JADE – A Step towards an Intelligent Debugger*

Cristinel Mateis and **Markus Stumptner**† and **Dominik Wieland** and **Franz Wotawa**‡
Technische Universität Wien,
Institut für Informationssysteme and
Ludwig Wittgenstein Laboratory for Information Systems,
Favoritenstraße 9–11, A-1040 Wien,
Email: {mateis,mst,wieland,wotawa}@dbai.tuwien.ac.at.

Abstract

The idea behind Model-based debugging is the notion of using the techniques of model-based diagnosis for locating errors in software. This requires the presence of a model of the software being debugged, which is generated automatically from the source code, and combined with test results and user input to identify statements or expressions that may be the cause of incorrect results. We describe the actual interactive use of model-based debugging support for Java programs, based on an extended version of a previously presented model for representing these programs. Even with an extremely simple model, the system uses the diagnosis outcomes to guide user interaction with the debugger and lead the user (programmer) towards the source of the error. We compare the behavior of the integrated debugger with the outcomes of methods from the programming community.

Introduction

In the last years various researchers have used model-based reasoning (MBR) for debugging of software (Console, Friedrich, & Dupré 1993; Bond & Pagurek 1994; Bond 1994; Missier, Xanthakis, & Travé-Massuyés 1994; Friedrich, Stumptner, & Wotawa 1999; Stumptner & Wotawa 1999; 1998; Mateis, Stumptner, & Wotawa 1999; 2000). All of the approaches have in common that they use a model derived from a program for locating (or, rarely, correcting) a bug. They differ in the considered programming language (ranging from purely logical languages, to hardware description languages, functional, sequential, and object-oriented languages), and the type of model (qualitative, dependency, or value-based models). While the purpose of previous research was to show the applicability of MBR in the software domain by introducing models, the purpose of this paper is to present first results of an implemented debugger prototype using different example programs. The JADE debugger, intended to be used for locating bugs in Java programs, currently implements a functional dependency model that extends our earlier work (Mateis, Stumptner, & Wotawa 1999; 2000). The granularity of the debugger, i.e., the elements of a program that are considered to be faulty or not, is set to the statement level.

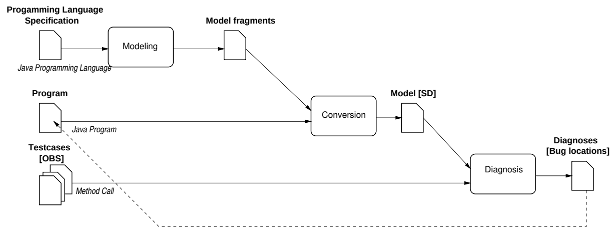

Figure 1: Model-based Diagnosis for Debugging

The JADE debugger combines the standard operation modes of diagnosis systems and standard debuggers. First, the program is converted into a functional dependency representation. This representation is compiled into a logical model. Once the program has been executed so that actual observations of its behavior can be provided, this logical model together with the specified behavior is used by the diagnosis engine to compute bug candidates that map back to positions (and statements) within the program to be debugged (see Figure 1). As mentioned, bugs are identified at the statement level. This reduces the number of diagnosis components and therefore debugging time.

The logical model described in this paper improves the model introduced in (Mateis, Stumptner, & Wotawa 1999). The original model could not be used for computing the most suitable points within a program to be next observed in order to distinguish between diagnosis candidates. (Observation points are formally characterized below by a pair consisting of the statement st and a query about the value of a variable after executing st). JADE uses a simplified version of the measurement selection algorithm introduced by (de Kleer & Williams 1987). Diagnoses are computed using the consistency-based approach (Reiter 1987; Console & Torasso 1990).

The contributions of this paper are: (1) extensions to the functional dependency model to improve measurement selection, (2) a demonstration of how the model is used for debugging and reduces the required user interactions, (3) a depiction of the properties of the model, (4) a comparison of the functional dependency model with program slic-

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

†Currently at Universität Linz, Data and Knowledge Engineering Group, Altenbergerstraße. 69, A-4040 Linz.

‡ Authors are listed in alphabetical order.

ing (Weiser 1982; 1984), a well known technique in the software debugging community, and finally (5) a presentation of empirical results. In the next section of this paper a short introduction in computing functional dependencies for Java programs is given. It is followed by the description of the improved functional model and a description of the functionality of the debugger using a small example. We conclude with theoretical and initial empirical results.

Computing functional dependencies

In this section, we give an overview of computing dependencies for Java program. For the sake of brevity we omit the discussion of object-oriented features such as dynamic binding in this paper and concentrate on the "basic" imperative features of the language. A more detailed technical description of the basic idea behind the conversion algorithm (excluding external side effects and method calls) can be found in (Mateis, Stumptner, & Wotawa 1999). A discussion about how to handle references and side effects is given in (Mateis, Stumptner, & Wotawa 2000). In order to compute dependencies we must consider that the variables occurring in the methods change their values during program execution. This is handled by assigning a unique index to all locations where a variable occurs as target of an assignment, i.e., 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. It is the various variable occurrences that dependencies are computed for. (Note that the variable occurrence indices are not, in general, the same as statement numbers, for more detail see (Mateis, Stumptner, & Wotawa 1999).)

Let x be an indexed variable occurrence of m and y either an indexed variable occurrence or a constant of m. We say that x *depends on* y iff a change of the value of y results in a change of the value of x. In other words, x depends on y iff there is an execution e_1 of m such that if at some point t during e_1 we alter the value of y (thus producing a new execution, e_2), then there is a later point t' such that the value of x in e_1 is different from its value in e_2. (Note that if conditions in m were dependent on the value of y, the code executed between t and t' may actually be different in e_1 w.r.t. e_2 since they may branch differently.) We say that x depends on a method n iff there is a method-call of n in m s.t. either (i) x is modified by side effects of that call of n, or (ii) x depends on the return value of that call of n.

Operationally, the notion of defining dependencies for sequential code has been used previously in software debugging. In (Korel 1988), three different types of dependencies (data, control, and potential dependencies) were distinguished, although the latter two categories are generally considered together: data dependencies are established by variable assignments, while control dependencies are due to branching or loop conditions that determine whether the subordinate code blocks (e.g., loop bodies or $then/else$ branches) are executed: variable occurrences in these blocks are dependent on variables in the branch or loop condition.

Formally, a functional dependency is a pair (x, M_x), where x is a variable occurrence and M_x is a set of variable occurrences, constants, and methods, such that x depends on every $y \in M_x$ (below we concentrate on variables for space reasons).

Whereas the variables correspond to the ports of traditional diagnosis components, the natural choice for components given the abstract nature of the dependency-based representation are statements (since representing expressions with a dependency-based model generally does not help discrimination: the outcome of the expression depends on the variables involved). The set of diagnosis components can be viewed as a diagnosis system where the connections are formed by the variable occurrences inside the components, i.e., components c_i and c_j are connected iff one component establishes the functional dependency (v_i, M) and the other, $(w_j, \{\ldots, v_i, \ldots\})$. Variable occurrences v_0 are inputs of the whole diagnosis system. A variable occurrence v_i is an output iff there is no other occurrence v_j such that $j > i$. Since during conversion indices are always increased, the resulting diagnosis system, i.e., the graph representation, is acyclic. No feedback loops occur in the system. This is different from the abstract dependency model in the domain of debugging concurrent programming languages (Friedrich, Stumptner, & Wotawa 1999), where explicit handling of feedback cycles is required.

Computing functional dependencies for Java programs requires compiling each method declared for a class. A method m of a Java program is converted by sequentially converting its statements into diagnosis components. Basically, a diagnosis component of m refers to a statement of m and contains the functional dependencies related to the occurrences of the variables which are modified in that statement. Note that from a single statement more functional dependencies may arise due to the fact that more variables may be modified by side effects. Consider for instance a method call which occurs in a mathematical expression whose ultimate value is assigned to a variable. Such a call can, apart from returning a numerical value, modify instance variables of some of its actual parameters of reference type.

There are four basic kinds of statements which may appear in the body of m: (i) assignment statements, (ii) selection statements, (iii) iteration statements, and (iv) method calls. There are also statements of other kinds which can be reduced to a basic statement, e.g., the return statement and the variable declaration statement can both be reduced to assignment statements. In the following, we sketch how each basic statement is converted into a set of functional dependencies. **Note:** For easier presentation, we omit the indices of variable occurrences from here on.

Assignments of the form $X = E$ are compiled into a set of dependencies at least including $\{X, vars(E)\}$, where $vars$ is a function delivering a set of variables used in the expression. If a method call is invoked inside the expression E then it is possible that additional dependencies occur. For example consider the assignment `s = m(a,b)` where `s`, `a`, `b` are variables and `m` is a method $m(x,y)$ such that the code of m includes the assignment `x=y`. In this case the set of dependencies for `s = m(a,b)` includes $\{a, \{b\}\}$.

Selection statements are converted by successively converting both branches and combining the dependency sets to form a single functional dependency set. In this set all variables occurring in the condition are added to every element, e.g., the element (x, M_x) is replaced by $(x, M_x \cup vars(E))$ where E is the condition expression.

Iteration statements are compiled in the same way as selection statements. The statements are converted and the variables occurring in the condition are added afterwards.

Method calls As indicated for the assignment conversion, method calls may introduce additional dependencies. These depend on the internal behavior (body) of the method and must be inserted appropriately into the dependencies of the block encompassing the method call.

A formal definition for compiling Java programs into the dependency representation can be found in (Mateis, Stumptner, & Wotawa 1999).

From functional dependencies to logical models

In the previous section we have recapitulated how functional dependencies can be derived using only the syntax of a given Java program. In this section we take the dependencies and show the resulting logical model. We assume that the statements are given as a set $COMP$, and that for all statements, the functional dependencies are defined. The set of functional dependencies for a statement st is written as $fd(st)$.

Functional dependencies describe behavior implicitly by describing influences between variable occurrences. Instead of speaking about real values, we can only speak about whether a value v is correct (written as $ok(v)$) or not (written $nok(v)$). We can further write that if a statement s, i.e., a diagnosis component, behaves correctly (i.e., $\neg AB(s)$ holds) and all input variable occurrences have a correct value then the value of variable occurrences 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, we know that it is impossible that a variable value is known to 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 SD, for each variable occurrence v in the program.

The described model can be used together with a standard MBD algorithm for computing bug locations. However, since the model as written above does not allow back propagation, the standard measurement selection algorithm (de Kleer & Williams 1987) sometimes will not 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 never 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 $nok(v) \wedge pnok(v)$.

The debugger

In this section we show how the functional dependency model can be used to locate a fault in a Java program, using the Java program given in Figure 2 as example. In Figure 2(b) the evaluation trace for the call *test(3,2,2,3,3)* is given. The trace only presents the lines of code which are involved in the current evaluation, and the new environments created. To distinguish different local variables they are indexed with the name of the method where they are declared. The (in this case nonexistent) return value of the method is also depicted, i.e., $test(3, 2, 2, 3, 3) = void$.

Most debuggers in use nowadays take the user through this evaluation trace (by letting the user interactively trigger individual execution steps or method calls). If an error occurs early in the trace, this approach can be quite effective. However, if the opposite happens, such a simple approach leads to the examination of unnecessary values with the additional possibility of ignoring a wrong value. Therefore, traditional debuggers allow the user to set break points, which somewhat reduces the amount of steps that have to be triggered explicitly. However, it is not always easy to place the break point at the desired location in a program. In addition, for recursive or looping functions a break point may be passed quite often during program execution in situations where the state of the program does not tell us anything about the fault. Hence, using such a debugger requires both experience and patience on the part of the user.

The functional dependency model can be used to avoid or at least alleviate the problems of a traditional debugger. Necessary for our approach is the availability of the source code, information about the expected outcome of a program, and the evaluation trace up to the detection of a misbehavior. From the source code the functional dependency model can be automatically derived. From this model and the misbehavior we can compute the subset of the program's statements which may be responsible for the wrong values. We illustrate our approach using the small example from Figure 2. The functional dependencies for the five statements (omitting indices) are given by:

$FD(C_4) = \{(s1, \{a, c\})\}$, $FD(C_5) = \{(s2, \{b, d\})\}$, $FD(C_6) = \{(s3, \{c, e\})\}$, $FD(C_7) = \{(f, \{s1, s2\})\}$, $FD(C_8) = \{(g, \{s2, s3\})\}$

where C_i denotes the statement in line i. From the dependencies, we get the logical model SD:

$$\begin{gathered}
\neg AB(C_4) \wedge ok(a) \wedge ok(c) \rightarrow ok(s1) \\
\neg AB(C_4) \wedge pnok(s1) \rightarrow (pnok(a) \wedge pnok(c)) \\
\neg AB(C_4) \wedge pok(s1) \rightarrow (pok(a) \wedge pok(c)) \\
\neg AB(C_5) \wedge ok(b) \wedge ok(d) \rightarrow ok(s2) \\
\neg AB(C_5) \wedge pnok(s2) \rightarrow (pnok(b) \wedge pnok(d)) \\
\neg AB(C_5) \wedge pok(s2) \rightarrow (pok(b) \wedge pok(d)) \\
\neg AB(C_6) \wedge ok(c) \wedge ok(e) \rightarrow ok(s3) \\
\neg AB(C_6) \wedge pnok(s3) \rightarrow (pnok(c) \wedge pnok(e)) \\
\neg AB(C_6) \wedge pok(s3) \rightarrow (pok(c) \wedge pok(e)) \\
\neg AB(C_7) \wedge ok(s1) \wedge ok(s2) \rightarrow ok(f) \\
\neg AB(C_7) \wedge pnok(f) \rightarrow (pnok(s1) \wedge pnok(s2)) \\
\neg AB(C_7) \wedge pok(f) \rightarrow (pok(s1) \wedge pok(s2)) \\
\neg AB(C_8) \wedge ok(s2) \wedge ok(s3) \rightarrow ok(g) \\
\neg AB(C_8) \wedge pnok(g) \rightarrow (pnok(s2) \wedge pnok(s3)) \\
\neg AB(C_8) \wedge pok(g) \rightarrow (pok(s2) \wedge pok(s3))
\end{gathered}$$

$$\begin{gathered}
ok(a) \wedge nok(a) \rightarrow \perp \\
ok(b) \wedge nok(b) \rightarrow \perp \\
ok(c) \wedge nok(c) \rightarrow \perp \\
ok(d) \wedge nok(d) \rightarrow \perp \\
ok(e) \wedge nok(e) \rightarrow \perp \\
ok(f) \wedge nok(f) \rightarrow \perp \\
ok(g) \wedge nok(g) \rightarrow \perp \\
ok(s1) \wedge nok(s1) \rightarrow \perp \\
ok(s2) \wedge nok(s2) \rightarrow \perp \\
ok(s3) \wedge nok(s3) \rightarrow \perp
\end{gathered}$$

In this example we assume that the method call *test(3,2,2,3,3)* should lead to values *f=12* and *g=0*, i.e., that line 8 should be *g=s2-s3* instead of *g=s2+s3*. For this case, we get observations OBS:

$$\begin{gathered}
ok(a) \wedge ok(b) \wedge ok(c) \wedge ok(d) \wedge ok(e) \wedge \\
ok(f) \wedge nok(g) \wedge pok(f) \wedge pnok(g)
\end{gathered}$$

Using $SD \cup OBS$ we get 3 diagnoses, each pinpointing a single possible bug location: $\{C_5\}, \{C_6\}, \{C_8\}$. The other statements can be ignored in this case. This initial result is similar to the one received by other techniques (Weiser 1982; 1984). Using the measurement selection algorithm from (de Kleer & Williams 1987) we can compute the optimal next question to be presented to the user in order to distinguish between the 3 candidates. The algorithm is based on minimizing the entropy and requires the existence of fault probabilities. For our example we assume that all statements are equally likely to be incorrect with probability 0.1. Using the 3 diagnoses and computing possible values we get the following results:

Variable	$ok(V) \vee pok(V)$	$nok(V) \vee pnok(V)$
s1	C_5, C_6, C_8	
s2	C_5, C_6, C_8	C_5, C_6
s3	C_5, C_8	C_5, C_6

Note that the values *ok*/*pok* (and *nok*/*pnok*) are combined because of their similar semantics. Using the diagnosis probabilities we receive the following entropy values:

Variable	HE
s1	-0.31993
s2	-0.58642
s3	-0.53298

So, it is recommended first to ask about the value of *s2* and afterwards about the value of *s3*. In the implementation, with the source code displayed in the background as reference, and the variable values of the current execution step available for examination, the question would be posed as follows (screen output in fixed type, user input in bold face):

`Is the outcome of statement 5 ok?`
`[yes/no]` **yes**
..after recomputing the diagnosis and optimal measurements..
`Is the outcome of statement 6 ok?`
`[yes/no]` **yes**
..again after computing the new diagnoses..
`The assignment statement 8 contains the bug.  Have a look at the expression being assigned.`

This small example shows that it is not necessary to go through the whole evaluation trace, it is sufficient to examine statements that can actually be responsible for wrong values. This process can be guided by the debugger in an optimal fashion using fault probabilities (even if these probabilities are wholly assumed, as is the case in our example). It should be noted that the dependency-based representation allows diagnosis times in the seconds range even for very large programs (Friedrich, Stumptner, & Wotawa 1999).

The interface of the current version of the JADE debugger is depicted in Figure 3. The JADE debugger asks the user questions about values of variables at specific positions within a program and recomputes diagnoses based on the given answers. The values of variables are computed using a Java run-time environment developed for our purpose. Although we are still experimenting with the best user interface configurations for programs that make heavy use of conditionals, loops, and recursive functions, the first results of the JADE debugger are quite promising.

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 (Weiser 1982).

Desirable properties of a diagnosis system for debugging support include completeness of results, and a reasonable

```
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

Line	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.	$test(3,2,2,3,3) = void$

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

Figure 2: A simple Java method

debugging time. Note that while completeness is obviously a very important requirement (consider a support system that tells the debugging programmer that he can ignore method X when in fact the error may lie there – not only will it have been of no help in this case, it would actually be misleading), soundness is not, for the simple fact that a debugger which computes a solution set that includes sound elements (i.e., all 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, i.e., the debugger's diagnosis support will not allow distinguishing between different candidates (in other words, as in a conventional debugger, all the work rests with the user). The requirement that a debugger should compute all solutions within a reasonable time is important. However, reasonable time is not well defined and depends on the intended application. For debugging very large programs "reasonable" 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 used interactively 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). Completeness means that all bugs can be potentially found. This must be the case for our model 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 2, together with the specified values $f = 12$ and $g = 0$.

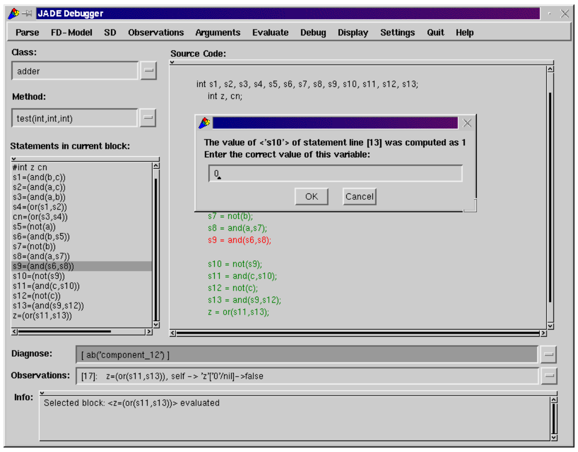

Figure 3: The JADE debugging window

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 2. 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 restricted to $O(n^2)$, where n denotes the number of diagnosis components, i.e., in our case statements. Using empirical results from (Fröhlich & Nejdl 1997) we can expect that computing all single bugs for Java methods with several hundred statements should be done in less than 1 second.

Program slicing (Weiser 1982; 1984) 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 possible influencing the value of given variables and not occurring within the program after a given position. For the example program above, the slice for variables $\{g\}$ and position 8 comprises the lines 8, 6, and 5. This result is equal to the one obtained by our dependency model and the question arise about the differences between both approaches. 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 the 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 slicing doesn't. Given the example, assume that line 5 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.

Empirical results

The example given below consists of a Java method *test* for a class *EmpiricalTests* implementing a binary full adder. The method maps three inputs (*a*, *b*, and *c*) to two outputs (*z* and *cn*). The source code comprises the *test* method and methods implementing a logical *and*, *not*, *or*. The following program (partly) shows the class declaration:

```
    class EmpiricalTests {
      public static void test(int a, int b, int c) {
1.        int s1, s2, s3, s4, s5, s6, s7, s8, s9, s10, s11, s12, s13;
2.        int z, cn;
3.        s1 = and(b,c);
4.        s2 = and(a,c);
5.        s3 = and(a,b);
6.        s4 = or(s1,s2);
7.        cn = or(s3,s4);
8.        s5 = not(a);
9.        s6 = and(b,s5);
10.       s7 = not(b);
11.       s8 = and(a,s7);
12.       s9 = or(s6,s8);
13.       s10 = not(s9);
14.       s11 = and(c,s10);
15.       s12 = not(c);
16.       s13 = and(s9,s12);
17.       z = or(s11,s13); }
// Function and, or not declarations ...
      static int not(int x) {
        if (x > 0) return 0;
        else return 1; }
...
    }
```

The in/out behavior of the *test* method can be easily described by a simple table mapping all binary inputs to outputs. In our case there are eight possible input vectors, which are mapped to four different output vectors. In the course of this debugging test session the test method was altered at randomly chosen code locations, i.e. statements. All in all, 8 different modifications to the source code were tested, all of which feature certain output anomalies with respect to the intended behavior (see Table 1).

An example run

The following example run shows a typical use of the JADE debugger. The interface of the current implementation is given in Figure 3. For instance, changing line no. 12 from *s9 = or(s6,s8);* to *s9 = and(s6,s8);* (file adder_f4.java in Table 1), the intended output will differ from the one actually computed by the Java runtime system. In this particular case an error can be observed by feeding the input vectors (010), (100), (011), and (101) to the system. Each such vector results in a wrong value of the output variable *z*. The incorrect program can now be parsed with the Java debugger in order to create an internal parse tree representation of the method in question. The next steps (performed automatically) are to generate the functional dependency model of the tested program and the system description belonging to the test method as described earlier.

The actual debugging process is started by specifying the observed output behavior of a particular method. If we choose 'Dynamic Observation' from the menu 'Observations' a dialog window will open up and allow the user to enter observations for all system outputs. In our case the variables *cn* of statement 7 and *z* of statement 17 are automatically considered outputs, since they are not connected to inputs of other components. As mentioned before, in this case an output deviation is spotted for variable *z*, whose value was computed as 1. Thus we set the target value of *z* to 0 indicating that an error occurred at exactly this output connection. As we are not aware of any errors involving the other output (*cn*), we leave this field empty.

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. This can give the user a rough idea of all possible locations of the error and the complexity of the problem, but will not locate a single candidate diagnosis. 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. This means that such statements are ranked highest in the measurement selection process, whose verification will eliminate the highest number of remaining diagnosis candidates. In our case variable *s10* of statement 13 was computed with the highest probability (see Figure 3). The value of *s10* as computed by the internal Java evaluator is 1. A close look at the intended behavior of our method shows that at this place *s10* should actually hold 0, causing the user to enter the corrected value of variable *s10*. Internally a new rule is created, added to the theorem prover and the diagnosis process is repeated again. This procedure is followed as long as no single diagnosis can be found, which solely explains the observed error. Once such a diagnosis is found, it is presented to the user as the only possible location of the bug provided the inputs to the system during the interactive debugging and measurement selection process were correct.

In order to find out more about the debugger's functionality and efficiency further tests with our test method were carried out. These tests included: (1) Mutations of function calls on the right hand side of the various assignments, (2) Mutations of parameters of function calls. Table 1 gives a detailed summary of all tests carried out with our simple test method.

Table 1 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

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
adder_f8.java	9	s6 = not(s5);	110	11	01	4	9

Table 1: Results from Java examples.

the system's output connections is not counted. If it takes for example 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., i steps are necessary to find a bug in statement (actually, execution step) i.

Conclusion

This paper presents the current implementation of the JADE debugger. This version of JADE implements a functional dependency model (Mateis, Stumptner, & Wotawa 1999) together with a model-based diagnosis engine for guiding the user (programmer) through a Java program for locating a bug. Although the used model is quite simple, the first tests indicate a reduction of the required number of user interactions when compared with traditional debuggers. In this paper we have modified a previous model in order to get better measurement selection results, and shown how user interaction should look like. In addition, we have given a comparison between the model-based approach and program slicing (Weiser 1982) showing the advantages of MBR.

Dependency models have the advantage that they can be used for debugging large programs at the expense of delivering too many bug candidates in some situations (Friedrich, Stumptner, & Wotawa 1999). Since a reduction in the number of candidates implies a further reduction in the number of user interactions, further improvements are possible (and, of course, highly desirable). This includes the incorporation of a value-based model for Java programs and an empirical comparison of their results when applied to programs.

References

Bond, G. W., and Pagurek, B. 1994. 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.

Bond, G. W. 1994. *Logic Programs for Consistency-Based Diagnosis*. Ph.D. Dissertation, Carleton University, Faculty of Engineering, Ottawa, Canada.

Console, L., and Torasso, P. 1990. Integrating models of correct behavior into abductive diagnosis. In *Proceedings of the European Conference on Artificial Intelligence (ECAI)*, 160–166. Stockholm: Pitman Publishing.

Console, L.; Friedrich, G.; and Dupré, D. T. 1993. Model-based diagnosis meets error diagnosis in logic programs. In *Proceedings of the International Joint Conference on Artificial Intelligence*, 1494–1499. Chambery: Morgan Kaufmann.

de Kleer, J., and Williams, B. C. 1987. Diagnosing multiple faults. *Artificial Intelligence* 32(1):97–130.

Friedrich, G.; Stumptner, M.; and Wotawa, F. 1999. Model-based diagnosis of hardware designs. *Artificial Intelligence* 111(2):3–39.

Fröhlich, P., and Nejdl, W. 1997. A Static Model-Based Engine for Model-Based Reasoning. In *Proceedings 15^{th} International Joint Conf. on Artificial Intelligence*.

Korel, B. 1988. PELAS–Program Error-Locating Assistant System. *IEEE Transactions on Software Engineering* 14(9):1253–1260.

Mateis, C.; Stumptner, M.; and Wotawa, F. 1999. Debugging of Java programs using a model-based approach. In *Proceedings of the Tenth International Workshop on Principles of Diagnosis*.

Mateis, C.; Stumptner, M.; and Wotawa, F. 2000. Locating bugs in Java programs – first results of the Java Diagnosis Experiments (Jade) project. In *Proceedings of the International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems*. New Orleans: Springer-Verlag. To appear.

Missier, A.; Xanthakis, S.; and Travé-Massuyés, L. 1994. Qualitative Algorithmics using Order of Growth Reasoning. In *Proceedings of the 11th European Conference on Artificial Intelligence (ECAI 94)*, 750–754.

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

Stumptner, M., and Wotawa, F. 1998. Model-based debugging of functional programs. In *Proceedings of the Ninth International Workshop on Principles of Diagnosis*.

Stumptner, M., and Wotawa, F. 1999. Debugging Functional Programs. In *Proceedings* 16^{th} *International Joint Conf. on Artificial Intelligence*.

Weiser, M. 1982. Programmers use slices when debugging. *Communications of the ACM* 25(7):446–452.

Weiser, M. 1984. Program slicing. *IEEE Transactions on Software Engineering* 10(4):352–357.