Observations and Results Gained from the Jade Project

Wolfgang Mayer[*] and **Markus Stumptner**[†] and **Dominik Wieland**[*] and **Franz Wotawa**[‡ §]

Abstract

This paper summarizes the work done in the course of the Jade project, which deals with automatic debugging of Java programs. Besides a brief introduction to the Jade project, models developed to debug Java programs are evaluated and results are presented. Furthermore, insights gained from the results are discussed and topics for further research are identified.

1 Introduction

For the last three years the Jade project has examined the applicability of model-based diagnosis (MBD) techniques to the software debugging domain. In particular, the goals of Jade were (1) to establish a general theory of model-based software debugging with a focus on object-oriented programming languages, (2) to describe the semantics of the Java programming language in terms of logical models usable for diagnosis, and (3) to develop an intelligent debugging environment for Java programs based on theoretic results.

The main practical achievement of the Jade project is the interactive debugging environment, which allows us to efficiently locate bugs in faulty Java programs. Currently, this debugger is fully functional with regard to nearly all aspects of the Java programming language and comes complete with a user-friendly GUI, the diagnosis system being integrated into a "normal" interactive debugger interface. The Jade debugger limits the search space of bug candidates by computing diagnoses for a given (incorrect) input/output behavior. This is done by using model-based diagnosis techniques, which in some cases have been adapted to suit the needs of an object-oriented debugging environment. Furthermore, the debugger can be used to unambiguously locate faults through an interactive debugging process, which is based on the iterative computation of diagnoses, measurement selection, and input of additional observations by the user.

This work is organized as follows: The next section briefly describes the program models used by the Jade debugging environment. Section 3 presents results obtained with the models introduced in Section 2. Section 4 analyzes the results from Section 3 and discusses some properties of the models. In Section 5, we point out interesting topics for further research. Section 6 briefly compares our approach to related work. Finally, we conclude the paper.

2 Program models

Since model-based diagnosis relies on the existence of a logical model description of the underlying target system, one of the most important components of the Jade system are its models. Currently, the Jade debugger makes use of two model classes, dependency-based models and value-based models. This section briefly describes these model types. More comprehensive descriptions can be found in [Stumptner *et al.*, 2001; Wieland, 2001; Mayer, 2000; 2001].

Dependency-based models are based on the collection of all data and control dependencies of a given Java program. As an example, we look at a single statement S_i, e.g., `int x=a*b;`. Informally, the variable dependencies arising from this statement can be specified by $S_i : x \leftarrow \{a, b\}$. A formal logical model can now automatically be derived from this dependency. For our example it reads $\neg AB(S_i) \wedge ok(a) \wedge ok(b) \Rightarrow ok(x)$, where the predicate AB stands for the assumption that a certain statement is incorrect, i.e., behaves abnormally. The predicate $ok(v)$ specifies that the value of variable v is correct without making use of the concrete value of v. Observations for such a model can be expressed by specifying the correctness or incorrectness of a certain variable, e.g., $\neg ok(x)$ in the above example. In the course of the Jade project different dependency-based models have been created that vary in their levels of abstraction and the amount of information used during their creation. These models are:

ETFDM: A dependency-based model, which makes use of a concrete execution trace [Wieland, 2001].

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

[†] University of South Australia, Advanced Computing Research Center, 5095 Mawson Lakes (Adelaide) SA, Australia, email: mst@cs.unisa.edu.au

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

[§] Authors are listed in alphabetical order

#	Test series	#TC	Diagnosis					Debugging					
			$\varnothing S_1$	$\varnothing D_1$	$\varnothing D_1(\%)$	$\varnothing D_2$	$\varnothing D_2(\%)$	$\varnothing S_2$	$\varnothing R$	$\varnothing T_1$	$\varnothing T_1(\%)$	$\varnothing T_2$	$\varnothing T_2(\%)$
1	Adder	14	17	8.14	48	8.14	48	17	10	3.9	39	3.9	39
2	IfTest	10	3.5	2.2 (1.9)	63 (54)	2.0	57	6.3	4.9	3	61	2.8	57
3	WhileTest	10	5.6	3.3	59	2.5	47	11.7	5.4	5.1	94	3.9	72
4	Numeric	9	4.6	4.6	100	4.6	100	6.2	3.6	4.4	120	5.3	147
5	Trafficlight	4	5	3	60	3	60	14	7.25	6.25	86	6.25	86
6	Library	5	26	20.6 (18)	79 (69)	20	77	33	18.6	7.8	42	7.6	41
$\varnothing$		1	10	6.3 (5.9)	**63 (59)**	6	**60**	13.4	7.6	4.6	**60**	4.5	**59**

Table 1: Diagnosis and debugging results of the dependency-based models

DFDM: A dependency-based model, which only makes use of static (compile-time) information, such as the Java source code and the programming language semantics [Stumptner *et al.*, 2001; Wieland, 2001].

SFDM: Another dependency-based model, which is based on either the ETFDM or the DFDM and involves a higher level of abstraction by removing the distinction between object locations and references [Stumptner *et al.*, 2001; Wieland, 2001].

Value-based models are models which make use of concrete execution values and propagate these values from the model's inputs to its outputs and (if possible) from the model's outputs to its inputs. A simple value-based model for the above example statement reads $\neg AB(S_i) \vdash x = a * b$, where x, a, and b stand for concrete variable values as computed at run-time. In the case of value-based models observations can be expressed by specifying the concrete value of a certain variable, e.g., $x = 6$ in the above example. The Jade system currently operates on the following two value-based model types:

VBM: A value-based model, which makes use of not only the underlying program dependencies, but also concrete evaluation values and the full programming language semantics [Mayer, 2000].

LF-VBM: A second value-based model, which is based on the unfolded source code for a particular program run [Mayer, 2001]. In particular, the loops are expanded into a set of nested conditional statements, where the conditional statements are modeled specially in order to provide better backward reasoning capabilities.

Although the expressiveness of the individual models is not exactly the same, all models support a considerable subset of the Java programming language. Currently, exception handling and programs using multiple threads are not supported. Furthermore, the value-based models do not support recursive method calls. The models are designed to locate functional faults, e.g. wrong operators or reversed conditions. They cannot reliably locate structural faults or more severe defects, such as wrong algorithms or data structures.

3 Results

In this section we describe results obtained by applying the models introduced above to a set of example programs and compare them with respect to their debugging and diagnostic accuracy. The tests were separated into two test sets, where one test set was used to compare the dependency-based models, whereas the other set was used to evaluate the value-based models. A comparison between the dependency-based models and the value-based models can be found in [Stumptner *et al.*, 2001]. Most of the example programs can be obtained from `http://www.dbai.tuwien.ac.at/proj/Jade/`.

3.1 Dependency-based models

The first test series aims at evaluating the performance of the used dependency-based models, i.e., DFDMs, ETFDMs, and SFDMs. Furthermore, we compare the results scored by these model types. In particular, the test series has two main goals: (1) to examine the ability of the Jade debugger to reduce the search space of bug candidates. In other words, we test which parts of a Java program can automatically be excluded from the fault localization process in a single diagnosis step and which parts of the search space remain for further debugging actions. (2) to evaluate the debugging performance of the Jade tool, i.e., determining the amount of user interaction needed to unambiguously locate a fault in a Java program.

In order to carry out these tests we implement a couple of test 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). We then construct test cases for each program P by specifying the correct input/output behavior of P and installing a single-fault into P. Overall 52 test cases are constructed and used for the evaluation of the system's performance. Table 1 shows all tests carried out with each row summarizing all tests performed in a single test series. Column $\#TC$ denotes the number of tests of the respective test series.

The diagnostic performance of the Jade system in the context of dependency-based models is given in columns 4 to 8 of Table 1. Column $\varnothing S_1$ shows the average number of top-level statements of the tested programs in a single test series. Since the Jade tool performs hierarchical debugging, only these top-level statements (this excludes statements nested in loops and selection statements) can be identified as the source of a fault in a single diagnosis step. Columns $\varnothing D_1$ and $\varnothing D_2$ present the number of top-level statements, which remain as possible fault candidates after a single diagnosis step has been performed using DFDMs and ETFDMs, re-

spectively. In other words, the difference between $\varnothing S_1$ and $\varnothing D_1$ ($\varnothing D_2$) shows the number of statements, which can be eliminated from the debugging scope in a single diagnosis step. Columns $\varnothing D_1(\%)$ and $\varnothing D_2(\%)$ show the number of remaining statements for both model types in relation to the total number of top-level statement, i.e., $\varnothing S_1$. These columns present the percentage of statements, which remain as possible fault candidates for further debugging actions. All tests are also performed with the simplified versions of the test programs' DFDMs. In cases where these tests yield results different from tests with the full DFDMs, the results obtained from the SFDMs are given in brackets. Note that no tests are carried out with simplified versions of ETFDMs, since these models are not yet fully supported by the Jade debugging tool.

The right side of Table 1 (columns 9 to 14) depicts the debugging performance of the Jade debugging environment. Since we are now interested in the exact localization of faults, we no longer deal with top-level statement only, but also take statements nested in loop and selection statements into consideration. Column $\varnothing S_2$ shows the average number of all statements of the respective tested program. Column $\varnothing R$ includes the average indices of those statements, in which the single fault has been installed during the test design phase. If we argue that with traditional debugging tools one has to step through the code manually statement by statement until the bug is located, the values in column $\varnothing R$ provide a reasonable reference value for the amount of user interaction needed by the Jade system to exactly locate a fault. The latter is presented in columns $\varnothing T_1$ (DFDMs) and $\varnothing T_2$ (ETFDMs). Columns $\varnothing T_1(\%)$ and $\varnothing T_2(\%)$ show the average number of user interaction relative to the average index of the buggy statement, i.e., $\varnothing T_1(\%) = \varnothing T_1/\varnothing R$ and $\varnothing T_2(\%) = \varnothing T_2/\varnothing R$.

3.2 Value-based models

In a second step we test the diagnostic performance of the more detailed and semantically stronger value-based models, i.e., VBMs and LF-VBMs. For this task we implement a second set of example programs which is designed especially to investigate the specific advantages and disadvantages of the value-based model variants. Whereas some examples are small and specifically designed to demonstrate different aspects of the models, most of the example programs implement well-known algorithms which could be part of larger programs. For example, programs executing a binary search procedure, computing the Huffman encoding of an array of characters, or applying Gauss elimination are part of this test suite. Similar to the tests carried out with the dependency-based models, faults were seeded into each program such that each test case is influenced by one fault. Again, 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 are 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 2. 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 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 is stopped because the seeded fault has been 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 cannot be located because the maximum time allowed for diagnosis is 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			LF-VBM				
		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
$\varnothing$	39	20	9	**65**	77	8	0.6	(1.6)	(**32**)

Table 2: Diagnosis results of the value-based models

4 Discussion

Based on the results from Section 3, in this section we discuss some important properties of the proposed models and present insights gained during the Jade project.

From the results it can be seen that the amount of code that has to be analyzed in order to locate a fault can be reduced significantly with all models. If we look at Table 1 we find that in the test series carried out with dependency-based models approximately 40% of the top-level statements can be eliminated from the debugging scope, leaving some 60% for further debugging actions. Interestingly, the average results obtained with different dependency-based model types were quite similar with slight advantages to ETFDMs (in comparison to DFDMs) and full model versions (in comparison to SFDMs). In the case of value-based models, the results lie in the same order of magnitude. In particular, between 40 and 80% of all statements have to be checked, with the average being at 65%. Note that this does not indicate a better performance of dependency-based models in comparison to value-based models, since completely different test programs were used to evaluate the different model types. In particular, the test series with the value-based variants in general used longer and more complex test methods. These methods result in only very few statements being removed from the suspect code in case of dependency-based models, but still yield remarkable results with VBMs. For a more detailed comparison of dependency-based and value-based models see [Stumptner *et al.*, 2001].

Dependency-based models One major advantage of dependency-based models is that they can be constructed and applied to actual diagnosis problems very quickly. This is also true for medium- to large-size programs. They are also easier to handle than their value-based counterparts, since they require observations only to state whether the value of a certain variable is correct or not, whereas with value-based models concrete execution values are needed. Generally, the use of ETFDMs results in fewer single diagnoses, because concrete execution traces are used during the collection of the dependencies. This becomes especially apparent for programs, which include loop and selection statements or recursive method calls. The improved debugging performance of ETFDMs in comparison to DFDMs comes with longer modeling times, since now the creation of a model not only depends on the underlying source code, but also on the existence of an execution trace, whose creation requires running the program. It was also shown that the full versions of DFDMs and ETFDMs are superior to their simplified counterparts. This is, because they model object locations and object references by separate model constructs and thus provide a finer-grained model architecture. On the other hand the computation of diagnoses with full model versions is computationally more expensive. Further on, the specification of observations is easier with simplified model versions.

The Value-Based Model However, dependency-based models did not prove to be an optimal solution for all tested programs due to their lack of run-time information. Note that even ETFDMs do not make use of concrete evaluation values directly, but only extract information about executed branches and numbers of iterations of loops from concrete execution traces. Therefore, the VBM was developed, which makes use of the full programming language semantics and propagates concrete evaluation values through the system. As already mentioned, in many cases VBMs score satisfying results with programs, which can hardly be diagnosed using dependency-based approaches only. [Stumptner *et al.*, 2001] indicates that in general value-based models are superior to their dependency-based counterparts. Therefore, although VBMs have the drawbacks of their high computational requirements, VBMs have proved as satisfying general-purpose alternatives and complements to dependency-based models.

Loop Handling A negative aspect of the dependency-based models and the VBM is the fact that these models provide good results for programs without loops but fail to compute satisfying diagnoses for programs that consist of large loop statements. This is due to the fact that loop statements are modeled hierarchically and discrimination between statements inside the loops is not possible. To overcome these problems, the LF-VBM expands loops into a set of nested conditional statements, with separate assumption variables for each statement. The number of conditional statements is derived from the initial execution of the test cases. Therefore, the model is able to reason about the statements inside the loop 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 2, switching from the VBM to the LF-VBM 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%[1] on average, which is quite low compared to the percentage of statements that was computed by the VBM. For the LF-VBM 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 greater than one, depending on the type of fault and the program structure. For most example programs the diagnosis cardinality required to locate a fault is less than or equal to two, which is usually computationally feasible when considering small- to medium-sized programs. Another aspect of the LF-VBM 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 to be executed from the evaluation of the selection condition. Therefore, faults in the condition cannot be located using the LF-VBM. Fortunately, such faults can in many cases be found with the VBM alone and do not require the LF-VBM to be applied.

In case of dependency-based models additional tests have been carried out to examine the overall debugging performance of the Jade tool. As Table 1 indicates, the average number of user interactions needed by the Jade tool is significantly smaller than the amount of user interactions needed by traditional debugging tools. On average some 40% of user interactions can be saved using the Jade tool. In general, the direct comparison of user interactions is problematic, since different user interactions require different types of inputs from the user, which vary in time, complexity, and knowledge

[1] 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.

needed by the user. The numbers given in Table 1 therefore include all user interaction performed by the Jade system. If only variable queries, i.e., the input of a new observation in the form of the value of a certain variable at a given source code position, are counted, the average amount of user interaction amounts to only 35% of the user interaction needed by traditional debugging tools. Since strictly speaking all other kinds of user interactions are not included in the reference value of traditional debuggers, this lower value probably provides a more accurate measurement of the debugging performance of the Jade system.

Comparison If we compare the results obtained with the Jade system to results obtained with other approaches for program analysis, it can be seen that the approaches described herein are comparable and in many cases even superior to other techniques. When comparing our approach to slicing [Weiser, 1984], we find that with dependency-based models we yield similar results to those obtained by slicing techniques. When value-based models are used, our results are much better, because for most of the example programs used during the evaluation of the value-based variants, static slicing is not able to eliminate any statement. This can be explained by the different levels of abstraction applied by dependency-based models and slicing on the one hand and value-based diagnosis techniques on the other hand. The value-based approach is somewhat closer to the actual execution semantics of the program than with both, program slicing and dependency-based models. Another improvement with respect to slicing is that we can provide more information to the user, if a loop has to be executed a different number of times to explain a fault. Those examples where no statements of the program can be eliminated are programs that are either very short (consisting of only an initialization statement and a loop) or 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).

5 Ongoing Work

Although the results presented in the previous section are already promising, there remain topics for further research. This section discusses possible enhancements of the models, to avoid some of the drawbacks mentioned in Section 4.

First, no single model is able to efficiently locate faults. Rather, a combination of models has to be applied to perform efficient reasoning. This multi-model-reasoning approach is not only applicable to a single level of abstraction, as in the case of the VBM and the LF-VBM, but can also be applied using multiple levels of abstraction or types of models. For example, the dependency-based models can be used to narrow the region of interest and then apply combinations of the VBM and the LF-VBM to exactly locate the fault. Also, models dealing with structural faults [Jackson, 1995; Wotawa, 2000] or various special-purpose models (e.g., to locate faults in loops, selection statements, etc...) 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 the models, as well as reduces the computational complexity of the diagnostic process, because 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 far as the fault classes which can be located with the Jade environment are concerned, it should already have become clear that we are interested in source code bugs which become observable as failures or output errors and manifest themselves as logical faults in the analyzed source code. This explicitly excludes compile-time and run-time failures as well as faults leading to the non-termination of a program. For a discussion about the fault classes handled by the Jade system we divide the class of analyzed faults into functional and structural faults. Functional faults are all faults, which result in a certain variable storing an incorrect value in at least one possible evaluation trace. In particular, these faults include the use of incorrect operators or the specification of incorrect literals, such as integer or boolean constants. Since these faults do not alter the structure of the program, faults belonging to this class can generally be found with the Jade debugging environment, once they become observable through a test case leading to an incorrect variable value.

Structural faults, on the other hand, are source code bugs which alter the structure of the underlying program. This is the case if the dependency graph [Ferrante *et al.*, 1987] of the program is not structurally equivalent to the dependency graph of the correct program. The result of these faults is that the system description, i.e., the model, differs from the system description obtained by the correct program. At the moment structural faults can only be located under certain circumstances. A discussion about different classes of structural faults and how they are handled by the Jade tool is given in [Wieland, 2001]. In the future special-purpose models have to be developed that handle different kinds of structural faults. As already discussed, these models then have to be combined with the general-purpose models described herein to increase not only the performance of the Jade debugger, but also the number of fault classes handled by the tool.

To aid the programmer in correcting a fault, an intelligent debugging environment should be able to provide possible corrections for a faulty part of a program. As described in [Stumptner and Wotawa, 1999], after a single diagnosis has been selected for further investigation, possible replacement expressions for the faulty expression can be inferred and presented as corrections.

Finally, intuitive means for specifying the expected behavior of a program have to be developed. This includes the construction of an intuitive graphical user-interface through which the user can easily switch between different levels of abstraction, test case specification, and other representations of the program (e.g., visualizations of heap structures, etc.).

6 Related Work

This section briefly summarizes related research in the area of program debugging and compares the approaches to our work.

Weiser's slicing approach [Weiser, 1984] is probably the most widely known approach to improve program debugging. His approach relies on the program dependencies and tries to eliminate those parts of a program that cannot contribute to an observed faulty program behavior. This approach is comparable to the dependency-based models presented here. Details on the relationship between these approaches can be found in [Wotawa, 2001].

Shapiro [Shapiro, 1983] introduces a theoretical framework for algorithmic program debugging and several algorithms suited to debug logic programs. However, the approach suffers from heavy user interaction, which is undesirable when debugging larger programs. In addition, the algorithms cannot locate faults inside procedures.

In [Console *et al.*, 1993] the application of model-based diagnosis to the software domain has been proposed for the first time. This paper introduces a way of using MBD by removing and adding Horn clauses to Prolog programs. Extensions of this approach were developed in [Bond, 1994].

Liver [Liver, 1994] discusses the use of a functional representation in the debugging of software to reduce the problem of structural faults in software, where statements are missing or superfluous parts of a program are the source of errors. The approach relies on symbolic execution of a functional specification, which has to be provided by the user.

Hunt [Hunt, 1998] applies the idea of MBD to the domain of object-oriented languages by building models for programs written in Smalltalk. The model used in this work is based on dependencies between instance variables and method calls that modify them. In contrast to our approach, [Hunt, 1998] is limited to single faults.

MBD concepts have also been applied to VLSI design languages, in particular VHDL [Friedrich *et al.*, 1999], using papers describe (abstract) models used for locating a concurrent statement, e.g., a VHDL process, responsible for a detected misbehavior. The Jade project builds on this work, but extends the previous approaches by modeling of object-oriented features.

Finally, Burnell and Horvitz [Burnell and Horvitz, 1995] present another approach to program debugging using probability measurements to guide diagnosis. As this approach relies on belief networks, which have to be initialized by domain experts, it is doubtable whether this approach can be applied to arbitrary programs.

7 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 summarize the work done during the Jade project and discuss some results obtained using the introduced model types. Besides the results, specific advantages and disadvantages of each of the models are discussed. Incorporating these models in a system with multi-model reasoning capability and ranking criteria for diagnoses holds the promise of wider applicability 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.

Acknowledgments

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

References

[Bond, 1994] Gregory W. Bond. *Logic Programs for Consistency-Based Diagnosis*. PhD thesis, Carleton University, Faculty of Engineering, Ottawa, Canada, 1994.

[Burnell and Horvitz, 1995] Lisa Burnell and Eric Horvitz. Structure and Chance: Melding Logic and Probability for Software Debugging. *Communications of the ACM*, 38(3):31 – 41, 1995.

[Console *et al.*, 1993] Luca Console, Gerhard Friedrich, and Daniele Theseider Dupré. Model-based diagnosis meets error diagnosis in logic programs. In *Proceedings* 13^{th} *International Joint Conf. on Artificial Intelligence*, pages 1494–1499, Chambery, August 1993.

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

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

[Hunt, 1998] John Hunt. Model-Based Software Diagnosis. *Applied Artificial Intelligence*, 12(4):289–308, 1998.

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

[Liver, 1994] Beat Liver. Modeling software systems for diagnosis. In *Proceedings of the Fifth International Workshop on Principles of Diagnosis*, pages 179–184, New Paltz, NY, October 1994.

[Mayer, 2000] Wolfgang Mayer. Modellbasierte Diagnose von Java-Programmen, Entwurf und Implementierung eines wertbasierten Modells. Master's thesis, Institut für Informationssysteme, Abteilung für Datenbanken und Artificial Intelligence, TU Wien, 2000. (only available in German).

[Mayer, 2001] Wolfgang Mayer. Evaluation of Value-Based Models for Java Debugging. Technical report, Technische Universität Wien, Institut für Informationssysteme 184/2, Paniglgasse 16, A-1040 Wien, Austria, 2001.

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

[Stumptner and Wotawa, 1999] 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.

[Stumptner *et al.*, 2001] Markus Stumptner, Dominik Wieland, and Franz Wotawa. Comparing Two Models for Software Debugging. In *Proceedings of the Joint German/Austrian Conference on Artificial Intelligence (KI)*, Vienna, Austria, 2001.

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

[Wieland, 2001] Dominik Wieland. *Model-Based Debugging of Java Programs Using Dependencies*. PhD thesis, Vienna University of Technology, Computer Science Department, Institute of Information Systems (184), Database and Artificial Intelligence Group (184/2), November 2001.

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

[Wotawa, 2001] Franz Wotawa. On the Relationship between Model-based Debugging and Programm Mutation. In *Proceedings of the Twelfth International Workshop on Principles of Diagnosis*, Sansicario, Italy, 2001.