Modeling Java Programs for Diagnosis[1]

Cristinel Mateis and **Markus Stumptner** and **Franz Wotawa**[2]

Abstract. A key advantage of model-based diagnosis is the ability to use a generic model for the production of system descriptions that can be used to derive diagnoses for differently structured individual systems from a domain. This advantage is nowhere more apparent than in the software error diagnosis (or debugging) area, where given a model, system descriptions can be automatically derived from source code. However, effective models for diagnosing programs have so far been limited to special-purpose languages. We describe a value-based model for Java programs that enables us to explicitly deal with imperative program execution (including loop execution), and compare the results to those obtained by using program slicing, a traditional technique from the software debugging community, and a simple dependency-based model for Java.

```
   public class Examples {
       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

1 Introduction

A key advantage of model-based diagnosis is the ability to use a generic model for the production of system descriptions that can be used to derive diagnoses for differently structured individual systems from a domain. This advantage is nowhere more apparent than in the software error diagnosis (or debugging) area, where given a model, system descriptions can be automatically derived from source code. However, the nature of software debugging as a design activity means that unlike hardware diagnosis, neither complete nor structurally equivalent specifications of the intended code behavior are available, making the task significantly harder than with hardware diagnosis. Nonetheless, advantages can be gained over other debugging methods, which labor under at least same restrictions. The model introduced in this paper is intended to be used for detecting functional faults in Java programs. For simplicity reasons we focus on a significant subset of Java. The subset comprises classes, methods, assignments, conditionals, and while-loops. We illustrate our approach using the small Java program from figure 1. The *demo* method comprises a conditional, assignments and a while-loop statement. Assuming that after execution, the value of variable *i* should be the same as the value of *stop*, we easily can prove that *demo* must have a bug. Say *demo(1,2)* is called. Then we get *i=3* after method execution which obviously contradicts our assumption that *i* should be equal to *stop* after execution. If we know nothing about the expected values of *start* and *stop*, then we can not distinguish whether the bug is located within the conditional, the assignment from line 6, or the loop statement. If the expected values are given, e.g., *start=1* and *stop=2*, the bug can be only located in line 6, or within the while-loop, because the assignment determines the start value of *i* and the while-loop the number of iterations influencing the value of *i*

We have previously described the use of a simple model of a Java program that is based on recording dependencies between variables [9]. This model delivers the result above, i.e., that the assignment (line 6) may be responsible for the bug, in addition to the while loop (line 7 to 9). The model introduced in this paper improves the result by providing more informations about the bug and how to repair it, as a result aiding the programmer in finding and fixing bugs more quickly. It has the potential to lead to an automatic debugger in the future. Similar to [9], we convert the Java program into a logical description that afterwards is used together with a model-based diagnosis engine [11, 3] for computing diagnoses, i.e., finding possible bug locations given an observed incorrect execution outcome. In contrast to [9], the model is not a dependency model but a value-based one, and includes models of faults which directly map to repair suggestions, e.g., replacing condition $i <= stop$ in line 7 with $i < stop$ (obviously leading to the correct version of *demo*).

2 The Model

The key of using model-based diagnosis to debugging lies in developing a component-connection model for programs. Such a model must be automatically derived from the source code. Further, a value-based model has to represent the semantics of the programming language, i.e., given the same inputs, both program execution and the corresponding model must lead to the same outcome. The model introduced in this section represents statements and expressions as components and variables as connections. This choice allows to detect functional faults but is not recommended for locating structural faults, in other words, completely missing statements in a program, or structural faults that occur whenever a wrong variable was used, are not explicitly covered in the model. Figure 2 gives the Java subset considered. Although it does not comprise the complete language,

[1] This work was partially supported by the Austrian Science Fund project P12344-INF and project N Z29-INF.
[2] Technische Universität Wien, Institut für Informationssysteme and Ludwig Wittgenstein Laboratory for Information Systems, Favoritenstraße 9–11, A-1040 Wien, Email: {mateis,mst,wotawa}@dbai.tuwien.ac.at. Authors are listed in alphabetical order.

Program ::= Classes
Classes ::= Class Classes | ϵ
Class ::= **class** Id [**extends** Id$_2$] { ClassStmnts }
ClassStmnts ::= ClassStmnt ClassStmnts | ϵ
ClassStmnt ::= VariableDecl; | MethodDecl
VariableDecl ::= Type Id
Type ::= **int** | **bool** | **float** | Id$_{Class}$
MethodDecl ::= Type Id (FormalParList) { JavaStmnts }
FormalParList ::= VariableDecl FormalParListRest | ϵ
FormalParListRest ::= **,** VariableDecl FormalParListRest | ϵ
JavaStmnts ::= JavaStmnt JavaStmnts | ϵ
JavaStmnt ::= Assignment | Selection | While | MethodCall | ReturnStmnt
Assignment ::= Id **=** Expr;
Expr ::= Id | Const | MethodCall | (Expr) | Expr$_1$ Op Expr$_2$ | **new** MethodCall
Selection ::= **if** (Expr) { JavaStmnts$_{then}$ } [**else** { JavaStmnts$_{else}$ }]
While ::= **while** (Expr) { JavaStmnts }
MethodCall ::= Id (ActualParList) | Id$_{Class}$.Id (ActualParList)
ActualParList ::= Expr ActualParListRest | ϵ
ActualParListRest ::= **,** Expr ActualParListRest | ϵ
ReturnStmnt ::= **return** Expr;

Figure 2. Java subset used for the model

it allows the use of object-oriented features such as dynamic binding and inheritance. Recursive functions, interface declarations, and some other language constructs are not considered.

2.1 Computing models

The compilation of programs into models is static from the debugger's point of view, i.e., it coincides with the actual compilation or initial execution of the program. The compiler successively computes models for each method of a class similar to the conversion of programs into byte-code. The obtained model can be divided into two parts. The structural part comprises the connections, components, and the connectivity relations. The behavior part defines the behavior of components using a logic-based language, e.g., first-order logic. While the structural part depends on the methods to be converted, the behavior part is determined by the language semantics and is therefore independent of a given program. Before we give the logic description of the behavior, we first informally introduce an algorithm for converting Java programs into components and connections.

In converting Java programs to a model, the key features of the Java program's classes are their variables and methods. Variables are divided into class and instance variables. While two instances of a class have separate sets of instance variables, they share the same class variables. Methods may introduce new local variables and consist of statements. All variables, regardless whether they are class, instance, or local, are mapped to connections. For each variable, a connection is created in the initialization phase of the compilation process. Whenever a variable occurs in an expression, this connection is used for connecting to the corresponding component. Each time a variable is used as target in an assignment statement, a new connection is created and used until a new connection arises. Consider for example the following program fragment:

```
1.    i = 1;
2.    o = i + 1;
3.    i = o + 1;
4.    x = i - 2;
```

Compilation starts with line 1 and converts the first assignment until line 4 is reached. Because i in line 1 is used as target, a new connection is created. This connection is used in line 2. In line 3, again a new connection for i is created that is used in line 4.

The conversion of variables to connections as described above is well defined whenever the sequence of statements to be executed is known in advance. This restriction does not require knowing which branch of a conditional statement will be executed, but it excludes Java programs using parallel processes.

The statements and expressions used in a Java method are mapped to components as follows:

- *Assignments* are mapped to assignment components with two ports, one input port and one output port. The input port is related to the evaluation of the expression on the right side of the assignment statement and is connected to the output port $result$ of the corresponding expression component (the expression component is introduced below). The output port is related to the target variable of the assignment.
- *Conditionals* are mapped to conditional components with a varying number of ports. An input port, $cond$, is connected from the output port $result$ of the component of the selection conditional expression. For every variable x used as target in an assignment occurring in either the $then$ or the $else$ branch, three ports are generated: $then_x$, $else_x$, and out_x. The input port $then_x$ ($else_x$) is connected from the output port dealing with x of the last statement from the $then$ ($else$) branch which modifies x. The output port out_x is connected to a new connection mapped to x. This connection is used in subsequent statements.
- *While loops* are mapped to loop components with several input and output ports given by the variables used within the loop or the conditional. For all variables X that are used in an expression a port in_X is assumed. Similarly, each variable X such that the value of X is changed by a statement or within a method call, has an associated port out_X. In addition to ports, the conditional and the statements block of the loop are converted separately, leading to two different diagnosis systems: (SD_C,$COMP_C$) for the conditional and (SD_B,$COMP_B$) for the block. This conversion procedure is the same as for statements described herein. We assume the following connections are used in SD_C and SD_B. For each variable X used in the conditional, we have a connection c_X in SD_C. In addition, SD_C has a connection $cond$ for delivering the value of the condition. For each variable X we assume a connection bi_X if X is used as input in the statements block, and a connection bo_X if used as output. Note that the set comprising all bo_X is equal to the set of all out_X, if no variable is changed within the condition. This assumption is valid for our model.
- *Return statements* are mapped to components with one input port and one output port. The input port is related to the evaluation of the returned expression and a connection runs to it from the output port $result$ of the corresponding expression component. The output port is related to an auxiliary variable, $return$, which takes the value of the returned expression evaluation.
- *Method calls* of type *void* are calls of methods which do not return any value. Instead they modify external variables through side effects (where "external variables" refers to instance variables of the object for whom the method is called, and variables derived from the actual parameters of reference type). Such method calls are statements on their own and are mapped to method call components MC with several input and output ports. The input ports are related to (i) the global variables used in the body of the called

method, and (ii) the actual parameters of the method call which induce variations of variables visible outside the method call. The output ports are related to all global variables used in the body of the called method and all variables derived from the actual parameters of reference type (e.g., all attributes $x.field$ of a variable x of reference type are variables derived from x) which are modified inside the body of the called method. Other output ports are related to variables derived from objects created inside the called method which are accessible in the environment where the method call occurs; note that these variables are newly introduced in the model, i.e., they did not exist before the method call.
If m is the called method, the component MC corresponding to the method call of m can be obtained from the component-based model M of m by substituting in M all ports related to the global variables and formal parameters with the ports related to the corresponding actual global variables and actual parameters. After substitution, the ports of M coincide exactly with the ports of MC.
- *Expressions* are constructs which are used in statements but which cannot be statements on their own (except function calls, but usually, their return values are always used in statements). Expressions are mapped to expression components which may have several input and output ports (at least one of each). The expressions are always evaluated to a final result which is then used in the statements where the expressions appear (usually an assignment or a selection or loop condition). An output port, $result$, of the expression component is related to the final result. In order to evaluate the expressions, the relations between the operands of the expressions (which in their turn may be other expressions, constants, variables, function calls, etc.) are evaluated, hence the variables appearing in the operands of the expressions can be viewed as inputs of the expressions. The input ports of the expression component are related to these variables. Other input ports are related to global variables which do not appear directly in the operands but are used by function call operands inside the body of the called function. Other output ports are related to variables which are derived from the actual parameters of function call operands and are modified through side effects by the called function.
There are three fundamental types of expressions.
Constants and **variables** are mapped to components with two ports. The input port is related to the corresponding constant (resp., variable). The output port is related to an auxiliary variable, $result$, whose value coincides with the value related to the input port.
Operators are mapped to components with two input ports and one output port. The input ports are connected from the output ports $result$ of the components corresponding to the two expression operands. The output port, $result$, is related to the evaluation of the relation between the two operands induced by the operator.
Function calls and **constructor invocations** are mapped to components FC which are similar to the method call components MC, but in addition, they have a further output port, $result$

Consider the example program *Examples.demo* from the Introduction. The conversion procedure described above returns the component-connection model depicted in figure 3.

The time for performing diagnosis is of course (among other parameters) dependent on the number of components. Therefore, an upper-bound of the number of created components for a specific program is needed. Assume that M denotes the number of method calls occurring in the method to be converted, let S be the maximum number of statements, and E be the maximum number of expressions in one method. Then the number of created components must be less than or equal to $M \cdot (S + E)$. Taken into account the average performance of modern diagnosis algorithms [5, 13], where systems with up to 10.000 components can be diagnosed within seconds, we estimate that the approach can be used for debugging of small and mid-size programs (several k-Byte of source code) in reasonable time.

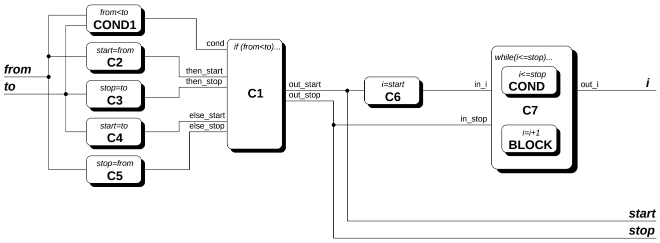

Figure 3. The component connection model for *Examples.demo*

2.2 Behavior

In the previous section we have shown how the component-based model of a given Java program is generated. In particular, each Java construct is transformed to a corresponding component. In order to detect possible faulty statements of the Java program, a description of the program behavior is needed. In this section we show how the behavior description of each component from the component-based model is generated. The predicate $AB(C)$ is used to indicate that a component C is behaving abnormally. A correctly behaving component C is therefore described by $\neg AB(C)$.

- *Assignments* (Assignment ::= Id = Expr;)
$\neg AB(C) \Rightarrow out(C) = in(C)$
where the input port $in(C)$ is connected to the output port $result(Expr)$ of the component corresponding to $Expr$, and the output port $out(C)$ is related to the variable Id.
- *Conditionals* (Selection ::= **if** (Expr) { Stmnts$_{then}$ } [**else** { Stmnts$_{else}$ }])
For each variable X that is modified in at least one of the selection branches *then* and *else*, the following behavior description constructs are generated.
$\neg AB(C) \wedge cond(C) = true \Rightarrow out_X(C) = then_X(C)$
$\neg AB(C) \wedge cond(C) = false \Rightarrow out_X(C) = else_X(C)$
where the input port $cond(C)$ is connected from the output port $result(Expr)$ of the component corresponding to the selection condition $Expr$, and the input port $then_X(C)$ (resp., $else_X(C)$ is connected from the output port related to X of the component corresponding to the last statement of $Stmnts_{then}$ (resp., $Stmnts_{else}$) which modifies X.
- *While loops* (While ::= **while** (Expr) { JavaStmnts })
The behavior corresponds to the intuitive expectation one would have of a loop construct. Whenever the diagnosis system of the conditional allows to derive *true* for the condition, the diagnosis system of the block statement is used to derive new values for the variables. The values for every loop iteration are stored in an auxiliary variable v_X^I where X denotes the variable (either used as input or output) and I the current number of iterations. At the beginning of the computation, the initial values are set using the values from the ports in_X. At the end of the computation, the values

are set equal to out_X. Formally, the behavior is given as follows:

$$
\begin{array}{l}
\neg AB(C) \Rightarrow \Bigg(\\
\quad \forall_X \, in_X(C) = v_X^0(C) \wedge \\
\quad \forall_X \, out_X(C) = v_X^{MAX}(C) \wedge \\
\quad \forall_{X \in (VC \cup VBI) \setminus VBO} \forall_{I < MAX} \, v_X^I(C) = v_X^{I+1}(C) \wedge \\
\left(\begin{array}{c}
\Big((SD_C(C) \cup \{\neg AB(D) | D \in COMP_C(C)\} \\
\cup \{v_X^I(C) = c_X(C) | X \in VC\} \models cond(C) = true) \rightarrow \\
\forall_{Y \in VBO} (SD_B(C) \cup \{\neg AB(D) | D \in COMP_B(C)\} \cup \\
\{v_X^I(C) = bi_X(C) | X \in VBI\} \models v_Y^{I+1}(C) = bo_Y(C)) \Big)
\end{array} \right) \wedge \\
\left(\begin{array}{c}
((SD_C(C) \cup \{\neg AB(D) | D \in COMP_C(C)\} \cup \\
\{v_X^I(C) = c_X(C) | X \in VC\} \models cond(C) = false) \wedge \\
\nexists J : (J < I \wedge \\
(SD_C(C) \cup \{\neg AB(D) | D \in COMP_C(C)\} \cup \\
\{v_X^J(C) = c_X(C) | X \in VC\} \models \\
cond(C) = false))) \rightarrow MAX = I
\end{array} \right) \Bigg)
\end{array}
$$

Here, VC denotes the variables used in the conditional expression, VBI (VBO) the variables used as input (output) of the block. Of the five conjunctive terms in the definition above, the first two describe the entry and exit mappings, respectively. The third describes the step from one iteration to the next; the fourth, a positive test followed by continuation of the loop; and the last a failure test of the loop condition, i.e., termination of the loop.

In addition to the correct behavior, we specify a fault behavior $Loop(C, MAX)$ setting the number of iterations to MAX:

$$
\begin{array}{l}
Loop(C, MAX) \Rightarrow \Bigg(\\
\quad \forall_X \, in_X(C) = v_X^0(C) \wedge \\
\quad \forall_X \, out_X(C) = v_X^{MAX}(C) \wedge \\
\quad \forall_{X \in (VC \cup VBI) \setminus VBO} \forall_{I < MAX} \, v_X^I(C) = v_X^{I+1}(C) \wedge \\
\Big(\forall_{I < MAX} \forall_{Y \in VBO} (SD_B(C) \cup \\
\quad \{\neg AB(D) | D \in COMP_B(C)\} \cup \\
\quad \{v_X^I(C) = bi_X(C) | X \in VBI\} \models v_Y^{I+1}(C) = bo_Y(C)) \Big) \Bigg)
\end{array}
$$

Assuming an example run for a given test case that should result in five iterations of the loop C, $Loop(C, 4)$ and $Loop(C, 6)$ would both be examples of fault modes for the loop that describe an incorrect number of iterations.

- *Return statements* (ReturnStmnt ::= **return** Expr;)
  $\neg AB(C) \Rightarrow return(C) = in(C)$
  where the input port $in(C)$ is connected to the output port $result(Expr)$ of the component generated for $Expr$, and the output port $return(C)$ is related to the returned auxiliary variable.
- *Method calls* support two variants.
  - MethodCall ::= Id (ActualParList)
    We assume that the behavior B_{Id} of the called method Id has already been computed, hence recursion is not supported. Then, for each rule $(A \Rightarrow B) \in B_{Id}$, the behavior description construct generated is: $\neg AB(C) \wedge A' \Rightarrow B'$, where A' (resp., B') is obtained from A (resp., B) by substituting all formal parameters and global variables with the corresponding actual parameters and global variables.
  - MethodCall ::= Id_{class}.Id (ActualParList)
    This case is similar to the previous one, the only difference consists of the origin of the actual global variables which substitute the formal ones. In the previous case, the actual global variables are the instance variables of the object for whom the current component-based model is generated. In the current case, the actual global variables are the instance variables of the object Id_{class}. Note that since the model is value-based, the actual class of the object in the variable Id is available at debugging time and therefore the correct called method will be used to for the model. This feature handles dynamic binding and also inheritance (since for each class the inherited methods can be statically precomputed).
- *Expressions* are subdivided according to the three categories mentioned in the section on the structural part of the model.
  - *Constants* and *variables*
    $\neg AB(C) \Rightarrow result(C) = in(C)$
    where both the input port $in(C)$ and the output port $result(C)$ are related to the constant (resp., variable).
  - *Operators* (Expr ::= Expr_1 Op Expr_2)
    $\neg AB(C) \Rightarrow result(C) = in_1(C) \;\; Op \;\; in_2(C)$
    where the input ports $in_1(C)$ and $in_2(C)$ are connected from the output ports $result(Expr_1)$ and $result(Expr_2)$ of the components corresponding to $Expr_1$ and $Expr_2$, respectively, and the output port $result(C)$ is related to the auxiliary variable whose value is the result of the evaluation of the operator *Op* with the operands $result(Expr_1)$ and $result(Expr_2)$.
  - *Function calls* and *constructor invocations*
    If m is the called function (resp., invoked constructor) and M is the behavior description of m, the behavior description construct of the function call (resp., constructor invocation) of m is obtained from M by using the same conversion mechanism previously presented for the behavior description of (void) method calls and by substituting the auxiliary variable $return$ with the auxiliary variable $result$.

3 Debugging

In this section we will use the model for debugging the *Examples.demo* program from figure 3 and show how use of the model results in improvements when compared with the outcome of a dependency-based model [9]. We assume that before executing *demo*, we have *from = 1* and *to = 2*. After program execution we expect to obtain *start = 1*, and *stop = i = 2*, but it can be easily seen that program execution leads to *i = 3*. We now examine the effects of using the different models in the debugging process.

Using only dependencies between variables as in [9], we see that *i* depends on *start* and *stop* (both of which have correct values). Therefore, the conditional statement and its sub-statements (line 1 to 5) cannot be the source of the faulty behavior. Only the assignment (line 6) and the while-loop (line 7 to 9) remain as diagnosis candidates. Because of the abstract representation used by the dependency model, almost no hints can be given to the user to distinguish between the two diagnoses, except that he or she should look first at the outcome of the assignment. Based on this information, an additional ranking for diagnosis discrimination can be delivered by a measurement selection algorithm.

Using the value-based model presented in this paper provides much richer information for discriminating diagnoses. Using the same specification as above, our value-based model allows to derive the following (minimal) diagnoses: $\{AB(C6)\}$, $\{AB(C7)\}$,

and $\{Loop(C7, 1)\}$. As it happens, the first two diagnoses give no hint to the fault. They only state that the assignment and the while-loop are candidates. The last diagnosis, on the other hand, provides more feedback to the user. Diagnosis $\{Loop(C7, 1)\}$ says that, if the body of the loop is only executed once (and not twice), then we receive a correct behavior. When using the principle that more specific information should be preferred over general information, we get $\{Loop(C7, 1)\}$ as the single most probable diagnosis candidate. The application of this principle is reasonable because it supports solutions having more evidence.

Based on the preferred single diagnosis, an intelligent debugger can further compute diagnoses that map back to expressions or statements within the loop statement (associated with $C7$). We have to distinguish between two possible bug locations, since either the conditional or the block of the loop could contain the bug. These are two independent diagnosis problems represented by two diagnosis systems SD_C, SD_B for the condition and the block, respectively. We derive the observations for the diagnosis problems directly from the original system description, the diagnosis $\{Loop(C7, 1)\}$, and the behavior definition of the component $C7$. For space reasons, we show only the diagnosis for the condition here. Using the model, we derive $v_i^0 = 1$, $v_{stop}^0 = 2$, $v_i^1 = 2$, $v_{stop}^1 = 2$. The system $(SD_C, COMP_C)$ comprises one component $CC1$ (implementing the less-or-equal function), and three connections (c_i, c_{stop}, $cond$). From the values of variables for each iteration we get two observation sets: $OBS_1 = \{c_i = 1, c_{stop} = 2, cond = true\}$ and $OBS_2 = \{c_i = 2, c_{stop} = 2, cond = false\}$. We finally get that $CC1$ is a candidate. If the system were extended by the techniques described in [14], we would additionally receive a replacement suggestion, e.g., $\{REPLACE(CC1, '<')\}$ as single diagnosis, which obviously when applied leads to a correct implementation. However, integrating these two approaches is the subject of future research.

Finally, we compare the outcome of our model with the results obtained by using a standard analysis and debugging technique from the program language community, program slicing [15]. A program slice is defined for a given program, a statement S, and a set of variables V and contains those program fragments, i.e., statements, influencing the value of a variable from V and occurring not after S. For our example, the slice for the last statement (line 9) and the variable i comprises all statements. When removing those statements included in the slice for line 9 and variables *start*, *stop*, only the statements from line 6 to 9 remain. A slice can be viewed as location for a potential bug. Therefore, using program slicing leads to the same results obtained by using the dependency model. Using the same arguments as before, we conclude that our value-based model provides better results than program slicing.

4 Related research and Conclusion

Automated debugging has been an active research area for several decades. Some of the proposed techniques use dependencies between variables and statements or knowledge about the program structure [8, 7], combined with specialized debugging algorithms. Approaches using dependencies include program slicing [15], where only statements causing a wrong variable value are considered, or approaches that compare a dependency specification which is explicitly written by the programmer [6]. In [1], the structure of a program together with a probability theoretic evaluation of possible error types has been proposed for debugging. In contrast to the latter two approaches, we only use information derived from the program itself. When compared to the pure dependency-based approach of [9], both the structure and the behavior of a program are considered for debugging. This leads to an increased debugging time but provides better results. As experience with the dependency-based approach for other languages [4] has shown, dependencies still provide worthwhile information when dealing with large programs, so that combining the two models (with the dependency-based model used for focusing in on a smaller part of the program before the value-based model is used) is a promising avenue of research. This combination is currently being implemented in our debugging prototype.

Other traditional approaches for debugging include [12] where the semantics of the program is used to optimize guidance of the user through the code. However, [2] showed that using model-based diagnosis for debugging reduces the number of questions necessary for locating a bug when compared with [12]. In [14, 4, 16] the application of model-based diagnosis has been extended to find bugs in the hardware description language VHDL, and a small functional language introduced for this purpose. Another approach for debugging recursive functions using qualitative reasoning was described in [10]. We think that the underlying ideas can be combined with our value-based model (when extended to handle recursive functions) and should provide better discrimination of bug candidates.

To conclude, in this paper we have introduced a model that can be used for locating and (partly) repairing bugs in Java programs. The model extends previously published models for software debugging. First, it handles object-oriented features of the language, such as inheritance. Second, the model is more general than previous models and value-based. Although we exclude recursive methods at the moment, we handle loop constructs. In addition, global variables can be used and are correctly captured by the model. Apart from the modeling aspects, this paper provides a comparison with other approaches including methods used by the software community for automated debuggers. The result of this comparison is that the model-based approach delivers more accurate results and improves flexibility.

The work presented here provides a basis for future research in the domain of applying model-based reasoning to debugging of object-oriented languages. Open problems include the development and use of different models capturing different faults in software, and to build an intuitive user-interface for a model-based debugger.

REFERENCES

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

[2] Luca Console, Gerhard Friedrich, and Daniele Theseider Dupré, 'Model-based diagnosis meets error diagnosis in logic programs', in *Proc. IJCAI*, pp. 1494–1499, Chambery, (August 1993).

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

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

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

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

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

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

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

[10] Antoine Missier, Spyros Xanthakis, and Louise Travé-Massuyés, 'Qualitative Algorithmics using Order of Growth Reasoning', in *Proc. ECAI 94*, pp. 750–754, (1994).

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

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

[13] Markus Stumptner and Franz Wotawa, 'Diagnosing Tree-Structured Systems', in *Proc.* 15^{th} *IJCAI*, Nagoya, Japan, (1997).

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

[15] Mark Weiser, 'Program slicing', *IEEE TSE*, **10**(4), 352–357, (July 1984).

[16] Franz Wotawa, 'Debugging synthesizeable VHDL Programs', in *Proc. DX'99 Workshop*, (1999).