Debugging of Java Programs using a model-based approach

Cristinel Mateis and **Markus Stumptner** and **Franz Wotawa**
Technische Universität Wien
Institut für Informationssysteme
Paniglgasse 16, A-1040 Wien, Austria,
Email: {mateis,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. This paper presents a model that is usable for the representation of imperative programming languages at an abstract level as part of a project for developing a model-based debugging system for Java. The representation is discussed and compared to other debugging representations.

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. We briefly recapitulate the basic tenets here. For a detailed introduction see [Reiter, 1987; de Kleer and Williams, 1987].

In model-based diagnosis a model of the *correct* behavior of a system (a device) is used to find a subset of system components that have caused a misbehavior. This subset is termed a diagnosis. A diagnosis is defined for a system, consisting of a behavioral description SD, a set of components $COMP$, and a set of observations OBS. Formally (see [Reiter, 1987]), a set $\Delta \subseteq COMP$ is a diagnosis iff

$$SD \cup OBS \cup \{\neg ab(C) | C \in COMP \setminus \Delta\} \cup \{ab(C) | C \in \Delta\}$$

is consistent. The predicate $ab(C)$ is used to indicate that a component C is behaving abnormally. A correctly behaving component C is therefore described by $\neg ab(C)$. A central issue is that of finding a model that is suitable for developing a system description of a domain.

For several years, researchers have worked on extending the scope of MBD to new domains, in particular that of software debugging. This is a natural step: software is an ubiquitous technical artefact with an already proverbial penchant for being incorrect or buggy. The potential gains of providing support for the software debugging process are considerable. Furthermore, unlike other domains (technical or medical), software has the advantage that, provided the source code is available, its behavior can be specified at any desired level of detail based on the semantics of the programming language. In other words, a system description of the program to be debugged can be generated automatically from the source code.

However, these advantages are balanced by a number of challenging restrictions. The most important is that, since the program itself is probably buggy, the system description represents the buggy semantics of the program, unlike the model in hardware diagnosis which represents correct behavior. Secondly, fault probabilities and fault modes play an important role in hardware diagnosis for focusing on promising candidate diagnoses. When debugging software, similar heuristics are significantly harder to come by.

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 extend this work into the domain of, by current standards, conventional programming languages, exemplified by the object-oriented language Java. Like most OO languages, Java is an imperative language: its seman-

```
   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

tics is based on sequential execution with side effects (i.e., variable assignments). We focus on these properties here; the object-oriented language properties are omitted for space reasons and are subject of future papers.

This paper is organized as follows. In Section 2, we discuss the properties of Java and their influence on possible bugs. In Section 3 we describe the modeling of Java programs in a manner suitable for MBD system descriptions, followed by the use of that description for diagnosis in Section 4. A discussion of related research concludes the paper.

2 Java and possible errors

Java is an object-oriented programming language 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. There is no need to explicitly allocate or free memory in Java.

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.

3 Constructing a model for Java programs

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. It is the various variable occurrences that dependencies are computed for.

3.1 Dependencies on Variables

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, 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 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,
$fd \cup fd' = \{(x, M_x \cup M'_x) | (x, M_x) \in fd \wedge (x, M'_x) \in fd'\} \cup \{(x, M_x) | (x, M_x) \in fd \wedge (x, M'_x) \notin fd'\} \cup \{(x, M_x) | (x, M_x) \in fd' \wedge (x, M'_x) \notin fd\}$.

3.2 Method Conversion

A method m of a Java program is converted independently from the other methods 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 function $conv$ can be applied recursively to the methods called in the components found to be faulty until the desired abstract level of diagnoses is reached.

The conversion algorithm uses the functions sum_{FD}, $addvars$, $rework$, and $rewrite$, and a local array, vn, which, during the conversion, stores the current highest used occurrence index for each variable that appears 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., $\{(v, M_v \cup vars) | (v, M_v) \in fd\}$ is given back as result. It is obvious that $addvars$ is polynomial.

The function $rework(fd_1, fd_2, vn)$ performs the following tasks in sequence:
(i) it replaces every variable occurrence v_0 in fd_1 or fd_2 by $v_{vn(v)}$ (i.e., the initial occurrence of v is the occurrence of

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'	$StatementList.conv := [Statement.conv(vn) \mid StatementList'.conv(vn)]$
$\mid \epsilon$	$\mid StatementList.conv := \{\}$
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	$c := new(DiagnosisComponent)$; $c.type := SELECTION$;
Statements_{then}	$c.then := Statements_{then}.conv(VAR)$;
else	$c.else := Statements_{else}.conv(VAR)$;
Statements_{else}	$c.fd := addvars(rework(sum_{FD}(c.then), sum_{FD}(c.else), vn), Expr.vars(vn))$;
	$\forall\ (v_i, M_{v_i}) \in c.fd\ :\ vn(v) := vn(v) + 1$; /* $i = vn(v) + 1$ */
	$SelectionStatement.conv := \{c\}$
IterationStatement ::=	**function** $IterationStatement.conv(vn)$
while (Expr)	$c := new(DiagnosisComponent)$; $c.type := ITERATION$;
Statements	$c.sub := rewrite(Expr, Statements, nestsize(Statements)).conv(VAR)$;
	$c.fd := addvars(rework(sum_{FD}(c.sub), \{\}, vn), Expr.vars(vn))$;
	$\forall\ (v_i, M_{v_i}) \in c.fd\ :\ vn(v) := vn(v) + 1$; /* $i = vn(v) + 1$ */
	$IterationStatement.conv := \{c\}$
	function $rewrite(Expr, Statements, size)$
	if $size = 1$ **then**
	$rewrite := Statements$
	else
	$rewrite := \{$ $Statements$ **if** $Expr$ **then** $rewrite(Expr, Statements, size - 1)$ $\}$
Expr ::=	**function** $Expr.vars(vn)$
id	$Expr.vars := \{id_{vn(id)}\}$
$\mid const$	$\mid Expr.vars := \{ident(const)_0\}$
| MethodCall	$\mid Expr.vars := MethodCall.pars(vn)$
$\mid \text{Expr}_1$ Operator Expr_2	$\mid Expr.vars := Expr_1.vars(vn) \cup Expr_2.vars(vn)$
MethodCall ::=	**function** $MethodCall.pars(vn)$
id ( )	$MethodCall.pars := \{\}$
$\mid id$ (ParList)	$\mid MethodCall.pars := ParList.vars(vn)$
ParList ::=	**function** $ParList.vars(vn)$
$Expr$	$ParList.vars := Expr.vars(vn)$
$\mid Expr$, ParList'	$\mid ParList.vars := Expr.vars(vn) \cup ParList'.vars(vn)$

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

v with the current highest index in the underlying block of statements calling $rework$),
(ii) it replaces every variable occurrence v_i in fd_1 or fd_2 s.t. $i > 0$ by $v_{vn(v)+1}$ (i.e., the final occurrence of v is the next (highest) occurrence of v in the underlying block of statements calling $rework$),
(iii) for each functional dependency $(v_{vn(v)+1}, M)$ from fd_1 (resp., from fd_2) s.t. there is no functional dependency $(v_{vn(v)+1}, M')$ in fd_2 (resp., in fd_1), it adds $v_{vn(v)}$ to M, and
(iv) it returns the union of fd_1 and fd_2 updated at the previous steps.

Clearly, $rework$ is polynomial.

In effect, what the combination of $rework$ and $addvars$ does is to insert the variable occurrences of an inserted statement (***if*** or ***while***) into the dependency set of the surrounding statement block, shifting the indices of the occurrences (which start out with index 0 relative to the beginning of the inserted statement) so they fit into the overall occurrence numbering of the method as a whole. Step (iii) above is needed to keep the ***then*** and ***else*** branches of a selection statement in sync with the preceding statements of the surrounding statement block.

3.3 Loops

A special case that needs to be examined in more detail is the conversion of loops. The repeated execution of the body of a loop can result in the creation of additional dependencies. For example, the sequence of statements "*a = b; b = c;*" results in the final value of a becoming dependent on the value of c, but only if the loop is executed at least twice. On the other hand, assuming the loop contains no other statements influencing the values of a and b, the ultimate value of a will depend on the value of b at the point of entry into the loop only if the loop body is only executed once! So far we are interested in a complete capture of dependencies without analyzing the actual number of iterations executed by the loop (which is undecidable anyway in the general case). Therefore, the dependency $(a, \{b, c\})$ should be part of the result of the conversion. The easiest way to attain this is by transforming the loop into an alternate control structure that provides the same set of dependencies for our analysis. This task is made easier by the fact that the structure does not have to provide the same outcome; we are merely interested in the dependencies. In the example above, we need two "passes" through the assignments of the loop to determine the overall set of dependencies regardless of the number of actual iterations. We now define this for the general case.

In order to compute the diagnosis component of a ***while*** loop with the loop condition $Expr$ and the loop block $Statements$, we rewrite the loop as a nested selection statement, where (i) the number of nested ***if***s is equal to the sum between the number of the assignment statements which are not in the body of any loop in $Statements$ and the number of the outermost loops of $Statements$, (ii) the selection condition of each ***if*** is $Expr$, and (iii) the selection execution statements block of each ***if*** is the concatenation of $Statements$ and the inner nested ***if***.

The function

$rewrite(Expr, Statements, nestsize(Statements))$

returns the selection execution statements block of the outermost ***if*** of the rewritten ***while*** loop described above. $nestsize(Statements)$ counts the sum of the number of assignment statements in the body of the loop (including those which are contained inside ***if*** statements, but not those included in the body of nested loops) and the number of nested top-level loops occurring in $Statements$. It is easy to see that $rewrite$ is polynomial in terms of the number of statements.

Example.

```
      while Expr {
(1)     Assignment1;
(2)     while Expr1 {
          Assignment2;
          if Expr2
            Assignment3;
        }
        if Expr3 {
(3)       Assignment4;
          if Expr4
(4)         Assignment5;
        }
        else {
(5)       Assignment6;
(6)       while Expr5
            Statements
        }
      }
```

```
if Expr {
  Statements_while
  if Expr {
    Statements_while
    if Expr {
      Statements_while
      if Expr {
        Statements_while
        if Expr {
          Statements_while
          if Expr
            Statements_while
        }
      }
    }
  }
}
```

The ***while*** loop on the left is rewritten into the nested selection statement on the right, where $Statements_{while}$ is the body of the loop (i.e., in the example, the concatenation of the assignment statement, the inner loop, and the selection statement). The size of the body, $nestsize(Statements_{while})$ is 6 (due to the assignment statements (1), (3), (4), and (5), and the inner loops (2) and (6)).

Proposition: Given that only static analysis is performed (i.e., no analysis of which branches inside the loop are executed in the different iterations of the loop), the set of dependencies determined from the nested **if** structure is the same as the set of dependencies determined from the original loop construct.

Proof (sketch): $\Rightarrow$ Assume there is a execution of the loop that results in a dependency that cannot be reconstructed using the ***if*** structure. To prove: At any step in the sequence, the correct branch can be chosen. If there are more steps in the loop sequence than nested iteration levels, then one branch must have been occurred twice and would not have to be counted again in the iteration. Proof of this property by induction.

$\Leftarrow$ If there is a particular path through the iteration structure, that path can be reconstructed directly by choosing the same branches in subsequent iterations of the loop. Proof by induction over the iterations. □

Note that the statement block $Statements_{while}$ is repeated in the expansion produced by $rewrite$, but the dependencies for every ***while*** loop contained in the block need only be computed once (by applying $rewrite$ to the body of that loop) and can then be inserted into every occurrence of $Statements_{while}$. Since the size of the output of rewrite is n^2 for a ***while*** loop with $nestsize$ n, the overall effort required is $\Sigma_{i=1...n} n_i^2$ if $Statements_{while}$ contains n loops of $nestsize$ n_i (regardless of the way in which they are nested). For example, for the shellsort sorting method from our Java

example library (containing ten assignments in a structure of while loops nested three deep), the successive calls to $rewrite$ produce nested ***if*** structures with $3^2 + 3^2 + 5^2 = 43$ individual statements overall.

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.

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 astute reader will note that most of the material presented in this section is not actually specific to Java.

4 Finding Faults

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 have 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.

The implementation follows the principles given in [Friedrich *et al.*, 1996] and is not discussed in more detail here. On the model 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 (at the moment this is cut off at method boundaries and nested loops). 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 Related research

The automatization of finding bugs in programs is a topic which has attracted a number of researchers. Some of the proposed techniques use dependencies between variables and statements or knowledge about the program structure [Kuper, 1989; Korel, 1988], combined with specialized search procedures. The analysis of dependencies in programs has a long history, such as in program slicing [Weiser, 1982; 1984], where only statements causing the value of a given individual variable are presented to the user. Since a bug can only occur within a program slice leading to an incorrect variable value, the overall search space for debugging is reduced (although the authors do not present how the method can be directly applied to situations with multiple incorrect variables, which in our case is handled directly by the standard diagnosis mechanism). Another approach based on analyzing dependencies, but requiring a separate specification, is that of [Jackson, 1995]. Other debugging approaches use the semantics of the programming language. In [Shapiro, 1983], algorithms for debugging logic programs by successive addition of user inputs were proposed.

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 like [Shapiro,

1983], [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. In [Missier *et al.*, 1994] a way of debugging recursive functions using order of growth reasoning has been proposed. [Burnell and Horvitz, 1995] introduce a debugging system for a very large legacy program using the program structure and a probability network for encoding the probabilities of bugs depending on path and dependency analysis as well as the observed misbehavior, with probabilities optimized for the legacy system at hand.

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.

The analysis of dependencies in programs also has a long history in other contexts such as optimizing compiler development, e.g., [Moriconi and Winkler, 1990; Ferrante *et al.*, 1987; Cytron *et al.*, 1991]. In the latter two the PDG is computed based on the concepts of post-dominance and dominance frontiers. A node V is post-dominated by a node W in G if every directed path from V to the terminal node contains W. Moreover, W is control dependent on V iff (i) there exists a directed path P from V to W with any U in P post-dominated by W, and (ii) V is not post-dominated by W. In [Ferrante *et al.*, 1987], the control dependencies are determined by constructing the post-dominator tree for an augmented control flow graph, and then examining certain control flow graph edges and annotating nodes on corresponding tree paths.

The approach presented in [Cytron *et al.*, 1991] is closely related to our transformation in that the statements of a program are organized into basic blocks and each assignment to a variable is given a unique name by assigning a unique index to the name of that variable. The main difference to our approach consists in that computing the PDG is a two-step process. In the first step, additional assignments are inserted at some of the join nodes in the control flow graph, with new variables generated for these assignments afterwards. Potential anomalies of simple dependency representations are discussed in [Moriconi and Winkler, 1990]. Further work will include the extension of our model to improve discrimination by avoiding such anomalies.

6 Conclusion

This paper has presented a model of Java programs that explicitly records dependencies between variables in the program. It 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 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. The reader will have noticed that at the moment there is actually very little Java-specific content in the model; this is due to the abstract nature of the representation and will change as the model is developed in more detail.

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.

Acknowledgement

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.

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.

[Cytron *et al.*, 1991] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.

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

[Ferrante *et al.*, 1987] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, 1987.

[Friedrich *et al.*, 1996] Gerhard Friedrich, Markus Stumptner, and Franz Wotawa. Model-based diagnosis of hardware designs. In *Proc. 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.

[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.

[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.