Can AI help to improve debugging substantially? Debugging Experiences with Value-Based Models[1]

Wolfgang Mayer[2] and **Markus Stumptner**[3] and **Dominik Wieland**[4] and **Franz Wotawa**[5,6]

Abstract. Finding and fixing faults in programs is usually an expensive and tedious task. Consequently the development of intelligent debugging tools that aid the programmer in this task is a topic of major industrial interest. This work describes two representations for applying model-based diagnosis to Java programs, a technique that permits locating (and partly correcting) faults without requiring a formal specification of the desired program behavior, since interaction can be limited to test cases and observations of variable correctness. One of the models uses a special transformation to provide more accurate diagnoses on programs with loops and this is borne out by the experiments. The presented results on actual debugging performance show clearly superior accuracy to classical debugging techniques, and better discrimination than dependency-based programs models. We discuss the results in terms of the properties of the two models and the various example programs and present avenues for further improvement.

1 INTRODUCTION

Debugging, i.e., to find a faulty behavior, to locate the causing fault within a program, and to fix the fault by means of changing the program, has been of interest for the last decades. Many papers have been published so far in the domain of finding faults in software, e.g., testing or formal verification [1], and locating them, e.g., program slicing [13] and automatic program debugging [11, 7]. More recently model-based diagnosis [10] has been used for locating faults in software by several researchers [2, 6, 4, 12]. [2] shows the relationship between automatic program debugging and the model-based approach. In [16] the author discusses the relationship between slicing and model-based debugging.

In this paper we rely on previous research in model-based diagnosis for locating bugs in Java programs. The idea behind the model-based debugging approach is (1) to compile a program to its logical model or to a constraint satisfaction problem, (2) to use the model together with test cases and a model-based diagnosis engine for computing the diagnosis candidates, and (3) to map back the candidates to their corresponding locations within the original program. The selection of diagnosis components, e.g., statements, expressions, or other program entities, influences the diagnosis granularity. Another important issue is the selection of the model. In the past two categories of models developed in the Jade project were published which represent two end points of the spectrum: dependency-based [8] and value-based models [9].

Several models from both categories have been implemented and tested in the Jade project. The tests consist of small- up to medium-sized Java programs together with their faulty variants and given test cases. This test collection was used to carry out the model evaluation. We present the results obtained for two different value-based models which are also described. Starting from the results we discuss some strengths and weaknesses of the current models. From the results and the discussion we show that model-based diagnosis indeed improves the debugging results, and we show on examples how modeling decisions influence not only the runtime but also the debugging results substantially, although at the price of a major performance hit compared to dependency-based models. Further improvements include the possibility of writing special models for different purposes. For example, it makes sense to write a model to locate bugs which are due to wrongly used variables in the code or to write models that specially treat the aliasing problem in object-oriented software.

The paper is organized as follows: In Section 2 two value-based models of Java programs are described. Both models closely follow the execution semantics of Java. Section 3 presents some results that are obtained from the implementation. This section is followed by an in-depth discussion of advantages and disadvantages of the models. Finally, we conclude the paper and describe avenues of further research.

2 MODELING FOR DEBUGGING

To apply MBD approaches to software debugging, a model representing the program under consideration has to be built. This section focuses on two models that closely simulate the execution semantics of the Java language. The models are currently limited to a subset of Java.

Given a program's source code, the simple value-based model (VBM) [9] is derived by transforming each element of the program into a set of components and connections between them. Statements and expressions are represented as components, where the connections between them correspond to the used and modified variables, respectively. For example, constants are represented as a single component with one output port that corresponds to the value of the constant. Whereas simple elements of the programming language are represented as single components, more complex structures such as conditional statements, loops or method calls have to be represented differently. Conditional statements are represented as components that forward the results of one of its branches to its outputs, depending on the evaluation of its condition, which is provided

[1] This work was partially supported by the Austrian Science Fund project P12344-INF.

[2] University of South Australia, School of Computer and Information Science, Mawson Lakes Boulevard 1, 5092 Mawson Lakes SA, Australia, email: mayer@cs.unisa.edu.au

[3] University of South Australia, School of Computer and Information Science, Mawson Lakes Boulevard 1, 5092 Mawson Lakes SA, Australia, email: mst@cs.unisa.edu.au

[4] Vienna University of Technology, Institute for Information Systems, Database and Artificial Intelligence Group, Favoritenstrasse 9-11, A-1040 Vienna, Austria, email: wieland@dbai.tuwien.ac.at

[5] Graz University of Technology, Institute for Software Technology, Inffeldgasse 16b/II, A-8010 Graz, Austria, email: wotawa@ist.tu-graz.ac.at

[6] Authors are listed in alphabetical order

```
abstract class Shape{
  abstract void rotate90();
  static void main(String[] args){
    Shape[] shapes = new Shape[5];
    shapes[0] = new Rect(1,1,3,5);
    shapes[1] = new Circle(4,0,1);
    for(int i = 0; i < shapes.length; ++i){
      if (shapes[i] != null)
        shapes[i].rotate90(); }}}
class Point{
  float x,y;
  Point(float x,float y){
    this.x = x;
    this.y = y; }
  void rotate90(){
    float t = x; x = -y; y = t; }}
class Rect extends Shape{
  Point ll,ur;
  Rect(float x0,float y0,float x1,float y1){
    ll = new Point(x0,y0);
    ur = new Point(x1,y1); }
  void rotate90(){
    float x0 = ll.x;
    float y0 = ll.y;
    float x1 = ur.x;
    float y1 = ur.y;
    ll.x = y1;  //-y1
    ll.y = x0;
    ur.x = -y0;
    ur.y = x1; }}
class Circle extends Shape{
  Point center;
  float r;
  Circle(float x,float y,float r){
    this.center = new Point(x,y);
    this.r = r; }
  void rotate90(){
    center.rotate90(); }}
```

Figure 1. Example Program

through an additional input connection. The statements belonging to the branches of the conditional are represented as separate components and connections, where the output connections of the then- and the else-branch are used as inputs for the component representing the conditional. The connections used as input values for the models of the two branches are chosen such that the last connection representing the used variable before the execution of the conditional is used. The conditional's output connections represent the values that are computed for the represented variables after the execution of the conditional.

The VBM represents loops and method calls as hierarchic components, where the input- and output-ports are derived from the loop's condition and the loop body or the called method's body, respectively. For polymorphic method calls, the component contains not just a single model fragment, but models of all method bodies that could possibly be invoked by the statement. This is necessary, as it cannot be determined at compile-time which method is called. The selection is done during the model's execution, when the actual type of the receiver is known.

Representing Objects To avoid similar problems regarding object references, objects and their instance variables are not directly represented as connections. Instead they are accessed indirectly through object identifiers (OIDs). The OIDs are assigned when an object is created and are used to uniquely identify an object throughout the model's execution. The values of the instance variables are stored in object environments (OEs) and are accessed using the OID of the corresponding object. Whenever an assignment to an instance variable is modeled, a new version of the OE has to be created that reflects the changes. The unchanged parts of the environment are copied from the OE that is valid before the execution of the assignment expression. To avoid copying the whole Java heap every time an instance variable is modified, the OEs is sliced such that for each instance variable of a class a separate OE is created that holds the values of the variable for the class' instances. In addition, a heap structure analysis [3] is performed to separate OEs associated with references for which it is known at compile time that they refer to different objects. Therefore, the OEs are formed according to may-alias relations between the reference variables accessing them and the copying overhead is kept at a minimum.

The modeling of the OEs can be formalized as follows: Each expression E of the program (i.e., constants, operators, etc.) can be described as a function $sem_E : ENV \rightarrow ENV'$ that transforms the environment ENV valid before the execution of the expression into an environment ENV' valid after the expression, which reflects the changes caused by the expression. Here, each environment is separated into two parts: One part, *SVAR*, contains the information about the local and the static variables, whereas the second part, OE, is responsible for the OE. Therefore, each ENV is a pair $\langle$*SVAR*, $OE_1 \cup \cdots \cup OE_k\rangle$, where OE_1 to OE_k are partitions of OE, according to a preceding heap structure analysis. In our modeling approach, each variable in *SVAR* and each OE_i in OE is modeled by a separate connection. Therefore, the behavior of each expression is given by sem_E, where the input is restricted to the subset of *SVAR* and the set of OE_i that contains all the relevant information to simulate the specification of E. Parts of ENV' that are not affected by the expression are not modeled through sem_E. For these parts of ENV', the connections representing the corresponding part in ENV are reused during the modeling process. In our representation, each OE_i is a collection consisting of tuples of the form $\langle$*OID*,*VNAME*,*IDX*,*VAL*$\rangle$, where OID denotes the identifier of the object the tuple corresponds to, and *VNAME* denotes the field name. IDX is optional and is used to distinguish values for different array indexes of an array. Finally, *VAL* denotes the value from OE that is referred to by the first three elements of the tuple.

Due to space limitations, the aspects of the model dealing with class instance creations and arrays are not described in this work, but they are straight forward extensions of the approach presented here.

Model Structure To illustrate the model building process, the model of the loop body of method main() in Figure 1 is depicted in Figure 2. First, the model of the condition of the if statement is built. The array access expression is modeled as three components, representing the access to the object reference shapes, the array index i, and the array access operator [], respectively. The slice of the OE holding the values of the array is passed as an input connection to the component representing the array access. Here it is assumed that the array has been associated with an abstract location L1 during the heap structure analysis. Next, the model of the conditional's then-branch is built. The modeling of the array access expression, which forms the receiver of the method call, is modeled exactly as the condition. The method call itself is modeled by first building the models of the methods Rect.rotate90 and Circle.rotate90 (the two possibly called methods) and then combining the models into a single hierarchically structured component. The object the method is invoked on and the required instance variables for both method models are passed as input parameters. Here it is assumed that the three instances of class Point belonging to the rectangle (location L2) and the circle (location L5) are associated with the abstract locations L3, L4, and L6, respectively. The output connections of the method call correspond to the instance variables that are modified by any of the methods. Once the model of the then-branch of the conditional has been obtained, modeling finishes with the introduction of the component representing the if statement. The input connections associated with the then-branch of the conditional are connected to the output connections of the model of the then-branch. As no else branch is present, the connections associated with the else-branch are con-

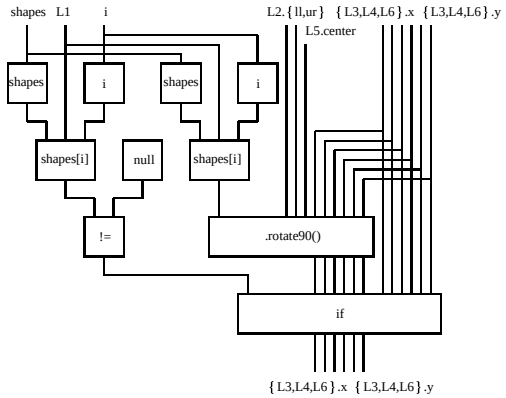

Figure 2. Model of loop body

nected to the connections that were valid before the execution of the conditional statement. The output connections correspond to the values of variables after the execution of the conditional statement.

Model Behavior Whereas the structure of the model is determined by the analyzed program, the behavioral description of the components is given by the Java Language Specification (JLS). The behavior sem_E of each component is described as sentences in predicate logic and simulates the behavior of the corresponding language element E from the JLS. For the sake of brevity, the discussion here is restricted to a few examples. Most of the remaining components are simple and their behavior is straightforward to obtain. Further details can be found in [9].

A component C representing a constant expression k introduces the constant into ENV'[7]: $\neg AB(C) \Rightarrow result(C) = k$. In a similar way, components representing an access to a variable $v \in SVAR$ is modeled as follows: $\neg AB(C) \Rightarrow result(C) = in(C)$, where $in(C)$ corresponds to the value of a connection representing the value of $v \in ENV$. For accesses to variables from OE, the accessed object and the relevant OE_i must be provided as input. We write $VName$ for the name of the variable that is accessed and $oe(C)$ for the value of the connection corresponding to OE_i:
$\neg AB(C) \wedge oid(C) \neq \epsilon \wedge (\exists_{j,v} \langle oid(C), VName, j, v\rangle \in oe(C)) \Rightarrow result(C) = v$.
Here, ϵ represents the fact that no value is known for the connection. If the variable references an array, an additional input connection $idx(C)$ is required, which corresponds to the index of the accessed array element and replaces the auxiliary variable j from above. Components representing assignments to variables are modeled similarly, with the addition of further output connections that represent the new versions of the OE partitions OE_i' that are possibly modified by the assignment. In this case, rules for the tuples inserted into OE_i', as well as rules dealing with the tuples copied from OE_i have to be specified.

As mentioned above, a correct conditional statement forwards the values computed by one of its branches to its outputs. The branch to be selected is determined by the value of the condition. The (simplified) behavior can be expressed as follows (C is a component representing a conditional statement; $cond$, $then_V(C)$, $else_V(C)$ and $out_V(C)$ denote the input and output ports of C; V represents a variable modified in one of the branches):

$$\forall_V \neg AB(C) \wedge cond(C) = true \Rightarrow out_V(C) = then_V(C)$$
$$\forall_V \neg AB(C) \wedge cond(C) = false \Rightarrow out_V(C) = else_V(C)$$

A call to a method M is modeled such that each sentence $\neg AB(C') \wedge \Gamma \Rightarrow \Delta$ of the model of M is replaced by $\neg AB(C) \wedge type(oid(C)) \in callertypes_M \neq \epsilon \wedge \Gamma' \Rightarrow \Delta'$, where Γ' and Δ' are derived from Γ and Δ, respectively, by replacing all connections of the model of M by connections of the component C modeling the call. $type()$ represents a function that retrieves the actual type of the object corresponding to $oid(C)$. $callertypes_M$ represents the set of all types for which M is defined but not overridden by another method.

Loop Free Model As will be demonstrated in the following sections, the diagnostic accuracy of the VBM is not sufficient when dealing with loops. Therefore, we developed a variant of the VBM where loops are expanded into nested conditional statements, resulting in improved backward reasoning capabilities and independent fault assumptions for statements inside the loop. This model is called (unsurprisingly) the loop-free model (LFM). The number of conditional statements that each loop is expanded into is derived from the number of iterations the loop executes during the initial execution of the faulty program using a test case.

In addition to the expansion of loops, the behavior of components representing conditional statements is modified such that the selection of the branch is independent of the evaluation of the condition. Instead, a default mode (comparable to the $\neg AB$-mode in the previous model) is replaced by a mode $Then(C)$ or $Else(C)$, which selects the corresponding branch of the conditional, depending on the initial execution. The $AB(C)$ mode is replaced by the remaining mode. This can be formalized as follows (the $default()$ predicate is necessary to avoid undesirable effects when setting the value of the condition):

$$\forall_V Then(C) \wedge default(Then(C)) \Rightarrow$$
$$\quad out_V(C) = then_V(C) \wedge cond(C) = true$$
$$\forall_V Else(C) \wedge default(Else(C)) \Rightarrow$$
$$\quad out_V(C) = else_V(C) \wedge cond(C) = false$$
$$\forall_V Then(C) \wedge \neg default(Then(C)) \Rightarrow out_V(C) = then_V(C)$$
$$\forall_V Else(C) \wedge \neg default(Else(C)) \Rightarrow out_V(C) = else_V(C)$$

Replacing the modes AB and $\neg AB$ with the modes from above, backward reasoning through conditionals is possible even though the value of the corresponding condition is unknown. Therefore, conflicts can be generated more often, which results in fewer diagnoses.

3 EMPIRICAL RESULTS

This section describes the results obtained by applying the models described in the previous section to a set of test programs and compares the models with respect to their diagnostic capabilities.

To evaluate the debugging accuracy of the models, a set of example programs has been created which is used to investigate the specific advantages and disadvantages of the model variants. Most of the example programs implement well-known algorithms which could be part of larger programs, each of them including a seeded fault. For example, programs executing a binary search procedure, computing the Huffman encoding of an array of characters, or applying Gauss elimination are part of the test suite. Throughout this work, we assume that the faulty program is a close variant of the correct program. We do not deal with wrong choice of algorithms, data structures or similar major design defects.

The diagnostic experiments were performed by specifying the inputs of the program together with the expected results as observations. A summary report of the obtained results for each example program is depicted in Table 1. Several aspects of the examples are listed: Stm denotes the number of statements in the program, C represents the number of components in the generated model. D stands

[7] In the following description it is assumed that $result(C) \in ENV'$ represents an auxiliary variable corresponding to the result of the expression.

for the number of diagnoses of minimal cardinality that are obtained and H represents the number of diagnoses from D that actually include the seeded fault. S denotes the cardinality at which the diagnostic process was stopped because the seeded fault was located. Finally, the %-column lists the percentage of the statements that have to be examined in the worst case until the seeded fault is found. Here it is assumed that the diagnoses are presented with increasing cardinality. Note that these numbers can further be improved by suitable heuristics, which present the diagnoses according to their 'likelihood' to explain the faults. For the VBM, the columns H and S are omitted because their value is always equal to one. Numbers in parentheses denote cases where the faults could not be located because the maximum time allowed for diagnosis was exceeded. In these cases the numbers are lower bounds to the actual results that would be obtained when continuing the diagnostic process to its completion.

Program	Stm	VBM			LFM				
		C	D	%	C	D	H	S	%
BinSearch	27	16	6	63	43	1	1	2	8
Binomial	76	26	9	42	255	24	1	1	32
BoundedSum	16	14	4	38	19	1	0	2	38
BubbleSort	15	10	6	93	34	7	1	1	47
FindPair	5	4	4	100	10	1	0	2	80
FindPositive2	17	13	3	41	20	2	1	1	12
FindPositive3	17	13	3	41	20	2	1	1	12
Hamming	27	19	11	70	95	9	1	1	33
Huffman	64	22	9	80	161	9	0	(2)	(25)
Huffman	64	22	6	59	164	12	1	1	19
Intersection	95	31	12	84	155	8	1	1	5
Library	24	21	6	38	36	5	0	2	34
Matrix	71	21	21	100	127	37	1	1	52
MaxSearch2	21	16	3	38	37	2	0	2	19
MultLoops	21	12	2	19	27	4	2	3	24
MultiSet	97	55	8	28	283	1	0	(2)	(11)
Permutation	24	17	14	96	29	3	1	1	13
Permutation0	26	19	12	69	33	1	1	1	4
Permutation1	26	19	12	69	32	8	0	3	100
Permutation2	26	19	15	85	33	9	1	1	35
Permutation3	24	19	12	67	33	2	0	3	50
Polynom	120	64	14	24	189	26	0	(3)	(13)
SearchTree	84	41	41	100	140	45	0	(1)	(54)
SkipEqual	5	4	4	100	11	2	1	1	40
Stat	23	17	3	39	42	2	0	4	48
Sum	5	4	3	80	10	3	1	1	40
SumPowers	21	12	8	81	36	5	1	1	24
average	39	20	9	**65**	77	8	0.6	(1.6)	(**32**)

Table 1. Summary of the example programs and results

4 DISCUSSION

VBM Analyzing the results obtained with the VBM, it can be seen that the amount of code that has to be analyzed in order to locate a fault can be reduced significantly. In most cases, only between 40 and 80 percent of all statements have to be checked, with the average being at 65 percent. Comparing these numbers to results obtained with other approaches for program analysis, it can be seen that the VBM is able to locate faults more accurately than previous approaches. In particular, when comparing our approach to slicing [13], our results are much better[8]. This can be explained by the different levels of abstraction the two approaches apply. Our approach is somewhat closer to the actual execution semantics of the program than with program slicing, which only considers program dependencies. Consequently, we are able to reason about concrete values and are thus able to detect conflicts more often than approaches operating on a higher level of abstraction. This does not only include data flow in the direction of program execution, but also includes backward computation, i.e. reasoning from the observed output values towards the input values. Another improvement with respect to slicing is that we can provide more information to the user in case a loop has to be executed a different number of times to explain a fault. Those examples where no statements of the program could be eliminated are programs that are either very short (consisting of only an initialization statement and a loop) or are programs where almost every part of the program depends on every other part (for example a binary search tree, where the program execution depends on the values that were inserted previously).

The VBM usually provides good results for programs without loops but fails to compute satisfying diagnoses for programs that consist of large loop statements. This is due to the fact that the loop statements are modeled hierarchically and discrimination between statements inside the loops is not possible. Hence, either the entire loop is considered correct or incorrect, which often results in inaccurate diagnoses. Further, backward reasoning through loops generally is not possible (unless the number of iterations is known), which causes the model to include larger parts of the program.

LFM To overcome these problems, the LFM has been developed. Loops are expanded into a set of nested conditional statements, with separate assumption variables for each statement. Therefore, the model is able to reason about the loop's statements independently, without considering the whole loop as an entity. This provides a finer-grained resolution, which avoids the problem of large diagnosis entities mentioned above.

As can be seen in Table 1, switching from the VBM to the LFM leads to much better results. In particular, the percentage of statements that has to be considered until the fault is located is reduced to 32-43[9] percent on average, which is quite low compared to the percentage of statements that was computed by the VBM. These improvements are mainly caused by the improved backward reasoning mechanisms and the strong behavioral modes of the conditional statements.

The LFM causes some undesirable side effects. In particular, the size of the diagnostic problem increases due to the changed representation of loops and the separate fault assumptions for the conditional statements. However, this is usually not a severe problem because for programs without deeply nested loops the number of conditionals together with the number of loop iterations usually is relatively small compared to the total number of statements in the program.

For the LFM it is no longer the case that every faulty statement is included in a diagnosis of cardinality one (as with the VBM). Therefore, the cardinality up to which diagnoses have to be computed is likely to be > 1, depending on the type of fault and the program structure. For most example programs the diagnosis cardinality required to locate a fault is ≤ 2, which is usually computationally feasible when considering small- to medium-sized programs.

For programs where multiple loops or conditional statements exhibit faulty behavior, the required diagnosis cardinality increases further. This causes the search space to increase considerably and also diminishes diagnostic performance, as diagnoses of larger cardinality often include unnecessary statements. These problems can be addressed by multi-model reasoning and by suitable ranking of diagnoses, as will be discussed later on.

Another aspect of the LFM that keeps the model from being blindly applicable is the fact that the strong fault modes of the conditional statements decouple the selection of the conditional branch

[8] For most of the example programs in this section, static slicing is not able to eliminate any statement.

[9] 43% is obtained when assuming the whole program has to be examined for the examples where no exact solution was found. Better estimates (37%) are obtained when taking the percentages obtained with the VBM as upper bounds.

to be executed from the evaluation of the selection condition. Therefore, faults in the condition cannot be located using the LFM. Here, again, multiple models have to be combined to locate such faults efficiently. Fortunately, such faults can in many cases be found with the VBM alone and do not require the LFM to be applied (e.g., examples MultLoops, Permutation1 and Stat).

5 SOLUTIONS AND OPEN ISSUES

When considering the solution quality of the two models beyond the context of a single diagnosis run as we have done in the previous section, the applicability actually increases, as experience with the much less discriminatory (but much faster) dependency-based models has shown that even these models can very quickly home in on a fault by suggesting measurement points and having the user provide interactive "observations" on the correctness of particular variable values [14].

However, the limitations of the models that were discussed previously can also be attacked directly, by applying several possible extensions. Most of the ideas presented here are subject to further research and should only provide an idea on how to extend the models, apart from the obvious issue of covering the nonsequential Java language features which are still excluded from modeling: exception handling, threads, synchronization and reflection.

First, as no single model is accurate to diagnose all types of faults in a program, several specialized models have to be applied concurrently, each specialized to detect a specific class of faults. This multi-model-reasoning approach is not only applicable to the two types of value-based models discussed here, but can also be applied using multiple levels of abstraction or types of models. For example, a dependency-based model (such as slicing [13] or the dependency-based models from [14]) can be used to narrow the possible locations to a part of the program with manageable size and then apply combinations of the VBM and the LFM to exactly locate the fault. Also, models dealing with structural faults [5, 15] could be incorporated in such a framework.

For this approach to be applicable, suitable strategies to decide under which conditions to apply certain kinds of models have to be developed and evaluated. Based on these criteria, the most efficient model can be selected based on the program structure, the test cases and the diagnoses computed so far. This approach overcomes the drawbacks of both models, as well as reduces the computational complexity of the diagnostic process, because more detailed models are only instantiated when needed.

To select candidates for further inspection, suitable criteria for ranking diagnoses according to their likelihood to explain the fault have to be developed. As a consequence, unlikely diagnoses, i.e. diagnoses that are not expected to be of much value to the user, are eliminated by assigning them low preference values. However, developing generally applicable criteria to rate diagnoses is an open research issue.

Besides presenting diagnoses with decreasing likelihood, providing more specific information about the types of faults and possible corrections can help to eliminate faults more quickly. As described in [12], after a single diagnosis has been selected for further investigation, possible replacement expressions for the faulty expression can be inferred and presented as corrections. This also is an important step towards an intelligent, (semi-)automatic debugging and program correction tool.

Finally, a limitation of the current models is given by their high computational effort required to execute them. Although this is to a large part caused by the inefficient implementation, the high overhead caused by copying OEs also contributes noticeably. As a consequence, the models can only be applied to small- to medium-sized programs dealing with a moderate number of objects. To speed up the models, further research and combination with other techniques are required.

6 CONCLUSION

Building intelligent debugging aids for programmers is an important goal repeatedly attacked by researchers during the last decades. Unfortunately, no generally applicable solution has been found so far. In this paper we describe an approach relying on model-based diagnosis and introduce two models for Java programs that closely follow the Java execution semantics, including dynamic object creation and referencing. As presented, the performance of these models on test programs is quite encouraging and significantly exceeds that of earlier dependency-based models. In particular, the Loop Free Model proved to be clearly more effective than the "plain" Value-Based Model due to its more effective backward reasoning ability. Incorporating these models in a system with multi-model reasoning capability and ranking criteria for diagnoses holds the promise of wider applicability (due to integration with dependency-based models) and even better discrimination. As our approach clearly outperforms classical debugging techniques for many example programs, the model-based approach can be considered a promising technique that should be further researched to obtain a generally applicable debugging tool.

REFERENCES

[1] 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).

[2] 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*, pp. 1494–1499, Chambery, (August 1993).

[3] James C. Corbett, 'Using shape analysis to reduce finite-state models of concurrent java programs', Technical report, Department of Information and Computer Science, University of Hawaii, (1998).

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

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

[6] Beat Liver, 'Modeling software systems for diagnosis', in *Proceedings of the Fifth International Workshop on Principles of Diagnosis*, pp. 179–184, New Paltz, NY, (October 1994).

[7] J. W. Lloyd, 'Declarative Error Diagnosis', *New Generation Computing*, **5**, 133–154, (1987).

[8] 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).

[9] 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).

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

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

[12] Markus Stumptner and Franz Wotawa, 'Debugging Functional Programs', in *Proceedings* 16^{th} *International Joint Conf. on Artificial Intelligence*, pp. 1074–1079, Stockholm, Sweden, (August 1999).

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

[14] Dominik Wieland, *Model-Based Debugging of Java Programs Using Dependencies*, Ph.D. dissertation, Vienna University of Technology, Institute of Information Systems, 2001.

[15] Franz Wotawa, 'Debugging VHDL Designs using Model-Based Reasoning', *Artificial Intelligence in Engineering*, **14**(4), 331–351, (2000).

[16] Franz Wotawa, 'On the Relationship between Model-Based Debugging and Program Slicing', *Artificial Intelligence*, **135**(1–2), 124–143, (2002).