DEBUGGING PROGRAM EXCEPTIONS

Wolfgang Mayer* **Markus Stumptner*** **Franz Wotawa****

** University of South Australia,*
Advanced Computing Research Centre
5095 Mawson Lakes SA, Adelaide, Australia
{mayer,mst}@cs.unisa.edu.au

*** Technische Universität Graz*
Institute for Software Technology
8010 Graz, Inffeldgasse 16b/2, Austria
wotawa@ist.tugraz.at

Abstract: Even with modern software development methodologies, the actual debugging of source code, i.e., location and identification of errors in the program when errant behavior is encountered during testing, remains a crucial part of software development. To apply model-based diagnosis techniques, which have long been state of the art in hardware diagnosis, for automatic debugging, a model of a given program must be automatically created from the source code. This work describes a model that reflects the execution semantics of the Java language, including exceptions and unstructured control flow, thereby providing unprecedented scope in the application of model-based diagnosis to programs. Notably, this approach omits the strict view of a component representing one statement of earlier work and provides a more flexible mapping from code to model.

1. INTRODUCTION

Debugging, i.e., detecting a faulty behavior within a program, locating the cause of the fault, and fixing the fault by means of changing the program, continues to be a crucial and challenging task in software development. Many papers have been published so far in the domain of finding faults in software, e.g., testing or formal verification (Clarke *et al.*, 1994), and locating them, e.g., program slicing (Weiser, 1984) and automatic program debugging (Lloyd, 1987). More recently model-based diagnosis (Reiter, 1987) has been used for locating faults in software (Console *et al.*, 1993; Mayer *et al.*, 2002*c*).

This paper extends prior research on model-based diagnosis for locating bugs in programs written in mainstream programming languages (e.g. Java). The idea behind the model-based debugging approach is (1) to automatically 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 the candidates back to their corresponding locations within the original program. Formally, given a set of test cases on which the program is run, a (minimal) diagnosis is defined as a (minimal) set of incorrectness assumptions $AB(C)$ on a subset $C \in \Delta$ of components $COMP$ in the program (usually statements) such that $\{AB(C) | C \in \Delta\} \cup \{\neg AB(C) | C \in COMP \setminus \Delta\} \cup SD$ is consistent (Reiter, 1987). Here, SD is a logical theory describing the program's behavior under the assumption that components work correctly. Since the computation depends on observations in terms of test case output, unlike formal verification approaches, no separate formal specification is necessary - everything but the test cases is computed automatically from the source code. Conversely, where verification model checkers produce counterexamples, the outcome of the diagnosis process are code locations. Model-based debugging thus complements, rather than replaces verification techniques.

The models presented in (Mateis *et al.*, 2000; Mayer *et al.*, 2002*a*) have successfully been applied in the Jade project to debug Java programs. The tests consist of small to medium sized Java programs together with their faulty variants and given test cases. A comparison of the models and their effectiveness relative to each other as well as compared to a normal interactive debugger was given in (Mayer *et al.*, 2002*c*). Whereas these methods have been applied to large scale programs in concurrent languages (Stumptner and Wotawa, 2000), so far, one important factor that has held up experimentation with "off the shelf" example programs in the Jade project has been the omission of certain frequently used language constructs from the models, most notably exception handling. The hierarchical structure of the models renders them unsuitable for diagnosing programs which do not conform to the assumption of a "structured" control flow based purely on loop and if statements, This excludes a wide array of language features which includes structured exception handling constructs, recursive method calls, return statements, and jump statements. This paper addresses that issue and presents a different modeling approach, where the restrictions on program structure are relaxed in order to overcome the limitations of previous models. As a re-

sult it represents a significant step towards a model of that language that can cover arbitrary Java programs.

The paper is organized as follows: In Section 2 we describe the construction of the extended diagnostic model. Sections 3 and 4 describe the behavior of the model components, with the latter focusing on control structures. Finally, we discuss the capabilities of the model.

2. MODELING EXCEPTIONS

The new modeling algorithm is based on building a modular, multi-directional version of the classical control flow graph (CFG) (Ferrante *et al.*, 1987), which represents an approximation of all possible execution paths. The nodes of this CFG represent *basic blocks* (i.e. a sequence of operations that always executes sequentially without branching), and the arcs represent the control flow between these blocks. In a following step, the CFG is enhanced by data flow information. The basic blocks in each method are then transformed into a component network. The components are grouped in two categories that represent the semantics of the language elements (e.g. the computational statements and expressions) on one hand and flow control restrictions on the other hand (e.g., turning off alternate execution paths to the one actually taken). The model variables are likewise partitioned, one group representing the actual program variables accessed and modified by the code, the other group representing the control flow (i.e., the selection of which statements are executed). As a result, control and data flow of a program can be reasoned about in a uniform framework. The representation of the program execution as a constraint system provides means for computing values in both directions: forward and backward. This is essential for applying model-based debugging approaches, which rely on an effective theorem prover generating small conflict sets.

The automatic transformation from the Java source code to the model used for diagnosis proceeds in three steps. For space reasons we only give a brief overview, full references to the algorithms and literature used are given in (Mayer *et al.*, 2002*b*).

Build CFG. The program is transformed into its CFG representation. Each method is represented as a directed graph with basic blocks, entry and exit points as nodes and all possible transitions between nodes as arcs.

Compute Data-flow Information. In this step the CFG is enhanced with additional data flow information, resulting in what is known as the *Static Single Information Form (SSI form)* (Ananian, 1999; Singer, 2002), an extension of the classic Static Single Assignment (SSA) form. Essentially, a program is in SSA form if every variable in the program is assigned exactly once and every use is reached by exactly one assignment. This is guaranteed by renaming the variables in the original program and introducing data flow ϕ functions at the beginning of each basic block that is reached via multiple paths in the control flow graph. To enable backward analysis, the SSI form adds complementary σ functions. The transformation algorithm (Ananian, 1999) traverses the Program Structure Tree (the hierarchy of single-entry-single-exit regions in the program) bottom-up and inserts σ and ϕ functions at every control flow branch (resp. join) for every variable used (resp. modified) in a subregion. This algorithm also computes the control dependence regions (Johnson *et al.*, 1993). A control dependence region is formed by all statements having the same control dependencies. [1]

As the above algorithm only uses syntactic names to compute the set of used and modified variables for each statement, heap data structures require special treatment. We follow (Mayer *et al.*, 2002*c*) and partition the heap into disjoint structures, each representing the values of a particular instance variable of a class. This allows to build a model for programs accessing heap data structures, even if the objects that are accesses through references are not known statically.

```
1  class Fac {
2    static int f(int x) throws OutOfRangeException {
3      if (x < 0) throw new OutOfRangeException();
4      else if (x == 0) return 1;
5      else return x * f( x - 1 );
6    }
7    static boolean exception = false;
8    static int main() {
9      int fac = 0;
10     try {
11       fac = f(3);
12     } catch(OutOfRangeException ex) {
13       exception = true;
14     }
15     return fac; } }
```

Fig. 1. Example Program

Transform CFG into Component Representation. For each primitive element in the model (e.g. variable access, assignments, operators, method calls, etc.) a component is then generated whose behavior (specified as a logic sentence that captures the Java semantics of the element) simulates the effects of that element in both forward and backward direction. For every assignment to a variable, a new model variable is generated and is subsequently used up to the point in the code where the variable is reassigned. A special placeholder component is inserted for a method call and loop statement that summarizes the effects of the method or loop. Similar to the CFG construction, these components have output connections for each exit node of the called method's (or loop's) model.

To obtain a model of the whole method, the models of the basic blocks are composed according to the control and data flow information computed previously, then outside variables accessed inside the method are connected to the entry vertex, and modified variables to the exit vertices.

[1] A node a is control dependent on node b if there is a path from b to a where a postdominates all nodes other than b on this path and a does not postdominate b (if $a \neq b$). (Johnson *et al.*, 1993)

The modeling of the control flow proceeds as follows. For each control dependence region a constraint variable is created that represents the control flow for the region. The domain of control variables is defined as $\{\top, \perp, ?\}$, with $\top$ meaning the region is definitely executed for the test case, $\perp$ representing the fact that a region is never executed, and for $?$ it is unknown whether the region is executed. Control variables are part of the precondition of the behavioral description of every component representing a statement that is contained in a block and can therefore be used to remove paths which are conflicting with a test case from the model simply by setting the associated control variable to $\perp$. The values of the control variables are determined by (1) statements which represent branches in the CFG, and (2) liveness components. To allow for case (1), the control variables for every successor region of a branch statement are set by the behavioral description of branch statements. For (2), liveness components ensure required properties of valid execution traces through the control flow graph are maintained during model execution. In particular, it must be satisfied that a region can be entered iff an exit transition from the region is possible.

The modeling of data flow largely proceeds as described in (Mayer *et al.*, 2002*c*). Variables used in a statement s are added to the input connections of the behavioral description of s, and variables assigned by the statement are added as output connections. A connection is used as input for other components until a new output connection is created by a subsequent assignment statement.

Data flow ϕ and σ functions are modeled by additional components that are not associated with any region, and, therefore, cannot be deactivated. The behavioral description is roughly equivalent to the intuitive behavior of the ϕ and σ functions: for ϕ functions, the values $data_i$ of the incoming paths i are combined into a new value for the output connection $data_u$ of the component. For σ functions, the behavior is reversed to duplicate an incoming connection for every successor region. Formal descriptions of the behavior of these components are presented in Section 3.

To combine the reasoning about data and control flow, each of the data flow variables v of a control region r is associated with r's control variable. [2] In case conflicting values are derived for a variable, the associated control variable is set to $\perp$ to indicate that the corresponding path in the model is invalid.

Example. To illustrate the modeling process, reconsider the program from Fig. 1. The model of method main is depicted in Fig. 3.

Control flow at the entry point of the method is represented by control variable a. The call of method f() splits the control flow in two separate paths. Therefore, σ components are inserted for the variables fac

[2] Variables associated with σ functions are assumed to be contained in the corresponding control region of their associated branch.

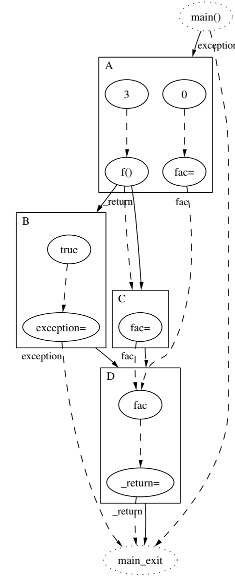

Fig. 2. Dependency graph of method main(). Solid lines represent control dependencies, dashed lines data dependencies (which are labeled with the variable they represent).

and exception. [3] The output variables for the components representing fac is associated with the control variable of Block B. For exception, the variable is associated with Block C's control variable.

The path represented by variable b represents the transition to the exception handler block. The normal return from the method is represented by c. As exactly one of these paths has to be followed if Block A can be reached, the liveness constraint live enforces this condition. The model of Block D requires variable fac and the control flow as input connections. As there are multiple incoming arcs for both dependencies in the CFG, ϕ components have to be inserted for fac and exception (as they are modified in the subregions B and C). The components for the variables are connected to the two preceding components providing the values of the variable. Further, the output of the ϕ component is connected to Block D, which requires the value of fac.

3. BASIC EXECUTION SEMANTICS

This section describes the behavior of the model components. For brevity, only essential components and differences to (Mayer *et al.*, 2002*c*; Mateis *et al.*, 2000) are presented.

First, the bijection between program statements and model components used in the earlier models is abandoned. This makes it possible to represent a single

[3] Note that for both the σ components the value of the represented variable is not used in the non-exceptional branch and therefore the variables representing these branches are omitted. Nevertheless, the components may not be removed from the model, as they form a propagation barrier for the backward reasoning algorithm in case it is unknown which path of the branch is followed and the values are different for any two paths.

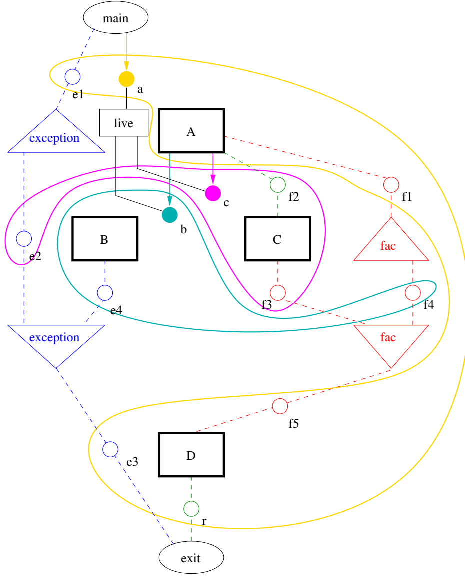

Fig. 3. Model of Method main(). For clarity, blocks are drawn without embedded components. Model variables are drawn as circles, control variables are drawn filled. σ and ϕ components are represented by triangles. The set of blocks and variables associated with a control variable are drawn as hyper-edges containing the control variable.

statement as multiple components and allows fine grained control over the model's assumptions, spanning multiple levels in the program structure. The assumptions for each component are grouped into a hierarchy that resembles the program structure. This provides means for making assumptions about single statements, blocks of statements, and hierarchically nested statements. In particular, the representation allows to distinguish the calling of a wrong method from a faulty implementation of the called method. Also, this representation is expected to reduce the complexity of the diagnostic process, as program structure can be exploited to speedup diagnosing (the approach is inspired by (Felfernig *et al.*, 1999)).

A description of the most important model components follows (C represents a component and $\neg AB(C)$ denotes that the C is assumed to behave correctly). The antecedent of every implication [4] implicitly contains an atom of the form $control(C) \neq \bot$, representing the dependency on the control variable ($control$ denotes the control variable of the control region containing component C). For brevity, this conjunct is omitted in the following description.

Instance Variable Access are modeled by components that restrict the value of the input heap partition to the required object instance (passed as input parameter) and provide the result as output value.

$$\neg AB(C) \wedge access(C) \wedge object(C) \neq null \\ \Rightarrow out(C) = in(C)|_{object(C)}$$

Instance Variable Assignments are modeled similarly to variable access components, with the value of the right hand side passed in as input parameter. The output heap partition is computed by replacing the value of the object in the input partition by the passed-in value.

$$\neg AB(C) \wedge assign(C) \wedge object(C) \neq null \\ \Rightarrow out(C) = in(C)[object(C) \mapsto value(C)]$$

4. MODELING CONTROL CONSTRUCTS

In this section we examine the more complex control constructs: while loops, method calls, ϕ and σ data flow components introduced to achieve SSI form, and finally liveness constraints.

While Loops are modeled as hierarchical components, where the loop component contains separate sub-components for the loop's condition and the body. If the condition evaluates to $\bot$, the loop component is removed from the model and its output connections are replaced with its input connections:

$$\neg AB(C) \wedge while(C) \wedge cond(C) = \bot \\ \Rightarrow \forall_v \left(out_v(C) = in_v(C)\right)$$

In case $cond$ evaluates to $\top$, the component is expanded and copies of the models of the body and the condition are inserted before the component (in this order). Therefore, this behavior is not expressed as a logical formula, but as a meta-rule to expand iterations of the loop and modify the model. This rule also has to take into account that loops might not terminate for particular input configurations. As our systems assumes termination of the program, this constitutes a conflict and the branch leading to this configuration has to be removed by setting the control variable containing the loop statement to $\bot$. To detect nontermination, the system currently assumes an (user-specified) upper bound on the number of iterations each loop may execute. In the future, this could be enhanced with more advanced analysis, such as Abstract Interpretation (Cousot and Cousot, 1977) based techniques or induction variable detection (Gerlek *et al.*, 1995).

Note that the behavior of the component is undefined for the case where the loop condition does not evaluate to $\top$ or $\bot$. In this case, we approximate the behavior of the loop by an abstract collecting semantics, similar to the approach presented in (Bourdoncle, 1993). This approach allows to exploit the values derived for the input and output values of the loop to obtain enhanced accuracy compared to solely static analysis frameworks. This is possible as the values for the environment before and after the execution of the loop are derived by a concrete program execution, which may result in more precise values than static analysis which abstracts from the concrete test case information. Also, the rule-guided model modification allows for determining the amount of context that is considered in the analysis to vary depending on the actual values computed during model execution. For example, consider the following program (the environment before the loop is denoted in angled brackets):

[4] Only for component representing statements of the program to be analyzed.

```
⟨a ← [−5, −3, 2, 0, 1], i ← 0, s ← 0⟩
while(i<5) {
  s = s + a[i]
  i = i + 1}
```

Using simple interval abstraction (Cousot and Cousot, 1977), static analysis cannot determine that $s \in [-8..\infty[$ after the loop. This is due to the abstraction of $i \in [0..4]$ and $a[0..4] \in [-5..1]$ in the last iteration of the fixpoint computation. Without further knowledge about the number of iterations, a lower bound for s cannot be established. However, if the first two iterations are considered separately (i.e. our model decides to expand the loop twice, after which the termination approximation takes effect), static analysis of the loop (after abstracting from the environment after the second iteration) can prove the lower bound for s. Therefore, the development of suitable heuristics to determine when to expand loops and method calls can lead to improvements compared to conventional static analysis.

Method Calls. Method calls are modeled similarly to loop statements, wit multiple alternate subcomponents for polymorphic calls).

Data-flow components. Each component dealing with n input or output paths has connections $data_i$ (each associated with a control variable $control_i$ ($i \in [1..n]$)), which represent the data variables for each of the paths. Further, the variable $data_u$ (associated with $control_u$) represents the unified connections of $data_i$. The behavior of these components can be formalized as follows (assume each of the $data$ connections holds a heap partition):

Let $Active$ denote the subset of indices i where $control_i(C) \neq \bot$. Then

$$dffunc(C) \Rightarrow \forall_i\, (data_u(C) \subseteq data_i(C) \vee control_i(C) = \bot)$$

$$dffunc(C) \Rightarrow data_u(C) \supseteq \bigcap_{i \in Active} data_i(C)$$

The first equation states that all the values of the unified connection must be supplied by every data connection that can possibly be executed. The second condition covers the case where values are propagated in the other direction: in this case the unified connection must be consistent with the intersection of all other data connections.

Liveness components ensure that if a statement (except the last statement of the program) is reachable then exactly one of its successors is executed. The component has an input variable $control_i$ for each of the statement's successor paths and is also connected to the surrounding region's liveness variable $control_r$.

$$liveness(C) \Rightarrow \Big(control_r(C) = \bigvee_i control_i(C)\Big) \wedge \Big(\bigwedge_{i \neq j} \neg control_i(C) \vee \neg control_j(C)\Big)$$

5. EXAMPLE AND DISCUSSION

To demonstrate them model's ability to effectively locate faults, reconsider the program in Fig. 1 together with a test case that specifies the value true for variable exception and 0 for the result of main(). Statements are identified by their line numbers. As a first step, the values provided by the test case are inserted into the model: e3 = true, [5] r = 0, and a = $\top$ (the entry and exit point of the method are reachable). Further, at the beginning of the diagnostic process it is assumed that all components are correct. By propagating values through the model, e4 = true and e1 = false are derived. For e2, both values are derived, which results in a conflict. Therefore, the corresponding branch of the model is eliminated by adding c = $\bot$ for the associated control variable c. a = $\top$ together with $\neg AB(\boxed{11})$ [6] derive b = $\bot$, which violates the liveness constraint (because of c = $\bot$). Therefore, the model is found inconsistent and the conflict $\{\neg AB(\boxed{7}), \neg AB(\boxed{11})\}$ is returned. To break the conflict, either $\{AB(\boxed{7})\}$ or $\{AB(\boxed{11})\}$ has to be valid. Assuming $\{AB(\boxed{7})\}$, true is derived for e2–e4 without causing a conflict. r = 0, a = $\top$, and $\neg AB(\boxed{15})$ derive 0 for f3–f5. On the other hand, a = $\top$ and $\neg AB(\boxed{11})$ derive c = $\top$ and 6 for f2 and f3, resulting in a contradiction for f3 (with the value 0 derived in the previous inference). Hence, $\boxed{7}$ cannot be a diagnosis and cannot be responsible for the misbehavior. For $\{AB(\boxed{11})\}$ values for e1–e3 are derived as in the first computation. However, b = $\bot$ cannot be derived and therefore the model is consistent. As a result, $\{AB(\boxed{11})\}$ is the only single fault diagnosis.

We close with an overview over the impact of some modeling decisions on the diagnostic process.

Conflicts. The model derives an inconsistency if and only if there is no possible and consistent path between the model's start node and its end node. In this case, to derive conflicts, the propagation algorithm must keep track of which components derive which values. Alternatively, approaches similar to (Junker, 2001) can be applied.

For efficiency, the component system propagates only deltas of values, so that execution in the next propagation cycle can be focused on the newly added values. Also, method calls and loops are expanded to full models only when reasonable input values are present at their inputs (e.g. only if the receiver is known).

Faults. The model proposed herein is able to locate general functional faults, where parts of a program are faulty without assigning to wrong variables. This also includes the subclass of structural faults where a wrong variable is *read*. Faults can be detected not only in the top level method, but also in hierarchically embedded program elements. However, the model assumes the control flow graph to be fixed and therefore faults causing changes in the CFG cannot be located reliably. For example, changing the type of the excep-

[5] For brevity, different versions of the variables exception, fac, and return are abbreviated as e, f, and r, respectively, and are annotated with a number to ensure uniqueness.

[6] Statements are denoted as boxes containing the statement's line number.

tion of a throw statement may cause a changed control flow, as a different catch block may be responsible for the new type.

Model Size. The size of the CFG constructed for a method is linear in the method's size (plus some smaller overhead proportional to the number of exceptions thrown and exception handlers) (Sinha and Harrold, 2000). The worst-case complexity of the SSI form is quadratic in the CFG size. However, it was shown that the SSI form is almost linear in the CFG's size for most programs (Ananian, 1999). As the model is derived from the SSI form, its size is also nearly linear in practice and the performance results of (Mayer *et al.*, 2002*c*) carry over to this model.

6. CONCLUSION

Model-based debugging carries considerable promise based on the concept of automatically deriving a diagnosis model from the program source code and using this, together with input-output test vectors, to identify faults in programs (Mayer *et al.*, 2002*d*). An effective model representation is a crucial step in building a framework for model-based debugging. This work represents the first time that non-structured control flow is represented in a diagnosis model. We have described the computation of the bidirectional model that permits reasoning about unstructured control flows on both directions as required for diagnostic reasoning. Further, the mapping from the graph into a logic-based model with additional rules for dynamic modification of the model has been described. Notably, this approach omits the strict view of a component representing one statement of earlier work (Mayer *et al.*, 2002*c*) and provides a more flexible mapping from code to model.

Since real-world programs make frequent use of constructs such as exceptions, this model represents a significant step beyond earlier models that represented Java programs for debugging, but were restricted to linear control flows. As it shares the basic model-based architectures, it can be smoothly incorporated into the implemented JADE intelligent debugging engine (Mayer *et al.*, 2002*c*).

REFERENCES

Ananian, S. (1999). The static single information form. Master's thesis. Department of Electrical and Computer Science, Princeton University.

Bourdoncle, F. (1993). Abstract debugging of higher-order imperative languages. In: *Proc. SIGPLAN Conf. on Prog. Lang. Design and Implementation*. pp. 46–55.

Clarke, E. M., O. Grumberg and D. E. Long (1994). Model Checking and Abstraction. *ACM TOPLAS* **16**(5), 1512–1542.

Console, L., G. Friedrich and D. T. Dupré (1993). Model-based diagnosis meets error diagnosis in logic programs. In: *Proceedings* 13th *International Joint Conf. on Artificial Intelligence*. Chambery. pp. 1494–1499.

Cousot, P. and R. Cousot (1977). Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixpoints. In: *POPL*. Vol. 4. Los Angeles, California.

Felfernig, A., G. Friedrich, D. Jannach and M. Stumptner (1999). Consistency based diagnosis of configuration knowledge-bases. In: *Proc. DX'99 Workshop*. Loch Awe.

Ferrante, J., K. J. Ottenstein and J. D. Warren (1987). The program dependence graph and its use in optimization. *ACM TOPLAS* **9**(3), 319–349.

Gerlek, M. P., E. Stoltz and M. Wolfe (1995). Beyond induction variables: Detecting and classifying sequences using a demand-driven SSA. *ACM TOPLAS* **1**(17), 85–122.

Johnson, R., D. Pearson and K. Pingali (1993). Finding regions fast: Single entry single exit and control regions in linear time. Technical Report CTC93TR141. Department of Computer Science, Carnell University, Ithaca, NY.

Junker, U. (2001). QUICKXPLAIN: Conflict detection for arbitrary constraint propagation algorithms. In: *IJCAI'01 Workshop on Modelling and Solving problems with constraints*. Seattle, WA, USA.

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

Mateis, C., M. Stumptner and F. Wotawa (2000). A Value-Based Diagnosis Model for Java Programs. In: *Proc. DX'00 Workshop*. Morelia, Mexico.

Mayer, W., M. Stumptner and F. Wotawa (2002*a*). Model-based Debugging or How to Diagnose Programs Automatically. In: *Proc. IEA/AIE*. Springer LNAI. Cairns, Australia.

Mayer, W., M. Stumptner and F. Wotawa (2002*b*). Modeling programs for exception debugging. Technical Report 02-02. University of South Australia, Advanced Computing Research Centre.

Mayer, W., M. Stumptner, D. Wieland and F. Wotawa (2002*c*). Can AI help to improve debugging substantially? Debugging Experiences with Value-Based Models. In: *Proc. ECAI*. Amsterdam.

Mayer, W., M. Stumptner, D. Wieland and F. Wotawa (2002*d*). Towards an Integrated Debugging Environment. In: *Proc. ECAI*. Amsterdam.

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

Singer, J. (2002). Efficiently computing the static single information form. Technical report. Computer Laboratory, Cambridge University.

Sinha, S. and M. J. Harrold (2000). Analysis and Testing of Programs with Exception Handling Constructs. *IEEE TSE* **26**(9), 849–871.

Stumptner, M. and F. Wotawa (2000). Using Model-Based Reasoning for Locating Faults in VHDL Designs. *Künstliche Intelligenz* **14**(4), 62–67.

Weiser, M. (1984). Program slicing. *IEEE TSE* **10**(4), 352–357.