On the relationship between model-based debugging and program slicing

Franz Wotawa [a]

[a] *Technische Universität Graz, Institut für Informationsverarbeitung und Computergestützte Neue Medien, Institut for Software Technology, Inffeldgasse 16b/2, A-8010 Graz, Austria, wotawa@ist.tu-graz.ac.at*

Abstract

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 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. In this paper we discuss the relationship between debugging using a dependency-based model and program slicing. As a result we obtain that slices of a program in a fault situation are equivalent to conflicts in model-based debugging.

Key words: program slicing, model-based diagnosis, debugging

1 Introduction

Program slicing [24,25] is a well-known technique for debugging. If we have a specific program, a location within this program, and a set of variables, then a slice is itself a program that is obtained from the original program by removing all statements that have no influence on the given variables at the specified position. Since slices are usually smaller than the original programs they focus the user's attention on relevant parts of the program during debugging. Beside slicing there are many other debugging techniques proposed so far, e.g., algorithmic software debugging [20,9], probability-based methods [4,5], and other dependency-based methods [18]. However, in this paper we focus only on the use of model-based diagnosis for debugging as introduced and described in [6,11,22]. Moreover, we consider only a dependency-based model similar to the one used by Friedrich et al. [11].

The objective of this paper is to show the relationship between model-based debugging using a dependency model and program slicing. The equivalence of slices and *program* ::= **procedure** *Id* { { *statement* }* }

statement ::= *assignment* | *conditional* | *while*

assignment ::= *Id* = *expr* ;

conditional ::= **if** (*expr*) { { *statement* }* } [{ { *statement* }* }]

while ::= **while**(*expr*){ { *statement* }* }

expr ::= *Id* | *Const* | (*expr op expr*) | (*mop expr*)

op ::= + | - | * | / | $\leq$ | $\geq$ | <= | >= | == | **and** | **or** | ...

mop ::= + | - | **not** | ...

Fig. 1. Syntax of the language L

conflict sets used in model-based diagnosis for computing diagnoses is the main result of this paper. This result has two implications. The first implication is that slices for more than one variable may contain too many statements. Some of the statements can be responsible for all discrepancies but others cannot. In this sense a slice for several variables is too pessimistic. The second implication is due to the computation of diagnoses from a dependency model. Using the result we do not need to compute an explicit logical model from the program dependencies. Instead we use the slices for every discrepancy as a conflict set. The obtained conflict sets are then used directly by a hitting set algorithm in order to compute diagnoses as described by Reiter [19]. Since at least good approximations of slices for single variables can be computed at compile time, diagnosis time is only bound by the hitting set algorithm. This result directly helps to improve the debugging performance whenever using a model-based diagnosis approach with a dependency model. The fact that only approximations of minimal slices can be computed is not a problem. As a consequence maybe too many diagnoses are computed which can be removed during debugging by using more observations, e.g., the knowledge of variable values at specific program locations. Moreover, it is well known that only approximations of dependencies between variables exist [15].

In this paper we use a small sequential programming language L which helps to focus on the important issues and avoids considering unnecessary detail. All programs used in this paper are written in L. Figure 1 depicts the syntax definition of L. The semantics of L is assumed to be equal to the semantics of any sequential imperative language like Pascal or C. Without restricting generality L contains three kinds of statements: the assignment statement, the conditional statement, and the while statement. Procedure calls are not allowed.

In the first two sections of this paper we recall the basic definitions of program slicing and model-based diagnosis, briefly show algorithms for computing slices and diagnoses. The next section introduces an algorithm for computing models of programs, discusses some examples, and states our main theorem. Afterwards we discuss the implications of our main theorem and show how model-based debugging relates to some extensions of slicing. Finally we conclude the paper.

2 Program Slicing

According to Weiser [25] a slice is defined for a program Π and a slicing criterion (n, V), where n is a line within Π and V is a set of variables.

Definition 1 (Slice) *A slice S of a program Π on a slicing criterion $C = (n, V)$ is any program with the following properties:*

(1) S can be obtained from Π by deleting zero or more statements from Π.
(2) Whenever Π halts on an input I, then S also halts on input I. The resulting values of variables $v \in V$ at position n must be equivalent for both programs.

This definition is not complete because it does not guarantee that a slice always contains the line with number n. To avoid this problem we assume that line n is always the last line in Π and always element of the slice, and that the line contains no statements. A minimal slice is a slice that cannot be further reduced by removing statements without violating the definition. Weiser [25] showed that computing minimal slices at compile-time is equivalent to the halting problem and therefore undecidable. However, he showed that it is possible to find a good approximation to a minimal slice.

There are several algorithms for computing slices. Danicic et al. [7] present a parallel algorithm, and Horwitz at al. [14] introduce an algorithm for computing interprocedural slices from a system dependency graph which is an extension of a program dependency graph. For the language L only program dependency graphs have to be used for computing slices. To be self contained we briefly examine how slices are computed from dependency graphs as given in [14].

A program dependency graph G_Π for a program Π is a directed graph. The vertices of G_Π represent assignment statements and control predicates that occur in program Π. In addition G_Π includes the distinguished *entry vertex*. The edges of the graph represent either control or data dependencies. The following informal definition of control dependencies is taken from Horwitz et al. [14]. *A control dependency edge is labeled either* true *or* false*, and the source of the edge is either a predicate vertex or the entry vertex. A control dependency edge between two edges v_1 and v_2 means that during execution, whenever the predicate represented by v_1 is evaluated and its value matches the label of the edge to v_2, then the program component represented by v_2 will be executed (although perhaps not immediately).* [10] introduces a method for computing dependency graphs from arbitrary programs. For the language L the control dependency edges correspond exactly to the nested structure of a program, i.e., a vertex p controls a vertex x if p is a predicate for an if or while

```
1.  procedure Main1 {
2.    sum = 0;
3.    i = 0;
4.    while (i < 10) {
5.      sum = sum + i;
6.      i = i + 1;
7.    } }
```

Fig. 2. The L program *Main1* and its dependency graph

statement S and x is in the body of the statement S.

In this paper we only use one kind of a data dependency edge, representing flow dependencies. Other kinds of data dependencies are not relevant for slicing. The dependency graph G_Π contains an edge from vertex v_1 to vertex v_2 iff the following rules hold:

(1) v_1 defines a variable x.
(2) v_2 uses the variable x.
(3) Control can reach v_2 after v_1 via an execution path that contains no vertex that defines x.

Figure 2 and Figure 3 depict two programs together with their program dependency graphs. The first program *Main1* sums up the numbers from zero to nine. The result is stored in the variable *sum*. The second program *Main2* computes the area and the circumference of a circle. The area is stored in *a* and the circumference in *s*. We will use the programs as our examples explaining the underlying theory and algorithms.

Program slices for a program Π on a slicing criterion $(n, \{x\})$ can be obtained from the dependency graph G_Π. In the first step the vertex v representing the last program position before n where variable x is defined must be localized. In the second step the algorithm collects all vertices that can reach v via a control or flow dependency edge. The statements represented by the collected vertices (including vertex v) are equal to the program slice for Π. According to Horwitz et al. [14] the slice extraction from dependency graphs can be extended to a set of variables. The slice for a set of variables V can be computed using the equation: $(n, V) = \bigcup_{x \in V}(n, \{x\})$.

We illustrate slicing using the *Main1* program and its dependency graph from Figure 2. The slice on $(7, \{i\})$ corresponds to the set of vertices lying on a path from the entry vertex to vertex 6. This set comprises 4 vertices: 6, 4, 3, and the entry vertex. The program slice corresponding to the vertices is given below:

```
1.  procedure Main1 {
```

```
1.  procedure Main2 {
2.    pi = 3.14;
3.    s = 2*r*pi;
4.    a = r*r*pi;
5.    }
```

Fig. 3. The L program *Main2* and its dependency graph

```
3.    i = 0;
4.    while (i < 10) {
6.      i = i + 1;
7.    } }
```

Note that line 7 (although it is not part of the dependency graph) is an element of the slice on $(7, \{i\})$ as assumed previously.

3 Model-based debugging

Model-based diagnosis [19,8] is a well known technique for locating malfunctioning devices in (physical) systems. The basic idea of model-based diagnosis is to have a logical description of a system, i.e., the model, and a set of observations of the system. The used model must be composed of components, that might be responsible for a misbehavior. A misbehavior is detected whenever the observed behavior contradicts the behavior that was derived directly from the model. As an advantage diagnosis is performed without explicit knowledge about how to locate faulty components. In addition, the model is only required to represent the correct behavior of components and the correct behavior of systems. In our context, i.e., finding bugs in programs at the level of statements, the components represent the statements. In this paper we assign every statement a unique number, i.e., the line number. This number is also used to identify a diagnosis component throughout this paper.

To be self contained we briefly recapitulate the basic definitions of model-based diagnosis given by [19]. In the following SD is a logical model describing the behavior of a system, i.e., the system description, $COMP$ a set of components, and OBS a set of observations. We further assume SD and OBS to be first-order-logic theories. The system description makes use of the predicate $AB(C)$ ($\neg AB(C)$) to specify the incorrectness (correctness) of a component C. The term $\neg AB(C)$ says that component C behaves correctly. In the context of software debugging this means that the statement in line C does not necessarily contain a bug. $AB(C)$ says that line C behaves faulty.

Definition 2 (Diagnosis, Reiter [19]) *Let $(SD, COMP)$ be a system and OBS a set of observations. A set $\Delta \subseteq COMP$ is a diagnosis iff*

$$SD \cup OBS \cup \{\neg AB(C) | C \in COMP \setminus \Delta\} \cup \{AB(C) | C \in \Delta\}$$

is satisfiable.

A diagnosis is said to be minimal if no proper subset is a diagnosis. From definition 2 follows that every superset of a diagnosis is a diagnosis. For example, if the program computes a wrong value for at least one variable and the assumption that statement 1 is incorrect leads to a consistent state, then the assumption that statements 1 and 2 are incorrect must also lead to consistency. Hence, the diagnosis $\{1\}$ is minimal whereas $\{1, 2\}$ is not.

The dual concept of a diagnosis which is used for computing diagnoses is a conflict.

Definition 3 (Conflict, Reiter [19]) *Let $(SD, COMP)$ be a system and OBS a set of observations. A set $CO \subseteq COMP$ is a conflict, iff*

$$SD \cup OBS \cup \{\neg AB(C) | C \in CO\}$$

is contradictory.

The computation of diagnoses from conflicts makes use of the concept of hitting sets.

Definition 4 (Hitting Set, Reiter [19]) *Let C be a collection of sets. A hitting set for C is a set $H \subseteq \bigcup_{S \in C} S$ such that $H \cap S \neq \emptyset$ for each $S \in C$. A hitting set is minimal if no proper subset of it is a hitting set.*

For example, all minimal hitting sets for $\{\{1, 2\}, \{1, 4\}\}$ are $\{1\}$ and $\{2, 4\}$. $\{1, 2, 4\}$ is also a hitting set but is not minimal. In [19] Reiter introduces the hitting set algorithm for computing diagnoses using a set of conflicts. This algorithm was improved by [12]. The relationship between diagnoses and conflicts is stated by the following theorem:

Theorem 5 (Reiter [19]) *The set $\Delta \subseteq COMP$ is a (minimal) diagnosis for $(SD, COMP, OBS)$ iff Δ is a (minimal) hitting set for the collection of conflict sets.*

Whereas model-based diagnosis research mainly focuses on modeling physical systems, several authors [6,2,3,11,22] have proposed the use of model-based techniques in software debugging. Console et al. [6] introduce a model for debugging Prolog-like languages. The authors claim that their approach improves Shapiro's algorithmic debugging [20] by reducing the required user-interaction necessary for locating a bug. Bond et al. [2,3] critically analyze the work done by Console et al. [6] and show that the exception form for diagnoses is not canonical, leading to an incomplete diagnosis computation procedure. To overcome this problem Bond et al. propose an improved algorithm for debugging which also generalizes the declarative error diagnosis approach from Shapiro [20]. Friedrich et al. [11] introduce a system for debugging hardware designs written in the hardware description language VHDL. The authors use a dependency-based model for debugging. Because of the simplicity of the model a prototype implementation is able to debug even very large programs. Stumptner and Wotawa [22] discuss the use of model-based diagnosis in debugging more theoretically. In this paper the authors present different models for a small functional language and some of the models' properties.

The basic idea behind using model-based diagnosis for debugging is to derive a model directly from the program and the programming language semantics. This model has to distinguish components, describe their behavior, and the structure of the program under examination. Although the objective of this paper is not to introduce a value-based model representing the whole semantics of L, we show how such a model for the following program would look like and how it could be used for debugging.

1. **procedure** Demo {
2. S = A **and** B;
3. D = **not** S;
4. E = S **or** C;
5. }

For modeling program *Demo* we assume that every statement is mapped to a diagnosis component of $COMP = \{2, 3, 4\}$. It easily can be seen that the following model SD really models the behavior of *Demo* given by the semantics of the language L. The model makes use of predicates $in_i(C, V)$ and $out(C, V)$ which are related to ports of components. The predicate $in_i(C, V)$ says that variable V is accessed in statement C, or that V is the ith input of C. The predicate $out(C, V)$ states that variable V is changed in statement C, or that V is the output of C.

$$\neg AB(2) \rightarrow out(2, S) = in_1(2, A) \textit{ and } in_2(2, B)$$
$$\neg AB(3) \rightarrow out(3, D) = \textit{not } in(3, S)$$
$$\neg AB(4) \rightarrow out(4, E) = in_1(4, S) \textit{ or } in_2(4, C)$$

The structural knowledge obtained from program *Demo* comprises only two facts:

$$out(2, S) = in(3, S) \wedge out(3, S) = in_1(4, S)$$

Without observations (represented as test-cases in the software domain) it does not make sense to search for a bug. Therefore, we assume $A = true$, $B = true$, $C = false$ before executing *Demo*, and $D = false$ and $E = false$ afterwards as our test-case. Obviously *Demo* must be buggy. Executing *Demo* using the given input results in $E = true$ contradicting the specified output value for variable E. The test-case can be represented as follows:

$$OBS = \left\{ \begin{array}{c} in_1(2, A) = true, in_2(2, B) = true, in_2(4, C) = false, \\ out(3, D) = false, out(4, E) = false \end{array} \right\}$$

In order to locate the bug we first compute the conflict sets. For this example we obtain 2 conflicts: $\{2, 4\}$, $\{3, 4\}$. Whereas the first conflict is obvious, the second is not and we explain it in more detail. From definition 3 follows that $\{3, 4\}$ is a conflict if and only if $\neg AB(3) \wedge \neg AB(4)$ contradicts $SD \cup OBS$. From $\neg AB(3)$, $out(3, D) = false$, and $\neg AB(3) \rightarrow out(3, D) = \textit{not}\, in(3, S)$ follows $in(3, S) = true$ and (using $in(3, S) = in_1(4, S)$) $in_1(4, S) = true$. This fact together with $\neg AB(4)$, $\neg AB(4) \rightarrow out(4, E) = in_1(4, S)\ \textit{or}\ in_2(4, C)$, and the observation $in_2(4, C) = false$ leads to $out(4, E) = true$. The latter contradicts the observation $out(4, E) = false$ because $out(4, E)$ cannot be true and false at the same time. Hence, $\{3, 4\}$ must be a conflict. Using theorem 5, we can compute diagnoses directly from the set of conflicts. The diagnoses are: $\{4\}$ and $\{2, 3\}$.

The single fault diagnosis $\{4\}$ indicates that the bug is located in statement 4. A possible correction is to replace the *or* operator with an *and*. The double fault diagnosis $\{2, 3\}$ says that statement 2 and statement 3 should be modified in order to make *Demo* bug-free. Since two bugs are more difficult to correct than a single one, we prefer single bug candidates over multiple ones. It should be noted that the above value-based model is only one model in the theoretically infinite model space. The dependency-based model used by [11] and described in the following section is a different model. Different models can be compared with respect to their discrimination capabilities or time requirements for computing conflicts. A more detailed discussion of this issue is not in the scope of this paper and has been left for future research. Other future research topics include the use of abstract models of programs for model-based debugging, the development of models that are specialized on specific bugs, e.g., faults regarding array accesses or wrong references, and the integration of probabilities into the model-based debugging approach. Moreover, more effort should be made in making model-based debugging applicable for debugging large programs.

4 The dependency model

After dealing with the basic definitions of model-based diagnosis and how to use them for debugging we introduce a dependency-based model for the language L. This model is based on a model specified in [11] for the hardware description language VHDL. The idea behind the model is to reflect dependencies between variables that are introduced by statements. For example, consider the statement $s \equiv$ 'X = **not** Y'. In this statement variable *Y* influences variable *X* because the value of *X* is determined by the value of *Y*. We say that *X* depends on *Y* and write $(X, \{Y\})$. Note that there are possibly more than one variable influencing a variable X. If M_X is a set of variables X depends on, we write (X, M_X). From dependencies for statements we directly derive a logical model. The corresponding logical rule for statement s is $\neg AB(s) \wedge ok(Y) \rightarrow ok(X)$. This rule expresses that if the statement s is bug-free (not abnormal, $\neg AB(s)$) and the variable Y has an expected value ($ok(Y)$), then the value of the variable X must be correct. An algorithm for compiling a program Π to a logical model has to reflect the sequential order of statements. For example, only the last statement in the program *Demo2* given below determines the value of variable *X*:

1. **procedure** Demo2 $\{$
2. X = 3;
3. X = 4;
4. $\}$

In order to distinguish between variable definitions occurring at different points in the program, we index every variable with the line number. For simplicity reasons we further assume that each statement can be identified by the number of the line where it occurs. The dependency-based model for *Demo2* is $\{\neg AB(2) \rightarrow ok(X_2), \neg AB(3) \rightarrow ok(X_3)\}$. Assuming that the resulting value of variable *X* after executing the program *Demo2* is incorrect, we compute the fault locations. The observations OBS are formally represented by the set $\{ok(X_0), \neg ok(X_3)\}$. Using OBS and the model we obtain one conflict $\{3\}$ and finally one diagnosis $\{3\}$, indicating that the statement in line 3 is a bug candidate. Note that the set OBS includes the fact $ok(X_0)$ which means that the variable X has the correct value before executing *Demo2*. This is a well founded assumption because otherwise a bug, which causes the wrong value for X, must be located elsewhere in the program.

Until now we have introduced a model for assignment statements. In the rest of this section we describe a model for conditionals and while statements. Every conditional and while statement has an expression causing the program execution to branch. In order to represent this behavior, we use a variable C with the line number of the statement as index. This variable represents a control dependency edge in the program dependency graph and can be seen as the result of the expression. For every statement in the sub-block of a conditional or while statement s, the variable

Fig. 4. The Strongly Connected Component Super-graph for *Main1*

C is an input variable. For the while statement s the variable C is an output. The inputs of s are the variables used in the expression.

Although we have declared all inputs and outputs of statements, the logical rules comprising the model need not be derived in a straight forward way. For example consider the program *Main1* depicted in Figure 2 and the following logical representation:

$$\neg AB(2) \rightarrow ok(sum_2)$$
$$\neg AB(3) \rightarrow ok(i_3)$$
$$\neg AB(4) \wedge ok(i_3) \wedge ok(i_6) \rightarrow ok(C_4)$$
$$\neg AB(5) \wedge ok(i_3) \wedge ok(i_6) ok(sum_2) \wedge ok(sum_5) \wedge ok(C_4) \rightarrow ok(sum_5)$$
$$\neg AB(6) \wedge ok(i_3) \wedge ok(i_6) \wedge ok(C_4) \rightarrow ok(i_6)$$

This model may lead to problems regarding debugging. If we know that the variable i is incorrect after executing *Main1*, represented by the observation $\neg ok(i_6)$, then we cannot derive $ok(C_4)$, $ok(sum_5)$, $ok(i_6)$ anymore. Hence, only the empty diagnosis can be computed, which is obviously not the expected result. The source of the problem is that the while statement causes the dependency graph to contain cycles. Moreover, not the single cycles are important. Instead strongly connected components (SCCs) are of interest. A SCC is a subgraph where there exists a path between two arbitrary vertices. There are algorithms for efficiently computing $SCCs$ (see [1]). In Figure 2 one SCC is spanned by the vertices $\{4, 6\}$. The super-graph spanned by all SCCs for the example program *Main1* is depicted in Figure 4.

Every SCC indicates that all statements contained in it have an influence on each other. It is not possible to make one of these statements responsible for an incorrect value without further knowledge. Therefore, we view all statements $\{s_1, \ldots, s_n\}$ of the SCC as bug candidates that can be logically represented by the rule $\neg AB(s_1) \wedge \ldots \wedge \neg AB(s_n) \rightarrow \neg ab(CC)$, where CC represents the SCC. The behavior of a SCC representing a while statement is defined relying on the principles explained above. We use the variables on edges from vertices outside CC to vertices inside the CC as inputs, and those vertices pointing from vertices inside CC to vertices from outside as outputs. E.g., the revised part of the model for *Main1* is fully specified by the following rules:

$$\neg AB(4) \wedge \neg AB(6) \rightarrow \neg ab(W)$$
$$\neg ab(W) \wedge ok(i_3) \rightarrow ok(C_4)$$
$$\neg ab(W) \wedge ok(i_3) \rightarrow ok(i_6)$$
$$\neg AB(5) \wedge ok(sum_2) \wedge ok(i_3) \wedge ok(i_6) \wedge ok(C_4) \rightarrow ok(sum_5)$$

The algorithm *COMPUTE_MODEL* introduced below summarizes the model building process for programs. The algorithm takes the program Π as input. In the first step *COMPUTE_MODEL* computes the program dependency graph. We assume that all edges from vertex v_1 to v_2 of the dependency graph are labeled with the variable defined in v_1. Afterwards, *COMPUTE_MODEL* takes the dependency graph and computes the strongly connected super-graph. Finally, the algorithm derives the logical rules directly from the super-graph.

4.0.0.1 Algorithm COMPUTE_MODEL(Π)

(1) Compute a program dependency graph G for program Π. The vertices of G are named by the line number of the associated statement i. The arcs are named by variables x with index i or C with index i for arcs representing control dependencies.

(2) The resulting graph G may contain cycles. Therefore, compute the strongly connected super-graph of G, named S_G. Every vertex v of S_G contains a set of vertices that are reachable by each other, i.e., strongly connected. v has at least 1 vertex from G. The arcs of the super-graph are named like the associated arcs of the original graph. The super-graph S_G is processed as follows:

   (a) For every vertex $n \in S_G$ having exactly one sub-vertex except the *Entry* vertex add the following rule to the system description SD:

$$\neg AB(n) \wedge \bigwedge_{x \in inputs(n)} ok(x) \rightarrow ok(y)$$

where $inputs(n)$ is the set of all input arcs of n, and y is the output arc from n.

(b) For every vertex $m \in S_G$ with $k > 1$ sub-vertices $(n_1, \ldots, n_k)$ add:

$$\neg ab(m) \wedge \bigwedge_{x \in inputs(m)} ok(x) \rightarrow ok(y)$$

to SD. $inputs(m)$ is the set of all input arcs of m. y is an output arc from m. In addition the correctness of vertex m is determined by the correctness assumptions of $n_1, \ldots, n_k$. Hence, the following rule must be also added to SD:

$$\neg AB(n_1) \wedge \ldots \wedge \neg AB(n_k) \rightarrow ab(m)$$

(3) Return the resulting model SD.

The COMPUTE_MODEL algorithm returns a logical sentence representing data and control dependencies. The logical model SD can be computed at compile-time and can be used at runtime. We illustrate the COMPUTE_MODEL algorithm and how to compute all diagnoses using the program *Main1* (see Figure 2). The algorithm returns as result the following model:

$$\neg AB(2) \rightarrow ok(sum_2)$$
$$\neg AB(3) \rightarrow ok(i_3)$$
$$\neg AB(5) \wedge ok(sum_2) \wedge ok(i_3) \wedge ok(i_6) \wedge ok(C_4) \rightarrow ok(sum_5)$$
$$\neg ab(W) \wedge ok(i_3) \rightarrow ok(C_4)$$
$$\neg ab(W) \wedge ok(i_3) \rightarrow ok(i_6)$$
$$\neg AB(4) \wedge \neg AB(6) \rightarrow \neg ab(W)$$

We now can compare the outcome of the model-based approach for *Main1* using a dependency model with the outcome obtained when using program slicing on the criterion $(7, \{i\})$. Then the slicing criterion is represented by the observations $\{ok(i_0), ok(sum_0), \neg ok(i_6)\}$. The reader can easily prove that $\{3, 4, 6\}$ is the only minimal conflict set. Using a model-based diagnosis engine we finally obtain three single diagnoses: $\{3\}$, $\{4\}$, and $\{6\}$. This result corresponds directly to the program slice $(7, \{i\})$ for the same program.

Now consider the *Main2* program given in Figure 3. Again we compute the model SD:

$$\neg AB(2) \rightarrow ok(pi_2)$$
$$\neg AB(3) \wedge ok(pi_2) \rightarrow ok(s_3)$$
$$\neg AB(4) \wedge ok(pi_2) \rightarrow ok(a_4)$$

Using the observations $\{\neg ok(s_3), \neg ok(a_4)\}$ we obtain two conflict sets: $\{2,3\}$ and $\{2,4\}$ and finally one single fault diagnosis $\{2\}$ and one double fault diagnosis $\{3,4\}$. We compare this result with the slice for $(5, \{s,a\})$ which can be computed from the single slices: $(5, \{s\}) \cup (5, \{a\})$ (see [25]). This slice comprises all statements of *Main2*. Hence, it allows not for discriminating between statements responsible for a faulty behavior. In this respect the model-based approach provides more information. However, when looking at the single slices for *Demo2* and the conflict sets, we see that they are equal. The following theorem states that this equivalence holds generally.

Theorem 6 *Let Π be a program of size n, SD a logical model of Π (SD=COMPUTE_MODEL(Π)), and V a set of variables having a wrong value at position n after executing Π. From V the set of observations is defined as $OBS = \{\neg ok(v_n) | v \in V\} \cup \{ok(v_0) | v \in variables(\Pi)\}$. Any slice $(n, \{x\})$ with $x \in V$ is a minimal conflict for $(SD, \{1, \ldots, n\}, OBS)$, i.e., $SD \cup \{\neg AB(s) | s \in (n, \{x\})\} \cup OBS$ is contradictory.*

Proof: ($\Rightarrow$): Given a slice $(n,\{x\}) = \{s_1,\ldots,s_k\}$ we have to show that $(n,\{x\})$ is a conflict, i.e., $SD \cup \{\neg AB(s) | s \in (n,\{x\})\} \cup OBS \models \bot$. Assume the contrary. In this case we know that the fact $ok(x_{max})$ cannot be derived, where max is the greatest index for variable x. The slice $(n,\{x\})$ is given by all vertices lying on paths from the ENTRY vertex to the vertex defining variable x immediately before or at position n. Because of algorithm COMPUTE_MODEL we know that rules correspond to arcs in the SCC super-graph of the dependency graph and vertices of SCC super-graph correspond to a non-empty set of statements. Because all statements are assumed to behave correctly and the precondition that all input variables are correct, we can conclude that all rules corresponding to the sub-graph of the dependency graph spanned by the slice must allow for deriving their consequent. Hence, $ok(x_{max})$ must be derivable, contradicting our initial assumption.

The conflict must be minimal. Assume that $(n,\{x\})$ is not minimal. So, there must be at least one element s of the conflict that can be removed without making $SD \cup \{\neg AB(s) | s \in (n,\{x\})\} \cup OBS$ satisfiable. This can only be the case if the corresponding vertex n_s is not on a path from ENTRY to the last vertex defining x. In this case s is not element of $(n,\{x\})$, again contradicting the assumption.

($\Leftarrow$): Given a minimal conflict $CO = \{s_1,\ldots,s_k\}$. We show that CO is a slice for $(n,\{x\})$. Because of the rules in SD determined by COMPUTE_MODEL we know that all vertices of the program dependency graph corresponding to $s_i \in CO$ must be connected. Moreover, they must lie on a path from ENTRY to the vertex defining x_{max}. Hence, they must be elements of the slice $(n,\{x\})$. If CO is not minimal, then the conflict might contain elements that are not in the slice. $\Box$

Using theorem 6 we criticize the usual understanding of what a slice is and how to combine slices as defined by Weiser [24,25]. In literature a slice is seen as part of a program influencing the value of variables at a given position. If there is a slice $(n,\{x\})$ for exactly one variable with an unexpected value, we would think of this slice as set of possible bug locations, i.e., diagnoses. Formally, this interpretation of a slice $\{n_1,\ldots,n_k\}$ can be stated as:

$$AB(n_1) \vee \ldots \vee AB(n_k)$$

This logical sentence means that at least one statement must be buggy, i.e., must have an abnormal behavior (AB). This interpretation is correct and corresponds to the result of theorem 6. The logical sentence above can be re-written as:

$$\neg\left(\neg AB(n_1) \wedge \ldots \wedge \neg AB(n_k)\right)$$

where $\neg AB(n_1) \wedge \ldots \wedge \neg AB(n_k)$ is a conflict. Whereas slices for single variables have a correct interpretation with respect to the diagnosis definition, this is not the case for slices of the form $(n,\{x_1,\ldots,x_k\}$ where $k > 1$ which is equivalent to

$\bigcup_{i=1}^{k}(n, \{x_i\})$ (see [25]). Although the resulting slice is a conflict, it is not a minimal conflict. Hence, considering an element of the slice as a single fault diagnosis candidate is not correct anymore.

Note that this problem about minimality of slices does only apply in the context of this paper, i.e., locating bugs in programs. Program slicing is also applied to lots of other tasks, e.g., reuse and comprehension. In those areas it is important that the slice preserves the effect of the program upon the slicing criteria. Tip [23] gives a good survey of program slicing techniques and their practical use.

5 Discussion

In the previous section we have shown the relationship between static program slicing and model-based debugging using a dependency model. In this section we discuss the implications of theorem 6, have a look at extensions of static slicing and generalize the results obtained so far.

5.1 Implications

The implications of theorem 6 for slicing can be best explained using a modified version of program *Main2* (see Figure 3) which is named *Main2b*. We assume that line 1 of *Main2b* is equal to '*1. pi = 0.14;*'. Executing *Main2b* leads to the computation of wrong values for variables *s* and *a*. If we use program slicing for debugging, we have to compute the slice on the slicing criterion $(5, \{s, a\})$. This slice comprises all statements of *Main2b* and does not allow to focus search for the bug location, anymore. This problem disappears if we take the slices for the single variables as conflicts and use Reiter's hitting set algorithm for computing all diagnoses.

The slice on $(5, \{s\})$ is $\{2, 3\}$ and on $(5, \{a\})$ is $\{2, 4\}$. If we now use the hitting set algorithm on the set $\{\{2, 3\}, \{2, 4\}\}$, we obtain the two diagnoses $\{2\}$ and $\{3, 4\}$. These diagnoses say either that statement 2 is buggy, or that statement 3 in combination with statement 4 is buggy. If we assume that diagnoses with a smaller cardinality should be preferred, then we focus on statement 2 unless no further knowledge about variable values at positions in the source code is available. Beside obtaining an order of precedence for diagnoses, we obtain knowledge about the nature of the bug, i.e., whether it is a single bug or a bug comprising several statements. This knowledge helps to determine all bugs in a program. For example, if we fix the misbehavior of *Main2b* in statement 3 by replacing *pi* with *3.14*, then we know that we have to fix statement 4 (again by replacing *pi* with *3.14*).

The implications of theorem 6 for program slicing can be summarized as follows:

- The theorem allows to introduce an order of precedence for diagnoses and thus improve focusing on relevant parts of a program.
- The resulting diagnoses allow to gain knowledge about bug correction. Possible faulty statements that must be corrected together can be identified.

Theorem 6 also allows to draw the conclusion that slices instead of the dependency model can be used for model-based diagnosis. But what are the advantages? One advantage is that it is not necessary to compute a logical model from the dependency graph. Another advantage is the reduction of the overall diagnosis time. If slices of single variables are used as conflicts, the diagnosis time depends only on the hitting set algorithm. If we use the logical model of a program, the conflicts are generated when they are required by the hitting set algorithm (see [19]). Each computation of a conflict requires at least a call to a theorem prover which checks the consistency of the current assumptions with respect to the given observations and the logical model. Hence, diagnosis time depends on the hitting set algorithm and the number of theorem prover calls.

Let us consider example program *Main2* and its variant *Main2b*. The logical model only comprises 3 rules: $\neg AB(2) \rightarrow ok(pi_2)$, $\neg AB(3) \wedge ok(pi_2) \rightarrow ok(s_3)$, and $\neg AB(4) \wedge ok(pi_2) \rightarrow ok(a_4)$. Given the observations $\{\neg ok(s_3), \neg ok(a_4)\}$ we obtain the hitting set graph given in Figure 5. The diagnosis process starts at vertex $n0$. A call to the theorem prover (indicated with **TP** in the figure) loaded with the observations and the model returns a minimal conflict $\{2, 3\}$. The hitting set algorithm creates two arcs and two vertices $n1$, $n2$. For vertex $n1$ the theorem prover is called again assuming that statement 3 and 4 are correct and that statement 2 is buggy. The theorem prover call returns that this assumption is correct and that we have found a diagnosis. A vertex storing a minimal diagnosis is marked with a $\surd$. The algorithm takes the next vertex and continues computation until no further new vertices can be generated. Vertices that store a not minimal diagnosis are labeled with a $\times$. Although the example program is very small, 4 theorem prover calls are necessary for computing all diagnoses. This computational overhead at runtime can be avoided, if using slices.

5.2 Extensions

Regarding slicing we consider two different directions of research. One line is due to computing static slices for different kind of programs and different fault situations. In [14] an algorithm for computing slices for programs with procedure and function calls was introduced. This paper makes static slicing applicable to almost all procedural programming languages. Another improvement for computing slices is given in [21] where the problem of implicit program states is solved. On the

Fig. 5. HS-DAG for program *Main2b*

other hand [13] shows how program slicing used for debugging can be extended to handle runtime faults like out-of-bound or division-by-zero exceptions. The second research line shifts static slicing to dynamic slicing [16,17]. Whereas static slices can be (approximated) computed at compile-time, dynamic slices are computed for specific test-cases. This results in a more concise slice because branches not executed can be ignored.

We now discuss how some of the extensions can be expressed in a logical model and argue why these extensions do not contradict theorem 6. In [13] the authors extend slices to handle abnormal situations detected during program execution. Consider the following program:

```
1.  procedure Example1 {
2.      x = 1 / z ;
3.      y = 2;
4.      }
```

A slice for $(4, \{y\})$ is a program only comprising statement 3. Now assume that $z = 0$ which causes a division-by-zero exception and prevents the execution of statement 3. This behavior is different from the behavior given by the slice. Because with the slice we can compute a value for y whereas this is not the case for the original program. Hence, our initial argument that a slice preserves the same behavior on the given variables is not longer valid! To avoid this problem [13] introduces the concept of error variables associated with statements, that may cause exceptions to be raised. Such variables Err are introduced for different exceptions, e.g., division-by-zero or out-of-bound. To catch all involved statements the special variables Err are assumed to be used and changed by the statements. Example: Assume that $Err_{division}$ is an error variable for statements including division operators. The slice for *Example1* on $(4, \{Err_{division}\})$ comprises statement 1.

The concept of error variables has two advantages: First, slices can retain the idea of behavioral equivalence with respect to the slicing criterion when error variables are included in this criterion, and finally, it is possible to compute slices in case of exceptions. Incorporating the same concept into our logical model can be done in a straight forward way. For the statements $n_1, \ldots, n_k$ where a variable Err has been used add the rule

$$\neg AB(n_1) \wedge \ldots \wedge \neg AB(n_k) \rightarrow ok(Err)$$

to the model SD. As a result a slice $(n, \{Err\})$ must be equivalent to the conflict obtained when assuming $\neg ok(Err) \in OBS$. Hence, theorem 6 is also valid, if error variables are used, and we obtain the following corollary.

Corollary 7 *Let Π be a program of size n, SD a logical model of Π (SD=COMPUTE_MODEL(Π)), E a set of error variables, and n a position n in the code. From E the set of observations is defined as $OBS = \{\neg ok(v_n) | v \in E\}$. Any slice $(n, \{x\})$ where $x \in E$ is a minimal conflict for $(SD, \{1, \ldots, n\}, OBS)$, i.e., $SD \cup \{\neg AB(s) | s \in (n, \{x\})\} \cup OBS$ is contradictory.*

Proof: Any slice for $(n, \{x\})$ comprises statements $n_1, \ldots, n_k$ for which an error variable E has been introduced. Hence, $\neg AB(n_1) \wedge \ldots \wedge \neg AB(n_k) \rightarrow ok(x)$ must be in SD. From $\neg ok(x) \in OBS$ follows that $OBS \cup SD \cup \{\neg AB(n_1), \ldots, \neg AB(n_k)\}$ must be contradictory. Hence, $\{n_1, \ldots, n_k\}$ must be a conflict. By reverting the arguments we get a proof for the other direction. $\square$

Finally, we discuss the relationships between dynamic slicing [16,17] and model-based diagnosis using a dependency model. A dynamic slice is computed from an execution trace [17] that consists of nodes associated with statements which are executed for some input during program execution. Since they use knowledge about program execution dynamic slices are often smaller than static slices. For example, if an else-branch of a conditional statement is not executed for a given input, the statements of the else-branch are not part of a dynamic slice but they can be part of a static slice.

The dynamic slicing criterion and a dynamic slice are defined in [16] using the executed program path which is named a program trajectory.

Definition 8 (Dynamic Slicing Criterion) *Let T be the trajectory of program Π on input x. A dynamic slicing criterion of program Π executed on input x is a triple $C = (x, I^q, V)$, where I is an instruction at position q on T and V is a subset of variables in P.*

The dynamic slicing criterion differs from the static slicing criterion in the additional input parameter x. In addition, the interpretation of the position of interest changes. In the static case a slicing criterion uses the instruction I in Π, whereas in the dynamic case I is an instruction at the execution position q in a trajectory T. We now informally define a dynamic slice for a slicing criterion C as any executable program Π' that is obtained from Π by deleting zero or more statements from it, while preserving relevant parts of the trajectory T (with respect to V) and computing the same values for V at position q.

Although there are some differences between the definitions of dynamic and static slices they have a great deal in common, e.g., both definitions cause a slice to preserve the program's behavior with respect to the original program and the variables of interest. Hence, it is possible to obtain a dynamic slice from a static slice for a given set of variables V and comparable positions by deleting all statements of the static slice that are not element of the execution trace, i.e., the trajectory of Π on input x. Although this procedure leads to a valid dynamic slice there are currently more precise dynamic slicing algorithms in use.

We now argue that theorem 6 for a modified logical model and a dynamic slice is still valid. First of all, we define the new model. Let $(SD, COMP)$ be the dependency model for a program Π, let x be an input and T the trajectory of Π on input x. The dependency model $(SD_D, COMP_D)$ for the dynamic slice Π'_D obtained by deleting all statements not in T from the static slice Π' is defined as follows:

$$SD_D = SD \cup \{\rightarrow \neg AB(n) | n \in COMP \wedge n \notin T\}$$

$$COMP_D = \{n | n \in COMP \wedge n \in T\}$$

This new model comprises components that are still candidates for being buggy. Statements n that are not in the trajectory T cannot be buggy which is represented by the fact '$\rightarrow \neg AB(n)$'. Hence, $(SD_D, COMP_D)$ directly corresponds to the dynamic slice obtained from the static slice. It is obvious that theorem 6 is still valid for $(SD_D, COMP_D)$ and Π'_D.

Corollary 9 *Let Π be a program of size n, SD a logical model of Π (SD=COMPUTE_MODEL(Π)), and V a set of variables having a wrong value at position n after executing Π. From V the set of observations is defined as $OBS = \{\neg ok(v_n) | v \in V\} \cup \{ok(v_0) | v \in variables(\Pi)\}$. Any dynamic slice $(n, \{x\})$ where $x \in V$, is a minimal conflict for $(SD_D, COMP_D, OBS)$, i.e., $SD_D \cup \{\neg AB(s) | s \in (n, \{x\})\} \cup OBS$ is contradictory.*

Proof: Both the dynamic slice and the model are obtain by removing (the same) statements. Removing of statements in the logical model is done by removing them from $COMP$ and adding the fact $\rightarrow \neg AB(n)$ to the system description. Hence, this corollary follows directly from theorem 6. $\Box$

6 Conclusion

This paper provides several contributions to model-based reasoning and software debugging. The first contribution is due to showing how model-based diagnosis can be used for debugging by describing a model with similar capabilities as program slicing. This is the first step in a more ambitious project aimed at clarifying the equivalence of different debugging approaches and models which can be used by a model-based diagnosis engine. We expect that the results would be of interest for both the software debugging and the model-based reasoning community.

Showing the equivalence of slices and conflict sets, which represents the main contribution of this paper, improves debugging using a model-based system. Instead of computing the logical model and afterwards computing conflict sets, only the slices for incorrect variables have to be considered. Since all slices for single variables can be computed at compile time (in the static case) the computation of the conflict sets at diagnosis time is no longer necessary. Hence, diagnosis time is equivalent to the time necessary for building a hitting set using a directed acyclic graph (see [19,12]).

The equivalence of slices and conflicts leads to the conclusion that slices for variable sets with more than one element are not minimal conflicts and every element of the slice is not a single fault diagnosis, i.e., a single bug candidate. A slice for $(n, \{x_1, x_{,2}, \ldots, x_k\})$ given by $\bigcup_{i=1}^{k}(n, \{x_i\})$ is too coarse in the sense that there might be elements not explaining all incorrect variables. Therefore, these elements cannot be single fault diagnoses. Since discriminating between different candidates is required in order to keep the search space for debugging as small as possible, smaller bug candidates should be preferred. Because slicing does not allow for this discrimination model-based diagnosis should be used. For example consider the following small program:

```
1.  procedure Demo3 {
2.      S = not CLK;
3.      O1 = I1 and CLK;
4.      O2 = I2 and CLK;
5.  }
```

We assume that the values of variables *O1* and *O2* are wrong after executing *Demo3*. If we use program slicing on the criterion $(5, \{O1, O2\})$ for debugging we obtain all statements as possible diagnoses. But if we use the model-based diagnosis as described herein, we obtain a single fault diagnosis $\{2\}$ and a double fault diagnosis $\{3, 4\}$. Hence, we can discriminate between the two different diagnoses.

Acknowledgments

I thank Dominik Wieland and Markus Stumptner for their comments on earlier drafts of this paper, and the referees for their extremely helpful comments and suggestions. The work described in this paper was partially supported by the Austrian Science Fund (FWF) under grants P12344-INF and N Z29-INF.

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Publishing Company, 1974.

[2] G. W. Bond and B. Pagurek. A Critical Analysis of "Model-Based Diagnosis Meets Error Diagnosis in Logic Programs". Technical Report SCE-94-15, Carleton University, Dept. of Systems and Computer Engineering, Ottawa, Canada, 1994.

[3] Gregory W. Bond. *Logic Programs for Consistency-Based Diagnosis*. PhD thesis, Carleton University, Faculty of Engineering, Ottawa, Canada, 1994.

[4] Lisa Burnell and Eric Horvitz. A Synthesis of Logical and Probabilistic Reasoning for Program Understanding and Debugging. In *Proceedings of the International Conference on Uncertainty in Artificial Intelligence*, pages 285 –291, 1993.

[5] Lisa Burnell and Eric Horvitz. Structure and Chance: Melding Logic and Probability for Software Debugging. *Communications of the ACM*, 38(3):31 – 41, 1995.

[6] Luca Console, Gerhard Friedrich, and Daniele Theseider Dupré. Model-based diagnosis meets error diagnosis in logic programs. In *Proceedings* 13^{th} *International Joint Conf. on Artificial Intelligence*, pages 1494–1499, Chambery, August 1993.

[7] Sebastian Danicic, Mark Harman, and Yoga Sivagurunathan. A parallel algorithm for static program slicing. *Information Processing Letters*, 56:307–313, 1995.

[8] Johan de Kleer and Brian C. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32(1):97–130, 1987.

[9] Gerard Ferrand. Error Diagnosis in Logic Programming, An Adaption of E.Y Shapiro's Method. *Journal of Logic Programming*, 1987:177–198, 1987.

[10] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, 1987.

[11] Gerhard Friedrich, Markus Stumptner, and Franz Wotawa. Model-based diagnosis of hardware designs. *Artificial Intelligence*, 111(2):3–39, July 1999.

[12] Russell Greiner, Barbara A. Smith, and Ralph W. Wilkerson. A correction to the algorithm in Reiter's theory of diagnosis. *Artificial Intelligence*, 41(1):79–88, 1989.

[13] Mark Harman, Dan Simpson, and Sebastian Danicic. Slicing programs in the presence of errors. *Formal Aspects of Computing*, 8:490–497, 1996.

[14] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural Slicing Using Dependency Graphs. In *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*, pages 35–46, Atlanta, Georgia, 1988.

[15] Daniel Jackson. Aspect: Detecting Bugs with Abstract Dependences. *ACM Transactions on Software Engineering and Methodology*, 4(2):109–145, April 1995.

[16] Bogdan Korel and Janusz Laski. Dynamic Program Slicing. *Information Processing Letters*, 29:155–163, 1988.

[17] Bogdan Korel and Jurgen Rilling. Applications of Dynamic Slicing in Program Debugging. In *Proceedings of the Third International Workshop on Automatic Debugging (AADEBUG-97)*, pages 43–58, Linköping, Sweden, 1997.

[18] Ron I. Kuper. Dependency-directed localization of software bugs. Technical Report AI-TR 1053, MIT AI Lab, May 1989.

[19] Raymond Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–95, 1987.

[20] Ehud Shapiro. *Algorithmic Program Debugging*. MIT Press, Cambridge, Massachusetts, 1983.

[21] Yoga Sivagurunathan, Mark Harman, and Sebastian Danicic. Slicing, I/O and the Implicit State. In *Proceedings of the Third International Workshop on Automatic Debugging (AADEBUG-97)*, pages 59–68, Linköping, Sweden, 1997.

[22] Markus Stumptner and Franz Wotawa. Debugging Functional Programs. In *Proceedings* 16^{th} *International Joint Conf. on Artificial Intelligence*, pages 1074–1079, Stockholm, Sweden, August 1999.

[23] Frank Tip. A Survey of Program Slicing Techniques. *Journal of Programming Languages*, 3(3):121–189, September 1995.

[24] Mark Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–452, July 1982.

[25] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.