Locating Bugs in Java Programs – First Results of the Java Diagnosis Experiments (Jade) project*

Cristinel Mateis, Markus Stumptner and Franz Wotawa
Technische Universität Wien
Institut für Informationssysteme
Favoritenstraße 9-11, A-1040 Wien, Austria,
Email: {mateis,mst,wotawa}@dbai.tuwien.ac.at

Keywords: Model-based Reasoning, Software Debugging

Abstract

This paper describes the use of model-based reasoning for locating bugs in Java programs. Model-based diagnosis is a technology that uses a declarative, generic description of the behavior of the components occurring in a domain to construct a model of the overall system which can then be used at the desired level of abstraction to predict a system's behavior and derive assumptions about which parts of the system are incorrect. This approach is particularly enticing when applied to software since the model can be constructed from the program automatically. However, the actual choice of models poses interesting challenges. We show a simple model based on dependencies that can be used to diagnose very large programs, and walk through an example debugging session.

1 Introduction

Debugging, i.e. locating and correcting faults in programs, is generally assumed to be a difficult task. This holds especially in the case where a programmer needs to debug a program written by a different person. One reason is the size of the search space for debugging and the implicit connection between parts of the specification (i.e., the intended behavior of the program), and parts of the program (i.e., the realization). These implicit links are usually manifested as part of the programmer's mental model. In order to overcome the complexity problem and to eliminate large portions of the program from the focus of interest, programmers use slices [Wei82, Wei84]. Slices are program fragments representing statements that influence the value of a given variable. They can be automatically derived from the syntactical structure of a program at compile and at runtime. Other approaches to software debugging use logical models of the program [Sha83, FN94, Nil98], or dependencies between variables or the flow of control [Kup89, Jac95]. Apart from general program debugging, other researchers present approaches for dedicated tutoring systems [Kor88, Mur88]. All of the above research directions of program debugging have in common that they rely on their own algorithms and modeling paradigms. Some of them are limited to a single programming language and some require an additional specification (causing additional learning and writing effort for the programmers using the tool).

Apart from 'traditional' debugging research, in the past years several attempts for using model-based diagnosis (MBD) [Rei87, dKW87] for locating faults have been published. Some of them [CFD93, BP94, Bon94] show how MBD can be used for debugging Prolog-like programs. They compare the outcome of the MBD approach with Shapiro's Algorithmic Debugging [Sha83] and claim that MBD requires less user interaction for locating the fault. Other papers [FSW99, SW96, Wot99, SW99a] show how MBD can be used for debugging hardware description languages (using VHDL as example) and functional programming languages. Some of the ideas described therein can be used for almost all programming languages including procedural languages. To use MBD principles for software debugging a model must be available. A model or system description SD in the terminology of MBD

*This project was partially supported by the Austrian Science Fund (FWF) under project grant N Z29-INF. Cristinel Mateis was funded by the FWF under grant P12344-INF.

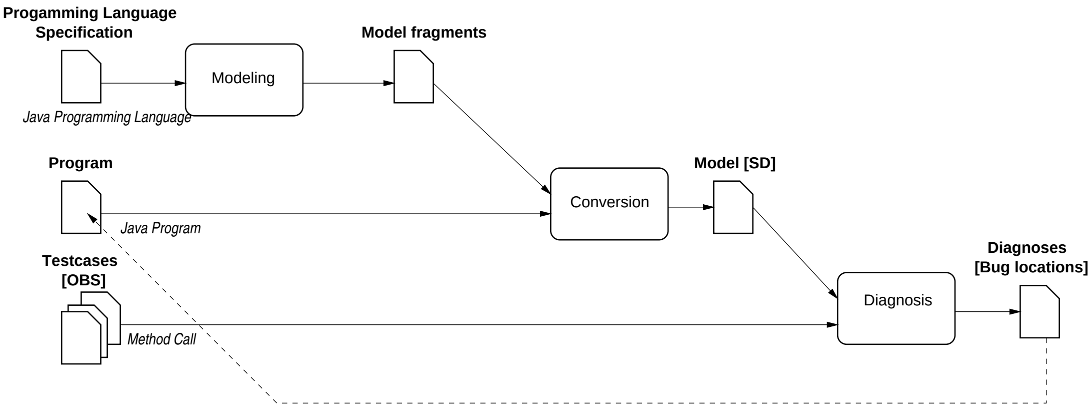

Figure 1: Model-based Diagnosis for Debugging

describes the behavior of system components $COMP$ and the structure of the system to be diagnosed. In model-based software debugging a model is the logical description of the potentially faulty piece of software, e.g., the set of statements and expressions used in the program. In addition to the model, a set of observations OBS of the system behavior is required. Then the model together with the observations is used directly for identifying the faulty parts. A basic advantage of the approach is that only the correct behavior of components must be known.

The necessary logical description, i.e., the model, of a program can be derived from the program and the programming language. The specification of a programming language describes the semantics of a program. The semantics are not given individually for every program, instead the behavior of the different language constructs is defined, and these constructs, e.g., conditional statements, expressions, or operators, can then be mapped to diagnosis components. For each of them, a logical model fragment is built. These model fragments together with the syntax of a concrete program allow to automatically derive a logical description for the program at compile time. Once a model is available, test cases, e.g., method calls with given argument values, can be used as observations in the search for a bug. The actual diagnosis step uses a standard model-based diagnosis algorithm, which returns diagnoses, i.e., sets of components that may be faulty. In our case these correspond to the above program constructs, i.e., the diagnosis maps back directly to the source code of the program.

The purpose of this paper is to give an overview of how Java programs can be modeled for model-based diagnosis. The overall modeling and diagnosis process is shown in Figure 1. Java is an object-oriented programming language. It is strongly typed and has a C-like syntax. A Java program is a collection of classes, and each class has a (possibly empty) set of instance 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 the semantics analyses. The methods described in this paper are for locating functional faults detected at runtime. Functional faults occur whenever the program terminates with a wrong variable value. In addition, we assume that the correct program is a small variant of the buggy program. Hence, faulty 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 previous research [FSW99, Wot99, SW99b, SW99a] can be used for locating a bug in a Java method. However, there are some differences. First, in Java the methods to be called need not to be known at compiletime. This behavior is called late binding. Consider for example the following program fragment.

```
GraphicsObject[] obj = new GraphicsObject[10];
setObjects(obj);
for(int i=0;i++;i<10) {
  obj[i].drawOn(graphicsContext);
}
```

and assume the existence of subclasses for `GraphicsObject`, e.g., `Circle`, `Point`, and `Line`, each implementing the method `drawOn`. If we detect a faulty behavior, e.g., an object is drawn in the wrong color, we must know the class of the object to choose the right `drawOn` method for debugging. The second difference is that Java allows the use of global variables, including class variables.

In the rest of this paper we present an abstract model which can be used for debugging Java programs. Statements are mapped to diagnosis components and variables are mapped to the connections between the diagnosis components. As a result, the granularity of the diagnosis result is set at the statement level, so that only statements can be classified as being correct or behaving abnormally. A finer classification, such as identifying individual expressions inside statement as the source of a bug, is possible by changing the model, leading to more diagnosis components and a corresponding increase of the size of the diagnosis model. Hence, diagnosis time increases as well, making this approach infeasible for larger programs. Restricting diagnosis granularity to the statement level provides enough information to the user and retains effective computability.

To be self contained we briefly describe the basic model-based diagnosis definitions. The theoretical background and algorithms are given in [Rei87, dKW87]. Given a system $(SD, COMP)$, where SD denotes a logical description of the system and $COMP$ the set of components used in the system, and observations OBS. A set $\Delta \subseteq COMP$ for a diagnosis problem $(SD, COMP, OBS)$ is a diagnosis if $SD \cup OBS \cup \{\neg AB(x) | x \in COMP \setminus \Delta\} \cup \{AB(x) | x \in \Delta\}$ is consistent. The predicate AB is used to express that a component behaves abnormally, i.e., is incorrect. The system description SD needs to contain the specification of the correct behavior based on the assumption $\neg AB(x)$ for a component x and the interconnection between components. The theory of fault models and the use of knowledge about physical impossibilities can be found in [dKMR92, FGN90].

2 A dependency model

The model described in this paper converts a given program into a set of diagnosis components by statically analyzing the statements of the methods appearing in the classes of that program. All classes of the program are converted by successively converting all methods of all classes. A method of a class is converted by successively converting all statements of that method. The set of diagnosis components is then used to derive the system description SD of the program which is used by the diagnosis algorithm.

The conversion of a program is performed at compile time, i.e., its output does not depend on values which arise at run time. The only values on which the result of the conversion depends are the single constants used in the program, which are known at compile time.

2.1 Defining Functional Dependencies

A Java method m may contain two kinds of variables: simple variables (i.e., variables whose values are of simple type, e.g., integer, boolean, etc.) and variables of reference type (i.e., variables whose values are not of simple type, e.g., objects and arrays).

The variables of m comprise the variables locally declared in m, the fields of the class owning the method m, the formal parameters of m, and the fields of the variables of reference type occurring in m. Because of the phenomenon of dynamic binding of the variables of reference type, i.e., that the same variable may point to different objects in the memory during different points in the execution of the same call of m, we assign auxiliary locations to variables of reference type. These locations are denoted by order numbers which correspond to the locations created in memory at run time whenever a constructor-invocation statement is encountered. When a new object is created by a constructor invocation, the variables arising from the fields of the new object are bound to the location referred by that object, i.e., the l-value of the object, and not to the value of the object, i.e., the r-value of the object[1]. In this way we can properly keep trace of the variables arising from the fields of the objects which can be accessed through different variables of reference type pointing to the same memory location (i.e., aliasing phenomenon). On the other hand, due to selection statements, the location assigned to a variable of reference type might not be known with precision at a given point of the method at compile time and there might be several locations, only one of which corresponds to the memory location assigned at run time to that variable. However, we consider a static analysis and therefore, we take into consideration all these possible locations.

A variable is represented in our model either by (i) its name (e.g., v) if it is a formal parameter or a local variable declared in the method, or (ii) by its name preceded by the number of a location (e.g., $4 :: v$) if it is the

[1] In Java, every variable has an r-value which is defined as the actual value stored in the variable, and it also has an l-value which is defined as the name of the variable associated to the location in the memory where the r-value is stored.

field of an object. In the following, whenever we talk about variables, we imply variables of one of the forms (i) and (ii).

We have to consider that the variables occurring in the methods change their values during program 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. It is the various variable occurrences that dependencies are computed for.

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 methodcall of n in m s.t. either (i) x is modified by side effects of that methodcall of n, or (ii) x depends on the return value of that methodcall of n.

Operationally, the notion of defining dependencies for sequential code has been used previously in software debugging. In [Kor88], 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 $then$ and $else$ branches) are executed: variable occurrences in these blocks are dependent on variables in the branching 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 $y \in M_x$.

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 [FSW99], where explicit handling of feedback cycles is required.

2.2 Method Conversion

A method m of a Java program is converted by sequentially converting its statements into diagnosis components.

Before starting the conversion of the statements of the method into diagnosis components, default locations are created for the formal parameters of reference type and for the fields of reference type deriving from the formal parameters. The object for which the method m is supposed to be called[2] we assign the default location with the order index 0, i.e., those fields v of o which are accessed in m are represented as $0 :: v$ in our model.

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 could, apart from returning a numerical value, modify instance variables of some of its actual parameters of reference type.

The functional dependencies of the diagnosis components of m are used to derive the system description SD of m in terms of logical rules involving (i) the modified variables, the constants, and the methods appearing in the functional dependencies, and (ii) the statements corresponding to the diagnosis components. The system descrip-

[2]Remember that the analysis is static and we do not actually know the object o for which the method m is invoked, therefore we have to use a generic placeholder object for the purpose.

tion SD of m along with a set of observations OBS (i.e., a set of atoms encoding the correctness or wrongness of some observed variables of m) is given in the input of a standard model-based diagnosis algorithm with a theorem prover which computes a set of diagnoses representing the possible faulty statements of m.

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 diagnosis components. A more detailed technical description of the basic idea behind the conversion algorithm (excluding external side effects and method calls) can be found in [MSW99].

2.2.1 Assignment Statement Conversion

An assignment statement consists of a left-hand side which always leads to a variable and a right-hand side which is an expression. An expression may be a constant, a variable, a method call, or an operation whose operands are expressions.

The assignment statement is converted into a diagnosis component containing the dependency of the variable occurrence corresponding to the left-hand side on the constants, variable occurrences, and methods appearing in the right-hand side. In addition, other functional dependencies, corresponding to variable occurrences visible in m and modified by side effects of method calls appearing in the assignment statement, on constants, methods, and variable occurrences visible in m may arise.

2.2.2 Selection Statement Conversion

A selection statement consists of a boolean selection expression and two sets of statements corresponding to the "then" and "else" branches, respectively.

First the diagnosis components due to side effects of the boolean selection expression are computed. Then the statements of each branch are converted into the corresponding sets of diagnosis components which are stored in the two appropriate instance variables of the diagnosis component corresponding to the selection statement.

The functional dependencies of the diagnosis component corresponding to the selection statement are obtained by summarizing the functional dependencies of the diagnosis components of the two branches corresponding to the variables visible outside the selection statement. To this set are added them the dependencies on the operands of the boolean selection expression.

2.2.3 Iteration Statement Conversion

For simplicity and without loss of generality, here we consider only "while" iteration statements. A "while" statement consists of an exit boolean expression and a body which is a set of statements executed at each iteration.

A loop can be viewed as a nesting of identical selection statements having an empty "else" branch and the same boolean expression and body of the "then" branch as the loop. We start by converting the outermost selection statement and by increasing the depth of the nesting until the addition of a new selection statement induces no new functional dependencies for variables visible outside the iteration statement. The diagnosis component corresponding to the iteration statement is the diagnosis component computed for the outermost selection statement of the equivalent nesting.

2.2.4 Method Call Statement Conversion

In order to convert a method call statement, a model (i.e., a set of diagnosis components) of the called method, call it n, is needed. The functional dependencies of the variables visible outside n (e.g., fields of the formal parameters of reference type, fields of the object owning n) are summarized and, after having replaced the formal parameters by the actual parameters, side-effects diagnosis components containing these functional dependencies are created. If n returns a value (i.e., n does not return the "void" type), the summarized functional dependency corresponding to the return statements of n is used for the conversion of the statement where the method call of n occurred.

Above, we have supposed that the model of a method n is available before a method call of n is encountered in the body of a method m. Unfortunately, this is not the case in general. If the model of n has not been computed yet, and if m is not mutually recursive with m, we can simply interrupt the conversion of m, compute the model of n, and continue then the conversion of m. However, this technique cannot be applied if m is recursive with n; in this case, the body of n contains at least one statement s containing a method call of m, but model of m has not been computed yet and we cannot convert s, hence we cannot compute the model of n. That is, an infinite sequence of alternative calls to the procedures for computing the models of n and m would arise. Therefore, a fixpoint computation is adopted. Since the non-recursion is a trivial special case of recursion, the algorithm works also for non-recursive methods.

First of all, the methods of the Java program are grouped in maximal sets of mutually recursive methods, i.e., all methods from a set are recursive with exactly all methods of that set. The fixpoint computation algorithm is employed separately for each set S of mutually recursive methods. The fixpoint computation algorithm for S starts by converting an arbitrary method r of S. The method r may have been chosen either (i) because it has been called in the body of a method not belonging to S before the set S has been processed, or (ii) because it was the first method stored in S.

At the i-th iteration of the fixpoint computation, in order to compute the i-th model M_i of a method m from S, the models N_{i-1} of the methods n from S computed at the previous iteration are used. The fixpoint is reached when the models of all methods of S computed at the current iteration have no new diagnosis components w.r.t. to the models computed at the previous iteration. Since the program has a finite number of methods and every method of the program has a finite number of statements and since a diagnosis component cannot be alternatively validated and invalidated at different iterations, it is clear that the fixpoint exists and it is reached in a finite amount of time. The question which has still to be answered is how the models of the methods from S are initialized, that is, what are the models M_0 of the methods m of S? The answer is as follows. The initial models M_0 of the methods m from S are the models computed by ignoring the block of statements in the body of m containing calls to methods from S.

One could say that all initial models M_0 of the methods m from S may be set to the empty set. This might work in some cases but it does not work in general. Consider for instance the case where m returns a value, i.e., it is not of the type "void". If M_0 is empty, we do not know anything about the returned object and at the first iteration we cannot get the model of the context where the method call of m occurs. A solution could be to set M_0 to the model corresponding to the initialization of the returned value with an imaginary object, but this has not an intuitive meaning.

3 Mapping functional dependencies to a logical model

In the previous section we describe how the 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 functional dependencies are defined. Accessing the dependencies is done using the fd function.

Functional dependencies describe behavior implicitly by describing influences between variables. Instead of speaking about real values, we only can speak about whether a value v is correct (written as $ok(v)$) or not (written $nok(v)$). We further can write that 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 variables 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, where v denotes a variable (with index) used 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 [dKW87] sometimes may not be able to provide a meaningful distinction between variable values that the

```
1. class Arithmethics {
2.     static int myMult(int x, int y) {
3.         if (x == 0)
4.             return 0 ;
5.         else
6.             return myAdd(y,myMult(x-1,y));
7.     }
8.     static int myAdd(int x, int y) {
9.         if (x == 0)
10.            return y ;
11.        else
12.            return myAdd(x-1,y)+1;
13.    }
14.}
```

(a) Source code

Line	Environment
	myMult(1,1)
2.	$x_{myMult} = 1, y_{myMult} = 1$
3.	
6.	myMult(0,1)
2.	$x_{myMult} = 0, y_{myMult} = 1$
3.	
4.	$myMult(0, 1) = 0$
6.	myAdd(1,0)
8.	$x_{myAdd} = 1, y_{myAdd} = 0$
9.	
12.	myAdd(0,0)
8.	$x_{myAdd} = 0, y_{myAdd} = 0$
9.	
10.	$myAdd(0, 0) = 0$
12.	$myAdd(1, 0) = 1$
6.	$myMult(1, 1) = 1$

(b) Evaluation Trace for myMult(1,1)

Figure 2: User-defined Addition and Multiplication Operations

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 in weaker in the sense that the pok and $pnok$ facts do no lead to a contradiction. The additional rules are:

$$\forall_{(o,M)\in fd(C)} \left[\neg AB(C) \wedge pok(o) \rightarrow \bigwedge_{x\in M} pok(x)\right] inSD$$
$$\forall_{(o,M)\in fd(C)} \left[\neg AB(C) \wedge pnok(o) \rightarrow \bigwedge_{x\in M} pnok(x)\right] inSD$$

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 Debugging with the dependency model

In this section we show how the functional dependency model can be used to locate the fault. We use the Java program given in Figure 2 as our running example. This program implements a multiplication and addition operator for non-negative integer values. In figure 2 the evaluation trace for the call *myMult(1,1)* 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 return value of method is also depicted, e.g., $myMult(1, 1) = 1$.

Most debuggers nowadays in use take the user through this evaluation trace. If an error occurs early in the trace this approach can be effectively used. 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. Obviously, this helps is some cases. However, it is not always easy to place the break point on the desired location in a program. In addition, for recursive 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 experience of the user.

The functional dependency model can be used to avoid 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 those statements which may be responsible for the wrong values. We illustrate our approach using the small example from Figure 3. The functional dependencies for the five statements (ignoring indices) are given by:

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

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

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

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

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., the line 8 should be *g=s2-s3* instead of *g=s2+s3*. For this case we get observations OBS:

$$ok(a) \wedge ok(b) \wedge ok(c) \wedge ok(d) \wedge ok(e) \wedge ok(f) \wedge nok(g) \wedge pok(f) \wedge pnok(g)$$

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 [Wei82, Wei84]. Using the measurement selection algorithm from [dKW87] we can compute the optimal next question to be ask 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 the all statements are equally likely to fail with probability 0.1. Using only 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, the questions might look like:

```
Debugger:  Is the outcome of statement line 5 ok?  [yes/no]
User:  yes
```

..after recomputing the diagnosis and optimal measurements..

```
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 3: A simple Java method

```
Debugger:  Is the outcome of statement line 6 ok?  [yes/no]
User:  yes
```
..again after computing the new diagnoses..
```
Debugger:  The assignment statement line 8 contains the bug.  Have a look
at the expression.
```

This small example shows that it is not necessary to go through the whole evaluation trace. It is sufficient to have a look at statements may influencing wrong values. This process can be guided by the debugger in an optimal fashion using fault probabilities. It should be noted that the dependency-based representation allows diagnosis times in the seconds range even for very large programs [FSW99].

5 Conclusion

This paper has described a dependency-based method for diagnosing Java programs. Once the behavior of the individual components has been defined, the model-based approach enables an automatic conversion of arbitrary application problems into a logical representation suitable for diagnosis. The approach is powerful enough to represent iterative code including recursive functions, abstract enough to allow diagnosis runs for large programs, and generic enough to be applicable to different imperative languages although individual features may differ from what we have presented here for Java.

This work builds on our previous experience with building model-based debugging systems for VHDL and functional languages. The main limitation of the representation shown here is that discrimination is limited, but this can be avoided as shown in our example by using debugging results to guide the developer through the evaluation trace in a manner comparable to conventional debuggers, but with semantic guidance. It is also possible to incorporate more detailed models in a comparable manner to our VHDL-oriented diagnosis systems, an approach we have not presented here for space reasons. Overall, the model-based approach provides an effective framework for modeling, a flexible, non-brittle approach to diagnosis that can accommodate multiple faults and provide help in measurement selection based on the structure of each individual application program. It also offers relatively high independence of actual diagnosis algorithms, and the ability to combine multiple models for more detailed reasoning about the sources of erroneous behavior in programs.

References

[Bon94] Gregory W. Bond. *Logic Programs for Consistency-Based Diagnosis*. PhD thesis, Carleton University, Faculty of Engineering, Ottawa, Canada, 1994.

[BP94] G. W. Bond and B. Pagurek. A Critical Analysis of "Model-Based Diagnosis Meets Error Diagnosis in Logic Programs". Technical Report SCE-94-15, Carleton University, Dept. of Systems and Computer Engineering, Ottawa, Canada, 1994.

[CFD93] Luca Console, Gerhard Friedrich, and Daniele Theseider Dupré. Model-based diagnosis meets error diagnosis in logic programs. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 1494–1499, Chambery, August 1993. Morgan Kaufmann.

[dKMR92] Johan de Kleer, Alan K. Mackworth, and Raymond Reiter. Characterizing diagnosis and systems. *Artificial Intelligence*, 56, 1992.

[dKW87] Johan de Kleer and Brian C. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32(1):97–130, 1987.

[FGN90] Gerhard Friedrich, Georg Gottlob, and Wolfgang Nejdl. Physical impossibility instead of fault models. In *Proceedings AAAI*, pages 331–336, Boston, August 1990. Also appears in Readings in Model-Based Diagnosis (Morgan Kaufmann, 1992).

[FN94] Peter Fritzson and Henrik Nilsson. Algorithmic debugging for lazy functional languages. *Journal of Functional Programming*, 4(3), 1994.

[FSW99] Gerhard Friedrich, Markus Stumptner, and Franz Wotawa. Model-based diagnosis of hardware designs. *Artificial Intelligence*, 111(2):3–39, July 1999.

[Jac95] Daniel Jackson. Aspect: Detecting Bugs with Abstract Dependences. *ACM Transactions on Software Engineering and Methodology*, 4(2):109–145, April 1995.

[Kor88] Bogdan Korel. PELAS–Program Error-Locating Assistant System. *IEEE Transactions on Software Engineering*, 14(9):1253–1260, 1988.

[Kup89] Ron I. Kuper. Dependency-directed localization of software bugs. Technical Report AI-TR 1053, MIT AI Lab, May 1989.

[MSW99] Cristinel Mateis, Markus Stumptner, and Franz Wotawa. Debugging of Java Programs using a Model-Based Approach. In *Proceedings of the Tenth International Workshop on Principles of Diagnosis*, Loch Awe, Scotland, 1999.

[Mur88] William R. Murray. *Automatic Program Debugging for Intelligent Tutoring Systems*. Pitman Publishing, 1988.

[Nil98] Henrik Nilsson. *Declarative Debugging for Lazy Functional Languages*. PhD thesis, Linköping University, April 1998.

[Rei87] Raymond Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–95, 1987.

[Sha83] Ehud Shapiro. *Algorithmic Program Debugging*. MIT Press, Cambridge, Massachusetts, 1983.

[SW96] Markus Stumptner and Franz Wotawa. Model-based program debugging and repair. In *Proceedings IEA/AIE*, Fukuoka, 1996.

[SW99a] Markus Stumptner and Franz Wotawa. Debugging Functional Programs. In *Proc.* 16^{th} *IJCAI*, Stockholm, Sweden, August 1999.

[SW99b] Markus Stumptner and Franz Wotawa. Detecting and locating faults in hardware designs. In *AAAI 99 Workshop on Intelligent Software Engineering*, Orlando, Florida, 1999.

[Wei82] Mark Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–452, July 1982.

[Wei84] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.

[Wot99] Franz Wotawa. New Directions in Debugging Hardware Designs. In *Proceedings IEA/AIE*, 1999.