Jade – Java Diagnosis Experiments Status and Outlook*

Markus Stumptner and **Franz Wotawa**
Technische Universität Wien
Institut für Informationssysteme
Paniglgasse 16, A-1040 Wien, Austria,
Email: {mst,wotawa}@dbai.tuwien.ac.at

Abstract

Model-based diagnosis is a successful AI technique for diagnosing physical systems but faces new challenges when applied to finding faults in software, i.e., debugging. Previous work has mostly dealt with modeling the behavior of programming languages with exploitable special properties: logic, concurrent, or functional. In this paper we describe the objectives, the current state, and the addressed research issues of the Jade project. The aim of the project is to develope the theory and practice of applying model-based diagnosis to software debugging of object-oriented programs. The Java language has been chosen as a vehicle for the project due to its relatively simple semantics, its ubiquity (if not now, then in the near future), and its imperative and object-oriented characteristics.

1 Introduction

Model-based diagnosis (MBD) [Reiter, 1987; de Kleer and Williams, 1987] is a technique traditionally used for identifying faults in physical (e.g., electrical, mechanical, and electronic) systems. The basic assumption of model-based diagnosis is that if the underlying principles of a particular domain can be captured in a model, that model can then be used to locate faults in arbitrary systems that are composed of the components of that domain, resulting in increased flexibility and better separation between domain knowledge and inference engine.

For several years, researchers have worked on extending the scope of MBD to new domains, in particular that of software debugging. Initially, papers in the application of model-based techniques to software focused on pure logic programs [Console *et al.*, 1993] - i.e., the programs themselves already were logical theories. This allowed using the program as its own system description, thus avoiding the problem of transforming the former into the latter. It suggested the use of clauses as components for diagnoses, thus providing systems with limited numbers of components. And, since clauses in a logic program are independent (the program as a theory is simply a set of clauses), alterations of the program are trivial to produce without violating the semantics of the language. Later work defined representation for types of software that are closer to mainstream programming languages, but still possess properties that allow focusing on declarative subsets: so-called hardware design languages such as VHDL that can be considered as general-purpose concurrent programming languages [Friedrich *et al.*, 1996], and functional languages [Stumptner and Wotawa, 1998a].

The goal of this paper is to give an overview of the *Java Diagnosis Experiments (Jade)* project and first results, e.g., a functional dependency model for a Java subset. The goal of Jade is to extend previous work into the domain of conventional object-oriented programming languages, exemplified by the language Java. Jade is intended to examine issues such as modeling, fault models and representation of background knowledge about system and programmers, measurement selection, and a number of other issues that have been identified as being relevant for future development of the approach.

2 Objectives and research issues

The goal of the Jade project is to extend and examine the application of model-based diagnosis techniques to programming languages in general, and in particular to object-oriented languages due to their widespread usage and their position at the forefront of programming language research. The vehicle chosen for the project is the language Java, for a number of reasons:

- Java is a general-purpose language with widespread usage predicted in the coming years
- Java is strongly typed and has relatively simple semantics, in particular compared to C++, the most widespread OOPL.
- Java provides specific features for developing and running applications over the World Wide Web, which is expected to present new challenges for the MBD approach
- Java is expected to be widely used in the future by people with little programming experience as it becomes the language of choice for developing small, ad-hoc WWW applications. In this context, a knowledge-based debugger that guides an inexperienced user through the debugging cycle would be especially important.

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

The goal of the Jade project is the examination of the formal underpinnings required for using Model-Based Diagnosis in a standard software development environment, and the development of an experimental system for solving actual diagnosis cases with Java programs.

The Jade objectives can be summarized as follows:

1. Development of a theory of model-based software debugging.
2. Description of Java semantics in terms of logical models usable for diagnosis.
3. Examination of the use of alternative models for scalable diagnosis and their flexible use in multi-model in an diagnosis environment.
4. Development of an intelligent debugging environment for Java programs based on the theoretic results.
5. Collection of example programs and establishment of a set of benchmarks for diagnosis evaluation.

The Jade project is divided into several parts, each considering some aspects of the problem.

- *Theoretical aspects:* This part of the project will aim at the development of a theory of model-based software debugging (MBSD), building on the results gained in our previous work [Friedrich *et al.*, 1996; Stumptner and Wotawa, 1998c; 1999]. Examination of different approaches to the modelling of programming language constructs, in particular with regard to object-oriented languages. The results gained will enter into the design of the actual diagnosis system for debugging Java code.
- *Applying MBSD principles to object-oriented (Java) programs:* The use of MBSD for Java requires the description of Java semantics in terms of logical models usable for diagnosis. Depending on a the characteristics of individual programs, different models may be developed for use in different situations (see next section). This part of the project will also deal with any required extensions to the general diagnosis principles for dealing with special Java language constructs. (An example for this would be the way in which the model defined in [Friedrich *et al.*, 1996] is based on the concurrent programming constructs of the VHDL language.)
- *Use of alternative models for diagnosis:* A topic of high relevance for the practical application of diagnosis algorithms to programs is the choice of the correct representation. This part of the project will examine the use of alternative models for diagnosis as mentioned in the previous section. For example, large programs may require a more abstract representation to make diagnosis feasible. A goal is also the definition and examination of multi-model diagnosis algorithms which can, e.g., switch between different levels of abstraction for a more efficient and flexible diagnosis process.
- *Development of an intelligent debugging assistant:* This is the most strongly implementation-oriented part of the project. It will deal with the development of an intelligent debugging environment for Java programs based on the theoretic results from the previous sections. This would also require (for experimentation and instrumentation purposes) the implementation of a Java interpreter. Work on this part of the project would proceed in parallel with the more theoretical stages to guarantee that the environment will be ready for use when needed for experimenting with different diagnosis models and algorithms.
- *Accumulating a set of example programs:* Evaluation and testing of the diagnosis algorithms will require the compilation of a set of example programs of different size, complexity, and application domain. This set of programs will be used as the base of a set of benchmarks for the (performance- and quality-related) evaluation of different diagnosis approaches. This part of the project has been concluded for the time being.

```
   public void demo ( int from , int to )
   {
      int start , stop , i ;
1.    if ( from < to ) {
2.       start = from ;
3.       stop = to ;
      } else {
4.       start = to ;
5.       stop = from ;
      }
6.    i = start ;
7.    while ( i <= stop ) {
8.       // Do something ...
9.       i = i + 1 ;
      }
   }
```

Figure 1: A small Java program

3 Java and possible errors

Java is an object-oriented programming language (OOPL) which, in contrast to its most obvious forebear C++, has much simpler semantics making it easier to use for experienced and non-experienced programmers. Java is largely platform independent since most environments generate instructions for a standardized virtual Java machine which can be then interpreted on the fly by the Java interpreter.

While Java is syntactically similar to C++ (see Figure 1 for a small example program), it is significantly simpler and smaller, with many features avoided because they were found to make C++ programs unsafe and hard to read, most importantly pointer arithmetic. Certain features were omitted primarily for the sake of simplicity or streamlining, such as $\#define$ (traditionally a good source for errors due to different semantics from the rest of the language), $typedef$, $enum$, $union$ and $struct$, global variables, standalone functions (everything is a method), friend functions (everything in a package is a friend of everything else in the package), operator overloading and multiple inheritance. A number of features have been added to Java to make it safer, including

array bounds checking, garbage collection, concurrency, interfaces and packages. Nonetheless, as with any other programming language (object-oriented or not), the discovery of bugs in Java programs is still a difficult task.

Certain bugs, such as division by zero or trying to access an object through a null reference, are detected at run time. On the other hand, so-called *functional bugs* that lead to an erroneous output or a non-terminating execution of the program cannot be naturally detected by the compiler. Examples of functional bugs are (i) omitting an operator (as writing "i" instead of "i+1"), (ii) using the wrong operator (as in writing "i++" instead of "++i") or the wrong variable (as in writing "a[i]" instead of "a[j]"), (iii) missing and improper initializing of variables, (iv) improper modification of a value stored in the variable, (v) errors with loop index initializations, adjustment, or exit tests that lead to the erroneous value of a variable (such as the classical "off-by one errors" studied in the debugging literature). Other sources of bugs in Java programs are the side effects which are due to the fact that objects and arrays are generally reference data types in Java, so that, e.g., given two instances $l1$ and $l2$ of an user-defined class $list$, if the assignment $l1 = l2$ is performed, the result will be aliasing of the two variables: $l1$ will point to the object pointed by $l2$, whereas a C++ programmer might instinctively expect a copy operation. Other examples of side effects are method parameters of reference data types which are modified inside the method. These parameters are always passed by reference, although it might seem that they are passed by value, and their alterations inside the method are viewed also outside the method.

4 The functional dependency model for Java

In this section we introduce the way Java programs are converted into a set of functional dependencies. Java programs consist of class declarations with defined methods and they are converted by successively converting all methods of all classes. We consider that the variables occurring in the methods change their values during the 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 occurence of v is assigned a different index. We denote by *VAR* the set storing the indexed variable occurrences of m. Let $x, y \in$*VAR*. 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, $e2$), 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 branched differently.) 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 $then$ and $else$ branches) are executed: variable occurrences in these blocks are dependent on variables in the branching or loop condition.

We do not discuss output statements or methods here that do not involve variables (e.g., some kind of screen output); for our purposes they can be adequately simulated by using variables as placeholders.

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, such that x depends on every $y \in M_x$. For each set fd of functional dependencies, we define a directed graph $DG_{fd} = \langle V, A \rangle$, called the dependency graph of fd. The vertices of V are the variable occurrences appearing in fd. There exists a directed arc in A from a vertex x to a vertex y iff $(x, M_x) \in fd$ and $y \in M_x$.

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, $(x, \{\ldots, v_i, \ldots\})$. 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 *et al.*, 1996].

Before examining the diagnosis process, we first present the conversion from method to component and dependency set. The union of two functional dependency sets fd and fd' is defined in the obvious way,
$fd \cup fd' = \{(x, M_x \cup M'_x) | (x, M_x) \in fd \wedge (x, M'_x) \in fd'\} \cup \{(x, M_x) | (x, M_x) \in fd \wedge (x, M'_x) \notin fd'\} \cup \{(x, M_x) | (x, M_x) \in fd' \wedge (x, M'_x) \notin fd\}$.

A method m is converted by applying the function $conv$ to m. This function returns a sequence of diagnosis components $[c_1, \ldots, c_n]$ in the same order as the original statements $st_1, \ldots, st_n$ occur within the method. Attributes of diagnosis components store the functional dependencies fd, the type $type$, and (depending on the type) sets of subcomponents (e.g., $then$ and $else$ branch for selection statements). The conversion algorithm is given in figure 2. The total set of components in the system is the set of components in the top level component sequence combined with the components occurring in the transitive closure of the subcomponent relationships (i.e., by following nested subcomponents to the lowest level).

The conversion algorithm uses the functions sum_{FD}, $addvars$, and $rename$, and a local array, vn, which, dur-

Statements ::=	***function*** $Statements.conv(VAR)$
{ StatementList }	$\forall X \in VAR : vn(X) = 0$; /* *note that* vn *is local* */
	$Statements.conv := StatementList.conv(vn)$
StatementList ::=	***function*** $StatementList.conv(vn)$
Statement StatementList'	$StatementList.conv := [Statement.conv(vn) \mid StatementList'.conv(vn)]$
$\mid \epsilon$	$\mid StatementList.conv := \emptyset$
Statement ::=	***function*** $Statement.conv(vn)$
Assignment	$Statement.conv := Assignment.conv(vn)$
| SelectionStatement	$\mid Statement.conv := SelectionStatement.conv(vn)$
| IterationStatement	$\mid Statement.conv := IterationStatement.conv(vn)$
Assignment ::=	***function*** $Assignment.conv(vn)$
id **=** *Expr* **;**	$c := new(DiagnosisComponent)$; $c.type = ASSIGN$;
	$index := vn(id) + 1$;
	$c.fd := \{(id_{index}, Expr.vars(vn))\}$;
	$vn(id) := index$;
	$Assignment.conv := \{c\}$
SelectionStatement ::=	***function*** $SelectionStatement.conv(vn)$
if *Expr Statements*$_{then}$	$c := new(DiagnosisComponent)$; $c.type = SELECTION$;
else *Statements*$_{else}$	$c.then := Statements_{then}.conv(VAR)$;
	$c.else := Statements_{else}.conv(VAR)$;
	$c.fd := addvars(rename(sum_{FD}(c.then), vn), Expr.vars(vn)) \cup$
	$addvars(rename(sum_{FD}(c.else), vn), Expr.vars(vn))$;
	$\forall (v_i, M_{v_i}) \in c.fd : vn(v) := vn(v) + 1$;
	$SelectionStatement.conv := \{c\}$;
IterationStatement ::=	***function*** $IterationStatement.conv(vn)$
while *(Expr) Statements*	$c := new(DiagnosisComponent)$; $c.type = ITERATION$;
	$c.sub := Statements.conv(VAR)$;
	$c.fd := addvars(rename(sum_{FD}(c.sub), vn), Expr.vars(vn))$;
	$\forall (v_i, M_{v_i}) \in c.fd : vn(v) := vn(v) + 1$;
	$IterationStatement.conv := \{c\}$
Expr ::=	***function*** $Expr.vars(vn)$
id	$Expr.vars := \{id_{vn(id)}\}$
$\mid const$	$\mid Expr.vars := \emptyset$
| Expr$_1$ *Operator Expr*$_2$	$\mid Expr.vars := Expr_1.vars(vn) \cup Expr_2.vars(vn)$

Figure 2: Converting Java methods into diagnosis systems (Corresponding syntax productions given on the left)

ing the conversion, stores the current highest used index for each variable occurring in the block of statements for which $conv$ was called. sum_{FD} summarizes the functional dependencies given for a sequence of statements to one functional dependency set. For example, the program fragment

x **=** *a* **;**
x **=** *x* **+** *b* **;**

is converted into $[c_1, c_2]$ with $c_1.fd = \{(x_1, \{a_0\})\}$, $c_2.fd = \{(x_2, \{x_1, b_0\})\}$ and $DG_{c_1.fd \cup c_2.fd} = \langle\{a_0, b_0, x_1, x_2\}, \{(x_1, a_0), (x_2, b_0), (x_2, x_1)\}\rangle$. The summarized dependencies computed by sum_{FD} for the whole program fragment are $\{(x_2, \{a_0, b_0\})\}$.

The function sum_{FD} is defined as follows.

1. Combine all dependencies: $fd := \bigcup_{x \in \{c_1, \ldots, c_n\}} x.fd$;
2. Collect all variable occurrences: $VAR := \bigcup_{(v, M_v) \in fd}(\{v\} \cup M_v)$;
3. Compute DG_{fd};
4. Let *IVAR* be the set of initial variable occurrences, i.e., those with index 0 (i.e., vertices of DG_{fd} with no outgoing arc) and *FVAR* be the set of final variable occurrences, i.e., those with the highest used index for that variable (i.e., vertices of DG_{fd} with no incoming arc);
5. Compute the set FD' containing a functional dependency (v, M_v) for each $v \in$ *FVAR*, where $M_v := \{w \in$ *IVAR* $\mid w$ is reachable in DG_{fd} from $v\}$;
6. Return FD'.

Note that the set of vertices of DG_{fd} coincides with VAR and that DG_{fd} is acyclic. Moreover, sum_{FD} is polynomial in the size of the input: Steps 1, 2, and 4 are clearly polynomial. In step 3, DG_{fd} can be computed in linear time [Tarjan, 1972]. Finally, in step 5, FD' can be computed in polynomial time as follows:

1. $FD' := \emptyset$; $D := DG_{fd}$;
2. For each $v \in VAR$, let $M_v = \{v\}$ if $v \in$ *IVAR* and $M_v = \emptyset$ otherwise;
3. While D is not empty do:
   (a) Remove a vertex w from D with no outgoing arc;
   (b) If $w \in$ *FVAR*, add (w, M_w) to FD';
   (c) For each arc $(v, w) \in D$, $M_v := M_v \cup M_w$;
4. Return FD'.

The function $addvars(fd, vars)$ adds all variables from $vars$ to every dependency from fd, i.e., $\{(v, M_v \cup vars) \mid (v, M_v) \in fd\}$ is given back as result. It is obvious that $addvars$ is polynomial.

The function $rename(fd, vn)$ replaces in fd every variable occurrence v_0 by $v_{vn(v)}$ (i.e., the initial occurrence of v in fd is the occurrence of v with the current highest index in the underlying block of statements calling $rename$) and every variable occurrence v_i s.t. $i > 0$ by $v_{vn(v)+1}$ (i.e., the final occurrence of v in fd is the next (highest) occurrence of v in the underlying block of statements calling $rename$). Clearly, $rename$ is polynomial.

We illustrate the conversion algorithm using the small Java example from figure 1. According to the algorithm, the first statement to be converted is the selection statement (labeled **1**). A new diagnosis component $c1$ is created with the type $SELECTION$. The statements for both branches are converted separately. For the *then* and *else* branches we get $[c2, c3]$ and $[c4, c5]$ with $c2.fd = \{(start_1, \{from_0\})\}$,

$c3.fd = \{(stop_1, \{to_0\})\}$ and $c4.fd = \{(start_1, \{to_0\})\}$, $c5.fd = \{(stop_1, \{from_0\})\}$, respectively. So we get $c1.fd = \{(start_1, \{from_0, to_0\}), (stop_1, \{from_0, to_0\})\}$. The next statement is assignment **6**. We obtain a component $c6$ with $c6.fd = \{(i_1, \{start_1\})\}$. Finally, to convert the loop, for assignment **9** we introduce a diagnosis component $c9$ with $fd = \{(i_1, \{i_0\})\}$. Combining and renaming leads to $\{(i_2, \{i_1\})\}$. So $c7.fd = \{(i_2, \{i_1, stop_1\})\}$. The conversion result is the sequence $[c1, c6, c7]$, with $c2, c3, c4, c5, c8, c9$ occurring as subcomponents.

The function $conv$ calls itself recursively a number of times which is linear in the number of selection and iteration statements of m. At any recursive call of $conv$, the diagnosis components corresponding to the assignment statements are clearly computed in polynomial time, so $conv$ is polynomial. This presentation of the conversion process is significantly simplified for space reasons.

Using model-based diagnosis for locating a bug within a program requires the existence of a model for the program. In the previous section we introduced the conversion of programs, in our case Java methods, into a sequence of diagnosis components together with functional dependencies between used variables. In order to generate a logical sentence, i.e., the model, we convert the dependencies using the following principle. A bug-free statement changes the variables according to the underlying (Java) semantics. In the case of functional dependencies we reformulate the idea as: *If the statement is assumed to behave correctly and all variables used as input are also correct, then the statements output must be correct.*

Given a sequence of diagnosis components $[c_1, \ldots, c_n]$ with associated functional dependencies, the system description SD_J used for diagnosis is defined as follows.

$\forall i \in \{1 \ldots n\}, \forall (v, M_v) \in c_i.fd$:

$$\neg ab(c_i) \wedge \bigwedge_{x \in M_v} \neg ab(x) \rightarrow \neg ab(v)$$

Consider again the example from figure 1 and assume that statement **3** is changed to *'stop = from ;'*. Then the functional dependencies remain the same as before, i.e.,
$c1.fd = \{(start_1, \{from_0, to_0\}), (stop_1, \{from_0, to_0\})\}$
$c6.fd = \{(i_1, \{start_1\})\}$
$c7.fd = \{(i_2, \{i_1, stop_1\})\}$.
The system description for the program given by the sequence $[c1, c6, c7]$ is given by the sentence: $SD_J =$
$\{\neg ab(c1) \wedge \neg ab(from_0) \wedge \neg ab(to_0) \rightarrow \neg ab(start_1)$,
$\neg ab(c1) \wedge \neg ab(from_0) \wedge \neg ab(to_0) \rightarrow \neg ab(stop_1)$,
$\neg ab(c6) \wedge \neg ab(start_1) \rightarrow \neg ab(i_1)$,
$\neg ab(c7) \wedge \neg ab(i_1) \wedge \neg ab(stop_1) \rightarrow \neg ab(i_2)\}$.

Assuming that the inputs are correct, i.e., $\neg ab(from_0) \wedge \neg ab(to_0)$ and the variable *stop* observed after the last statement has the wrong value, i.e., $ab(stop_1)$, we have the following diagnosis problem:
$(\{c1, c6, c7\}, SD_J, \{\neg ab(from_0), \neg ab(to_0), ab(stop_1)\})$
It is easy to see that the only diagnosis that can be computed is $\{c1\}$, which includes the actual error as a subcomponent. Diagnosis at the subcomponent level can be achieved separately or combined hierarchically with the top level.

Compared to the dependency-based model for VHDLDIAG [Friedrich *et al.*, 1996], this representation provides a finer granularity due to the separation of a single dependency between variables into dependencies between multiple variable occurrences. The next step will be to test and expand the model based on results with real-world programs. Based on experience with VHDLDIAG, diagnosis times for large systems can be expected in a comparable range, i.e., several seconds per example.

5 Conclusion and related research

This paper has presented the goals of the Jade project and a model of Java programs that explicitly records dependencies between variables in the program. The model builds upon our earlier work on dependency-based representations and shares their advantage of being suitably abstract to make possible effective diagnosis computation for large (real-world) programs. However, the earlier representations were designed for special purpose programming languages where particular semantic aspects, e.g., concurrency, could be exploited [Friedrich *et al.*, 1996], whereas the current model explicitly considers the sequential nature of the Java code.

The automatization of finding bugs in programs is a topic which has attracted researchers in the past. Some other proposed techniques use dependencies between variables and statements or knowledge about the program structure [Kuper, 1989; Korel, 1988], combined with specialized search procedures. Program slicing [Weiser, 1982; 1984] is a technique to reduce the overall search space for debugging. Dependency analysis, albeit requiring a separate specification, is discussed in [Jackson, 1995]. Other debugging approaches use the semantics of the programming language, such as [Shapiro, 1983]. Unlike these more traditional debugging approaches, each with its own algorithms and heuristics, a debugging framework based on model-based diagnosis can be used for a variety of languages and model requirements. It has been shown that dependency models [Friedrich *et al.*, 1996] and more detailed models with value-based semantics [Console *et al.*, 1993; Bond and Pagurek, 1994; Bond, 1994; Stumptner and Wotawa, 1998c; 1998a] can be represented using the same underlying principles and algorithms. While [Console *et al.*, 1993; Bond and Pagurek, 1994; Bond, 1994] introduce models for debugging logical programs, [Stumptner and Wotawa, 1998a] gives a model for a simple functional language, and [Stumptner and Wotawa, 1998c] for VHDL, a hardware description language similar to Ada. A more detailed discussion of model-based debugging approaches can be found in [Stumptner and Wotawa, 1998b].Other approaches for debugging include the use of qualitative models and probabilities for order of growth reasoning [Missier *et al.*, 1994] and debugging of very large legacy systems [Burnell and Horvitz, 1995].

The model for Java programs described in this paper also uses model-based diagnosis as framework. Unlike the dependency model from [Friedrich *et al.*, 1996] for concurrent programs, we have specified a model for sequential programs extending [Friedrich *et al.*, 1996] by considering the sequence of statements during model construction. Further work will include the extension of the model to improve discrimination by avoiding the anomalies of simple dependency representa-

tions described in [Moriconi and Winkler, 1990].

The model therefore allows to localize certain bug occurrences in Java programs by using knowledge about incorrect outputs (or incorrect parts of the output), specified in terms of observations of incorrect values. Compared to previous dependency-based representations [Kuper, 1989; Korel, 1988], it is an improvement because it utilizes standard MBD implementation and standard software MBD representation techniques, although language-dependent heuristics will still be important in focusing search.

Compared to earlier work in model-based program diagnosis, it directly tackles the issue of modelling sequential behavior of programs at a level of detail commensurable with effectively diagnosing larger systems.

Compared to traditional debugging techniques such as program slicing and dependency analysis [Weiser, 1984; Korel, 1988], the model-based approach provides higher flexibility in terms of errors involving multiple variables and in terms of providing a ready-made theory for actually defining and identifying diagnoses.

References

[Bond and Pagurek, 1994] 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.

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

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

[Console *et al.*, 1993] 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.

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

[Friedrich *et al.*, 1996] Gerhard Friedrich, Markus Stumptner, and Franz Wotawa. Model-based diagnosis of hardware designs. In *Proceedings of the European Conference on Artificial Intelligence (ECAI)*, Budapest, August 1996.

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

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

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

[Mateis *et al.*, 1999] Cristinel Mateis, Markus Stumptner, and Franz Wotawa. A Functional Dependency Model for Java Programs. Technical Report DBAI-TR-99/xx, Technische Universität Wien, Institut für Informationssysteme 184/2, Paniglgasse 16, A-1040 Wien, Austria, 1999.

[Mateis, 1999a] Cristinel Mateis. Jade – Java Test Cases. Technical Report DBAI-TR-99/xx, Technische Universität Wien, Institut für Informationssysteme 184/2, Paniglgasse 16, A-1040 Wien, Austria, 1999.

[Mateis, 1999b] Cristinel Mateis. Jade – Subsets of Java considered for Debugging. Technical Report DBAI-TR-99/xx, Technische Universität Wien, Institut für Informationssysteme 184/2, Paniglgasse 16, A-1040 Wien, Austria, 1999.

[Missier *et al.*, 1994] Antoine Missier, Spyros Xanthakis, and Louise Travé-Massuyés. Qualitative Algorithmics using Order of Growth Reasoning. In *Proc. ECAI 94*, pages 750–754, 1994.

[Moriconi and Winkler, 1990] Mark Moriconi and Timothy Winkler. Approximate reasoning about the semantic effects of program changes. *IEEE Transactions on Software Engineering*, 16(9):980–992, September 1990.

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

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

[Stumptner and Wotawa, 1998a] Markus Stumptner and Franz Wotawa. Model-based debugging of functional programs. In *Proc. DX'98 Workshop*, Cape Cod, May 1998.

[Stumptner and Wotawa, 1998b] Markus Stumptner and Franz Wotawa. A Survey of Intelligent Debugging. *AI Communications*, 11(1), 1998.

[Stumptner and Wotawa, 1998c] Markus Stumptner and Franz Wotawa. VHDLDIAG+:Value-level Diagnosis of VHDL Programs. In *Proceedings of the Ninth International Workshop on Principles of Diagnosis*, Cape Cod, May 1998.

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

[Tarjan, 1972] R. Tarjan. Depth-first search and linear graph algorithm. *SIAM Journal of Computing*, 1(2), June 1972.

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

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