

Model-based Debugging or How to Diagnose Programs Automatically*

Franz Wotawa¹, Markus Stumptner², and Wolfgang Mayer² **

¹ Graz University of Technology, Institute for Software Technology, Inffeldgasse 16b/11,
A-8010 Graz, Austria, wotawa@ist.tu-graz.ac.at

² University of South Australia, Advanced Computing Research Centre, 5095 Mawson Lakes,
Adelaide, SA, Australia, {mst, mayer}@cs.unisa.edu.au

Abstract. We describe the extension of the well-known model-based diagnosis approach to the location of errors in imperative programs (exhibited on a subset of the Java language). The source program is automatically converted to a logical representation (called model). Given this model and a particular test case or set of test cases, a program-independent search algorithm determines a the minimal sets of statements whose incorrectness can explain incorrect outcomes when the program is executed on the test cases, and which can then be indicated to the developer by the system. We analyze example cases and discuss empirical results from a Java debugger implementation incorporating our approach. The use of AI techniques is more flexible than traditional debugging techniques such as algorithmic debugging and program slicing.

1 Introduction

Debugging, i.e., the task of detecting, locating, and correcting faults in programs, is generally considered a very difficult and time consuming task that has, at the same time, remained crucial to software development [1]. Whereas most research deals with techniques for fault *detection*, e.g., test case generation and verification or *avoidance* (most software design methodologies fit in here), relatively little has been done in the domain of locating and correcting bugs. The main lines of research in bug location date back to the seminal papers on program slicing [2] and algorithmic software debugging [3]. In this paper we present a logic approach to this problem that is based on model-based diagnosis [4]. In this approach a logical model of the program together with the expected behavior of the program are used directly to locate and (sometimes) correct errors in the program. The logical model is automatically extracted from the program without further user interaction and without requiring a separate formal specification. The expected behavior is assumed to be given in the test cases, i.e., basically input-output vectors of used variables.

For demonstration purposes, in this paper we make use of a (non-object-oriented) subset of Java. However, it should be noted that the implementation used in the debugger actually covers the object-oriented parts of the language. The syntax of this subset is depicted in Fig. 1. Consider the following program:

* Work was partially supported by Austrian Science Fund projects P12344-INF and N Z29-INF.

** Authors are listed in reverse alphabetical order.

```

program ::= id '{ stmnts }'
stmnts ::= stmnt stmnts | ε
stmnt ::= assignment | conditional | while
assignment ::= id '=' expr ';'
conditional ::= if expr '{ stmnts }' [ else '{ stmnts }' ]
while ::= while expr '{ stmnts }'
expr ::= '(' expr ')' | expr op expr | id | const

```

Fig. 1. Syntax of \mathcal{L}

```

1. test {
2.   if (X=1) {
3.     Y = 1;
4.   } else {
5.     Y = 0;
6.   } }

```

and the test case $(X_2 = 0, Y_6 = 1)$ where $V_i = v$ mean that variable V has value v before executing the statement in line i . It is obvious that program *test* computes the wrong value for variable Y at line 6. This can be proven by sequentially executing statement by statement according to the control flow of the program. In this case variable Y is assigned the value 0 which contradicts the given test case.

In this situation the programmer would first search for the last statement defining variable Y – in this case, statement 5. Without further inspecting the code, it could be concluded that statement 5 should be changed to $Y = 1$. However, the addition of another test case, e.g., $(X_2 = 1, Y_6 = 0)$, may lead to a situation where changing statement 5 is not the best solution. When considering both tests the programmer would most likely then go back on the trace and finally visit statement 2. Changing the condition to $X = 0$ will lead to a program that passes both test cases.

The search for a possible bug location as described above is influenced by both the control flow of the program and the dependencies between variables that are given by the semantics of the language. Statements that are executed but do not lead to a wrong value of a different variable can be excluded from the list of bug candidates.

In our approach we consider dependencies, the control flow and the whole semantics of the program. We make use of *correctness assumptions* about the behavior of statements during the search for bug candidates. The use of assumptions is quite natural. Consider the above example where a programmer assumes that some statements are correct and others are not. During debugging the assumptions are changed and adapted until a consistent state is reached, i.e., the test cases do not contradict the assumptions. Hence, debugging can be seen as the problem of assigning correctness and incorrectness assumptions to all statements and checking the consistency of these assumptions and given test cases in the standard Model-Based Diagnosis [4, 5] approach.

The paper formally characterizes the debugging process and introduces a specific logical model of the assignment language \mathcal{L} . While-statements are converted to nested if-statements while preserving result equivalence for the given test cases. This results in a simpler model that can be restricted to loop-free variants of programs, in contrast to our previous work [6–8], resulting in more effective pruning of the search space and better discrimination between diagnosis candidates. Our empirical results confirm that

the approach reduces the search space for debugging and thus helps the user to focus his or her attention on relevant parts of the program.

2 Model-based Diagnosis

In Model-Based Diagnosis, a logical sentence that represents the behavior of a system is used to determine a fault. The logical sentence is called a system description SD . In software debugging SD has to specify the behavior (or at least the interesting parts) of the program. A fault of the system, i.e., the bug in the program, is a set of system components, i.e., statements or expressions, that are responsible for a detected misbehavior, i.e., a difference between the actual and the specified behavior. Before formally defining diagnosis we first show how system descriptions are directly derived from programs.

To illustrate the resulting description, Fig. 2 shows the graphical representation of the model of program `test`. The structural part of the model is the system description SD which describes the connection between the components we want to diagnose, in this case Java statements, for the set of statements $STMNTS = \{S_1, S_2, S_3\}$. Each component has a set of input ports on which values produced earlier in the program execution are propagated, and an output port, on which values computed and possibly assigned during execution of the component are propagated further.

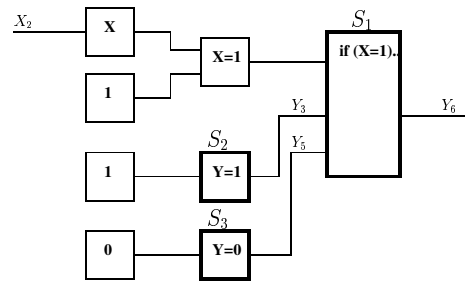


Fig. 2. Graphical representation of model `test`

This is complemented by the behavior model which captures the semantics of statements and expressions in a generic manner. The semantic descriptions of the various statement types are based on explicitly representing the abovementioned correctness or incorrectness assumptions in terms of predicates assigning appropriate modes to the statements. In the model described here, expressions are not assigned a mode since we only consider diagnosis at statement level. Assignment statements have the modes *Corr* and *Incorr* referring to the assumption of correct and incorrect behavior, respectively, with correct behavior being the default mode. Conditional statements have modes *Then* and *Else*, corresponding to execution of the corresponding branch of the statement. For them there is no general default mode; the default depends on the branch chosen by a particular execution. The approach can be generalized to more than two modes per statement but this requires a change in the formalization [9] which is why we do not include it here.

The generic behavior model is given by the following logical sentences which must be added to the system description.

Expressions: There are 3 different types of expressions which must be taken into account for the logical model.

Constants: The correct behavior of a constant is to evaluate to itself. In component-oriented terminology, the constant itself is propagated to the output port *out*.
 $const_k(C) \Rightarrow out(C) = k$

Variable accesses: If a variable access is correct, the input port and the output port must have the same values. $vara(C) \Rightarrow out(C) = in(C)$

Operators: For each operator *op* the following rule must be added to the system description. It says that the output value is determined by the associated to the operator, and by the input values. $fun_{op}(C) \Rightarrow out(C) = in_1(C) op in_2(C)$

Note that for some operators not all input values need be specified in order to determine the output value. E.g., the output value of an and-operator is \mathcal{F} if at least one input value is known to be \mathcal{F} .

Assignments: The behavior of assignments is to determine the new value of the target variable. The target variable in our model is represented by a connection with the output port. The new value itself is determined by the value of the evaluated expression.
 $assign(C) \Rightarrow (Corr(C) \rightarrow out(C) = in(C))$

Conditionals: A conditional either requires the then-block or the else-block to be executed depending on the value of the condition. We model this behavior by introducing two modes *Then* and *Else*. If a conditional component *C* is in *Then*-mode, the value of each variable that is defined in one of the statement blocks is determined by the then-block and the condition must be true. Otherwise, *C* must be in *Else* mode, the behavior is determined by the else-block, and the condition must be false.

$$\begin{aligned} if(C) &\Rightarrow (default(Then(C)) \wedge Then(C) \rightarrow out(C, V) = in_T(C, V) \wedge cond(C) = \mathcal{T}) \\ if(C) &\Rightarrow (default(Else(C)) \wedge Else(C) \rightarrow out(C, V) = in_E(C, V) \wedge cond(C) = \mathcal{F}) \\ if(C) &\Rightarrow (\neg default(Then(C)) \wedge Then(C) \rightarrow out(C, V) = in_T(C, V)) \\ if(C) &\Rightarrow (\neg default(Else(C)) \wedge Else(C) \rightarrow out(C, V) = in_E(C, V)) \end{aligned}$$

The conditional statement's behavior depends on the current default mode. For example, assume that the else-branch is executed but it really should be the then-branch which is executed. In this case, the condition should evaluate to true and not to false. If the component is in mode *Then* during diagnosis the value true would be propagated to the expression. Since an expression is not a component and cannot have an incorrect mode, this propagation may lead to unexpected results. The last assignment changing the value of a variable that is used in the expression also becomes part of the diagnosis which is not intended.

Note that there exist programs for which the default mode of a conditional cannot be determined using information obtained during the execution of a test case. Consider a conditional expression that is part of the then-branch of another conditional and suppose that the else-branch of the latter conditional is executed. In this case, the statements forming the then-branch are not executed and therefore no value is derived for the embedded conditional. As default mode for the statement part of such branches we choose the mode that is derived when the mode of the containing conditional is selected such that the previously ignored branch is executed.

The advantage of this conditional model is that the branch to be executed is determined by the mode of *C* and not by the connected expression components. Hence, in cases where no value of the conditional is derivable, the conditional component still has

a well-defined behavior and, moreover, the output of the condition is given. This helps to restrict the number of diagnosis candidates.

In the previous section we stated that the diagnosis process is characterized by assumptions of the correctness or incorrectness of components. These assumptions are asserted and retracted during the process. The goal is to find a set of assumptions that is consistent with the given test cases:

Definition 1 (Mode assignment). A mode assignment for statements $\{C_1, \dots, C_n\} \subseteq STMNTS$, each having an assigned set of modes ms and a default mode $default$ such that $default \in ms(C_i)$ for each $C_i \in STMNTS$, is a set of predicates $\{m_1(C_1), \dots, m_n(C_n)\}$ where $m_i \in ms(C_i)$ and $m_i \neq default(C_i)$.

E.g., $\{Then(S_1)\}$ and $\{Corr(S_2)\}$ are mode assignments for program `test`. In the default case, assignment statements are assumed to be correct. Conditional statements are assumed to be executed according to the actual test case. If the test case causes the then-block to be executed, the default behavior of the conditional is the *Then* mode. Since we are only interested in deviations from the default behavior — which is the actual behavior of the program given a test case — mode assignments only collect the deviations, i.e., statement modes that are not equal to the default behavior.

Based on mode assignments we now define the concept of diagnosis with respect to a given test case.

Definition 2 (TC-Diagnosis). A test case diagnosis (tc-diagnosis) Δ for a diagnosis problem $(SD, STMNTS, TC)$ is a mode assignment that is consistent with the given test case, i.e., the logical sentence $SD \cup TC \cup \{m(C) | m(C) \in \Delta\} \cup \{d(C) | C \in STMNTS \wedge m(C) \notin \Delta \wedge d = default(C)\}$ must be consistent.

Consider the test case $\{X_2 = 0, Y_6 = 1\}$ of program `test`. The mode assignments $\{Then(S_1)\}$ and $\{\neg Corr(S_3)\}$ are both tc-diagnoses. For the test case $\{X_2 = 1, Y_6 = 0\}$ the tc-diagnoses are $\{Else(S_1)\}$ and $\{\neg Corr(S_2)\}$.

From tc-diagnoses we can generalize to diagnoses for a given set of test cases. However, since default modes of components and therefore tc-diagnoses depend on test cases, the definition of a diagnosis for all test cases must abstract from modes. For example, consider program `test` and the two test cases. There are two diagnoses that are associated with the conditional statement, $\{Then(S_1)\}$ and $\{Else(S_1)\}$, but $\{Then(S_1), Else(S_1)\}$ cannot be a diagnosis.

Definition 3 (Diagnosis). A set $\Delta \subseteq STMNTS$ is a diagnosis for a diagnosis problem $(SD, STMNTS, SPEC)$ iff there exists a tc-diagnosis Δ_{TC} for at least one test case $TC \in SPEC$ where $\Delta \cap \{C | m(C) \in TC\}$ is not empty.

We further say that a diagnosis (tc-diagnosis) is minimal if no proper subset of it is a diagnosis (tc-diagnosis). We compute diagnoses using Reiter's hitting-set algorithm[4]. Using this approach the notion of a conflict set characterizes a set of statements that cannot be all correct. In our case we relax this definition to statements that cannot be all in the default mode.

Definition 4 (TC-Conflict). A set $\{C_1, \dots, C_n\} \subseteq STMNTS$ is a tc-conflict iff $SD \cup TC \cup \{d_i(C_i) | d_i = default(C_i), i = \{1, \dots, n\}\}$ is inconsistent.

Diagnoses are the hitting sets of the set of conflicts. Therefore, the minimal tc-conflicts can be directly used to compute the tc-diagnoses of the same test case. We only have to add the non-default modes of the statements to the hitting sets. For example, the only tc-conflict of `test` and the test case $\{X_2 = 0, Y_6 = 1\}$ is $\{S_1, S_3\}$. The

hitting sets are $\{S_1\}$ and $\{S_3\}$. If we consider the non-default modes of the statements we finally obtain $\{Then(S_1)\}$ and $\{\neg Corr(S_3)\}$ as tc-diagnoses. Moreover, we can compute all diagnoses directly from all tc-conflicts as stated in the following theorem.

Theorem 1. *Let T be the set of all tc-conflicts of all test cases $SPEC$. A diagnosis for $(SD, STMNTS, SPEC)$ is a hitting set of T .*

Proof. (Sketch): Every tc-diagnosis is a hitting set of that subset of T that corresponds to the same test case. Such a tc-diagnosis or a superset of it must be a tc-diagnosis of all other test cases. This follows from the definition of tc-diagnosis. If the tc-diagnosis directly is a diagnosis, then other tc-conflicts do not have an influence on this diagnosis and the diagnosis must clearly be a hitting set of T . If a superset is required to be a tc-diagnosis of all test cases, then there are other tc-conflicts that have an influence. In this case the superset must be also a diagnosis which would be detected if using T directly. Hence, all diagnoses are hitting sets of T .

The tc-conflict of `test` and the other test case is $\{S_1, S_2\}$. If we use both tc-conflicts to compute the hitting set, we receive $\{S_1\}$ and $\{S_2, S_3\}$ as diagnoses.

3 Loop-free Programs

Programs with loop-statements are not easy to debug. A reason is the lack of a good model that allows to reason backwards in the program. Therefore some statements that cannot be responsible for the faulty behavior are still part of the set of computed diagnoses. This problem does not occur in programs that are comprised only of assignments and conditionals because there are good models available for both statement types. Since interesting programs contain loop-statements we have to show how to compile them to their loop-free variant without changing their semantics (as far as diagnosis is concerned).

One way of representing loops is to map them to an infinite nested sequence of conditionals. For practical purposes we can restrict the nesting depth to the maximum number of loop iterations to be considered. For our purpose, i.e., locating bugs in programs, this is always possible because we only have to consider the specified test cases. We can run the program on all test cases and measure the maximum number of iterations. The only requirement for the approach is that the resulting program must be (semantically) equivalent to the original program with respect to the given test cases. This type of equivalence is softer than the usual definition of program equivalence. In our case the variant need not be equivalent for all possible inputs.

Definition 5 (S-Equivalent). *Let V be a set of input environments. Two programs $\Pi_1, \Pi_2 \in \mathcal{L}$ are soft equivalent (s-equivalent) iff they halt on every input environment $I \in V$ with the same output environment, i.e., $\forall I \in V : eval(\Pi_1, I) = eval(\Pi_2, I)$.*

The s-equivalence of a program and its loop-free variant is an important requirement. Without s-equivalence the computed diagnoses of the variant are not guaranteed to be diagnoses of the original program. Because of our assumptions and the restricted semantics of our Java subset, the following corollary holds (given without proof).

Corollary 1. *The program Π is s-equivalent to its loop-free variant $\Gamma(\Pi)$.*

Beside s-equivalence the size of the variant and its expected runtime are important factors that influence the practicability of the approach. The size of the variant is given by the number of while-statements in the original program. In the worst case the size is determined by the maximum number of nested while-statements.

Note that in practical terms, this size increase is not a problem. The subroutines (Java methods) we are considering for debugging tend to be rather small and therefore the nesting depth of while-statements is generally quite low. Moreover, the size has (almost) no influence on the expected runtime because only those statements are executed that would be executed by the original program when using the same input environment. For example, if a loop is executed twice, then only two of the corresponding then-branches of the nested if-statements are executed. Hence, the runtime of both versions should be approximately the same.

4 Diagnosis Complexity, Remarks, and Results

Note that the approach is correct but not complete. All diagnosis results do explain the faulty behavior, but there are some cases where either the correct diagnosis is not part of the result or no diagnosis can be computed. The latter case is due to the limitations of the mapping I that restricts the number of possible iterations. If a multiple fault diagnosis requires more iterations than expected, it can be the case that no diagnosis is delivered back as result. Moreover, the used model is mainly for locating bugs that are specific to the wrong use of operators. A faulty variable access maybe found but this is not guaranteed. Handling this kind of error requires ways of changing the program's structure automatically during diagnosis.

Although our approach is not complete, the empirical results given in the next section why it is nonetheless useful.

4.1 Debugging examples

This section illustrates the debugging process with the model based on a small example program shown below. The intended behavior of the program is to read a sequence of values from a file, until a value different from its predecessor is encountered. The reading is performed by an auxiliary function `read_from_file`, which is assumed to be a primitive of the language. Note that the program is not correct, as the condition of the loop doesn't test for equality, but is true for any value that is less than or equal to the previous value.

It is easily observed that the behavior of the program is incorrect for the test sequence (3, 3, 2, 2, 5). As the model requires the program to be loop-free, the program must be expanded into a sequence of `if` statements as described in the previous sections. The number of iterations the loop executes is derived from the execution of the faulty program, given the test data. Using the sequence from above, the loop is executed (and is thus expanded) four times, resulting in the loop-free variant of the program that is shown in Fig. 3.

The loop-free program is transformed into a representation based on components and connections between them, according to the description in the previous sections. For this program, the model consists of 36 different components, each of them representing a statement or an expression in the program. As the diagnosis process is applied

```

1 skip_equal {
2   p = read_from_file ();
3   c = p;
4   while(c <= p) {
5     p = c;
6     c = read_from_file ();
7   }}

1 skip_equal_loopfree {
2   p = read_from_file ();
3   c = p;
4   if (c <= p) {
5     p = c;
6     c = read_from_file ();
7     if (c <= p) {
8       p = c;
9       c = read_from_file ();
10      if (c <= p) {
11        p = c;
12        c = read_from_file ();
13        if (c <= p) {
14          p = c;
15          c = read_from_file (); }}}}}

```

Fig. 3. Example Program

at the statement level, the components that correspond to expressions of the program are not considered diagnosis candidates, reducing the number of components that are subject to mode assignments to 14.

If the diagnosis engine is given the expected output of the program ($p = 3$ and $c = 2$) as observations, it derives 8 tc-diagnoses³:

$\{Else([if]_{10})\}$, $\{Ab([c =]_3),Else([if]_4)\}$, $\{Ab([c =]_6),Else([if]_7)\}$,
 $\{Ab([c =]_9),Else([if]_{13})\}$, $\{Ab([p =]_{11}),Else([if]_{13})\}$, $\{Ab([p =]_{14}),Ab([c =]_{15})\}$,
 $\{Ab([c =]_9),Ab([c =]_{12}),Ab([c =]_{15})\}$ and $\{Ab([p =]_{11}),Ab([c =]_{12}),Ab([c =]_{15})\}$.

Each of the tc-diagnoses represents a subset of the program's statements that can possibly be responsible for the deviations between the expected and the observed behavior of the program. The number of components assigned the *Then-* or *Else-* mode can be used as an estimate for the 'distance' between the behavior of a corrected version of the program and the behavior of the faulty program. Under the assumption that small deviations (with few statements involved) are more likely to occur, the tc-diagnosis $\{Else([if]_{10})\}$ can be considered a preferred diagnosis and presented to the user for inspection. In this example, the preferred diagnosis identifies the statement that contains the fault and, in addition, information is given about the iteration in which the fault occurs. The diagnosis indicates that the loop should exit after the second iteration (the mode of the *if*-statement representing the test after the second iteration is set to *Else*, which corresponds to the termination of the loop). Further inspection of the conditional expression (using the observation that the conditional should evaluate to \mathcal{F} and that the values of c and p are correct), the operator is identified as the single location of the fault. Based on the expected input- and output-values of the operator, the program can be corrected by computing a replacement for the faulty operator.

To evaluate the effectiveness of the model, first experiments with a set of example programs of varying size and structure have been performed, using a debugger prototype incorporating the diagnosis algorithm. The table below contains a brief description of the example programs, their models and the computed results. All programs im-

³ The components are represented by their corresponding statement, together with its line number.

plement basic algorithms (e.g. binary search, sorting, finding the maximum or sum of a sequence of numbers, computing power series and permutations, gauss-elimination, etc.) and contain a single faulty statement. The column 'Statements' contains the number of statements of the original program and the number of statements after expanding all loops. 'Components' contains the number of components the model of the program consists of and the number of components representing statements (i.e. those are considered when searching for diagnosis candidates). 'Diagnoses' lists the number of diagnoses with minimal cardinality⁴ that are obtained when specifying the expected values at the end of the program. 'Hits' indicates the number of diagnoses that actually refer to a faulty statement and 'Code %' lists the fraction of the original program that has to be examined in the worst case until the fault is detected.

Name	Statements	Components	Diagnoses	Hits	Code %
sum	5 (11)	34 (11)	2	1	40
find_pair	5 (11)	34 (11)	5	2	80
skip_equal	5 (14)	36 (14)	1	1	20
bin_search	26 (59)	253 (63)	1	1	8
library	24 (39)	161 (56)	3	0	34
permutation	24 (32)	118 (32)	4	2	13
sum_powers	21 (61)	200 (61)	2	1	10
bubblesort	15 (51)	235 (54)	1	1	7
matrix	71 (191)	970 (199)	67	3	46

The results show that the model is able to locate faults in programs relatively precisely. In most cases, the number of statements to be examined is reduced significantly, even when the results are translated back to the original (unexpanded) programs. On the set of example programs, the fraction of a program that has to be examined to locate the fault is reduced to 29% on the average.

For some examples, considering only the diagnoses with minimal cardinality does not lead to the fault and diagnoses with larger cardinality have to be examined. In the `library` example, the inaccuracy can be explained by looking at the algorithm the program implements: the maximum value of a sequence is computed. Hence, changing the initialization of the algorithm (restricting the sequence to be considered to the element that contains the maximum value) corrects the observed behavior for the given test case. The diagnoses representing these changes are smaller than the diagnoses that contain the true fault inside the loop. This is caused by the expansion of the loop, when the faulty statement is copied for each iteration. Therefore diagnoses with larger cardinality are required to locate the fault. For the example program it is sufficient to consider the 15 minimal diagnoses with cardinality 2.

Spurious diagnoses as in the `library` example can be eliminated by performing an iterative diagnosis process and querying the user about the expected values of variables during the execution of the program (the results were obtained by using only the output values of the program), or by using multiple test cases concurrently. A further,

⁴ In most cases the diagnoses with minimal cardinality consist of one component, but for some examples (e.g. `bin_search`) the cardinality can be larger. Note that the set of diagnoses with minimal cardinality generally doesn't cover all minimal diagnoses, as some of them may be of larger cardinality.

more powerful solution is to apply the diagnosis process on a meta-level. For example, the size of a fault indicated by a diagnosis is not measured on the loop-free program. Instead, the diagnoses can be rated based on their size after mapping them back to the unexpanded program. This approach enables the model to effectively locate faults that could not be detected when considering only diagnoses with minimal cardinality.

5 Related Research

Slicing [2] is a well known technique that is not only used for debugging but also for other applications, i.e., program analysis, software maintenance, testing, and compiler tuning. Many different slicing definitions and algorithms exist (see [10] for a survey). If we use slicing for debugging, then each slice corresponds to all statements that potentially determine the (wrong) variable value at a given point in the program. Therefore, we can view slices as conflicts. A conflict is a set of statements that if assumed to work correctly contradicts a given test case. This is exactly the case for a slice. But a slice is not required to be a minimal conflict. Moreover, there are conflicts with no corresponding slice. Consider for example the following program:

```
1. test2 {  
2.   R = D / 2;  
3.   A = 3.14 * R * R;  
4.   C = 3.14 * R };
```

and the test case $\{D_1 = 2, A_5 = 3.14, C_5 = 6.28\}$. A slice for `test2` and variable `C` at location 5 is:

```
1. test2 {  
2.   R = D / 2;  
4.   C = 3.14 * R }
```

With model-based diagnosis we obtain 2 conflicts. The conflict $\{St_2, St_4\}$ is equal to the slice. For the second conflict $\{St_3, St_4\}$ there is no corresponding slice. As a result, debugging using slices as conflicts may lead to the computation of too many single bug candidates which is not the case for model-based debugging. Hence, slicing is weaker for diagnosis than our approach. A similar result can be obtained for other dependency-based techniques, e.g., [11, 12].

Another well-known technique for debugging is algorithmic or declarative software debugging [3, 13]. Similar to the model-based approach, these also use the semantics of the language and a given test case, but do not clearly separate the knowledge about behavior from the knowledge of diagnosis. Moreover, Console et al. [14] has shown that model-based diagnosis techniques can outperform algorithmic debugging of logic programs due to the required number of user interactions before identifying a single bug.

6 Conclusion

In this paper, we have described an approach that uses Model-Based Diagnosis for the location of erroneous statements in imperative programs. We assume faults are detected

due to discrepancies found in program execution on a set of test cases. The diagnosis system automatically transforms a program into a logical model by analyzing its source code. This model is then used together with a set of test cases showing desired input-output behavior to locate statements that are eligible to be the source for that fault. The language used is a subset of Java (use of a larger subset is described in [6]). Compared to earlier work, the model used is based on a transformation of the original program into a loop free form, which provides for effective search and diagnosis discrimination. Diagnosis candidates are sets of program statements which are then mapped back to locations in the source code for programmer interaction.

We have shown an empirical evaluation of different example programs using a debugger augmented with our diagnosis system, resulting in quick and direct focusing on the potentially faulty locations in the code.

Note that our approach, in contrast to verification techniques, aims at locating faults based on test cases instead of formally proving certain program properties.

Compared to traditional debugging (error location) techniques like Algorithmic Debugging and Program Slicing, our approach provides higher flexibility, the use of a generic and efficient problem solving algorithm, the ability to incorporate different models, and the ability (inherent in the problem solving algorithm) of diagnosing multiple faults.

References

1. Lieberman, H.: The debugging scandal and what to do about it. *Communications of the ACM* **40** (1997)
2. Weiser, M.: Programmers use slices when debugging. *Communications of the ACM* **25** (1982) 446–452
3. Shapiro, E.: *Algorithmic Program Debugging*. MIT Press, Cambridge, Massachusetts (1983)
4. Reiter, R.: A theory of diagnosis from first principles. *Artificial Intelligence* **32** (1987) 57–95
5. de Kleer, J., Williams, B.C.: Diagnosing multiple faults. *Artificial Intelligence* **32** (1987) 97–130
6. Mateis, C., Stumptner, M., Wotawa, F.: Modeling Java Programs for Diagnosis. In: *Proceedings of the European Conference on Artificial Intelligence (ECAI)*, Berlin, Germany (2000)
7. Stumptner, M., Wotawa, F.: Debugging Functional Programs. In: *Proceedings 16th International Joint Conf. on Artificial Intelligence*, Stockholm, Sweden (1999) 1074–1079
8. Friedrich, G., Stumptner, M., Wotawa, F.: Model-based diagnosis of hardware designs. *Artificial Intelligence* **111** (1999) 3–39
9. de Kleer, J., Mackworth, A.K., Reiter, R.: Characterizing diagnosis and systems. *Artificial Intelligence* **56** (1992)
10. Tip, F.: A Survey of Program Slicing Techniques. *Journal of Programming Languages* **3** (1995) 121–189
11. Murray, W.R.: *Automatic Program Debugging for Intelligent Tutoring Systems*. Pitman Publishing (1988)
12. Kuper, R.I.: Dependency-directed localization of software bugs. Technical Report AI-TR 1053, MIT AI Lab (1989)
13. Lloyd, J.W.: Declarative Error Diagnosis. *New Generation Computing* **5** (1987) 133–154
14. Console, L., Friedrich, G., Dupré, D.T.: Model-based diagnosis meets error diagnosis in logic programs. In: *Proceedings 13th International Joint Conf. on Artificial Intelligence*, Chambéry (1993) 1494–1499