Comparing Two Models for Software Debugging*

Markus Stumptner[1], Dominik Wieland[2], and Franz Wotawa[2]**

[1] University of South Australia, School of Computer and Information Science, 5095 Mawson Lakes, Adelaide, SA, Australia, mst@cs.unisa.edu.au

[2] Technische Universität Wien, Institut für Informationssysteme, Database and Artificial Intelligence Group, Favoritenstraße 9–11, A-1040 Wien, Austria, {wieland,wotawa}@dbai.tuwien.ac.at

Abstract. This paper extends 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

Detecting, locating, and repairing faults in software is a difficult and time consuming task. Detecting an incorrect behavior of a given program is done by using testing techniques (e.g., [1]) or formal verification methods, e.g., model checking [3]. Whereas much effort has been made on test theory, test methodology, and algorithms for automatic test-case generation, somewhat less work has been published on locating and repairing software faults. Because debugging is not only performed in the implementation and test phases of a project, but also in maintenance, saving debugging time naturally results in saving time and money over the whole product life cycle. Especially in maintenance, where the original developers may no longer be involved, debugging is very costly. An automated debugger for locating and fixing faults can help in such a situation.

Automatic debugging approaches introduced in the past include program slicing [22, 23], algorithmic debugging [19], dependency-based techniques [13, 12, 11], probability-based methods [2], and others. An overview of automatic debugging techniques can be found in [6]. These traditional approaches are either specific to a programming language, use specialized algorithms, or require explicit user-interaction to locate a bug. In order to overcome these drawbacks and to improve the results of the above-mentioned approaches, the use of model-based diagnosis (MBD) for debugging was

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

** Authors are listed in alphabetical order

suggested [4]. Model-based diagnosis [18, 5] provides a general theory for diagnosis that has sucessfully been applied to various engineering areas. Apart from the diagnosis of technical systems, e.g., [14, 24], MBD has been used for knowledge bases [20, 7] and other less technical domains, e.g., in ecology [10].

In this paper we build on the results obtained by the MBD community, especially as its application to the localisation of software bugs is concerned ([4] and more relevantly [9, 21]). Section 2 gives a brief description of the basics of MBD and shows how it can be used for fault localisation in programs. Section 3 presents some general thoughts about the modeling of Java programs and lists a sample program, which will be used throughout this paper to describe various model properties. Section 4 describes how a functional dependency model (FDM) of Java programs can automatically be derived from the program's source code. Furthermore, the transformation into a simplified functional-dependency model and the handling of object references (and therefore aliasing) are discussed. Section 5 shows how semantic information can be incorporated into the model by introducing a value-based model (VBM). Such a model allows for better results at the cost of an increased debugging time. Section 6 describes how the two models can be used to actually locate faults in programs. Furthermore, it presents empirical results produced by the `jade` prototype debugger. The `jade` debugging tool is a model-based debugger for Java programs featuring both, the use of functional dependency and value-based models. The diagnosis engine used by `jade` is based on Reiter's algorithm [18]. The `jade` user interface is designed to guide the user through the code in an optimal fashion, with optimality defined as minimizing the user interaction. An analysis of the debugging potentials of our approach and some future extentions conclude this work.

2 Model-Based Diagnosis and debugging

The model-based approach is based on the notion of providing a representation of the correct behavior of a technical system. By describing the structure of a system and the function of its components, it is possible to ask for the reasons why the desired behavior was not achieved. In the diagnosis community, the model-based approach has achieved wide recognition due to the following advantages:

- once an adequate model has been developed for a particular domain, it can be used to diagnose different actual systems of that domain
- the model can be used to search for single or multiple faults in the system without alteration
- different diagnosis algorithms can be used for a given model
- the existence of a clear formal basis for judging and computing diagnoses

Using the standard consistency-based view as defined by Reiter [18], a diagnosis system can be formally seen as a tuple $(SD, COMP)$ where SD is a logical theory sentence modeling the behavior of the given system (in our case the program to be debugged), and $COMP$ a set of components, i.e., statements or expressions. A diagnosis system together with a set of observations OBS, i.e., a test-case, forms a diagnosis problem. A diagnosis Δ, i.e., a bug candidate, is a subset of $COMP$, with the property

that the assumption that all statements (expressions) in Δ are incorrect, and the rest of the statements (expressions) is correct, should be consistent with SD and OBS. Formally, Δ is a diagnosis iff $SD \cup OBS \cup \{\neg AB(C) | C \in COMP \setminus \Delta\} \cup \{AB(C) | C \in \Delta\}$ is consistent. A component not working as expected, i.e., a statement (expression) containing a bug, is represented by the predicate $AB(C)$.

The basis for this is that an incorrect output value (where the incorrectness can be observed directly or derived from observations of other signals) cannot be produced by a correctly functioning component with correct inputs. Therefore, to make a system with observed incorrect behavior consistent with the description and avoid a contradiction, some subset of its components must be assumed to work incorrectly. In practical terms, one is interested in finding minimal diagnoses, i.e., a minimal set of components whose malfunction explains the misbehavior of the system (otherwise, one could explain every error by simply assuming every component to be malfunctioning). Basic properties of the approach as well as algorithms for efficient computation of diagnoses are described in [18].

The principles of model-based debugging are depicted in Figure 1. The program, in our case written in Java, is compiled into an internal representation. From this representation (together with a set of *model fragments*) a converter computes logical models for diagnosis. Model fragments represent a logical description of parts of a model, e.g., the behavior description of functions. Such knowledge has to be derived from the programming language semantics. For Java for example, a value-based model requires the model fragments of all basic functions and types of statements, e.g., for arithmetic functions, behaviors (e.g., $Nab(C) \rightarrow out(C) = in_1(C) + in_2(C)$ for the + operator) must be defined. The predicate Nab stands for *not abnormal*, saying that a function or statement is not responsible for an incorrect behavior. After building the model, which is done automatically, the model together with the specified behavior of the program, e.g., test-cases, is used by the diagnosis engine for finding bug candidates. The candidates can be further discriminated by adding additional knowledge, i.e., values of variables at specific locations within the program. The selection of the variable and location is done by a measurement selection algorithm. The information about the value must be delivered by the user (or another oracle). The remaining candidates provide a link back to the original source code.

3 Modeling for debugging

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

Note that variable declarations are not counted as statements, because no diagnosis components can arise from them. We classify the entities in the program in terms of

- The set $\mathcal{C}$ of *constants* (e.g., 0, 1, 2). Although constants in Java are not objects, they can logically be seen as objects with a fixed content of primitive type. Each occurrence of such a constant has to be considered separately, but we omit indexes for simplicity in the paper.

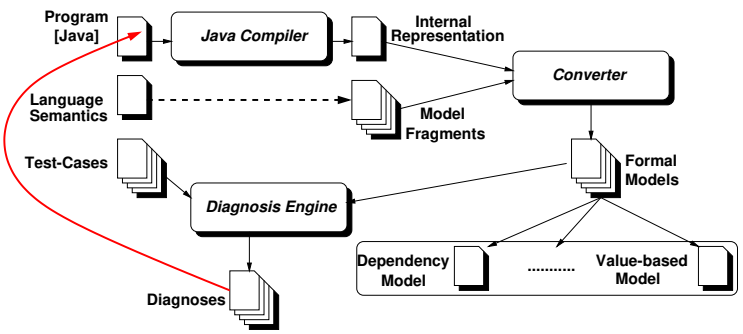


Fig. 1. MBD in software debugging

```
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); }
```

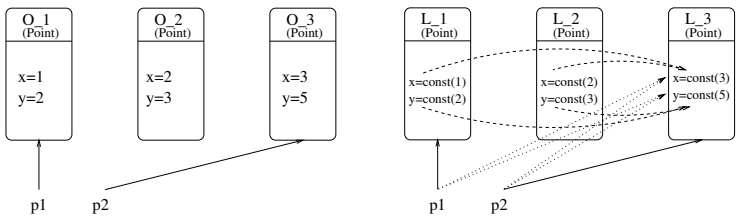
```
public static void test() {
    Point p1, p2;
1.  p1 = new Point(0,0);
2.  p2 = new Point(2,3);
3.  p1.x = 1;
4.  p1.y = 2;
5.  p2 = p1.plus(p2); } }
```

Fig. 2. Example program *Point.java*

- 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 [11]. Their internal state is user defined and alterable (in our example 3 locations of type *Point* are created), but it is important to note that locations, which are introduced during static analysis of the functional dependencies, do 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, individual objects are used instead.
- The set $\mathcal{V}$ of *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 3 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).



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

Fig. 3. Representing objects

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 4) and a dynamic value-based model (Section 5).

4 The dependency-based model

A dependency model for Java programs was presented in [15], defining 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 (e.g, [17]), but are quite relevant in Java due to the language's general reference semantics for object-valued variables.

4.1 The detailed functional dependency model

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* if *x* denotes a class

variable or a local variable of the currently modeled block; 0::x if *x* stands for an instance field of the receiver of the method; *n::x* where *n*> 0 is a location representing a different object than the receiver. *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 and $V \subseteq \mathcal{V}$ is the set of variables influencing *var*, 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 1: 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 2: 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 3: p1.x=1
    FD ( '1::x_2' ⇐ { consts(1), vars('p1_1'), methods(), locs() })
% Statement 4: p1.y=2
    FD ( '1::y_2' ⇐ { consts(2), vars('p1_1'), methods(), locs() })
% Statement 5: 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) })
```

Six of the above 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)*, the others are directly derived from the assignment statements in *test()*. Note that since the FD model is static, its creation does not involve or require any information about a particular evaluation trace.

4.2 A simplified functional dependency model

The advantage of the DFDM which records 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. However, the DFDM is difficult to read and understand, and the large size of FDs slows down diagnosis performance. Also, the user needs 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. FDs in SFDM again have the simpler structure of [15], 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 and class variables and only those instance variables, which are defined for the method's owner class, i.e., variable occurrences of the form *n::y* for $n > 0$ are deleted.

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

As a result, FDs for variable occurrences of reference type now no longer simply denote that reference. Instead the dependency directly refers to the 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:

```
FD ( 'p1_1' ⇐ { vars()})                  % Statement 1: p1=new Point(0,0);
FD ( 'p2_1' ⇐ { vars()})                  % Statement 2: p2=new Point(2,3);
FD ( 'p1_2' ⇐ { vars('p1_1')})            % Statement 3: p1.x=1;
FD ( 'p1_3' ⇐ { vars('p1_2')})            % Statement 4: p1.y=2;
FD ( 'p2_2' ⇐ { vars('p1_3','p2_1')}) % Statement 5: p2=p1.plus(p2);
```

These two FD models introduce the notion of aliasing, which describes any situation 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 code 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 an incorrect value is observed in p2.x (which is the same as p1.x since they refer to the same Point object). The model from [15] would suggest statement 1 as a potential source of the error, but would omit statement 2. Both, the DFDM and the SFDM introduce a location l on which both p1 and p2 depend and express the change caused by the method call in line 2 in terms of a change to l.x, thus making the aliasing between the two variables explicit.

5 The value-based model

In the value-based diagnosis model both expressions and statements are represented as diagnosis components, with the semantics of the expressions and statements described in terms of logical sentences. Components are connected if there is a flow of information between the corresponding expressions and statements, e.g., if a changes a variable v that is accessed in s and there is no assignment to v in between.

5.1 Description

Figure 4 shows the graphical representation of the model of example program *test*. The source code of line 1 is mapped to four diagnosis components: two components for the constants ($C1.2, C1.3$), one for the *new Point* method call ($C1.1$), and one ($C1$) that corresponds to the assignment statement. Instance variable accesses are also represented as components ($C3$ 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). We refer to a component C as "correct" if it is assumed to be correct, i.e., $\neg AB(C)$ holds.

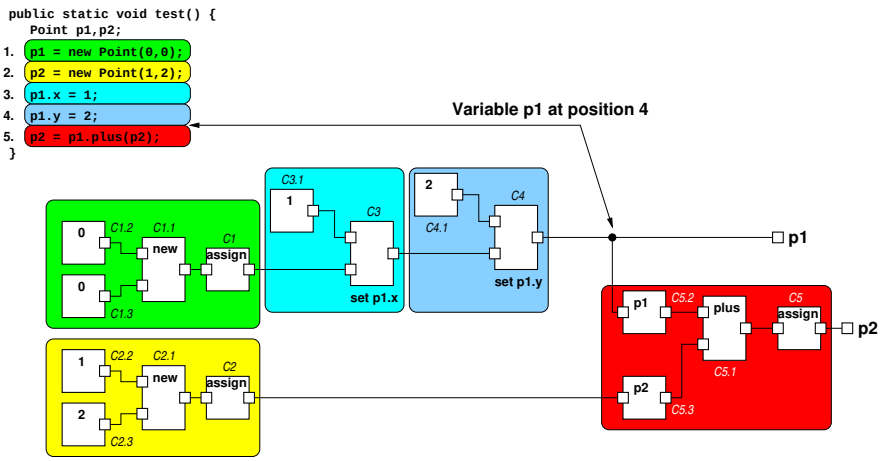


Fig. 4. Graphical representation of the value-based model

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 its value Val_i. Note that Val_i itself can be an object identifier.

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

Variable access: This has one input, connected to the output of the last assignment having the same variable as target, and one output. A correct access propagates the input value to the output and known output values 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 a correct method call is determined by the diagnosis model constructed from the method's source code to propagate the input parameter values to output parameters or return value.

Assignments: If assumed to be correct, assignments $v = e$ to locally declared variables propagate the single input value produced by expression e to the single output and vice versa. Assignments to instance variables (e.g., `p1.x = 3;`) have a second input carrying the object. The output port is connected to components which access the 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 output port of e.

Control statements: Loop components have as input connections the variables used inside the loop, and as output connections the variables altered by the loop, propagating values repeatedly (as needed) across the components of the body of the loop. Selection statements propagate values to either the **then** or **else** branch depending on condition evaluation.

Diagnoses are computed in the usual consistency-based manner [18]: 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.

To locate a bug in program *test*, assume that the call *test()* should compute the object $[?\ [x\ 1]\ [y\ 1]]$ to be stored in variable *p1*. Instead, *test* returns the object $[?\ [x\ 1]\ [y\ 2]]$ for variable *p1*. The test case produces four diagnoses: $\{AB(C1.1)\}, \{AB(C1)\}, \{AB(C4, 1)\}, \{AB(C4)\}$ which correspond to statements 1 and 4. If using a fault model for the *new Point* method and the assignment statement, we can rule out the diagnoses $\{AB(C1.1)\}$ and $\{AB)(C1)\}$ by propagating an empty object $[?\ [x\ ?]\ [y\ ?]]$ to the output in case of an unknown fault AB. For more details of the value-based model see [16].

5.2 Refining conditional statements

The value-based model treats conditional statements loosely, i.e., in some cases too many diagnoses can be derived from the model. Consider the following program fragment with expected value for *result* being 0:

```
1.    x = 1;
2.    if (x < 1) {
3.       result = 1; }
4.    else { 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 subblocks) 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. A remedy is to extend the model of conditional statements:

1. If there is a variable x such that the **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 v for x, propagate v to the output.

Formally, the new model can be expressed as follows:

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

When using this model, line 1 is excluded as a diagnosis candidate.

6 Results

In this section we compare the dependency-based and value-based model, first in terms of the Java language constructs covered, then in terms of tests carried out with the `jade` debugger on both types of models using actual programs. Note that during all tests with FDMs a SFDM was used. Whereas a DFDM could produce more accurate results, its handling would be more difficult, especially as far as the exact specification of observations is concerned. We therefore use the DFDM as a generic model, which represents the basis for the SFDM and in future other possible dependency-based models. Finally we analyze some routes for future work.

Both models currently support classes and instances, static and instance methods and variables, method calls, polymorphism and aliasing. The value-based model currently does not allow for recursive functions. Both models support sequential code exclusively (no processes).

Since the value-based model expresses the full semantics of the covered language features, it can predict values by forward propagation (from inputs to outputs) or backward propagation (from outputs to inputs) whenever sufficient values are present on a component's connections. During debugging, it allows the user to specify intended actual values as observations instead of merely observing whether a value is correct or incorrect. Also, the value-based model's diagnosis granularity is lower and individual subexpressions can be identified as incorrect.

6.1 Using the models for debugging

The diagnosis models were implemented in the context of the `jade` debugger and tested on a Sun UltraSPARC II/360. Both models are automatically derived from the source code and 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 program is evaluated and the user is then asked to specify expected values (observations) for the connections of the system description. This means that when using the SFDM he has to state whether the value of a certain variable (directly derived from the observed in/out

behaviour of the tested method) at a particular position within the source code has the correct value or not, and when using the VBM he can specify the intended values. The observations are converted into logical sentences and added to the debugger's system description. The user can 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 (the `jade` system uses a slightly modified version of [5]) 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 s containing the bug is found by the debugger. In this case the user has the following options:

1. If s contains subblocks (a selection or loop statement), the user can interactively guide the diagnosis process into the condition or one of the subblocks. If s is a loop, the user also has to select the first incorrect iteration from a displayed list. When stepping into a subblock, information from the enclosing statement s is propagated downwards. **2.** In all other cases the bug has been found at statement or expression level, and in the case of a method or constructor call, debugging can continue inside that call. **3.** Stop the debugging process, continue manually from the current diagnosis candidate list.

6.2 Empirical results

The relative debugging potential of both models was tested on programs demonstrating simple variable dependencies (simulating a binary adder, numeric examples), making use of control structures (if and while statements), and finally multiple objects and instance fields together with linked lists and general processing (a small library application).

Table 5 shows the tested method (in which a single error has been installed), the total application code length (which determines the model size; esp. in the library example most of this code is also called by the buggy function), the number of statements in the tested 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 (variable setup, variable observation queries, program flow control queries) which are needed to exactly locate the bug[1]. The latter are defined as follows:

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

V1 shows the performance of the VBM in statement level debugging. Selection statement queries are not needed with the VBM, because the system automatically guides the diagnosis process into the faulty subbranch. For each measurement selection query only one observation was specified.

V2 shows the number of interactions needed to locate the faulty *expression* using the VBM. In most cases it took one or two extra steps from statement level to expression level.

[1] The average is included for overview purposes; it has no statistical significance since it depends on the specific selection of program and error examples.

In general both debugging strategies perform better than a manual walkthrough. On average it takes 1.1 additional queries to find the solution at expression level. Therefore the VBM usually finds a solution at expression level quicker than the FDM at statement level.

In general the VBM achieves better results than the FDM, mainly due to the additional run-time information used by the VBM debugger, which allows for a more effective elimination of wrong diagnoses (see adder and numeric tests). With a growing number of variables and more complex dependency structures this effect is likely to increase. If the error appears within an if statement, the VBM clearly outperforms the FDM (see if-tests), mainly due to the VBM debugger's ability to debug the body of a selection statement together with its enclosing block, whereas the FDM debugger performs a hierarchical diagnosis involving some overhead (control and setup GUIs). With loops, both models require extra queries on loop condition correctness (mainly to identify the first incorrect iteration) in combination with setup queries for the subblock's variables. This leads to more queries for both, reducing the advantages of the VBM. In the numeric examples the apparently high number of queries is somewhat misleading since the complex computations automatically produce good focusing and result in identifying detailed subexpressions quickly. With the library program both models seem to be equally good. Although the VBM uses more information than the SFDM, this does not pay off here because of the complex object structure of the application combined with the relatively simple operations possible in the example (mostly lookup operations concerning lists of books and customers which are highly dependent on the input values provided).

6.3 Discussion

The `jade` debugger utilizes a model-based diagnosis framework to incorporate multiple automatically derived models of programs into a standard debugger interface, allowing to switch between traditional step-based debugging and between models at the flick of a switch and providing iterative error location without the need for external specifications. The latter properties are the main distinction compared to formal verification methods such as model checking, which requires a separate formal specification and provides counterexamples but no indication of the error locality.

Current implementation work includes better runtime for the VBM model, since loops with complex object structures such as large lists can result in diagnosis runtimes of several minutes. Howevere, most of the examples in the table were diagnosed in the 1-10 second range. Another goal is better visualization for indicating correctness of complex object structures.

In modeling, the identification and modeling of specific problem classes can be expected to lead to still more effective debugging. The dependency-based representation can be expected to scale up well to medium-sized programs (thousands of lines of code). (The somewhat simpler dependency-based representation described in [9] provided acceptable performance for programs with hundreds of thousands of lines of code.)

Faulty location structure. Either two variables point to different locations, i.e., objects, but should refer to the same object, or a variable refers to the same object as

Test	Method	LOC	Lines	Error	FD	V1	V2
1	*adder_f*1	71	17	4	3	3	4
2	*adder_f*2	71	17	4	3	3	4
3	*adder_f*3	71	17	7	3	2	3
4	*adder_f*4	71	17	7	3	2	3
5	*adder_f*5	71	17	7	3	3	4
6	*adder_f*6	71	17	12	5	4	5
7	*adder_f*7	71	17	12	5	4	5
8	*adder_f*8	71	17	11	5	3	4
9	*adder_f*9	71	17	11	7	4	5
10	*adder_f*10	71	17	14	5	2	3
11	*adder_f*11	71	17	14	5	3	4
12	*adder_f*12	71	17	16	5	5	6
13	*adder_f*13	71	17	12	5	4	5
14	*adder_f*14	71	17	9	5	4	5
15	*ifTest_f*1	109	5	4	4	3	4
16	*ifTest_f*2	109	5	4	3	2	3
17	*ifTest_f*3	109	9	8	4	4	4
18	*ifTest_f*4	109	6	5	5	2	3
19	*ifTest_f*5	109	8	8	5	2	3
20	*ifTest_f*6	109	12	10	5	3	4

Test	Method	LOC	Lines	Error	FD	V1	V2
21	*whileTest_f*1	119	12	6	3	3	3
22	*whileTest_f*2	119	12	10	6	6	7
23	*whileTest_f*3	119	13	4	2	3	3
24	*whileTest_f*4	119	13	6	3	3	5
25	*whileTest_f*5	119	13	10	10	8	9
26	*differntiat_f*1	98	4	3	4	4	5
27	*differntiat_f*2	98	4	3	4	4	5
28	*differntiat_f*3	98	4	4	4	2	4
29	*differntiat_f*4	98	4	4	4	2	4
30	*differntiat_f*5	98	4	4	4	2	4
31	*differntiat_f*6	98	4	4	4	2	4
32	*integrate_f*1	98	8	2	3	2	3
33	*integrate_f*2	98	8	2	2	2	2
34	*integrate_f*3	98	8	2	4	2	3
35	*integrate_f*4	98	8	6	7	7	10
36	*integrate_f*5	98	8	6	7	7	10
37	*trafficLight_f*1	106	15	12	9	7	8
38	*trafficLight_f*2	106	15	7	7	7	8
39	*library_f*1	215	31	24	8	8	8
Σ			575	342	237	143	186
Av.			11.98	7.12	4.94	3.67	4.77

Fig. 5. Debugging results from Java examples

another variable but should point to a different object (possibly with the same content), as in this 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 its simple functional dependency model

```
FD('p1_1' <- {vars()})                                          % Statement 1: p1 = new Point(0,0);
FD('p2_1' <- {vars('p1_1')})                                    % Statement 2: p2 = p1;
FD('p1_2' <- {vars('p1_1','p2_1')}, 'p2_2' <- {vars('p1_1')}) % Statement 3: p1.x = 1;
```

the debugger will list 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 two different models, one with and one without aliasing. The best model to be used for a specific problem could then be chosen by the user (if a-priori knowledge about the problem exists) or by the use of given error class statistics. Similar observations hold for the DFDM and VBM.

Structural faults. The wrong variable in the program is accessed or changed, i.e., the dependency graph [8] of the program is not structurally equivalent to the dependency graph of the correct program. Such faults can be repaired by replacing the variable. The following fragment depicts an incorrect assignment target:

```
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, but in general more powerful solutions need to be applied, such as the introduction of replacement fault modes for assignments [21].

The second subclass of structural faults is the wrong use of variables in expressions:

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

Because no value for *tmp* is specified, the debugger would return statement 2 and 3 as diagnosis candidates regardless of model. With the VBM, we obtain the variable access *tmp* in line 3 as a single fault. Hence, this second class of structural faults can implicitly be handled by our models.

7 Conclusion

In this paper we have extended 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, the implementation of a value-based model of imperative programs. Both models represent an improvement over the earlier dependency-based model in terms of diagnosis discrimination. Runtime performance of the dependency-based models is satisfactory, while the value-based model still has to be improved in runtime performance. However the experiments have shown it to be superior to the dependency based models and stepwise manual debugging in terms of required user interaction.

References

1. Boris Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, 1990.
2. Lisa Burnell and Eric Horvitz. Structure and Chance: Melding Logic and Probability for Software Debugging. *Communications of the ACM*, 38(3):31 – 41, 1995.
3. Edmund M. Clarke, Orna Grumberg, and David E. Long. Model Checking and Abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.
4. Luca Console, Gerhard Friedrich, and Daniele Theseider Dupré. Model-based diagnosis meets error diagnosis in logic programs. In *Proceedings* 13^{th} *International Joint Conf. on Artificial Intelligence*, pages 1494–1499, Chambery, August 1993.

5. Johan de Kleer and Brian C. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32(1):97–130, 1987.
6. Mireille Ducassé. A pragmatic survey of automatic debugging. In *Proceedings of the 1st International Workshop on Automated and Algorithmic Debugging, AADEBUG '93*, Springer LNCS 749, pages 1–15, May 1993.
7. Alexander Felfernig, Gerhard Friedrich, Dietmar Jannach, and Markus Stumptner. Consistency based diagnosis of configuration knowledge-bases. In *Proceedings of the Tenth International Workshop on Principles of Diagnosis*, Loch Awe, June 1999.
8. 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.
9. Gerhard Friedrich, Markus Stumptner, and Franz Wotawa. Model-based diagnosis of hardware designs. *Artificial Intelligence*, 111(2):3–39, July 1999.
10. Ulrich Heller and Peter Struss. Conceptual Modeling in the Environmental Domain. In *Proceedings of the 15^{th} IMACS World Congress on Scientific Computation, Modelling and Applied Mathematics*, volume 6, pages 147–152, Berlin, Germany, 1997.
11. Daniel Jackson. Aspect: Detecting Bugs with Abstract Dependences. *ACM Transactions on Software Engineering and Methodology*, 4(2):109–145, April 1995.
12. Bogdan Korel. PELAS–Program Error-Locating Assistant System. *IEEE Transactions on Software Engineering*, 14(9):1253–1260, 1988.
13. Ron I. Kuper. Dependency-directed localization of software bugs. Technical Report AI-TR 1053, MIT AI Lab, May 1989.
14. A. Malik, P. Struss, and M. Sachenbacher. Case studies in model-based diagnosis and fault analysis of car-subsystems. In *Proceedings of the European Conference on Artificial Intelligence (ECAI)*, 1996.
15. Cristinel Mateis, Markus Stumptner, and Franz Wotawa. Debugging of Java programs using a model-based approach. In *Proceedings of the Tenth International Workshop on Principles of Diagnosis*, Loch Awe, Scotland, 1999.
16. Cristinel Mateis, Markus Stumptner, and Franz Wotawa. Modeling Java Programs for Diagnosis. In *Proceedings of the European Conference on Artificial Intelligence (ECAI)*, Berlin, Germany, August 2000.
17. 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.
18. Raymond Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–95, 1987.
19. Ehud Shapiro. *Algorithmic Program Debugging*. MIT Press, Cambridge, Massachusetts, 1983.
20. Markus Stumptner and Franz Wotawa. Model-based reconfiguration. In *Proceedings Artificial Intelligence in Design*, Lisbon, Portugal, 1998.
21. Markus Stumptner and Franz Wotawa. Debugging Functional Programs. In *Proceedings 16^{th} International Joint Conf. on Artificial Intelligence*, pages 1074–1079, Stockholm, Sweden, August 1999.
22. Mark Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–452, July 1982.
23. Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.
24. Brian C. Williams and P. Pandurang Nayak. A reactive planner for a model-based executive. In *Proceedings 15^{th} International Joint Conf. on Artificial Intelligence*, pages 1178–1185, 1997.