Towards an Integrated Debugging Environment[1]

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

Abstract. With recent research showing that consistency based diagnosis can be used to model programs written in imperative programming languages for debugging purposes, it has been possible to develop debugging environments that provide interactive support to the developer, homing in on individual faults within a few interactions. In addition to complexity results, this paper discusses how the results of the straightforward application of diagnosis models can be improved upon by incorporating information obtained from multiple test cases (i.e., input/output vector specifications). The information from executing such test cases can be used to support heuristic statement selection by assigning fault probabilities, and for elimination of diagnosis candidates. We also discuss how the extended algorithms can be integrated with the consideration of multiple diagnosis models during the diagnosis process.

1 Introduction

For the last three years the JADE project has examined the applicability of model-based diagnosis 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 the 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 user-friendly GUIs. 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 steps, and input of additional observations by the user.

[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 of 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

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 following model classes are used by the JADE debugger:

ETFDM: A dependency-based model, which makes use of a concrete execution trace (see [10]).

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 (see [8, 10]).

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 locations and references (see [8, 10]).

VBM: A value-based model, which makes use of concrete evaluation values and the full programming language semantics. It computes possible fault scenarios not only by making use of the underlying program dependencies, but also by actually propagating concrete run-time values from the model's inputs to its outputs and (in some cases) from the model's outputs to its inputs (see [6]).

LF-VBM: A second value-based model, which is based on the unfolded source code for a particular program run (see [7]).

Empirical results have shown that in almost all tests the JADE debugger was able to substantially decrease the amount of suspect code in a single diagnosis step. This is true for all used models with each model having its individual strengths and weaknesses. Moreover, the JADE debugger was able to exactly locate source code faults in interactive debugging sessions and in most cases outperformed traditional debugging tools as far as the amount of user interaction was concerned. However, there still remain open issues, the most important of which are:

- The different model types have their individual strengths and weaknesses. For example, the dependency based models can be created and applied very quickly. However, in some cases the dependency structure of the underlying program is so complex that in each iteration only few statements can be removed from the debugging scope automatically. Value-based models, on the other hand, are much more detailed and in general score significantly better results than dependency-based models. However, these models in their current implementation are very slow and can thus be applied only to small Java programs. One goal of future research activities has therefore to be to effectively combine multiple models within a single debugging session.
- A common practice in hardware diagnosis is the ranking of diagnoses by probability, a practice that is based on the evaluation of failure rates of the individual components. Computing such estimates for software is significantly harder, because fault probabilities for particular program constructs are highly context-dependent. Most descriptions of errors found during debugging

are of limited scope and mainly anecdotal in structure [4]. The ability to rank diagnosis probabilities, even if estimated, could be used in different phases of the debugging process in order to improve the debugger's performance, e.g., to eliminate highly unlikely diagnoses, compute optimal measurement points, or select the best models to be used in a multi-model debugging session.
- The ultimate goal is to embed the JADE debugger within a complete software engineering environment which supports its user during all phases of the software development process. Synergies with techniques used in other areas of software development (testing in particular) can further improve the system's debugging performance.

This paper builds a framework for addressing the issues described above. We extend the normal diagnosis process to the creation of an integrated debugging tool, which (1) is part of a more general software development environment, (2) makes use of probabilities to rank diagnoses, and (3) supports the combination of multiple models in a single debugging session.

2 Debugging Complexity

We start with a brief discussion of the time and space complexity of debugging programs. Since this complexity depends on the type of model used, we distinguish the main two model classes used by the JADE debugger, i.e., dependency-based models (e.g., ETFDM, DFDM, SFDM) and value-based models (e.g., VBM, LF-VBM). With the value-based model, in the worst case all possible combinations of statements have to be checked for being a bug candidate, with the check time being the time for the program to run. We assume a fixed finite runtime for the program under consideration. This might be considerably reduced when working in an environment that allows restarting the program at some later point instead of at the beginning, but we do not consider this issue here.

Theorem 2.1 (Time complexity; value-based model) *Let* Π *be a program,* t *a test case, and* $T(\Pi, t)$ *the maximum time necessary to apply program* Π *to* t*. The worst-case time complexity for debugging is* $O(T(\Pi, t) \cdot 2^{size(\Pi)})$*. If we limit ourselves to single diagnoses, the worst-case time complexity is* $O(T(\Pi, t) \cdot size(\Pi))$*.*

When using a dependency-based model, the time complexity again depends on the number of possible combinations which is bound by $2^{size(\Pi)}$ for each program Π. Checking a combination, however, now does not depend on the maximum runtime. Instead, checking is performed by a propositional horn clause theorem prover, and the length of the logical sentence to be proved depends on the size of the model, whose complexity is given by $O(size(\Pi) \cdot |VS|)$. VS stands for the set of all variables of Π.

Theorem 2.2 (Time complexity; dependency-based model) *Let* Π *be a program,* t *a test case, and* VS *the set of all variables of* Π*. The worst-case time complexity for debugging is* $O(size(\Pi) \cdot |VS| \cdot 2^{size(\Pi)})$*. If the diagnosis size is bound to 1, the worst-case time complexity is* $O(size(\Pi)^2 \cdot |VS|)$*.*

As experimentation shows, in the average case diagnosis runtimes for the dependency-based model are significantly lower.

3 Fault Detection (Testing & Diagnosis)

The first step towards an integrated software development and debugging tool is to combine testing and debugging. By this we mean that a single integrated user interface must provide user-level functionality that allows to (1) specify test cases for a given program, (2) run these test cases, (3) automatically compute diagnoses in case of "unsuccessful" test cases, i.e., test cases exhibiting an incorrect program behavior, and (4) rank all diagnoses by their probabilities. This ranking can use information obtained during testing and is described in detail in this section. The advantages of such an approach are:

- As a prerequisite, each diagnosis process needs the specification of the expected input/output behavior for each test case. In our current implementation either the user enters it on screen, or it is read from a file containing ground facts in a simple assertion language. By combining testing and model-based debugging these observations can directly be taken from the test case specification.
- Multiple test cases can be used to compute diagnoses. This technique has been proposed in [9] (see also [10]). Again, coupling testing and debugging reduces the overhead of specifying and selecting these test cases.
- Testing and debugging can be performed using the same interface, including starting the diagnosis process automatically.
- The computed diagnoses can be ranked using information obtained directly from testing. This technique is described below.

Ranking based on multi-test case performance The results reported in, e.g., [3, 5] contained only test cases that failed to deliver the expected results. This means that diagnoses were computed for all test cases exhibiting incorrect program behavior, whereas all other test cases were simply ignored.

```
S1.  if (Y < 0)
S2.      X = 1;
     else
S3.      X = 1; // Should be X = -1
```

Figure 1. The first example program

However, successfully concluded test cases can also contain valuable information. The concept is demonstrated by the very simple program given in Figure 1 and the two test cases $(Y_0 = 1, X_4 = -1)$, $(Y_0 = -1, X_4 = 1)$. In the examples in this paper, we index variables based on their position in the code, with Y_i referring to the value of variable Y at position i, and $i = 0$ indicating the initial situation before executing the first statement in the program. We also generally talk of statements being correct or incorrect even when the program part in question is actually an expression. Thus, to describe Figure 1, we talk of line S_1 as a "statement", even though it really contains an expression that is part of an if statement reaching down to S_3. We consider statements individually even if nested (such as S_2 and S_3 in the example). Note also that the principles discussed in this paper apply as well to the debugging of smaller program fragments or individual methods, but for simplicity we use the term "program" throughout.

In general, given a set of test cases (given in terms of specifications of input and output values for a program), we can automatically compute the evaluation trace for every program Π and test case t. This evaluation trace comprises all statements and top-level expressions (such as the condition of the if statement in the example) that are executed during evaluation. For our small example we have two different evaluation traces. One is $\langle S_1, S_3 \rangle$ and the other is $\langle S_1, S_2 \rangle$. We see that in both cases the condition is contained in the trace. Statement S_2 is in the trace and produces a correct value, and statement S_3 is in the trace and produces an incorrect value. Hence, we conclude that S_2 cannot be a bug candidate, S_1 may contain the bug, but S_3 is most likely. We now formally represent this concept.

Given a program Π with a set of input variables $\mathcal{I}$ and output variables $\mathcal{O}$, and $dom(v)$ being the domain of each variable $v \in \mathcal{I} \cup \mathcal{O}$, we define a *test case* t to be a pair $\langle I_t, O_t \rangle$, where I_t (with domain $\mathcal{I}$) and O (with domain $\mathcal{O}$) are functions mapping variables to some value from their domain. I_t is total, but O_t need not be. The totality requirement on I_t guarantees that the program is executable on t. Now consider the set of statements of a program Π. For a set of test cases TS, for each statement S, we define the *correctness count*, c_S, to denote the number of successful test cases that caused S to be executed, and the *faulty count*, f_S, to denote the number of unsuccessful test cases. The counters imply a probability $p_C(S)$ and $p_F(S)$, respectively.

$$p_C(S) = \frac{c_S}{c_S + f_S} \qquad p_F(S) = \frac{f_S}{c_S + f_S}$$

Note that $p_C(S) + p_F(S) = 1$ holds. The probability estimates resulting from the example are:

S_i	c_{S_i}	f_{S_i}	$p_C(S_i)$	$p_F(S_i)$
S_1	1	1	0.5	0.5
S_2	1	0	1.0	0.0
S_3	0	1	0.0	1.0

We can make use of these probabilities to determine the most likely diagnosis as stated by the following lemma:

Lemma 3.1 *If the diagnosis procedure is complete with regard to the set of possible errors in* Π*, the probability* $p_F(S)$ *must be greater than 0, if statement* S *contains a bug that is detectable by the given test cases.*

From the above lemma we can directly obtain a more precise characterization.

Lemma 3.2 *If the diagnosis procedure is complete with regard to the set of possible errors in* Π *and* $p_C(S) = 1$ *(i.e.,* $p_F(S) = 0$*) holds for a statement* S*, then* S *cannot contain a bug that is detectable by the given test cases.*

The reverse of course does not hold, i.e., if a statement S has the probabilities $p_C(S) = 0$ and $p_F(S) = 1$, it still does not need to be buggy, although they do indicate that S is most likely to be the source of the faulty behavior. This holds especially if f_S is equal to the number of faulty test cases. The $p_C(S)$ and $p_F(S)$ values, however, neither guarantee finding a most likely diagnosis nor give an optimal ranking of competing diagnoses. In addition, the case where both probability values are greater than 0, neither correctness nor incorrectness of statements can be judged. Moreover, the probability values heavily depend on the test cases used. For example, consider the *Min* program depicted in Figure 2 which searches for the minimum *min* in an array of integers *X*. It can easily be shown that the program is correct and therefore passes the following test cases: $(X_0 = \{6\ 2\ 3\ 5\ 1\}, min_7 = 1)$, $(X_0 = \{1\ 5\ 2\ 3\ 6\}, min_7 = 1)$, and $(X_0 = \{2\ 1\ 3\ 5\ 6\}, min_7 = 1)$.

Now consider the following two variants *Min1* and *Min2*. *Min1* is equal to *Min* except statement S_1 was changed to `'min = X[2];'`. The program *Min2* is *Min* with statement S_5 replaced by `'min = x[1];'`.

Using the counter values leads to good results for program *Min2* but not for *Min1*. Figure 3 shows the probability values for the two variants *Min1* and *Min2*. The probability values of the statements in *Min1* give no clear indication about the cause of the misbehavior and are in fact misleading. The best choice would be to prefer the correct statement S_5 as a diagnosis because $p_F(S_5)$ is higher than the faulty count of all other statements. For program *Min2*, the faulty statement is identified correctly by the probabilities resulting from the test cases.

```
                // X is an 5-element array of integers.
S1.  int min = X[1];
S2.  int i = 2;
S3.  while (i<=5) {
S4.      if (X[i] < min) {
S5.          min = X[i]; }
S6.      i = i + 1; }
```

Figure 2. The program *Min*

S_i	Min1				Min2			
	c_{S_i}	f_{S_i}	$p_C(S_i)$	$p_F(S_i)$	c_{S_i}	f_{S_i}	$p_C(S_i)$	$p_F(S_i)$
S_1	2	1	0.67	0.33	1	2	0.33	0.67
S_2	2	1	0.67	0.33	1	2	0.33	0.67
S_3	2	1	0.67	0.33	1	2	0.33	0.67
S_4	2	1	0.67	0.33	1	2	0.33	0.67
S_5	1	1	0.50	0.50	0	2	0.00	1.00
S_6	2	1	0.67	0.33	1	2	0.33	0.67

Figure 3. Probability values of programs *Min1* and *Min2*

The above principles of reducing the number of diagnoses by using multiple test cases are incorporated in the following algorithm *MultTC*. *MultTC* is called with a set of subset-minimal diagnoses DS, the test cases TS, and the program Π as inputs, and returns an ordered set of diagnoses where impossible diagnoses are removed.

Algorithm $\mathbf{MultTC}(\mathbf{DS}, \mathbf{TS}, \mathbf{\Pi})$

1. Let D be the empty set.
2. Compute the probabilities $p_C(S_i)$ and $p_F(S_i)$ for each statement S_i, by executing the program on all test cases in TS.
3. Add each diagnosis $\Delta \in DS$ to D if there is no statement S_i with $p_C(S_i) = 1$ in Δ.
4. Order the diagnoses in D according to the $p_F(S_i)$ values of the contained statements S_i. We assign a p value to every $\Delta \in D$ where p is defined as $\kappa(S_1, ...S_n)$ with $\Delta = \{S_1, ..., S_n\}$.
5. Return D.

Step 3 of *MultTC* is needed to handle diagnoses that include a statement S_i that is never executed in an unsuccessful test case, i.e., $p_C(S_i) = 1$ holds. Since we do not have any evidence that S_i might be faulty with the given set of test cases TS, we eliminate S_i from the debugging scope. If we only consider minimal diagnoses, the elimination of S_i from a diagnosis Δ results in Δ being inconsistent with the used model. Therefore, we remove Δ from the set of diagnosis candidates. Step 3 might be superfluous for some models (e.g., value-based models), but is necessary if only a static analysis (e.g., DFDM) is performed. Step 4 of *MultTC* uses function $\kappa(S_1, ..., S_n)$ to compute the probability value of a diagnosis $\Delta = \{S_1, ...S_n\}$. If we assume $S_1, ...S_n \in \Delta$ to be independent we can use $\kappa(S_1, ..., S_n) = \prod_{S_i \in \Delta} p_F(S_i)$ as a good heuristic.

The time complexity of *MultTC* is $O(|TS| \cdot \max_{t \in TS} T(\Pi, t))$. However, in an integrated debugging environment the probability values can be collected during testing and automatically be used to rank diagnoses after an individual diagnosis step. Therefore, apart from the negligible overhead of computing the probability values of the diagnoses, no extra computations have to be performed. Of course, the computed probabilities depend on the test cases used. Note that TS is defined as a set of test cases, i.e., each test case can

only occur once in it (as it would distort the resulting probabilities otherwise). We also assume that path testing techniques (see [1]) are used, which guarantee that all branches are executed during testing roughly equally often.

Variable-based weighting In order to improve the ordering of statements and diagnoses we make use of the knowledge about correctly computed variables. If a variable v is correct after executing the program, statements involved in the computation of v can be seen as less likely to be responsible for a misbehavior. These statements, however, cannot be excluded from the list of suspicious statements. The set of statements influencing the value of v can be computed using a functional dependency model and the observation $\neg ok(v_k)$, where k is the highest index of variable v. The value of variables before executing the program are assumed to be correct. The model together with the observation can be used to determine all single diagnoses which are equivalent to the statements influencing the value of v. The following algorithm *AdaptP* makes use of the knowledge about correctness of values and adapts the probabilities of the given diagnoses DS. Other parameters of *AdaptP* are the program Π, the test cases TS, and the set of all variables of Π, i.e., VS.

Algorithm $\mathbf{AdaptP}(\mathbf{DS}, \mathbf{TS}, \mathbf{VS}, \mathbf{\Pi})$

1. Let A be the set of all variables, i.e., $A = VS$.
2. For every test case $t \in TS$ do the following:
   (a) If the program Π does not pass the test case t, compute the set of variables B that are correctly computed.
   (b) Set A to $A \cap B$.
3. For all $v \in A$ do the following:
   (a) Use a dependency model together with the observation $\neg ok(v_k)$ to determine the set Σ of all statements that have an influence on the value of v.
   (b) If there exists a statement $s \in \Sigma$ such that $s \in \Delta$ for some $\Delta \in DS$, then the new fault probability is $p(\Delta) = \xi \cdot p'(\Delta)$. $p'(\Delta)$ denotes the current fault probability of Δ.
4. Return DS.

In the computation above, $\xi \in [0, 1]$ is a constant value that determines the factor by which the probability of a diagnosis Δ is devaluated, if Δ contains a statement influencing a variable which is correctly computed in all test cases. In general, ξ has to be determined empirically. In the following we work with $\xi = 0.5$. The time complexity of the adaption is bound by $O(|TS| \cdot \max_{t \in TS} T(\Pi, t) + size(\Pi)^2 \cdot |VS|)$, where the term $size(\Pi)^2 \cdot |VS|$ is due to the fact that we are interested in all single diagnoses explaining the observation $\neg ok(v_k)$. We illustrate *AdaptP* using the *Min1* and *Min2* programs where the initial sets of diagnoses comprise all statements and the adaption parameter ξ is set to 0.5. For both programs and all test cases variable *i* is correctly computed. The value of *i* is determined by the statements S_2, S_3, and S_6. Whereas the probability values of the other statements remain the same, the fault probabilities of statements S_2, S_3, and S_6 are $0.5 \cdot 0.33 = 0.17$ for *Min1* and $0.5 \cdot 0.67 = 0.33$ for *Min2*, respectively. When ordering the statements according to their probability values, for *Min1* and *Min2* we receive the same sequence, i.e., $\langle S_5, S_1, S_4, S_2, S_3, S_6 \rangle$. In both cases the correct diagnosis is ranked first or second.

In an integrated testing and debugging environment the algorithms $MultTC$ and $AdaptP$ do not cause significant overhead. Most of the required information can be obtained during testing of Π, and no extra program executions are required. The additional overhead is of the same order as computing the diagnoses using dependency-based models. Using this ranking not only means that the user is given an intuitive feeling of the likelihood of individual fault explanations, indicating which diagnoses should be evaluated first (e.g., by code inspection or creation of further test cases), but the ranking diagnoses can also be used to increase the overall performance of the interactive debugging tool, as discussed in detail in Section 4.

4 Fault Localization

We are now at a point in the debugging process where all diagnoses for the set of specified test cases have been computed and ranked by their probability values as defined in Section 3. Generally, at this stage the number of diagnoses is still too large to immediately exactly locate given faults or evaluate these diagnoses by manual code inspection. This section describes various ways of reducing the number of diagnoses with the ultimate goal of eventually pointing the user at the exact fault location(s).

Additional test cases One way to reduce the number of diagnoses is to specify additional test cases. This can be done very efficiently in an integrated debugging environment as described in Section 3, since new test cases can automatically be executed and divided into unsuccessful and successful test cases. All unsuccessful test cases can then be used to compute additional conflict sets and thus reduce the current number of diagnoses using standard model-based diagnosis techniques. Furthermore, the set of all test cases can be used to adapt the probability values of all diagnoses using the algorithms given in Section 3. However, this approach can only be expected to work as long as test cases can be found, which provide additional information, e.g., because they exhibit additional failures or execute additional paths.

Additional observations Another way to decrease the current number of diagnoses is to specify additional observations of the behavior of the debugged program. In contrast to the specification of additional test cases, here we are talking about observations inside the analyzed program, i.e., values of local variables, etc... Whereas new observations result in the (possible) elimination of some of the current diagnoses, the probability values of the remaining diagnoses are not affected. Currently, the JADE debugger uses a measurement selection algorithm to determine the location in the program, where an observation should be specified in order to eliminate a maximum amount of diagnoses. This algorithm is a simplification of the algorithm proposed in [2] and is based on the computation of entropy values (see [10]). However, this measurement selection algorithm assumes that all diagnoses are equally likely. By using the probability values defined in Section 3 more efficient measurement selection algorithms can be designed speeding up the overall debugging time.

Alternative models Due to the generality of the definitions and algorithms of model-based diagnosis, the approaches proposed herein are not limited to the two modeling paradigms of dependency-based models and value-based models that are currently represented in the JADE environment. In principle there exists an infinite model space, which contains not only general-purpose models, but also special-purpose models, such as models designed for the debugging of loops, arrays, or other data type specific errors. In order to further decrease the number of diagnoses, multiple models can be used within a single debugging session. This can be achieved in the following way:

- Start debugging with a general-purpose model with a high level of abstraction, e.g., a dependency-based model.
- Use more complex and detailed models, e.g., value-based models, to (successively) reduce the number of diagnosis candidates.

- Once the search space of bug candidates has been narrowed sufficiently, use special-purpose models to exactly locate a fault.

This approach has the advantage that diagnoses can be computed faster than with the more detailed models alone. We combine two models by using the results of the more general approach as focus set for the more detailed approach. Only in the case where the more abstract model is not capable of reducing the number of statements does the proposed algorithm not help to decrease runtime. In the following we show how to combine dependency-based and value-based models to speed up the overall debugging time. Note that the same approaches can be used to combine any two models suitable for debugging.

The following algorithm $locate$ computes a set of candidates for the given program Π and the set of test cases TS. We assume that all minimal diagnoses are computed up to a pre-specified diagnosis size ds. Moreover, we assume that the value-based model SD_{VB} only uses knowledge about the correct behavior of statements. Diagnosis is performed at the statement level, i.e., only statements are diagnosis components. These assumptions do not restrict the generality of our approach.

Algorithm locate($\mathbf{TS}, \mathbf{\Pi}$)

1. Use the given test cases TS and the dependency-based model SD_{FD} and compute the diagnosis set DS.
2. Reduce DS by applying the $MultTC$ and $AdaptP$ algorithms. Use a minimum probability value to cut away unlikely diagnosis candidates. Let DS' be the resulting set of diagnoses. If DS' is empty, decrease the minimum probability value until DS' is no longer empty.
3. Compute the focus set $FS = \bigcup_{\Delta \in DS'} \Delta$.
4. Use the value-based model SD_{VB} of Π, the test cases, and the given focus set FS to compute a set of diagnoses DS^*, i.e., for each $\Delta^* \in DS^*$, $\Delta^* \subseteq FS$.
5. Return DS^* as result.

The worst-case time complexity of $locate$ is of the same order as the time complexity of diagnosis using the value-based model. Due to the effects of the focus set limiting the overall search space the overall runtime should be decreased, although this cannot be guaranteed in every case. What we know is that $\bigcup_{\Delta* \in DS*} \subseteq \bigcup_{\Delta \in DS'}$ must hold. For example, a statement that only occurs in one multiple fault diagnosis in DS' could turn out to have disappeared from DS^*, leading to correct diagnoses being omitted. For example, consider the program *Circle* given in Figure 4 which computes area a and circumference c of a circle. Obviously, the program computes a wrong value for variable c. If we use the dependency-based model, we obtain statements S_1 and S_3 as bug candidates, which are used as focus set by $locate$. If we use the test case $(d_0 = 2, a_4 = 3.14, c_4 = 6.28)$, we see that statement S_1 can no longer be a single diagnosis using the value-based model. Even though statement S_1 no longer tells us the correct value of r, from the test case and the correctness assumption of statement S_2 we get $r = 1$. Using the correctness of statement S_3 and $r = 1$ we finally obtain $c = 3.14$ which contradicts the expected value of 6.28. Hence, for the given example we have only one diagnosis remaining in the focus set, which pins the blame on statement S_3. In fact there is another valid diagnosis using the value-based model, which would indicate two faults, in statements S_1 and S_2 (in statement S_1, because the assumption is that the value of r is incorrect, and accordingly in statement S_2 because S_2 computes a correct value for a out of an incorrect value for r). Note that this diagnosis will not be computed using $locate$ because statement S_2 is not in the focus.

```
S1. int r = d/2;
S2. int a = r*r*3.14;
S3. int c = r*3.14; // BUG! should be c = d*3.14;
```

Figure 4. The program *Circle*

As already discussed, special-purpose models can be used, once the search space of bug candidates has been limited enough to have clear expectations of a fault's nature. General-purpose and special purpose models can be combined exactly as described above. The more detailed and specialized the used models get, the harder it becomes to select the right model. However, if we make use of probability values of diagnoses, this task can be made much easier. For example, if we know that loop statements are part of the remaining diagnoses with the highest probability values, then a model designed especially for the debugging of loops could help to separate correct diagnoses from incorrect ones.

5 Conclusion

The goal of this paper has been to extend the by now well-established basis of model-based debugging by adapting a number of model-based diagnosis techniques to the debugging domain, namely the integration of multiple test cases, heuristics for ranking diagnoses based on these test cases, and the integration of multiple models in the diagnosis process. We have presented a set of algorithms that incorporate these extensions in the work done in [9, 3]. By moving beyond the consideration of individual diagnosis runs and considering the work situation of an actual developer working with an integrated software development environment, more information can be brought to bear on the debugging process and be used to gain better diagnosis performance and discrimination, while using the same diagnosis reasoning engine underneath.

REFERENCES

[1] Boris Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, 1990.

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

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

[4] D. Knuth. The Errors of TeX. *Software - Practice and Experience*, 19(7):607–686, July 1989.

[5] 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.

[6] 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).

[7] 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.

[8] 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.

[9] 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.

[10] 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.