JADE – AI Support for Debugging Java Programs*

Cristinel Mateis, Markus Stumptner, Dominik Wieland and Franz Wotawa†

Technische Universität Wien, Institut für Informationssysteme
Favoritenstraße 9–11, A-1040 Wien,
Email: {mateis,mst,wieland,wotawa}@dbai.tuwien.ac.at.

Abstract

Model-based diagnosis is a successful AI technique for locating and identifying faults in technical systems. Extending previous research on model-based diagnosis support for fault search in technical designs, we are building a model-based debugger for Java programs to provide intelligent support for the programmer trying to locate the source of an error. By using one or more models derived from the source code of the program without additional specifications except the Java semantics, the debugger guides the user towards potential sources for incorrect program behaviors, i.e., bugs.

1 Introduction

Finding, locating, and repairing faults in software is a difficult and time consuming task. Incorrect behaviors of programs are found by using test-cases or verification techniques, e.g., model checking [2]. While much effort has been spent on test theory, test methodology, and algorithms for automatic test-case generation, somewhat less work has been published on locating and repairing faults. Debugging approaches introduced in the past include program slicing [15], algorithmic debugging [13], dependency-based techniques [9, 8, 7], probability-based methods [1], and others. These traditional approaches are either specific to a programming language, use specialized algorithms, or require explicit user-interaction to locate a bug. Especially in maintenance, where the original developers may no longer be involved, debugging is very difficult and time consuming. An automated debugger for locating and fixing faults can help in such a situation.

In order to provide a well-founded theory for debugging and to improve results, the use of model-based diagnosis (MBD) for debugging was suggested [3]. Model-based diagnosis [12, 4] provides a general theory for diagnosis that has been used in a variety of domains, e.g., [10, 16], but is restricted to finding faults in hardware The idea behind using MBD in debugging is that programs are compiled to a logical description which is used by a diagnosis engine for computing bug locations and repair suggestions. Compilation is automatic and while models of programs may vary, the diagnosis engine remains stable. The capabilities of the approach depend highly on the mapping used, i.e., the resulting model. A model capturing all semantical aspects of a program allows for better results but at the cost of increased debugging time. A simpler model, e.g., one using dependency relations between variables, computes bug candidates for large programs in a reasonable time, but delivers too many candidates requiring additional user-interaction. However, it was claimed in [3] that MBD outperforms algorithmic debugging [13] when comparing the number of questions a debugger has to ask an *oracle* (i.e., any outside source of information, such as the user), in order to locate and fix a bug.

The principles of model-based debugging are depicted in figure 1. The program, in our case written in Java, is compiled into an internal representation. From this representation (together with a set of *model fragments*) a converter computes logical models for diagnosis. Model fragments represent a logical description of parts of a model, e.g., the behavior description of functions. Such knowledge has to be derived from the programming language semantics. For Java for example, a value-based model requires the model fragments of all basic functions and types of statements, e.g., for arithmetic functions, behaviors (e.g., $Nab(C) \rightarrow out(C) = in_1(C) + in_2(C)$ for the + operator) must be defined. The predicate Nab stands for *not abnormal*, saying that a function or statement is not responsible for an incorrect behavior. After building the model, which is done automatically, the model together with the specified behavior of the program, e.g., test-cases, is used by the diagnosis engine for finding bug candidates. The candidates can be further discriminated by adding additional knowledge, i.e., values of variables at specific locations within the program. A measurement selection algorithm selects the variable and location. The information about the value must be delivered by the user (or another oracle). The remaining candidates provide a link back to the original source code.

In this paper we describe the first `jade` prototype imple-

* This work was partially supported by the Austrian Science Fund project P12344-INF and project N Z29-INF.

† Authors are listed in alphabetical order.

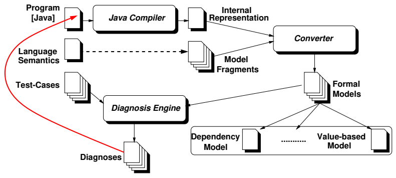

Figure 1: MBD in software debugging

mentation, a model-based debugger for Java programs. The model for Java programs and the basics rely on previous research [3, 5, 14]. `jade` currently uses a functional dependency model for debugging. The diagnosis engine used by `jade` is based on Reiter's algorithm [12]. Measurement selection, i.e., selection of variables that should be examined at a specific location within the program, uses a slightly modified version of [4]. The `jade` user interface is designed to guide the user through the code in an optimal fashion, with optimality defined as minimizing user interaction. Interface and diagnosis engine are designed to be as general as possible, i.e., changing the model does not cause changes in the diagnosis part and most of the interface. Apart from the development of the `jade` debugger, the *Java Diagnosis Experiments (Jade)* project aims at providing a theoretical basis for debugging Java programs and other object-oriented languages. Currently, we have developed a value-based model for Java and will add it to the `jade` debugger.

2 Modeling Aspects

Java is a strongly typed object-oriented programming language. A Java program is a collection of classes, and each class has a (possibly empty) set of instance variables, class variables, and methods used to implement the desired behavior of the class. For our purposes, we assume that the programs to be debugged can be compiled, i.e., they must be syntactically correct and pass semantic analysis. The methods described in this paper are used for locating functional faults detected at runtime, by the program terminating with a wrong variable value (but for the purpose of this paper, excluding aborts due to runtime errors). In addition, we assume that the correct program is a small variant of the buggy program. Hence, incorrect programs requiring substantial changes, e.g., new instance variables or classes, are out of the scope of this paper. Using this assumptions, the source of a misbehavior must be located in a method of a class. Since methods are similar to procedures or functions, the results of research concerning other languages [5, 14] can be adapted to locate a bug in a Java method. However, there are important differences, due to Java features such as Global variables, Inheritance, Late binding, and Interfaces. In addition, loop statements and recursive functions must be handled. In the following section we show how `jade` computes the dependency model and discuss these issues.

3 Dependency Model

The functional dependency model used by `jade` encodes causal relations between variables. We say that a variable y depends on a variable x iff changing the value of x may cause a change of variable y. Usually, several variables potentially influence a single variable x, written as (x, M_x), where M_x is the set of the variables influencing x. In general, dependencies can only be computed based on the actual behavior of programs. However, a good approximation of the dependency sets can be derived at compile time. In the following, we show how dependency sets are computed from programs and afterwards how they induce a logic model for debugging.

3.1 Dependency computation at method level

Each assignment statement of a method produces at least one functional dependency. Formally, a functional dependency arising from an assignment statement is a pair (v, M_v), where v is the target variable of the assignment and M_v is the set of the variables appearing in the right-hand side of the assignment and which v obviously depends on. Since the analysis of the program is performed statically, we do not deal with run-time values. Consequently, it is not always possible to unequivocally determine the object which a variable of reference type, occurring in a certain line of code, is bound to, because of both (i) dynamic binding, and (ii) selection and loop statements where the run-time program execution depends on run-time values which the conditional expressions evaluations depend on. Since we do not deal with run-time values, we can decide neither which branch of a selection statement is executed nor how many times the body of a loop statement is executed. For this reason, we consider all possible dynamic bindings resulted from all possible executions of a selection or loop statement. In case of a selection statement there are only two possibilities (e.g., once the "then" branch is executed and once the "else" branch). The case of a loop statement can be reduced to the case of a selection statement by rewriting the loop statement as a nesting of selection statements (cf. [11]). For each nested selection statement, (i) the selection condition is the loop condition, (ii) the "then" branch contains the body of the loop statement followed by the next nested selection statement, and (iii) the "else" branch is empty. The nesting depth is initially set to 1 and successively incremented until the addition of a further nested selection statement does not induce any new dependency.

Consider now an assignment statement $v.x = \langle Expr \rangle$ which occurs immediately after a selection statement which dynamically bound v in at least one of its branches. After that selection statement has been modeled, the variable v is assigned all possible bindings. Therefore, $v.x$ may be bound to several variables, one for each binding of v, and for each variable a separate functional dependency may arise. Additional functional dependencies may arise from the assignment statement if $Expr$ includes any call to any method n which modifies program variables by side effects (note that these dependencies will be introduced via other assignment statements in the body of n).

The model of an assignment statement s consists of the set of functional dependencies arising from s as shown above.

The model of a selection statement s consists of the set of functional dependencies computed by appropriately summarizing the functional dependencies arising from the selection condition and the statements of both branches of s (cf. [11]).

The model of a loop statement s consists of the model of the equivalent nesting of selection statements obtained from s according to the technique described above.

The model of a return statement s consists of the model of the assignment statement obtained from s such that the target variable is an only internally used variable and the right-hand side is the returned expression.

The model of a (void) method call statement consists of the set of the functional dependencies arising by side effects from that method call, as we have explained above.

The functional dependency model of a method m consists of the set of the models arising from the statements of m. The functional dependency model (FD model) of a Java program p consists of the models of the methods defined in the classes of p. Consider for instance the Java method $test$ from Figure 5. Each assignment statement i, $i = 4, \ldots, 8$, induces a single functional dependency FD_i which is contained by the model M_i of the assignment statement i, i.e., $M_i = \langle i, \{FD_i\}\rangle$. The models M_i, $i = 4, \ldots, 8$, turn out to be $M_4 = \langle 4, \{(s1_1, \{a_0, c_0\})\}\rangle$, $M_5 = \langle 5, \{(s2_1, \{b_0, d_0\})\}\rangle$, $M_6 = \langle 6, \{(s3_1, \{c_0, e_0\})\}\rangle$, $M_7 = \langle 7, \{(f_1, \{s1_1, s2_1\})\}\rangle$, and $M_8 = \langle 8, \{(g_1, \{s2_1, s3_1\})\}\rangle$. The model M of $test$ is the set of the models $M_i, i = 4, \ldots, 8$.

The model of a method is computed statically, hence no concrete objects are available and whenever they are needed, we simulate them by assigning auxiliary memory locations to the variables of reference type. These locations are uniquely indexed in increasing order at the moment when they are introduced in the model.

The use of auxiliary locations is needed in order to avoid to definitively assign objects to variables, because (i) one given variable might refer to different objects during the execution due to the phenomenon of dynamic binding, and (ii) an object may be assigned simultaneously to several variables due to the phenomenon of aliasing. The locations are exactly what we need in order to give the objects an identity and, whenever required, to dynamically bind a given variable to different objects in different places of the program. Three distinct types of locations may arise during modeling.

First, since the model of a method m is computed statically, we imagine an environment where a method call $o.m(\langle\text{actual parameter list}\rangle)$ occurs, where o is an imaginary object instance of the class where m is defined. For simplicity, we assume here that the names of the actual parameters coincide with the names of the formal parameters. To the object o and to the parameters of reference type, as well as for their fields, we assign locations even if no explicit statement (i.e., constructor invocation or array initialization) induces their creation. These locations are called *default locations* and are always assigned the index 0.

Second, auxiliary locations are introduced in the model of m whenever a statement of m directly leads to the creation of a new object, such as constructor invocations and array initializations. These are called *explicitly created locations*.

Finally, other auxiliary locations are introduced in the model of m whenever a call to a method n occurs in a statement of m and locations introduced in the model of n are transmitted outside n, hence visible in m. Obviously, the locations introduced in the model of m corresponding to the locations from the model of n visible outside n, are assigned indexes obtained by incrementing the index of the last location, regardless of its type, introduced in m. These locations are called *imported locations*.

The variables occurring in the methods change their values during the execution. This is handled by assigning a unique index to all occasions where a variable occurs as target of an assignment. That is, a variable v occurring in a method m may change its value several times during the execution of the same call of m and each time, the occurrence of v is assigned a different index. During the conversion, the current highest used occurrence index for each variable v that appears in the method m is stored in a global environment table env at the entry v and is updated each time v occurs as target of an assignment. Whenever v occurs in an expression, its last indexed variable occurrence stored in the table env is used for modeling the statement containing that expression.

A variable is represented in the model by its name (e.g., v) if it is a formal parameter or a local variable declared in the method, or by its name preceded by the index of a location, $i::v$, if it is the field of a variable of reference type which has been bound to the object of the location with index i.

Whenever we speak about variables in connection with a functional dependency, we actually refer to (indexed) variable occurrences (e.g.,v_3). If the variables are fields of objects, these occurrences will include also the location of the object containing the variable (e.g., $2::w_4$) in order to distinguish it from the same fields of other objects of the same class.

For space reasons, we omit the description of dependency computation (including recursive methods) at program level.

3.2 Mapping functional dependencies to logic

We now map the FD model derived in the previous section to a resulting logical model. We assume that the statements are given as a set $COMP$. For all $s \in COMP$, the set of dependencies $fd(s)$ is obtained from the FD model.

Functional dependencies describe behavior implicitly by describing influences between variables. Instead of speaking about real values, we can only speak about whether a value v is correct (written as $ok(v)$) or not (written $nok(v)$). Moreover, if a statement s (i.e., a diagnosis component) behaves correctly (i.e., $\neg AB(s)$ holds) and all input variables have a correct value then the value of the variable used as target in an assignment statement must be correct. Formally, the system description is given by:
$\forall_{(o,M) \in fd(C)} \left[\neg AB(C) \wedge \bigwedge_{x \in M} ok(x) \rightarrow ok(o)\right] \in SD$
where $C \in COMP$ is a statement. In addition, it is clear that a variable value cannot be correct and incorrect at the same time. Therefore, we have to add the rule $ok(v) \wedge nok(v) \rightarrow \perp$ to the model, where v denotes a variable occurrence.

The described model can be used together with a standard MBD algorithm for finding a set Δ of bug candidates such that, for a set of observations OBS, find a $SD \cup OBS \cup \{ok(s)|s \in \Delta\} \cup \{nok(s)|s \in COMPS - \Delta\}$ is consistent [12]. However, since the model as written above does not allow back propagation, the standard measurement selection algorithm [4] may sometimes not be able to provide a meaningful distinction between variable values that the user is to be queried for. Therefore, we introduce additional rules to avoid this problem. Since *ok* and *nok* are semantically too strong, we use new predicates *pok* and *pnok* to say that a value is *possibly ok* and *possibly nok*, respectively. The new predicates are weaker in the sense that the *pok* and *pnok* facts do no lead to a contradiction. The additional rules are:
$\forall_{(o,M) \in fd(C)} \left[\neg AB(C) \wedge pok(o) \rightarrow \bigwedge_{x \in M} pok(x)\right] \in SD$
$\forall_{(o,M) \in fd(C)} \left[\neg AB(C) \wedge pnok(o) \rightarrow \bigwedge_{x \in M} pnok(x)\right] \in SD$
As far as the measurement selection algorithm is concerned, we place diagnoses leading to *ok* and *pok* (and *nok* and *pnok*) together in one class.

Apart from the system description, the MBD approach requires a set of observations OBS. For software debugging, the observations are given by the specified behavior, in our case the expected input/output vectors. By comparing the specified output with the computed output, we can classify the correctness of variables. Variables v that are assumed to have the correct value lead to the observation $ok(v) \wedge pok(v)$. Variables with an incorrect value are represented by $nok(v) \wedge pnok(v)$.

4 Debugger Interface

The `jade` user interface should not be affected by changes to the diagnosis engine or the used language model. Moreover, the interface should present the user only such information relevant in a current situation. If a specific statement is a bug candidate and it is recommended to ask the user about variables changed or used by that statement, then all other variables are less important and should not figure prominently in interaction. Finally, the number of questions should be reduced and adapted to the current kind of statement, e.g., assignments or conditional statements.

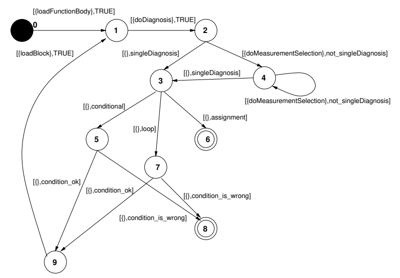

Figure 2: State chart of the `jade` debugger

The objective of the user interface is to guide the user through the debugging process. Therefore, in addition to presenting the outcomes produced by the diagnosis engine, e.g., diagnosis and measurement selection, the interface must provide functions for navigating through the code. This includes stepping through a block or body of a function depending on diagnosis results or program execution states. The state chart in Figure 2 gives an impression of the debugging process. Debugging starts with loading a method into `jade` and performing a diagnosis step (from state 1 to state 2). If diagnosis results in a single diagnosis (maybe comprising several components) it is checked whether a statement within the diagnosis is a loop or a conditional statement. If this is the case, the bug must be inside either the condition expression or a block. If the condition is correct, the debugger loads the block of statements and continues debugging. In the case of selection statements, the correct selection branch is chosen because the value of the expression can be computed by a Java evaluator. If the condition of the selection or loop statement is wrong, then debugging finishes. The same happens in the case of an assignment statement. In the current version, expressions cannot be debugged by `jade` (this is part of future research). Let us go back to state 2. If the result of diagnosis is not a single diagnosis, then the debugger has to discriminate between the bug candidates, using measurement selection [4]. The debugger asks the user about values of variables for given locations inside the program. Depending on the answers, diagnoses can be removed. Eventually, only one diagnosis should remain.

The current `jade` user interface is shown in figure 3. In the next section the usage of `jade` is demonstrated using a small Java program, and first empirical results are given.

4.1 An example debugging run

To demonstrate the efficiency of the `jade` debugger we implement a class library which provides the framework for

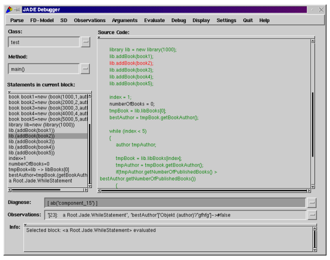

Figure 3: The `jade` user interface

modeling a small library project (which includes classes like *author*, *book*, and *library*). The method *test* of the class *EmpiricalTests* is known to behave incorrectly in certain cases and therefore has to be debugged. *test* creates a small sample library and then determines the author who has published the most books out of all authors whose books can be found in the library, by iterating over all books (respectively their authors) currently in the library and comparing them with the hardest-working author so far (cf.Figure 4).

The bug is located in the 'while' loop of the top-level block (statement 23) in Figure 4. For a more accurate location of the faulty statement we could specify the selection statement within the loop (w4) and finally the statement within the selection statement's then-branch (t1). This strategy, i.e. some kind of hierarchical bug location, is exactly what the `jade` debugger is performing. It starts with debugging the out-most block of a given method. Once a statement containing a bug is found the debugging method is called again for one of the statement's sub-blocks. The debugging sequence (i.e. the sequence of blocks in the debugging process) is controlled by the `jade` evaluator, that evaluates all statements of a block given an appropriate variable environment to start with.

Before a program can be debugged with the `jade` debugger it has to be loaded into the application. Choosing 'Parse Java Program' from the menu 'Parse' results in an internal parse tree representation of the Java source code, that can be seen as the base for the creation of the functional dependency model and the system description of the method in question. These models are automatically generated by the debugger using techniques described previously in this paper.

The debugging process can now be started by choosing 'Debug Selected Method' from the menu 'Debug'. This will generate all possible diagnoses, i.e. all sets of statements, which could be used as fault explanations. In our case 10 out of the 23 statements of the top-level block are marked as possible diagnoses. Obviously, this only gives a rough idea of all possible locations of the error and the complexity of the problem, but will not locate the desired single candidate diagnosis we would hope for. In order to exactly locate a bug in our method, we choose 'Debug Selected Method (Multiple Passes)' from the 'Debug' menu. This will result in computing all possible diagnoses (just as before) and automatically determining the ideal measurement point (i.e. statement) for the next step in the debugging procedure, i.e., the statement whose verification will eliminate the highest number of remaining diagnosis candidates. From now on we will focus on the latter technique of determining a single fault location using measurement selection.

```
class EmpiricalTests {
      public static void test() {
1.        int index;
2.        int numberOfBooks;
3.        author bestAuthor;
4.        book tmpBook;
5.        author author1 = new author(111,456,45623455,5);
6.        author author2 = new author(222,343,62093800,12);
7.        author author3 = new author(333,353,57343848,3);
8.        book book1 = new book(1000,1,author1);
9.        book book2 = new book(2000,2,author3);
10.       book book3 = new book(3000,3,author3);
11.       book book4 = new book(4000,4,author2);
12.       book book5 = new book(5000,5,author1);
13.       library lib = new library(1000);
14.       lib.addBook(book1);
15.       lib.addBook(book2);
16.       lib.addBook(book3);
17.       lib.addBook(book4);
18.       lib.addBook(book5);
19.       index = 1;
20.       numberOfBooks = 0;
21.       tmpBook = lib.libBooks[0];
22.       bestAuthor = tmpBook.getBookAuthor();
23.       while (index < 5) {
      w1.  author tmpAuthor;
      w2.  tmpBook = lib.libBooks[index];
      w3.  tmpAuthor = tmpBook.getBookAuthor();
      w4.  if (tmpAuthor.getNumberOfPublishedBooks() >
             bestAuthor.getNumberOfPublishedBooks()) {
        t1.   bestAuthor = bestAuthor; /* should be tmpAuthor !!! */ }
             else {
        e1.   numberOfBooks = numberOfBooks +
                 tmpAuthor.getNumberOfPublishedBooks(); }
      w5.  index = index + 1; } }
```

Figure 4: Method test

4.2 Top-level debugging

As mentioned above, the first task of the debugging process is to locate a faulty statement within the top-level block of a method. If one chooses 'Debug Selected Method (Multiple Passes)' this block is evaluated and a dialog box will allow the user to enter observations for all system outputs. Table 1 shows all outputs of the block's system description (i.e. statement/variable pairs) together with their observed values as computed by the `jade` evaluator for our example method.

From test runs we know that the variable *bestAuthor* at the method's output is incorrect. Therefore we set its value to false. As we cannot make any statements about the other outputs, we simply leave these fields open. This automatically sets the output connections of the internal system description to the given values. Note that all inputs are supposed to be correct and therefore set to true.

Now the diagnosis process is started returning all possible

Statement	Variable	Observed Value
[23] While Loop	tmpBook	Object(Book)
[23] While Loop	numberOfBooks	11
[23] While Loop	bestAuthor	Object(Author)
[23] While Loop	index	5

Table 1: System output observations.

```
1.  class SWExamples {
2.      public static void
              test(int a,b,c,d,e) {
3.          int f,g,s1,s2,s3;
4.          s1=a*c;
5.          s2=b*d;
6.          s3=c*e;
7.          f=s1+s2;
8.          g=s2+s3;
9.      }
10. }
```

(a) Source code

	Environment
	test(3,2,2,3,3)
2.	$a_{test} = 3, b_{test} = 2,$ $c_{test} = 2, d_{test} = 3,$ $e_{test} = 3$
3.	
4.	$s1_{test} = 6$
5.	$s2_{test} = 6$
6.	$s3_{test} = 6$
7.	$f_{test} = 12$
8.	$g_{test} = 12$
9.	
10.	$test(\ldots) = void$

(b) Evaluation Trace for test(3,2,2,3,3)

Figure 5: A simple Java method

diagnoses of the top-level block together with a set of system connections ranked by their capabilities to classify correct and incorrect diagnoses. If we always choose the one with the highest value, in our case we get variable lib in statement 18 (lib.addBook(book5);) as an optimal measurement point As this variable is of reference type, we might opt for inspecting its internal structure (i.e. instance fields, class variables, etc) what shows that at this particular point within the method the variable's state is correct. Therefore we press the 'correct' button. Internally a new rule is created, added to the theorem prover and finally the diagnosis process is started again. This is repeated as long as no single diagnosis can be found that explains the observed error. In our case 4 queries of the above type are needed to find statement 23 (i.e. the while loop) as the only possible location of the bug provided the inputs to the system during the interactive debugging and measurement selection process were correct, showing the effectiveness of the combination of diagnosis algorithm and measurement selection as a software debugging strategy.

4.3 Debugging loops and selection statements

As demonstrated above, the faulty statement within the outermost block of our test method could be located quite efficiently. In our case the debugging process proved the while loop to be buggy. Nevertheless a user might want to continue the debugging process to locate the bug within the loop body provided the loop condition is correct. The user is therefore asked to check this condition to make sure the bug can actually be found within the loop body. The next step is to determine the exact iteration in which the error occurred for the first time. To do so the user is presented a list of all evaluation environments which can be observed after each iteration of the loop body. Iteration 0 shows the input environment for the first iteration. By inspecting the environments' variables, the user has to select the first iteration in which the error can be spotted. In our case the error occurs in iteration 3 when the author of book4 (containing the overall best author) is compared with $bestAuthor$, but not stored in $bestAuthor$. So we select iteration 3 and click the 'apply' button. This results in the loop body being debugged using the environment of the beginning of the third loop iteration. The debugging process of the block itself works exactly as before. A new system description is created, incorrect outputs have to be set by the user (in our case $bestAuthor$ is again set to be false) and the debugging process is started. Among the five statements of the while loop we find the correct location of the bug (i.e. the selection statement w4) with a single user query. (Note that other types of loops, e.g., 'for' loops, can of course be debugged in a similar manner.)

We now know that the bug can be found within a selection statement (in our case w4). Just as before, the user is asked to verify the correctness of the selection statement's conditional before one of the two sub-blocks of the selection statement is debugged. In our case the evaluator leads the user into the then-branch. Theoretically a new system description would be generated, output connections would have to be set and debugging would continue. As there is only one possible location of the bug in our case, finding the solution is trivial. Statement t1 (assignment statement) of statement w4 (selection statement) of statement 23 (loop statement) contains the bug. The solution is presented to the user.

5 Results

In this section, apart from empirical results using Java example programs, we give some theoretical results for the functional dependency model. First, we show how the properties of the FD model influence the obtained diagnosis results. Afterwards we compare the solutions of the FD model with the outcomes produced by program slicing [15].

Desirable properties of a diagnosis system for debugging support include completeness of results, and a reasonable debugging time. Note that while completeness is obviously a very important requirement (a support system that tells the debugging programmer that he can ignore method X when in fact the error may lie there would be worse than useless), soundness is not a necessity. A debugger which computes a solution set that includes too many solutions (i.e., statements responsible for the actual fault, but also solutions that can be eliminated using more information) can still be used. However, again obviously, the larger the result set, the more difficult it is to find the actual bug locations. In the worst case, the debugger's diagnosis support will not allow distinguishing between different candidates (in other words, as in a conventional debugger, all the work must be done by the user). Next, a debugger has to compute all solutions within a reasonable time, although, "reasonable" is not well defined and depends on the intended application. For debugging very large programs it may mean a debugging time ranging from minutes to hours (because it may still cut down on the amount of code that has to be examined by the programmer). For smaller programs and debuggers interactively used by the user, debugging time should never exceed 1 second.

The `jade` debugger with dependency model can be proven to be complete (though of course not sound), i.e., it will not exclude incorrect statements from the set of potential errors, because all statements that may cause a wrong value are considered and therefore are diagnosis candidates. However, the number of candidates will be too high. Consider the example program from Figure 5, together with the specified values $f = 12$ and $g = 0$. In this case, the debugger returns the candidate $\{C5\}$ which could be eliminated when using a value-based model. Now assume $C5$ is incorrect, all other statements are correct, and apply the test case from Figure 5. From $C4$ and $C6$ and the input values we derive $s1_{test} = s3_{test} = 6$. Using these values together with $C7$ and $f_{test} = 12$, we get $s2_{test} = 6$. Now using this value together with the assumption $C8$ is correct leads to $g_{test} = 12$, contradicting our specified value $g_{test} = 0$. Hence, $\{C5\}$ is not a diagnosis wrt the value-based model. Using this example leads to the conclusion that the dependency model is not sound with respect to underlying program semantics. Finally, we give a limit for the debugging time. It is obvious that searching for all single bugs using our model is of the order $O(n^2)$, where n denotes the number of diagnosis components, i.e., in our case statements. Using empirical results from [6] we can expect that computing all single bugs for Java methods with several hundred statements will require less than a second.

Program slicing [15] and our dependency model are both static models, i.e., they are computed using the program structure and do not use the program behavior for fault localization. A program slice is defined as the part of a program that may influence the value of given variables up to a given position in the program. The slice for our example for variables $\{g\}$ and position 8 comprises the lines 5, 6, and 8. This result is equal to the one obtained by our dependency model and the question of the differences between both approaches arises. Because of a different presentation of results a direct comparison is not appropriate. Our dependency model is more hierarchically organized, e.g., a conditional statement is viewed as a single diagnosis component and sub-divided only after being identified as faulty. To allow a comparison of slicing with MBD using the functional dependency model, we assume an appropriate mapping. In this respect we obtain similar results from both techniques except in cases where several variables have a faulty value after program execution. In this situation MBD tries to minimize the source of the misbehavior leading to fewer solutions, while this is not the case with slicing. Assume that line 5 in the example method is faulty, leading to wrong values for variables f and g. The slice for f and g is the whole program, while our dependency model would deliver only line 5 as single fault candidate.

5.1 Empirical results

This section contains some results from tests of the `jade` debugger interface, to show the number of user interactions needed to find certain bugs in a Java method. This number can then be compared with the number of steps performed in a traditional debugging process, where the user normally has to go through the code statement by statement.

The first test method shown implements a binary full adder, that maps three inputs (parameters a, b, and c) to two outputs z and cn. The in/out behavior of such a method can easily be described by a table mapping all binary inputs to outputs. The tests carried out include mutations of function calls on the right-hand side of the various assignments, and mutations of parameters of function calls. Table 2 gives a detailed summary of all tests for the tested method.

Table 2 shows all modifications of the test method together with the new methods' behavior. 'Input' describes the input vector, which produces a faulty output ('Output') in relation to the target output ('Target'). The second column from the right shows the number of actual steps taken in the debugging process. The initialization, i.e. the verification of the system's output connections is not counted. E.g., if it takes 4 steps to find the buggy statement, this means that four queries to the user about the correctness of particular statement/variable pairs (see Figure 3) are necessary. The rightmost column shows the number of steps a normal debugger would have to go through to find the correct solution - if no breakpoints are set, that would correspond to evaluating the whole method statement by statement, i.e., bearing in mind that we are only looking at one method here (corresponding to a breakpoint set upon entering the method) i steps are necessary to find a bug in execution step i.

6 Conclusion

Debugging, i.e., finding, locating, and correcting faults in software, is a difficult and time consuming task. Therefore, an automated debugger would provide savings in multiple phases of a project, i.e., implementation, testing, and mainte-

File	Modification		Input	Output	Target	Steps I	Steps II
	line	code					
adder_f1.java	4	s2 = or(a,c);	001	11	10	2	4
	4	s2 = or(a,c);	100	11	10	2	4
adder_f2.java	7	cn = and(s3,s4);	011	00	01	2	7
	7	cn = and(s3,s4);	101	00	01	2	7
	7	cn = and(s3,s4);	110	00	01	2	7
adder_f3.java	12	s9 = and(s6,s8);	010	00	10	4	12
	12	s9 = and(s6,s8);	101	11	01	4	12
adder_f4.java	11	s8 = or(a,s7);	110	11	01	4	11
	11	s8 = or(a,s7);	111	01	11	6	11
adder_f5.java	14	s11 = or(s10,c);	101	11	01	4	14
	14	s11 = or(s10,c);	000	10	00	4	14
adder_f6.java	16	s13 = and(s9, s10);	010	00	10	4	16
adder_f7.java	12	s9 = or(s6,s7);	000	10	00	4	12

Table 2: Results of Java examples.

nance, and help reduce software development costs. We propose the use of model-based diagnosis for debugging. MBD provides a well-founded theory and algorithms for finding and fixing faults. The `jade` debugger is the first implementation of a Java debugger using MBD techniques. Because of the simple underlying dependency model, significant improvements are possible, but without a need for significantly altering the architecture and interface of the debugger. Experience with dependency based models for VHDL programs [5] also shows that these models can be used for very large programs.

In summary, the contributions of this paper are: a description of problems a debugger for object-oriented languages has to deal with, the description of an abstract dependency model for OO languages and its derivation from the program, an analysis of the capabilities and the problems underlying this model, a description of the debugger's interface and the examination of actual small test cases. Further research will concentrate on developing and evaluating value-based models optimized for different situations, e.g., structural faults, and on developing a methodology for reasoning with multiple models. The ultimate goal is the significant reduction of user interaction required for finding a bug, i.e., to provide a largely automated debugger.

References

[1] Lisa Burnell and Eric Horvitz. Structure and Chance: Melding Logic and Probability for Software Debugging. *CACM*, pages 31 – 41, 1995.

[2] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model Checking and Abstraction. *ACM Transactions on Database Systems*, 16(5):1512–1542, September 1994.

[3] Luca Console, Gerhard Friedrich, and Daniele Theseider Dupré. Model-based diagnosis meets error diagnosis in logic programs. In *Proc. IJCAI*, pages 1494–1499, Chambery, August 1993.

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

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

[6] Peter Fröhlich and Wolfgang Nejdl. A Static Model-Based Engine for Model-Based Reasoning. In *Proc.* 15^{th} *IJCAI*, Nagoya, Japan, August 1997.

[7] Daniel Jackson. Aspect: Detecting Bugs with Abstract Dependences. *ACM TOSEM*, 4(2):109–145, April 1995.

[8] Bogdan Korel. PELAS–Program Error-Locating Assistant System. *IEEE TSE*, 14(9):1253–1260, 1988.

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

[10] A. Malik, P. Struss, and M. Sachenbacher. Case studies in model-based diagnosis and fault analysis of car-subsystems. In *Proc. ECAI*, 1996.

[11] Cristinel Mateis, Markus Stumptner, and Franz Wotawa. Debugging of Java programs using a model-based approach. In *Proc. DX'99 Workshop*, Loch Awe, Scotland, 1999.

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

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

[14] Markus Stumptner and Franz Wotawa. Debugging Functional Programs. In *Proc.* 16^{th} *IJCAI*, Stockholm, Sweden, August 1999.

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

[16] Brian C. Williams and P. Pandurang Nayak. A reactive planner for a model-based executive. In *Proc.* 15^{th} *IJCAI*, pages 1178–1185, 1997.