Analysing models for software debugging*

Markus Stumptner and **Dominik Wieland** and **Franz Wotawa**†
Technische Universität Wien
Institut für Informationssysteme
Database and Artificial Intelligence Group
Favoritenstraße 9–11, A-1040 Wien, Austria
Email: {mst,wieland,wotawa}@dbai.tuwien.ac.at

Abstract

This paper extends our previous work on the representation and analysis of Java programs for diagnosis in a new direction by providing a description and analysis of the issues arising from handling object references in dependency-based models of Java programs. We empirically compare dependency-based models with a value-based model using a set of example programs in terms of required user interaction (questions put to the user) and examine and incorporate specific interesting error categories. Apart from being based on experience with an actual implementation of the various models, the model extensions and analysis deal with aliasing, an issue that the programming language community has been examining for a long time, and that is also crucial to object-orientedness.

1 Introduction

Automatic and algorithmic debugging has been an active research area for several decades and many different techniques have been introduced and discussed in the past (see [Ducassé, 1993] for an overview). In recent years the use of model-based diagnosis for debugging has been examined [Console, Friedrich, & Dupré, 1993; Bond & Pagurek, 1994; Bond, 1996]. Research in model-based debugging includes work dealing with debugging of hardware designs [Friedrich, Stumptner, & Wotawa, 1999], an analysis of the modeling space [Stumptner & Wotawa, 1999], and first steps in developing models for the object-oriented programming language Java [Mateis, Stumptner, & Wotawa, 1999; Mateis *et al.*, 2000a; 2000b]. In this paper we build on the results of the latter work. We discuss new aspects of dependency-based models for modeling Java code as well as a value-based model. We also discuss some hard problems for these models.

To demonstrate the modeling process and the properties of the resulting models we implement a short example program *Point.java*, which provides the basic data structure and functionality of a 2-dimensional point (see Figure 1).

```
    class Point {
      int x;
      int y;
      Point(int x, int y) {
1.      this.x = x;
2.      this.y = y;
      }
      Point plus(Point p) {
1.      return new Point(x+p.x, y+p.y);
      }
      public static void test() {
        Point p1, p2;
1.      p1 = new Point(0,0);
2.      p2 = new Point(2,3);
3.      p1.x = 1;
4.      p1.y = 2;
5.      p2 = p1.plus(p2);
      }
    }
```

Figure 1: Example program *Point.java*

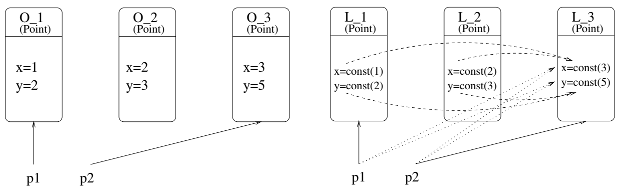
Java source code consists of instance field definitions, static initializers, and method declarations (including constructors) (one class with 2 instance fields and 3 methods in the example). The method *test()* itself contains 5 statements and the declaration of 2 local variables. Note that variable declarations are not counted as statements, because no runtime diagnosis components can arise from them. We classify the entities in the program in terms of

Constants: The set $\mathcal{C}$ of literals that are defined in the Java source code and do not change their values during program execution. Although in Java these structures are not objects, they can logically be seen as objects with a fixed content of primitive type (in our example we find the constants 0, 1, 2, and 3). Note that each occurrence of such a constant has to be considered separately (we omit indexes for simplicity in the paper).

Locations: The set $\mathcal{L}$ of memory locations which represent the stored state of Java objects. A location is a placeholder for objects that would be produced dynamically during runtime by constructor calls [Jackson, 1995]. Their internal state is user defined and alterable (in our

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

†Authors are listed in alphabetical order



(a) Value-based (b) Dependency-based

Figure 2: Representing objects

example 3 objects of type *Point* are created). It is important to note that locations which are introduced during static analysis of the functional dependencies, does not represent individual objects, but instead are generic representations covering all objects that might be assigned to a particular variable occurrence. (In the value-based model explained later in the paper, we will use actual, individual objects instead of locations.)

Variables: The set $\mathcal{V}$ of Java variables (including class and instance fields) can be of primitive or of reference type. Primitive variables (e.g. *x* and *y* above) hold values, reference variables (e.g. *p1* and *p2*) reference objects, i.e. they point at a particular memory location.

The state of a Java system at any time during execution can be specified by defining the three sets of system components given in the list above. Figure 2 shows the state of our example program after the execution of all five statements of method *test()*. We can see the current run time state of three objects (memory locations) given by their instance fields which are all assigned to constants. The two variables of reference type, *p1* and *p2* reference the first and third location, respectively. The object at location 2 is not referenced by any of the system's variables and will be eliminated by the garbage collector. In the dependency-based view, the values of the instance variables of *p2* are recorded as being derived from the values of those of location 2 (and location 1).

Given some representation of the behavior of a Java program in terms of a set of components (usually statements or expressions), and using observations of the state of the program as described above, diagnoses, i.e., potential error locations in the program can be computed in the usual consistency-based manner [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. $AB(C)$ indicates that a component C is behaving abnormally, and a correctly behaving component C is described by $\neg ab(C)$. In general we want to compute diagnoses which are subset-minimal.

The following sections show how a Java system as defined above can be modeled for debugging using a static functional dependency model that for the first time incorporates object references (in Section 2) and a dynamic value-based model (Section 3). Section 4 gives empirical and analysis results based on the implementation of both models.

2 The dependency-based model

In [Mateis, Stumptner, & Wotawa, 1999], we presented a functional dependency model for Java programs. We defined a functional dependency (x_i, M_{x_i}), to express the fact that the value of the i-th occurrence of variable x in the current method depends on the values of the variables (actually, variable occurrences) in $M_{x_i})$ if their values were used to compute the value stored by x_i (data dependency) or if they were used in selection or loop conditions that could have influenced the computation of that value (control dependency). In this paper, we describe two extensions of that model. The central issue in both is the explicit consideration of the semantics of object references, allowing a treatment of aliasing situations, which are generally excluded in classical dependency research (see, e.g, [Moriconi & Winkler, 1990]), but are quite relevant in Java due to the language's general reference semantics for object-valued variables.

2.1 The detailed functional dependency model (DFDM)

When creating the detailed functional dependency model (DFDM) of a particular method we consecutively transform each statement of the method in question. Functional dependency components (hereinafter FDs) arise from assignment statements or program structures, that themselves include assignments, such as method calls (side-effect FDs), selection statements (selection FDs) or loops (loop FDs). Note that there might exist more than one FD for one statement, e.g., side-effect, selection, or loop FDs.

An FD in the detailed model is of the form $FD(var, DEP)$, where *var* is the variable occurrence whose value is determined in the particular statement and therefore depends on the entities in DEP. *var* can be of the following form:

- *x* — *x* denotes a local variable of the currently modeled block
- *0::x* — *x* stands for an instance field of the receiver of the method (self)
- *n::x* — *n* refers to the location of an object different from 0 (self). *x* denotes an instance field of this particular object.

Note that in all 3 cases *x* can be of primitive or of reference type.

DEP is a tuple $DEP = < C, V, M, L >$, where $C \subseteq \mathcal{C}$ is a set of constant values (e.g., 10, true) that may possibly influence the FD's left-hand side, $V \subseteq \mathcal{V}$ is the set of variables influencing the FD's left-hand side, M is a set of method declarations (i.e., of the methods possibly called in the process of computing the new value for *var*), and $L \subseteq \mathcal{L}$ is a set of locations representing those objects that might be referenced by *var*.

Method *test()* from the example is modelled by collecting all FDs arising from its five statements. These FDs can be seen in the following report, which is automatically generated by the jade modeling component:

```
% Jade System Description Global Report
%
% Statement Index: 1
% Statement: p1=new Point(0,0)
```

FD ('1::y_1' ⇐ { consts(0), vars(), methods(Point::Point(int,int)), locs() })
FD ('1::x_1' ⇐ { consts(0), vars(), methods(Point::Point(int,int)), locs() })
FD ('p1_1' ⇐ { consts(), vars(), methods(Point::Point(int,int)), locs(1) })

% Statement Index: 2
% Statement: p2=new Point(2,3)
FD ('2::y_1' ⇐ { consts(3), vars(), methods(Point::Point(int,int)), locs() })
FD ('2::x_1' ⇐ { consts(2), vars(), methods(Point::Point(int,int)), locs() })
FD ('p2_1' ⇐ { consts(), vars(), methods(Point::Point(int,int)), locs(2) })

% Statement Index: 3
% Statement: p1.x=1
FD ('1::x_2' ⇐ { consts(1), vars('p1_1'), methods(), locs() })

% Statement Index: 4
% Statement: p1.y=2
FD ('1::y_2' ⇐ { consts(2), vars('p1_1'), methods(), locs() })

% Statement Index: 5
% Statement: p2=p1.plus(p2)
FD ('3::x_1' ⇐ { consts(), vars('p1_1','1::x_2','p2_1','2::x_1'), methods(Point::Point(int,int),Point::plus(Point)), locs() })
FD ('3::y_1' ⇐ { consts(), vars('p1_1','1::y_2','p2_1','2::y_1'), methods(Point::Point(int,int),Point::plus(Point)), locs() })
FD ('p2_2' ⇐ { consts(), vars('p1_1'), methods(Point::Point(int,int),Point::plus(Point)), locs(3) })

All in all 11 FDs are created for a total of 5 statements. 5 of these FDs can directly be associated with the 5 assignments appearing in the source code of method *test()*. The other 6 FDs are side-effect components and imported into the model of *test()* via the calls of *Point(int x, int y)* and *plus(Point p)*, respectively.

Note that since the FD model is static, its creation does not involve or require any information about a particular evaluation trace. As described above, the detailed FD model distinguishes between references (i.e. variables of reference type) and object locations. By doing this we solve aliasing problems occurring in the Java source code.

2.2 A simplified functional dependency model (SFDM)

The advantage of the DFDM approach of recording locations and references separately is that a broad range of program bugs can be located at statement level, due to the distinction being made between memory locations and variables. On the other hand, the DFDM is difficult to read and understand, and the large amount of constants, variables, and locations on the right-hand sides of FDs sometimes leads to slow diagnosis processes. Further on a potential user has to have detailed knowledge about the underlying object structure when specifying an incorrect variable observation.

This section therefore introduces a simplified functional dependency model (SFDM), which can automatically be derived from the DFDM. It is easier to understand, includes only variables on the FDs' right-hand sides, and makes it easier for the user to specify observations. FD's in SFDM again have the simpler structure of [Mateis, Stumptner, & Wotawa, 1999], but due to being derived from DFDM, still incorporate the effects of locations.

The conversion from DFDM to SFDM is obtained by successively simplifying each FD of the DFDM. Consider a FD $d = (vo, DEP)$, where $DEP =< C, V, M, L >$, to be converted to a SFD $d' = (vo', DEP')$, where DEP' is the subset of V that contains all local or instance variable occurrences of the class that the method is defined for, i.e., variable occurrences of the form *n::y* for $n > 0$ are deleted.

The variable occurrence on the left hand side is computed as follows. First, assume that vo is of the form x, *0::x* (i.e., it is a local variable occurrence or instance variable occurrence of the analyzed class), then $vo' = vo$. Now, assume that *vo* is of the form *n::x* referring to $l \in L$. Then all (local or instance) variable occurrences y which at the given point within the program possibly reference l are resolved by introducing a new FD of the form (y, DEP') (with DEP' defined as above).

During this process the indices of all variable occurrences are adapted in order to guarantee a correct sequence of occurrence indices from the method's beginning to its end.

As a result, FDs for variable occurrences of reference type now no longer simply denote that reference, but instead the dependency beyond that reference to locations and references representing the local state of the referenced object (since these locations are no longer explicitly present in the model).

The overall set of FDs occurring at run-time can now be covered by $\mathcal{C}$ and $\mathcal{V}'$. The locations $\mathcal{L}$ are no longer explicitly part of the model, but implicitly covered through $\mathcal{V}'$.

The resulting SFDM for method *test()* reads as follows:

% Jade System Description Global Report
%
% Statement Index: 1
% Statement: p1=new Point(0,0)
FD ('p1_1' ⇐ { vars()})

% Statement Index: 2
% Statement: p2=new Point(2,3)
FD ('p2_1' ⇐ { vars()})

% Statement Index: 3
% Statement: p1.x=1
FD ('p1_2' ⇐ { vars('p1_1')})

% Statement Index: 4
% Statement: p1.y=2
FD ('p1_3' ⇐ { vars('p1_2')})

% Statement Index: 5
% Statement: p2=p1.plus(p2)
FD ('p2_2' ⇐ { vars('p1_3','p2_1')})

These two FD models introduce the notion of aliasing. Previously we considered as aliasing the situation of two variables of reference type referring to the same object. Now we extend this to the case where two variables of reference type refer to two different object structures, which in turn have references to the same object in common.

Consider the following program fragment using two variables p1, p2 of class Point:

1. p2 = p1;
2. p1.doubleXValue();
3. output(p2.x);

Assume the call to method doubleXValue() is erroneous, and this is detected by observing an incorrect value in p2.x (which is the same as p1.x since they refer to the same Point object). The model from [Mateis, Stumptner, & Wotawa, 1999] would suggest statement 1 as a potential source of the error, but omit statement 2. The introduction of a location l on which both p1 and p2 depend and expressing the change caused by the method call in line 2 in terms of a change to l.x solves this problem.

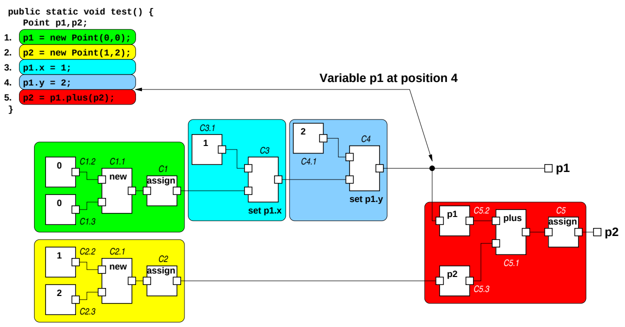


Figure 3: Graphical representation of the value-based model

3 The value-based model

In [Mateis, Stumptner, & Wotawa, 2000], we described a value-based model for a subset of Java. In our model analysis we use this model as our second reference model. To be self-contained, we briefly discuss the basics behind the value-based model. In the value-based model expressions and statements are represented as diagnosis components, with the semantics of the expressions and statements described in sets of logical sentences. Components are connected if there is a flow of information between the corresponding expressions and statements. An information flow between an assignment and another statement occurs for example, if the assignment changes a variable that is accessed by the other statement, and there is no assignment changing the same variable in between.

3.1 Description

Figure 3 shows the graphical representation of the model of example program *test1*. The source code of line 1 is mapped to four diagnosis components: two components for the constants ($C1.2, C1.3$), one component for the *new Point* method call ($C1.1$), and one component ($C1$) that corresponds to the assignment statement. Accessing of objects' instance variables is also represented as a component *set* ($C3$ or $C4$) with two parameters: the variable storing the object, and the name of the instance variable (see line 3 or 4). If variables are used in an expression, then they are mapped to a variable access component (see components corresponding to line 5).

In the following behavior description, we use tuples of the form $[Id[Var_1 Val_1] \ldots [Var_n Val_n]]$ to represent objects with the unique object identifier Id, and the pairs $[Var_i$ $Val_i]$ that map each instance variable Var_i to their value Val_i. Note that Val_i itself can be an object identifier.

Constants: If assumed to be correct, constants propagate their constant value, e.g., an integer or even an object, to the output port.

Variable access: A variable access has one input and one output. The input is connected with the output of the last assignment having the same variable as target. If assumed to be correct, the input value is propagated to the output and vice versa, known output values are propagated to the input.

Operators: An operator has an input port for each argument and one output. The inputs are connected to the components that correspond to subexpressions. The output is connected to a statement port, e.g., the expression port of an assignment, or to one input of the superexpression. The behavior of an operator is specified by the semantics of the corresponding statement. For example the Java *and* (`&&`) operator component has the following behavior: $\neg AB(C) \Rightarrow (in_1(C) \wedge in_2(C) = out(C))$

Method calls: The behavior of method calls is determined by the source code of the method declaration. This source code itself is represented by a diagnosis model consisting of components and connections. Currently, our implementation does not support dynamic binding, i.e., the methods called must be known at compile time. If the corresponding method call component is assumed to be correct, all components of the diagnosis model of the method declaration are also assumed to be correct and values are propagated.

Assignments: Assignments are represented differently depending on whether they change the value of locally declared or instance variables (e.g., `p1.x = 3;`). In the first case, if working correctly, the single input value is propagated to the single output and vice versa. The second type has one input that is connected to the compiled expression, and one input that is connected to the corresponding connection of the variable. The output port is connected to components where the corresponding statements are using this variable. The correct behavior is to change only the instance variable V_i of the object $[Id$ $[Var_1\ Val_1] \ldots [Var_n\ Val_n]]$ occurring on the input port of the variable. In the back propagation step the expected object is propagated from the output to the input port of the variable, and the value of the used instance variable is propagated to the port connected to the expression of the statement.

The described model can now be used to locate a bug in the *test* program. For this purpose assume that the call *test()* should compute the object $[?\ [x\ 1]\ [y\ 1]]$ to be stored in variable *p1*. But *test* returns the object $[?\ [x\ 1]\ [y\ 2]]$ for variable *p1*. The model together with the test-case produces four diagnoses: $\{AB(C1.1)\}, \{AB(C1)\}, \{AB(C4,1)\}, \{AB(C4)\}$ which correspond to the statements 1 and 4. Note that if using a fault model for the *new Point* method and the assignment statement, we can further remove the diagnoses $\{AB(C1.1)\}$ and $\{AB)(C1)\}$. This can be done by propagating an empty object $[?\ [x\ ?]\ [y\ ?]]$ to the output in case of an unknown fault AB. For a more detailed introduction to the value-based model including discussion of how it is used for debugging we refer to [Mateis, Stumptner, & Wotawa, 2000].

3.2 Refining conditional statements

The value-based model described in [Mateis, Stumptner, & Wotawa, 2000] treats conditional statements loosely, i.e., in some cases too many diagnoses can be derived from the model. Consider for example the following program fragment where the expected value of *result* is 0 after execution:

```
1.    x = 1;
2.    if (x < 1) {
3.       result = 1;
```

```
      } else {
4.       result = 1; // 1 should be 0 !!!
      }
```

With the original value-based model we derive diagnoses saying that either statement 1 or statement 2 (including the sub-blocks) causes the misbehavior. But it is obvious that the assignment in line 1 is not a single fault. This effect is due to the missing behavior in cases where the correct value of the condition expression of a conditional statement is known, which is the case when we assume that line 1 is faulty. To avoid this problem we extend the model of conditional statements:

1. If there is a variable x such that *then* and *else* block compute different values for x and both values are different from the expected value, then raise a contradiction.
2. If there is a variable x such that *then* and *else* block compute the same value for x, propagate this value to the output.

Formally, the new model can be expressed as follows:

$$\neg AB(C) \wedge cond(C) = true \Rightarrow (then_X(C) = out_X(C))$$
$$\neg AB(C) \wedge cond(C) = false \Rightarrow (else_X(C) = out_X(C))$$
$$\neg AB(C) \wedge cond(C) = unknown \wedge then_X(C) \neq out_X(C) \wedge else_X(C) \neq out_X(C) \Rightarrow \bot$$
$$\neg AB(C) \wedge cond(C) = unknown \wedge then_X(C) = else_X(C) \Rightarrow (out_X(C) = then_X(C))$$

With this improvement line 1 is not longer a diagnosis candidate.

4 Results

In this section we present results that are obtained from analysing the dependency-based (in the form of SFDM) and value-based model. In the first part we compare the two models with respect to their capabilities of debugging Java language constructs. Then we show how the models can be used together with a debugging GUI to actually locate bugs in Java methods, and we present some results from test carried out with both types of models. Finally we present some shortcomings of the described models together with possible solutions.

4.1 Model comparison

We first consider the Java language features currently supported by both models. Figure 1 shows the results of this comparison. We see that both models nearly support all features (including object-oriented ones). Only the value-based model currently imposes syntactical and semantical restrictions on programs. For example not all loop statements are currently supported, and recursive functions are not modeled. However, meaningful programs far away from toy examples can be implemented if we consider the restrictions.

The Java features currently not supported are interfaces, exceptions, and processes. Whereas exceptions and interfaces will be easy to incorporate into the dependency-based model, we expect problems for programs using parallel processes communicating with each other. The debugging of single processes, however, i.e., of standard sequential Java programs, is not a problem at all. In case of the current value-based model, more severe changes are required to handle exceptions and interfaces. The latter requires handling dynamic binding in the value-based model.

	Dependency-Based	Value-Based
Full Java syntax	$\surd$	
Classes/Instances	$\surd$	$\surd$
Static methods	$\surd$	$\surd$
Instance methods	$\surd$	$\surd$
Static variables	$\surd$	$\surd$
Instance variables	$\surd$	$\surd$
Inheritance	$\surd$	$\surd$[a]
Method calls	$\surd$	$\surd$[b]
Polymorphism	$\surd$	$\surd$[c]
Aliasing	$\surd$	$\surd$
Exceptions		
Interfaces		
Processes		

[a] Dynamic binding is currently not supported

[b] The value-based model is not defined for recursive functions. *while* statements are allowed.

[c] The methods must be determined by the compiler. See note *a*.

Table 1: Comparing different models

Independent of particular language features, we can compare results and requirements of the models.

Observations The dependency-based model requires a classification into correct and incorrect variables, based on the expected and computed results. The value-based model requires a test case (including the expected values of the observed variables), but no explicit classification.

Prediction The dependency-based model cannot determine the value of variables at specific points in the program. However, correctness can be judged by the user by examining the values obtained from program execution. The value-based model can predict values whenever possible.

Diagnosis The dependency-based model diagnoses individual statements, but not expressions within statements. The value-based model identifies bugs at the statement and expression level.

Diagnosis process Both approaches use a hierarchical approach for diagnosis. In the first step, a statement (or expression) of a block is identified as being faulty. If this statement is a conditional statement, diagnosis can be performed for one of the sub-blocks. If the statement or expression is a method call, the body of the method can be debugged.

4.2 Using the models for debugging

The program models described in sections 2 and 3 have both been implemented in Smalltalk (Visualworks 5i) and tested on a Sun UltraSPARC II/360 512 MB Solaris 2.6. Together with the `jade` debugging GUI they can be used to locate bugs in the source code of the modeled Java programs.

When using the SFDM described in section 2, the FD model of the selected method is automatically compiled into a system description, which can be represented as a set of logical sentences and thus be loaded into a standard theorem prover. The user is then asked to specify observations for the connections of the system description. This means that he has to state whether the value of a certain variable at a particular position within the source code has the correct value or not. These observations, which can directly be derived from the observed in/out behaviour of the tested method, are then transformed into logical propositions and added to the internal theorem prover. All statements which might contain the actual program bug are finally computed as possible diagnoses by the debugger.

In a second, more interactive debugging option, the user has the opportunity to continue specifying additional variable observations , in order to successively reduce the set of possible diagnoses and eventually find a single bug location. a measurement selection algorithm automatically selects the measurement point, i.e. variable occurrence, whose evaluation will reduce the number of diagnoses in an ideal way. After a certain number of such steps a single statement containing the bug should be found by the debugger. In such a case the user has the following options:

- In case of statements containing blocks (faulty selection or loop statement) the user is asked to evaluate the statement's condition. If the condition is correct, the debugging process is guided into the statement's subblocks, where the debugging process described above is started again. Note that no variable observations have to be specified by the user in the first step, because the debugger uses information from the block enclosing the statement.
- In all other cases the bug has been found at statement level. A list of all sub-expressions of the faulty statement enables the user to inspect the internal structure of the statement and thus guide the debugging process into a method or constructor call.
- Stop the debugging process and continue manually from the current diagnosis candidate list

If the value based model (VBM) described in section 3 is used to debug the selected method, the debugging process looks similar to the one described above for the FD model. Internally a constraint propagation system is used instead of a theorem prover to cope with the dynamic nature of value-based debugging. Furthermore it is no longer enough to specify the correctness of a variable occurrence; when working with the VBM the exact intended value of the observed variable has to be known and specified. In the current implementation of the `jade` debugger it is possible to specify multiple observations in one diagnosis step (unlike when using the FD model).

The VBM works at expression level, i.e. diagnoses no longer consist of faulty statements, but contain expressions of various sizes from assignment statements to simple variable occurences. This on the one hand allows for a more accurate bug location, but on the other hand leads to an increase in system connections (and thus in variable observation queries). Once a single faulty location has been found by the debugger, the debugging process is terminated. Selection statements and their branches can directly be debugged together with the enclosing block. While loops are not currently being treated specially, but this is being implemented.

Test	Method	Lines	Error	Interactions		
				FD Model	VBM	
				Total 1	Total 2	Total 3
1	$adder\_f1$	17	4	3	3	4
2	$adder\_f2$	17	4	3	3	4
3	$adder\_f3$	17	7	3	2	3
4	$adder\_f4$	17	7	3	2	3
5	$adder\_f5$	17	7	3	3	4
6	$adder\_f6$	17	12	5	4	5
7	$adder\_f7$	17	12	5	4	5
8	$adder\_f8$	17	11	5	3	4
9	$adder\_f9$	17	11	7	4	5
10	$adder\_f10$	17	14	5	2	3
11	$adder\_f11$	17	14	5	3	4
12	$adder\_f12$	17	16	5	5	6
13	$adder\_f13$	17	12	5	4	5
14	$adder\_f14$	17	9	5	4	5
15	$ifTest\_f1$	5	4	4	3	4
16	$ifTest\_f2$	5	4	3	2	3
17	$ifTest\_f3$	9	8	4	4	4
18	$ifTest\_f4$	6	5	5	2	3
19	$ifTest\_f5$	8	8	5	2	3
20	$ifTest\_f6$	12	10	5	3	4
21	$library\_f1$	31	24	8	8	8
Sum		314	203	96	70	89
Av.		14.95	9.67	4.57	3.33	4.24

Table 2: Debugging results from Java examples

4.3 Empirical results

The debugging potential of the FD model has been tested in previous work. Results of tests with programs demonstrating variable dependencies (*adder.java*), multiple objects and instance fields (*library.java*), and various sorting algorithm of linked lists (*list.java*) can be found in [Mateis *et al.*, 2000b].

In order to test the VBM together with the debugger and compare the results with those obtained from the FD model, all tests with *adder.java* and *library.java* were repeated using the VBM. The modeling of selection statements was tested by introducing *ifTest.java* [1]. Note that tests with *list.java* heavily rely on the debugging of the body of loop statements, which is currently not implemented for the VBM, only for the dependency-based models. These tests will be carried out in the next version of the `jade` debugging system.

Table 2 shows (from left to right) the tested method (in which a single error has been installed), the total number of statements in each method, the index of the buggy statement within the method (which can directly be used to compare the outcomes of the `jade` debugger tests with "manual" use of a debugger where the user steps through the code sequentially until the erroneous line is found), and finally the number of user interactions which are needed to exactly locate the bug. The latter are defined as follows:

Total 1 shows the total number of user interactions needed to locate a single error at statement level using the FD

[1]Note that all the testcases can be downloaded from our project page `http://www.dbai.tuwien.ac.at/proj/Jade/`

model. Such interactions are the variable setup (evaluation of the method's output variables), variable observation queries, and selection statement queries. The latter are used to guide the user into the incorrect subbranch of a faulty selection statement. Note that no hierarchical debugging in case of incorrect loops and method calls was performed.

Total 2 shows the performance of the VBM at statement level debugging and includes the variable setup and all variable observation queries. Selection statement queries are not needed with the VBM, because the system automatically guides the diagnosis process into the incorrect subbranch. Note that for each measurement selection query only one observation was specified. Debugging was stopped once a single buggy statement was found.

Total 3 shows the number of user interactions needed to locate the faulty expression using the VBM. In most cases it took one extra step from statement level to expression level fault localization.

The following behaviour of the debugger could be observed:

- In general both debugging strategies seem to perform reasonably well. Whereas on average the error is installed in line 9.67 (i.e. a normal debugger would need 10 steps to assuredly find the bug), it only takes 4.57 or 3.33 user interactions, respectively, to locate the bug at statement level.
- With program *adder.java* and tests with selection statements (*ifTest.java*) the VBM achieves better results at statement level than the FD model. This is mainly due to the additional run-time information used by the VBM debugger, which allows for a more effective elimination of wrong diagnoses.
- As mentioned above in most cases only one additional query was used to find the solution at expression level with the VBM. On average the VBM debugger finds a solution at expression level quicker than the FD debugger at statement level.
- With program *library.java* both models seem to be equally good. Although the VBM uses more information than the FD model, this does not pay off in this case, because of the complex object structure of *library.java*.
- At the moment the debugging process with the VBM is still very slow. Whereas this was no problem with *adder.java*, a single debugging step takes up to 9 minutes with *library.java*. It is planned to speed up the VBM debugging process through both methodic and implementation improvements.
- As mentioned above, in each step of the VBM debugging only one variable observation was added.

4.4 Analysis

Although a large number of debugging problems can be solved by at least one of the two models, there are still classes of programs that cause problems as described below. We always assume that the programs pass the compiler and terminate on all test-cases.

Faulty location structure A large problem class, consisting of two cases:

1. A variable refers to the same object as another variable but should point to a different object (maybe with the same content).
2. Two variables point to different locations, i.e., objects, but should refer to the same object.

This class of faulty programs is characterized by a wrong structure in the location space of a program. The following program fragment is an example for the first case.

```
1.    p1 = new Point(0,0);
2.    p2 = p1; // Should be p1.copy()
3.    p1.x = 1;
        // Expected results: p1=(1,0), p2=(0,0)
```

When using the functional dependency model

```
% Statement: p1 = new Point(0,0);
    FD('p1_1' <- {vars()})
% Statement: p2 = p1;
    FD('p2_1' <- {vars('p1_1')})
% Statement: p1.x = 1;
    FD('p1_2' <- {vars('p1_1','p2_1')}, 'p2_2' <- {vars('p1_1')})
```

the debugger lists all statements as bug candidates. However, because statement 3 changes the value of *p1* and its value is correct at the end of the program, the statement can actually not be responsible for the faulty behavior. This additional candidate is caused by the dependency for 'p2_2'. If we eliminate this dependency from statement 3, the dependency-based model delivers the expected results, but can no longer deal with aliasing. One solution would be to use the two models (with and without aliasing) in combination.

The problem is worse for the DFDM and the VBM. The expected values of the above program indicate that we have two different values for the same object. From the model-based perspective, we would conclude that the observations are contradictory and we are not able to diagnose. Again, a solution of this problem must introduce means for allowing to speak about missing or redundant object locations.

Structural faults We say that a program has a structural fault, if the wrong variable in the program is accessed or changed. Such faults can be repaired by replacing the variable. The term structural faults comes from the observations that the corresponding dependency graph [Ferrante, Ottenstein, & Warren, 1987; Horwitz, Reps, & Binkley, 1988] of the program is not structurally equivalent to the corresponding dependency graph of the corrected program.

The class of structural faults can be further subdivided. The first subclass comprises programs where the target of an assignment is wrong. Consider for example the following program fragment:

```
1.    int x=0, y=0 ;
2.    x = 2;
```

3. x = 2; // Should be `y=2`
// Expected results: x=2, y=2

For both models the debugger returns statement 1 as the only candidate but the expected bug cannot be located. In some cases, e.g., when line 3 reads `x=3` and we expect the outputs *x*=2 and *y*=3, the debugger is able to locate the bug. In general this cannot be guaranteed. In such cases there are several possible solutions. One is to search for equivalent statements that occur very closely in the program. Another one is to provide the user with assignments changing a variable to the expected value. A combination of both solutions could also be a promising approach. A different solution is to introduce replacement fault modes for assignments, that test different variables during diagnosis [Stumptner & Wotawa, 1999], or to make use of program mutation.

The second subclass of structural faults is due to the wrong use of variables in expressions. An example is shown in the following code:

1. x = 3;
2. tmp = 2;
3. y = 2 * tmp; // Should be `y=2*x`
// Expected results: x=3, y=6

Because no value for *tmp* is specified, the debugger would return statement 2 and 3 as diagnosis candidates, if using one of our models. Moreover, if we use the value-based model, we obtain the variable access *tmp* in line 3 as a single fault. Hence, this second class of structural faults can implicitely be handled by our models.

5 Conclusion

In this paper we have extended our earlier work on dependency-based models of imperative programs by the description of dependencies for object references that allow the diagnosis of situations involving aliasing, and discuss the tradeoffs inherent in the models. We also present results gained from experimenting with the implementation that incorporates both these models as well as, for the first time, for the value-based model [Mateis, Stumptner, & Wotawa, 2000]. Both models present an improvement over the earlier dependency-based model [Mateis, Stumptner, & Wotawa, 1999]. As predicted, the performance of the dependency-based models is satisfactory and can be expected to scale up well to medium-sized programs (thousands of lines of code). (The even simpler dependency-based representation described in [Friedrich, Stumptner, & Wotawa, 1999] provided acceptable performance for programs with hundreds of thousands of lines of code, and was useful despite the weak discrimination of that model since it allowed to rule out larger parts of the program structure as potential error sources just as here.) The value-based model still has to has to be improved in runtime performance. However, also as predicted, the experiments have shown it to be superior to the dependency based models in terms of required user interaction, and both are superior to manual debuggers.

References

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

[Bond, 1996] Bond, G. W. 1996. Top-down consistency based diagnosis. In *Proceedings of the Seventh International Workshop on Principles of Diagnosis*.

[Console, Friedrich, & Dupré, 1993] Console, L.; Friedrich, G.; and Dupré, D. T. 1993. Model-based diagnosis meets error diagnosis in logic programs. In *Proceedings* 13^{th} *International Joint Conf. on Artificial Intelligence*.

[Ducassé, 1993] Ducassé, M. 1993. A pragmatic survey of automatic debugging. In *Proceedings of the 1st International Workshop on Automated and Algorithmic Debugging, AADEBUG '93*, Springer LNCS 749, 1–15.

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

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

[Horwitz, Reps, & Binkley, 1988] Horwitz, S.; Reps, T.; and Binkley, D. 1988. Interprocedural Slicing Using Dependency Graphs. In *Proc. SIGPLAN Conf. on Prog. Lang. Design and Implementation*.

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

[Mateis *et al.*, 2000a] Mateis, C.; Stumptner, M.; Wieland, D.; and Wotawa, F. 2000a. AI Support for Debugging Java Programs. In *3rd Workshop on Intelligent SW Eng.*

[Mateis *et al.*, 2000b] Mateis, C.; Stumptner, M.; Wieland, D.; and Wotawa, F. 2000b. Model-Based Debugging of Java Programs. In *Proceedings of the 4th International Workshop on Automated and Algorithmic Debugging, AADEBUG '00.*

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

[Mateis, Stumptner, & Wotawa, 2000] Mateis, C.; Stumptner, M.; and Wotawa, F. 2000. Modeling Java Programs for Diagnosis. In *Proceedings of the European Conference on Artificial Intelligence (ECAI)*.

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

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

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