MODEL-BASED DEBUGGING WITH HIGH-LEVEL OBSERVATIONS

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

Abstract Recent years have seen considerable developments in modeling techniques for automatic fault location in programs. However, much of this research considered the models from a standalone perspective. Instead, this paper focuses on the highly unusual properties of the testing and measurement process, where capabilities differ strongly from the classical hardware diagnosis paradigm. In particular, in an interactive debugging process user interaction may result in highly complex input to improve the process. This work extends the standard entropy-based measurement selection algorithm proposed in (de Kleer and Williams, 1987) to deal with high-level observations about the intended behavior of Java programs, specific to a set of test cases. We show how to incorporate the approach into previously developed model-based debugging frameworks and to how reasoning about high-level properties of programs can improve diagnostic results.

Keywords: Model-based reasoning, Software engineering and AI, Diagnosis, Debugging

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) work that is being continued in a series of workshops on debugging (Ducassé, 2000). In the 1990s model-based diagnosis techniques (Reiter, 1987) started to be examined for locating faults in software, originally for logic programs (Console et al., 1993), later for the debugging of large-scale concurrent hardware designs written in VHDL (Friedrich et al., 1999), and more recently imperative programs (Mayer et al., 2002a).

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 diagnostic 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, and $AB(C)$ expresses that the program part modeled by C is possibly faulty (*ABnormal*) and can show arbitrary effects. 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., 2002c) have successfully been applied in the Jade project to debug Java programs, with tests performed on 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., 2002a). More recent work (e.g., (Mayer and Stumptner, 2003)) has added models based on the Abstract Interpretation Framework (Cousot and Cousot, 1977; Cousot and Cousot, 2000) and also moved to more efficient models that are based on test case specific representation of individual traces.

While considerable improvements in the modeling and diagnostic algorithms have been achieved, the interactive aspect of (semi-) automatic debugging has so far taken second seat behind the computational aspects. In particular, previous research prototypes combined a standard debugger-like interface with

a variant of the standard entropy-based selection of measurements to identify points during program execution where the debugger user would be queried about the correctness of (parts of) the program status at that point in execution (the user serving as "oracle"). The experience was that answering these oracle queries posed by the system could be difficult in many cases. In particular, if the model of the program closely reflects the program's semantics, most queries can only be answered if the program behavior is hand-simulated up to that point. Therefore, to be useful for interactive debugging, an approach to queries is needed that is both more powerful and simple for the developer (debugger user) to apply. This work takes a step in this direction in that the measurement selection algorithm proposed in (de Kleer and Williams, 1987) is extended to deal with certain issues specific to the software domain. Furthermore, we introduce the notion of what we call *high-level observations* (HLO's) about the expected behavior of the program. Dedicated HLO predicates provide high level descriptions of program execution beyond the classical diagnosis test of whether a given part of the program state is correct or incorrect. We refer to an observation as high-level if it constrains multiple program states and/or locations. HLO's thus allow for debugging capabilities beyond current modeling approaches while keeping the information that has to be provided by the user at a minimum.

In addition to missing observations needed to refute unwanted diagnoses (we assume that there is a threshold on the maximal cardinality of fault candidates. Therefore, additional conflicts remove diagnosis candidates once the threshold is exceeded) an additional source of spurious fault candidates is approximate modeling. In contrast to many diagnostic models, where the behavioral description of each component is known in full, for programs approximations need to be made to avoid undecidability issues related to loops, recursive method calls, and dynamic data structures. The high-level observation approach also helps in this case, as refined modeling may diminish negative effects cause by coarse approximations.

Lastly, more effective control over the queries asked of the user is important. The application of the standard measurement selection proposed in (de Kleer and Williams, 1987) to the software domain is problematic. In particular, changing control- and data flow for multiple test cases and fault candidates results in non-comparable sets of variables and possible values, which renders the standard algorithm unsuitable. Furthermore, multiple test cases with widely varying value ranges need to be integrated.

Under these conditions entropy alone is not sufficient to effectively pinpoint measurements that discriminate between fault candidates. To make queries easily answerable, query complexity and reasoning over all test cases needs to be incorporated.

The rest of the paper is organized as follows. Section 2 investigates the sources of query complexity in model-based debugging and describes the concept of high-level observations. Section 3 provides an algorithm to select the "best" measurement and describes how to incorporate our measurements in the approach of (de Kleer and Williams, 1987). Finally, we discuss future extensions before we conclude the paper.

2. DEBUGGING WITH HIGH LEVEL OBSERVATIONS

In our current scheme, HLO's are produced by presenting queries to the user that have been ranked high as measurements to be selected (see Section 3); the query specifies a high level condition and the user provides a HLO, or measurement outcome, by answering it. A query is a condition on the program state or actual execution. A query schema describes the possible HLO's the debugging system can deal with.

DEFINITION 1 *A* query schema $\mathcal{S} = \langle \mathcal{P}, C, \mathcal{Q} \rangle$ *is a 3-tuple, where* $\mathcal{P}$ *denotes a first order formula representing the scope of* $\mathcal{Q}$, C *is the complexity of the query, and* $\mathcal{Q}$ *represents a logical schema describing the actual condition (or "query") to be tested.*

For all components L in a model run (or program execution) R where $P(L)$ is true, $Q(L)$ is applicable. C is a positive number, predefined or assigned by the system, that estimates the complexity of the query as it will be experienced by the user. This is later used to rank easier queries first. We make the simplifying assumption that all query instances of the same query schema have equal complexity. Q (also denoted by $Q(\mathcal{S})$) denotes the condition that is to be confirmed or refuted by the user. We will see later that these conditions often coincide with the traditional notion of an invariant.

EXAMPLE 2 $\langle \_loop(L), 5, \mathsf{allUpdated}(L) \rangle$ *denotes a query schema that tracks the property* $\mathsf{allUpdated}(L)$ *for all loops (all parts of the program code for which* $\_loop(L)$ *is true). The complexity associated with the query is 5.*

DEFINITION 3 *A* query $q = \langle T, L, C, Q, R_T, R_F, R_U \rangle$ *is an instance of a query schema* $\mathcal{S}$, *where* T *is the test case that q is valid for,* L *denotes the location where the query takes place in the model describing* T, C *is the complexity (taken from the query schema), and* Q *is the query expression (*invariant*) specified by the query schema* $\mathcal{Q} = \mathcal{Q}(\mathcal{S})$ *with all free variables instantiated with information describing the context of the query (test case, location, context, complexity, question).* R_T *and* R_F *are the sets of fault candidates that predict* Q *to be true and false, respectively. Candidates in* R_U *are those that do not predict a value for* Q.

When debugging using a model that follows the program semantics closely, three main causes of complex queries can be identified. First, unless the observed variable is at a point close to the start or the end of the program execution, the user needs to simulate much of the programs behavior to compute the desired value. Second, frequent switching between different execution states makes it much harder for the user to build a model of the correct execution of a program. This is especially true if the execution states are deep in the middle of some complex computation. Finally, the user cannot rely on values of variables provided by the debugger, as these may have been influenced by the true program fault or the diagnostic assumptions.

Our current approach to tackling above problems is to abstract from concrete values of variables as much as possible and present simple high-level queries to the user, i.e., individual predicates but chosen for their suitability to

T_f : list.price $= [2, 10, 4, 8, 100, 40]$. T_c : list.price $= [0, 1, 7, 5]$

```
1   class PriceList {
2     float price;
3     PriceList next;
4   }
5   class Test {
6     static void increase(PriceList list) {
7       while (list != null) {
8         if (list.price < 10.0f)
9           list.price += 1.0f;
10        else
11          list.price *= 0.05f; // 1.05f
12        list = list.next;
13      }
14    }
15  }
```

T_f : list.price $= [3, 10.5, 5, 9, 105, 42]$. T_c : list.price $= [1, 2, 8, 6]$

Figure 1. Example program and test case.

extract parts of the design information from the user's mental model and use it for diagnostic purposes. Concrete values (heretofore our standard type of observations) are mainly used to define input values and to check the result of a program run. The HLO's provide the necessary information to prune conflicts and build refined models that help avoiding some approximations. Thus, while in general software diagnosis is more difficult than hardware diagnosis due to the absence of a correct model, in this case the ability to obtain a description of *intent* from the user (developer) makes observations potentially more powerful than with hardware. Generation and selection of suitable high level queries are described in sections 2 and 3, respectively.

EXAMPLE 4 *For illustration, consider the program in figure 1. Using a value-based model (Mateis et al., 2000) and the input and expected output values depicted in the figure for test case T_f, statements* list!=null$_7$, list.price*=0.05f$_{11}$, *and* list=list.next$_{12}$ *are identified as possible faults. In addition to the true fault in line 11, two spurious diagnoses are reported. By adding the observation that each member of the list must be assigned exactly once, it is derived that the loop must be bounded (because the data structure that is traversed is acyclic). (We assume that every program eventually terminates.) This information allows to build a refined model (Mayer and Stumptner, 2004), where* list!=null$_7$ *and* list=list.next$_{12}$ *are no longer considered as explanations.*

2.1 The HLO Catalogue

While there are no intrinsic requirements on the structure and subexpressions used in HLO's, in our current implementation each type of observation is separately implemented, in keeping with the assumption that it is advantageous to let the users deal with individual, dedicated predicates rather than asking them to form complex expressions interactively. Currently, our models can deal with the following different types of HLO predicates. (We omit the formal definition for space reasons. "A" refers to the attributes used by the predicate. Optional attributes are in brackets.)

Traversal properties: Elements of arrays and dynamic data structures are often processed such that either all of them are read or updated. If the underlying data structure is monotonic, this allows to bound the number of times the iterating loop or recursive function is executed, which in many cases allows to refine the model to avoid imprecision due to approximation and to exclude unwanted fault candidates. **A**: data structure (variable), type of access, [order of traversal].

Acyclicity: Knowledge that a data structure should be acyclic can be exploited to bound the number of loop iterations and method calls. Code that builds such data structures is also constrained in that any diagnosis candidate that implies a cycle is refuted. **A**: data structure (variable), path.

Read- and write-only: Observations that all elements of a data structure used as input to a loop or a method call are either only read or updated only. **A**: data structure, [type or path constraints]. The latter limit the effect of the assertion to a subset of all reachable elements.

Read-only assertions allow to ignore any attempted update to data structures passed to a loop or method call, and exclude diagnostic candidates implying such updates. This is most useful to bypass model limitations due to aliasing between references to dynamic data structures.

Write-only access is used to decouple the previous values of the data structure from the updated values, leading to smaller conflicts in case the update operation is abnormal.

Subproblem (in)dependence: Loops and recursive method invocations can be modeled differently if it is known that computations in different loop iterations and disjunct sub-structures of dynamic data structures are independent (Mayer and Stumptner, 2004) (modulo the loop variable, which is of course updated based on the previous value). This allows to construct behavioral descriptions with fewer approximation operators and fewer data dependencies between components, resulting in more precise values and smaller conflicts. **A**: loop statement.

Variable (in)dependence: Information about possible dependencies between variables can be utilized to infer missing statements or uses and updates of wrong variables (Jackson, 1995). In our approach, this is also used to shortcut conflict computation, especially if the dependencies relate method parameters and side-effects at the method exit. **A**: variable set. Dependency information could also be used to chose a suitable abstraction for dependent variables. For example using a relational model for local dependent variables may reduce spurious values (and fault candidates) compared to a purely non-relational approach.

Loop specific invariants: Loops based on counters or other induction variables (Gerlek et al., 1995) can often be bounded if monotonicity of the induction variable is assumed. Although it is possible to infer that property for large classes of loops using syntactic pattern-based and Abstract Interpretation (Cousot and Cousot, 1977) approaches, manual specification also eliminates the fault candidate where the update expression of the induction variable is assumed abnormal.

UML invariants: Invariants taken from UML class diagrams, such as type constraints and cardinality constraints for relations, are useful to detect conflicts as soon as the invariant is violated. Consequently, the number of components in

a conflict is reduced. A further benefit of type invariants is that dynamic types can often be derived even if the concrete object is unknown.
Region reachability: Here, statements that should be executed always (or never, or at least once) are marked to remove paths that would otherwise contribute to spurious fault candidates. This is often used to indicate that a particular method invocation should not raise an exception. **A**: Statement set, frequency specification.

2.2 Generating High Level Queries

Queries about high-level properties are generated using an approach borrowed from the Daikon (Ernst et al., 2000) invariant detection tool. The idea is to track a set of properties while test cases are executing, and to eliminate all invariants that do not imply all properties. Statistical measures are used to discard invariants without sufficient support. Tracked properties consist of built-in predicates, such as $\geq$ and =null, and can be extended with user-specified predicates. For this work, we also add properties necessary to infer high-level observations. During model simulation, auxiliary variables are created to track the valid predicates. One important aspect is that tools like Daikon do not generate *real* invariants, but merely guesses based on a set of test cases. In our case that is not a problem as the purpose of queries is not to serve as explanations to the user, but as tests to obtain his agreement or disagreement.

While Daikon only supports forward execution where all values of a program state are known, our approach also allows backward reasoning and variables with unknown values. This is necessary to support invariant detection even if fault assumptions are present. Consequently, the predicates may not evaluate uniquely and are ignored in that case.

EXAMPLE 5 *Reconsidering the program in figure 1. Assume a HLO predicate allUpdated(L) is tracked to find loops L where all elements of the data structure references by L are traversed and updates. This can easily be checked by examining the assignments and the loop update expression in each iteration. Note that it is not necessary that the loop* always *traverses all values. It is sufficient that it does given the values predicted by the* current *test case and fault candidate.*

For each fault candidate, the model is simulated and the predicate is tracked:

Fault candidate	allUpdated(while$_7$)
$\emptyset$	*true*
{*ab*(list!=null$_7$)}	*?*
{*ab*(list.price*=0.05$_{11}$)}	*true*
{*ab*(list=list.next$_{12}$)}	*?*

For the second and the last candidate, the predicate cannot be evaluated uniquely, as the number of loop iterations is not known. The loop is modeled as an approximation only, where targets of assignments in different iterations cannot be determined reliably.

HLO's are instantiated into queries if the tracked property is not strongly refuted and the assumption of the HLO allows to eliminate some fault candidates. A property P is *strongly refuted* if there exist a test case T and a fault

candidate that predicts false for P given T. The sets R_T is determined by assuming that the HLO predicate holds, to obtain a refined model P' which is subsequently used to analyze each candidate where the predicate may not hold. All candidates that imply contradictions in the new model are eliminated from R_T. R_F and R_U are empty, as there must not exist any candidates that predict false for P'. While simulation and candidate elimination is an expensive operation, this is alleviated to a certain extent by the measurement selection, which can rule out many potential queries based on the queries' locations alone.

EXAMPLE 6 (CONT. FROM EX. 5) *When checking either of the two candidates from example 5 for consistency, it is discovered that the invariant is never strongly refuted. Consequently, a refined model for the loop is built assuming* allUpdated(while$_7$) *holds. As a consequence, the statements in lines 7 and 12 are no longer valid single faults, as the incorrect values are still derived (independent of the effects of* list!=null$_7$ *and* list=list.next$_{12}$*).*

Consequently, the query $\langle T_f, \text{while}_7, 5, \text{allUpdated}(\text{while}_7), R_T, \emptyset, \emptyset \rangle$, *with* $R_T = \{\text{list.price*=0.05f}_{11}\}$ *is instantiated.*

3. SELECTING MEASUREMENTS

A solution for selecting good measurements given a test case and a set of fault candidates was presented in (de Kleer and Williams, 1987). The algorithm utilizes entropy to find the variable that, when observed, on average eliminates the most candidates. Only fault probabilities for components and the values predicted by the fault candidates are required.

For the software domain, we obtain fault probabilities from the execution paths of correct and faulty test cases (Mayer et al., 2002b). Components that are executed for few correct test cases and many failing test runs are assigned a higher fault probability.

To integrate the HLO's and the IIO's into the measurement selection, we define auxiliary variables o_i for each query Q_i, with $\text{dom}(o_i) = \{true, false\}$. The sets R_T, R_F, and R_U are used to compute the entropy for o_i: $H(o_i) = p(o_i = true) \log p(o_i = true) + p(o_i = false) \log p(o_i = false) + p(R_U) \log 2$ (H denotes the entropy, $p(o_i = v) = p(S_v) + \frac{p(R_U)}{2}$, where S_v is the set of selected candidates where $o_i = v$, and $p(X)$ denotes the summed probability of all fault candidates in X). Finally, the measurement with maximal entropy is selected.

While this approach selects the variable that (on average) optimally discriminates fault candidates, it proved insufficient for interactive debugging, mainly due to the queries being too complex to answer with reasonable effort. For interactive debugging, selected queries must also conform to the following properties:

1 Queries that are deep in the execution trace of a program are difficult to answer, as the values predicted by fault candidates cannot be relied upon to judge correctness of invariants.

2 The selected queries should be focused (from a user's point of view). In particular, queries should not "jump around" in the execution trace, as

this prohibits the user from building a model of the correct execution of the program.

3 Subsequent queries should use the same test case, if possible.

To elude these problems, we propose to extend the entropy-based measurement selection with a heuristic approach, that sacrifices optimality in favor of low query complexity.

To minimize query complexity, for each query we compute a "distance" d_i between the location $l_i = L(Q_i)$ and the location of the closest answered query (or the program start or end point) for the same test case. d_i is derived from the execution profile of the test cases and all the fault candidates: Each statement that is executed for at least one fault candidate is marked. Also, the union of all the call graphs for each fault candidate is computed.

Starting at l_i, transitions between statements are explored to find the path to an answered query Q_j where the difference between the minimal nesting depth and the maximal nesting depth is minimal. To find the closest query, we apply a simple best-first search algorithm, following only transitions between marked statements. The nesting depth of a statement in a method is computed from the source code. For called methods, all possible call and return transitions are followed, according to the call graph generated earlier. The nesting depths for each method are summed.

EXAMPLE 7 *Reconsider the query from example 6 (for test case T_f). Assume further that no other queries have been answered. The nesting depth of line 7 is 0, as there is no enclosing scope other than the method. Lines 8 and 12 and lines 9 and 11 are at nesting depth 1 and 2, respectively. The call graph is empty, as there are no method calls. All statements are executed for at least fault candidate.*

The query is located at line 7 and has thus distance 0 to the start of the program.

Using the entropy, the query complexity, and d_i, the best possible measurement that is easy to answer and not too far from the a previously observed position is selected. The algorithm proceeds as follows:

1 **Compute entropy.** For each query Q_i, compute the entropy e_i using $R_T(Q_i)$, $R_F(Q_i)$, and $R_U(Q_i)$. For variables that are not associated with HLO's and IIO's use the standard algorithm from (de Kleer and Williams, 1987).

2 **Mark executed statements.** For each test case and each fault candidate, mark the statements that are executed.

3 **Compute call graph.** Build the graph containing all possible call sequences between methods and statements marked in the previous step.

4 **Compute distances.** For each Q_i, compute $d_i = \forall_{Q_j} \min \text{distance}(L(Q_i), L(Q_j))$. Start at $L(Q_i)$ and perform a best-first search, traversing only statements that are marked. Move between method according to the graph computed in the previous step.

5 **Find a good, close query.** Find the k queries where $e_i \cdot r + d_i \cdot (1 - r)$ is smallest. Discard all other queries.

6 **Pick the simplest.** From the remaining queries, return the one with minimal $C(Q_i)$. If there is more than one candidate, select the one with larger entropy (or lower complexity if entropy does not discriminate).

The parameters r and k are currently pre-specified, but could also be adapted according to the history of (un)answered queries.

EXAMPLE 8 *In our example, the fault probabilities for all statements are 0.5, with the exception of line 9, where the probability is 0.0. Given the sets R_T, R_F, and R_U from example 6, the entropy of the* allUpdated(while$_7$) *observation is greater or equal than for any other observation.*

The predicate allUpdated(while$_7$) *is (by definition of our HLO library) simpler than queries of type 'Subproblem Independence'. Other HLO queries are not considered as either they are refuted by a test case, or they are always strongly derived. It is also simpler than the IIO queries for the same reason.*

The HLO query is also the closest to the program start, as all other statements are embedded into the loop. It is therefore selected in step 6 of the above algorithm.

4. FURTHER WORK & CONCLUSION

While the approach described in this paper has provided promising results on a set of small benchmark programs, there are open issues requiring further work. Currently, the measurement selection is not fully integrated in our debugging prototype, which makes further evaluation difficult. Further, it is not clear whether the parameters k and r must be preset or if there exist good heuristics to choose and update those values. The user interaction aspect of interactive debugging also requires more investigation, in particular, the question on what and how much context needs to be provided to the user to allow efficient query answering.

In contrast to other debugging and verification approaches (for example (Groce and Visser, 2003)), our method has the advantage that it does not require to specify the behavior of the program in a formal language. Rather, it is sufficient to provide properties and invariants that are *specific to a set of test cases*, which is usually much easier, especially as the behavioral description need not be complete. Also, complexity is lower than for verification, as we do not follow *all* possible executions of a program. Instead we focus on the program behavior specific to a set of test cases, which is usually good enough if the test set is large.

In summary, we have presented an approach to semi-automatic debugging that builds on a fixed library of predicates which can be used to analyze high-level properties of the program. Combined with the inference of test-case specific invariants, this allows for more expressive user interaction, leading to a more powerful oracle describing the expected behavior of the program. Interaction between specific types of HLO's and specific types of program constructs enables effective heuristics and provides a gateway for the choice of a correct model in a specific situations (a long-term goal of our research (Mayer et al., 2002b)).

References

Clarke, Edmund M., Grumberg, Orna, and Long, David E. (1994). Model Checking and Abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542.

Console, Luca, Friedrich, Gerhard, and Dupré, Daniele Theseider (1993). Model-based diagnosis meets error diagnosis in logic programs. In *Proc.* 13^{th} *IJCAI*, pages 1494–1499, Chambery.

Cousot, Patrick and Cousot, Radhia (1977). Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixpoints. In *POPL'77*, pages 238–252, Los Angeles.

Cousot, Patrick and Cousot, Radhia (2000). Abstract interpretation based program testing. In *Proceedings of the SSGRR 2000 Computer & eBusiness International Conference.*

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

Ducassé, Mireille, editor (2000). *Proceedings of the* 4^{th} *International Workshop on Automated and Algorithmic Debugging, AADEBUG '00*. Munich.

Ernst, Michael D., Czeisler, Adam, Griswold, William G., and Notkin, David (2000). Quickly detecting relevant program invariants. In *Proceedings of the* 22^{nd} *International Conference on Software Engineering*, pages 449–458, Limerick, Ireland.

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

Gerlek, Michael P., Stoltz, Eric, and Wolfe, Michael (1995). Beyond induction variables: Detecting and classifying sequences using a demand-driven SSA. *ACM Transactions on Programming Languages and Systems*, 1(17):85–122.

Groce, Alex and Visser, Willem (2003). What went wrong: Explaining counterexamples. In *SPIN Workshop on Model Checking of Software.*

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

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

Mateis, Cristinel, Stumptner, Markus, and Wotawa, Franz (2000). A Value-Based Diagnosis Model for Java Programs. In *Proc.* 11^{th} *Int'l Workshop on Principles of Diagnosis*, Morelia, Mexico.

Mayer, Wolfgang and Stumptner, Markus (2003). Model-based debugging using multiple abstract models. In *Proceedings of the* 5^{th} *International Workshop on Automated and Algorithmic Debugging, AADEBUG '03*, pages 55–70, Ghent.

Mayer, Wolfgang and Stumptner, Markus (2004). Approximate modeling for debugging program loops. In *Proceedings of the Fifteenth International Workshop on Principles of Diagnosis*, Carcassonne.

Mayer, Wolfgang, Stumptner, Markus, Wieland, Dominik, and Wotawa, Franz (2002a). Can AI help to improve debugging substantially? Debugging Experiences with Value-Based Models. In *Proc. ECAI*, pages 417–421, Lyon.

Mayer, Wolfgang, Stumptner, Markus, Wieland, Dominik, and Wotawa, Franz (2002b). Towards an Integrated Debugging Environment. In *Proceedings of the European Conference on Artificial Intelligence (ECAI)*, pages 422–426, Lyon.

Mayer, Wolfgang, Stumptner, Markus, and Wotawa, Franz (2002c). Model-based Debugging or How to Diagnose Programs Automatically. In *Proc. IEA/AIE*, Springer LNAI, pages 746–757, Cairns, Australia.

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

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