No., 41, pp. 79-88.

Hamscher, W.C (1991). Modeling Digital Circuits for Troubleshooting. *Artificial Intelliegence,* Vol., No., 51 (October & 1995) pp. 223-271.

Hamscher, W.C; Console, L. & Kleer, J. (1992). Readings in Model-Based Diagnosis. *Morgan Pulisher,* (October & 1992).

Jackson, D. (1995). Aspect Detecting Bugs with Abstract Dependences. *Transaction on Software Engineering and Mrthodology,* Vol., No., 3, (April & 1995) pp. 109-145.

Kleer, J. & Williams, B C (1987). Diagnosing multiple faults. *Artificial Intelligence*, pp. 97–130.

Malik., A., Struss., P. & Sachenbacher., M. (1996). Case Studies in Model-based Diagnosis and Fault Analysis of Car-Subsystems. *Proceedings of European Conference ofArtifical Intelligence.* Pp. 1322-1334. 1996.

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

Mayer., W., Stumptner., M., Wieland., D. & Wotawa., F (2002a). Can AI help to improve debugging substantially? Debugging experiences with value-based models. *Proceedings of European Conference ofArtifical Intelligence.* Pp. 417-421. IOS Press, Lyon, France 2002.

Mayer., W., Stumptner., M., Wieland., D. & Wotawa., F (2002b). Towards an Integrated Debugging Environment *Proceedings of European Conference ofArtifical Intelligence.* Pp. 422-426. IOS Press, Lyon, France 2002.

Peischl., B., Soomro., S. & Wotawa., F. (2006). Towards Lightweight Fault Localization in Procedural Programs. *Proceedings of the 19th Conference on International Conference on Industrial, Engineering and Applications of Applied Intelligent Systems (IEA/AIE).* Lecture Notes in Compter Science, Springer 2006.

Reiter, R. (1987). A theory of diagnosis from first principles. *Artificial Intelligence*, pp. 57–95.

Stumptner., M & Wotawa., F. (1999a). Debugging Functional Programs. *Proceedings of Joint International Conference on Artifical Intelligence,* pp. 1074-1097, August, Stockholm, Sweden, 1999.

Stumptner., M & Wotawa., F. (1999b). Reconfiguration Using Model-Based Diagnosis. *Proceedings of Diagnosis Worksshop Series* , Loch Awe, Scotland.

Stumptner., M & Wotawa., F. (1999c). Detecting and Locating Faults in Hardware Designs. *Proceeding AAAI 99 Workshop on Intelligent Software Engineering,* Orlando, Florida.

Stumptner., M & Wotawa., F. (1999d). Jade -- Java Diagnosis Experiments -- Status and Outlook. *IJCAI '99 Workshop on Qualitative and Model Based Reasoning for Complex Systems and their Control* August, Stockholm,Sweden, 1999.

Stumptner., M., Wieland., D. & Wotawa., F. (2001a). Analysis Models of Software Debugging.*Proceedings of Diagnosis Worksshop Series ,* Sansicario, Italy.

Stumptner., M., Wieland., D. & Wotawa., F. (2001b). Comparing Two Models for Software Debugging. *Proceedings of the Joint German/Austrian Conference on Artificial Intelligence (KI),* Vienna, Austria.

Soomro., S. (2007). Using Abstract Dependences to localize faults from procedural Programs *Proceeding of Artificial Intelligence and Applications*. pp. 180-185, Inssbruck, Austria

Soomro., S., Shah. A.A & Shah., Wajiha (2008). Localize Faults from ALias-free Programs using Verification Based Model. *Proceeding of Artificial Intelligence and Soft Computing ASC 2008.* pp. 190-196, Palma-DelMalorca, Spain. 2008.

Soomro., S. (2008). Verification Based Model Localizes Faults from Procedural Programs. *Frontiers in Robotics and Automation Control* International Book of Advanced Robotic System. ISBN 978-953-7619-17-6, October 2008.

Soomro. S. & Wotawa. F. (2009). Detect and Localize Faults in Alias-free Programs using Specification Knowledge. *Proceedings of the 19th Conference on International Conference on Industrial, Engineering and Applications of Applied Intelligent Systems (IEA/AIE).* Lecture Notes in Compter Science, Springer, Taina,, Taiwan 2009.

Tip, F. (1995). A Survey of Program Slicing Techniques. *Journal of Programming Languages,* Vol., No., 3, (September & 1995) pp. 121-189.

Wieland., D. (2001). Model Based Debugging of Java Programs Using Dependencies. *Phd. Thesis Dissertation*, Vienna University of Technology, Computer Science Department, Institute of Information Systems (184), Database and Artificial Intelligence Group (184/2) Austria , November 2001.

Williams., B.C. & Nayak., P.P. (1996). Immobile Robots -- AI in the New Millennium. *Artificial Intelligence Magzine,* AAAI Press, pp. 16-35.

Weiser. M. (1982). Programmers Use Slices when Debugging. *Communication of The ACM,* Vol. No. (7), pp. 446-452. July 1982.

Weiser. M. (1984). Programmers Use Slices when Debugging. *IEEE Transaction of Software Engineering,* Vol. No. (4) , pp. 352-357. July 1984.

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