Abstract Interpretation of Programs for Model-Based Debugging

Wolfgang Mayer **Markus Stumptner**
Advanced Computing Research Centre
University of South Australia
[mayer,mst]@cs.unisa.edu.au

Abstract

Developing model-based automatic debugging strategies has been an active research area for several years. We present a model-based debugging approach that is based on Abstract Interpretation, a technique borrowed from program analysis. The Abstract Interpretation mechanism is integrated with a classical model-based reasoning engine. We test the approach on sample programs and provide the first experimental comparison with earlier models used for debugging. The results show that the Abstract Interpretation based model provides more precise explanations than previous models or standard non-model based approaches.

1 Introduction

Developing tools to support the software engineer in locating bugs in programs has been an active research area during the last decades, as increasingly complex programs require more and more effort to understand and maintain them. Several different approaches have been developed, using syntactic and semantic properties of programs and languages.

This paper extends past research on model-based diagnosis of mainstream object oriented languages, with Java as concrete example [14]. We show how abstract program analysis techniques can be used to improve accuracy of results to a level well beyond the capabilities of past modelling approaches. In particular, we discuss adaptive model refinement. Our model refinement process targets loops in particular, as those have been identified as the main culprits for imprecise diagnostic results. We exploit the information obtained from abstract program analysis to generate a refined model, which allows for more detailed reasoning and conflict detection.

Section 2 provides an introduction to model-based diagnosis and its application to automatic debugging. Section 3 outlines the adaption of the program analysis module of our debugger and discusses differences between our and the classical analysis frameworks. Section 4 contains results obtained though a series of experiments and compares the outcome with other techniques. A discussion of related and future work concludes the paper.

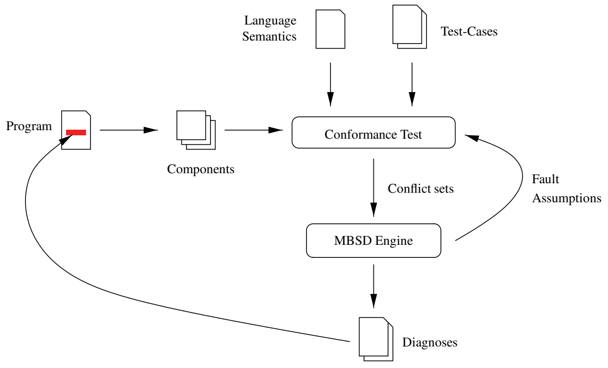


Figure 1: MBSD cycle

2 Model-based debugging

Model-based software debugging (MBSD) is an application of *Model-based Diagnosis (MBD)* [19] techniques to locating errors in computer programs. MBSD was first introduced by Console et. al. [6], with the goal of identifying incorrect clauses in logic programs; the approach has since been extended to different programming languages, including VHDL [10] and JAVA [21].

The basic principle of MBD is to compare a *model*, a description of the correct behaviour of a system, to the *observed behaviour* of the system. Traditional MBD systems receive the description of the observed behaviour through direct measurements while the model is supplied by the system's designer. The difference between the behaviour anticipated by the model and the actual observed behaviour is used to identify components that, when assumed to deviate from their normal behaviour, may explain the observed behaviour.

The key idea of adapting MBD for debugging is to exchange the roles of the model and the actual system: the model reflects the behaviour of the (incorrect) program, while the test cases specify the correct result. Differences between the values computed by the program and the anticipated results are used to compute model elements that, when assumed to behave differently, explain the observed misbehaviour. The program's instructions are partitioned into a set of *model components* which form the building blocks of explanations. Each component corresponds to a fragment of the program's source text, and diagnoses can be expressed in terms of the original program to indicate a potential fault to the programmer (see

Figure 1).

Each component can operate in *normal mode*, denoted $\neg AB(\cdot)$, where the component functions as specified in the program, or in one or more *abnormal modes*, denoted $AB(\cdot)$, with different behaviour. Intuitively, each component mode corresponds to a particular modification of the program.

Example 1 *The program in Figure 2 can be partitioned into five components, $C_1, \ldots, C_5$, each representing a single statement. For components representing assignments, a possible AB mode is to leave the variable's value undetermined. Another possibility is to expand the assignment to a set of variables, rendering their values undetermined. For loops, the number of iterations could be altered.*

The model components, a formal description of the semantics of the programming language and a set of test cases are submitted to the conformance testing module to determine if the program reflecting the fault assumptions is consistent with the test cases. A program is found *consistent* with a test case specification if the program *possibly* satisfies behaviour specified by the test case.

In case the program does not compute the anticipated result, the MBSD engine computes possible explanations in terms of mode assignments to components and invokes the conformance testing module to determine if an explanation is indeed valid. This process iterates until one (or all) possible explanations have been found.

Formally, our MBSD framework is based on Reiter's theory of diagnosis [19], extended to handle multiple fault modes for a component. MBSD relies on test case specifications to determine if a set of fault represents is a valid explanation, where each test case describes the anticipated result for an execution using specific input values.

Definition 1 (Debugging Problem) *A* Debugging Problem *is a triple $\langle P, \mathbf{T}, \mathbf{C}\rangle$ where P is the source text of the program under consideration, $\mathbf{T}$ is a set of test cases, and $\mathbf{C}$ denotes the set of components derived from P.*

The set $\mathbf{C}$ is a partition of all statements in P and contains the building blocks for explanations returned by the debugger. For simplicity of presentation, it is assumed that there is a separate component for each program statement.

A set of fault assumptions $\mathbf{\Delta} \,\hat{=}\, \{AB(C_1), \ldots, AB(C_k)\}$ is a valid explanation if the model modified such that components C_i may exhibit deviating behaviour, while the remaining components exhibit normal behaviour, no longer implies incorrect behaviour.

Definition 2 (Explanation) *A fault assumption $\mathbf{\Delta}$ is a* consistent explanation *for a Debugging Problem $\langle P, \mathbf{T}, \mathbf{C}\rangle$ iff for all test cases $T \in \mathbf{T}$, it cannot be derived that all executions of P (altered to reflect Δ) violate T:*

$$\forall T \,\hat{=}\, \langle I, O\rangle \in \mathbf{T} \quad [\![P \oplus \mathbf{\Delta} \oplus O]\!](I) \neq \bot.$$

I and O represent the input values and assertions describing the correct output values specified in a test specification, respectively. $[\![\cdot]\!]$ denotes the semantic function of the programming language and the fault assumptions. $\oplus$ denotes the application of the fault assumptions in $\mathbf{\Delta}$ and test assertions O to P. $\bot$ denotes an infeasible program state.

```
  I : n ↦ 4
1 int i = 1
2 int x = 1
3 while (i ≤ n) {
4    x = x + i      "correct is x = x * i"
5    i = i + 1
6 }
  O : assert (x == 24)
```

Figure 2: Example program and test specification

Example 2 *Assume component C_1 representing line 1 of the program in Figure 2 is abnormal. In this case, there is no execution that terminates in a state satisfying $x == 24$. Therefore, a fault in line 1 cannot explain the program's misbehaviour. In contrast, if the value of x is left undetermined after line 2, an execution exists where the program satisfies the assertion. Therefore, C_2 is a potential explanation.*

While the definition of an explanation is intuitive, the precise test is undecidable and must be approximated. Different abstractions have been proposed in the literature, ranging from purely dependency-based representations [10] to Predicate Abstraction [13]. The following section introduces an approximation based on Abstract Interpretation [8], a technique borrowed from program analysis.

3 Model construction

Models developed for VHDL and early models for Java were based on static analysis and represented the program as a fixed model representing the program's structure. As test cases were not taken into account, it was necessary to represent *all* possible executions and fault assumptions. While this approach has been used successfully to debug VHDL programs, the dynamic method lookup and exception handling lead to large models for Java programs.

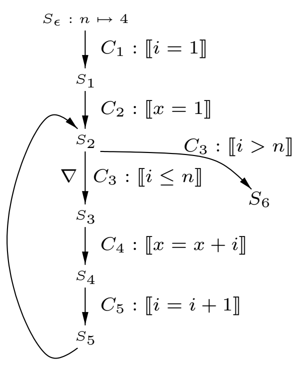
Mayer and Stumptner [15] propose an approach where the model is constructed dynamically, taking into account only executions that are feasible given a test case. The approach is based on Abstract Interpretation [8] of programs, generalised to take fault assumptions and test case information into account. In the following, the basic principles are introduced. More detailed presentations are given in [15; 16].

3.1 Abstract interpretation

The model is derived from a graph representing the effects of individual statements of the program.

Definition 3 (Program Graph) *A program graph for a program P is a tuple $\langle \mathbf{V}, S_\epsilon, S_\omega, \mathbf{E}\rangle$ where $\mathbf{V}$ is a finite set of vertices, $E \subseteq \mathbf{V} \times \mathbf{V}$ a set of directed edges, and $S_\epsilon \in \mathbf{V}$ and $S_\omega \in \mathbf{V}$ denote distinct entry and exit vertices.*

The vertices of the program graph represent sets of program states; the edges represent the effects of program instructions, transforming sets of program states into new sets of states. The initial vertex ϵ is associated with the initial program state I described by the test specification. An execution of a program starts in the initial state in S_ϵ and proceeds following the



$$\hat{S}_\epsilon = \hat{I}$$
$$\hat{S}_1 = [\![\widehat{i = 1}]\!](\hat{S}_\epsilon)$$
$$\hat{S}_2 = [\![\widehat{x = 1}]\!](\hat{S}_3) \sqcup \hat{S}_5$$
$$\hat{S}_3 = \nabla([\![\widehat{i \leq n}]\!](\hat{S}_2), \hat{S}'_3)$$
$$\hat{S}_4 = [\![\widehat{x = x + i}]\!](\hat{S}_3)$$
$$\hat{S}_5 = [\![\widehat{i = i + 1}]\!](\hat{S}_4)$$
$$\hat{S}_6 = [\![\widehat{i > n}]\!](\hat{S}_2)$$
$$\hat{S}_\omega = [\![\widehat{x == 24}]\!](\hat{S}_6)$$

Figure 3: Program graph and abstract equation system

edges until the final vertex ω has been reached. The program states in S_ω represent the result computed by the program.

Example 3 *Figure 3 depicts the program graph derived from the program in Figure 2 (where $\mathbf{\Delta} = \emptyset$). Each program statement is represented as an edge, connecting the program state before the statement is executed to the successor state. The loop statement in line 3 induces two possible successors: one transition representing the case where the loop is entered, and one transition representing loop termination. Starting in the initial state where $n \mapsto 4$, the flow graph induces a trace that traverses the loop four times and the stops in S_ω with $x \mapsto 12$.*

In case the initial state is not fully specified or parts of an intermediate state become undetermined due to fault assumptions, a potentially infinite number of paths must be followed to determine the states in S_ω.

To ensure every analysis of the graph is finite, the concrete program states and the concrete semantics of the program are replaced with abstract versions. Formally, the Abstract Interpretation framework [8] uses complete lattices to represent concrete ($\mathbf{S}$) and abstract states ($\hat{\mathbf{S}}$) such that $\langle \hat{\mathbf{S}}, \perp, \top, \sqsubseteq, \sqcup, \sqcap \rangle$ is a safe abstraction of $\langle \mathcal{P}(\mathbf{S}), \emptyset, \mathbf{S}, \subseteq, \cup, \cap \rangle$. $\perp$ denotes infeasibility and $\top$ represents all possible states. $\sqsubseteq$ represents the abstract state ordering, and $\sqcup$ and $\sqcap$ denote the least upper bound and greatest lower bound operator, respectively.

The key component in this framework is a pair of functions $\langle \alpha, \gamma \rangle$ ("Galois connection"), where α maps sets of concrete program states to their abstract representation and γ implements the reverse mapping. The semantic functions operating on the abstract domain can then be defined as $\widehat{[\![\cdot]\!]} \doteq \alpha \circ [\![\cdot]\!] \circ \gamma$. An approximation of the program execution can be obtained by translating the program graph into a set of equations and computing the fixpoint solution. In case the abstract lattice is of infinite height and the program graph is cyclic, *widening operators* (∇) must be applied to ensure termination. Widening operators selectively discard information from the abstract states to guarantee that the computation of mutually dependent equations eventually converges.

Similar equations systems and fixpoint solutions can be derived for backward analyses, where for any given $\hat{S}_\omega$ an approximation of S_ϵ is computed such that the program is guaranteed to terminate in a state in $\gamma(\hat{S}_\omega)$. In our work we apply a variant of Bourdoncle's bidirectional method [2] where forward and backward analyses are applied repeatedly to refine approximations.

Example 4 *Using backward analysis it can be derived that if the program in Figure 2 should terminate in S_ω with $i \mapsto 24$, then $n \leq 4$ must hold in S_ϵ (assuming that n has been left undetermined in I).*

3.2 Dynamic model creation

To utilise Abstract Interpretation for debugging, Definition 2 must be lifted to the abstract domain, replacing $[\![\cdot]\!]$ and I with their abstract counterparts $\widehat{[\![\cdot]\!]}$ and $\hat{I}$, respectively. For O translation is not necessary, as it is assumed that O is represented as assertions in the programming language and can be evaluated using $\widehat{[\![\cdot]\!]}$.

In our framework, we apply the well-known non-relational interval abstraction [8] to approximate concrete program states. Further, we use the simple abstraction described in [7] to ensure dynamically allocated data structures are represented finitely.

In contrast to purely static program analysis, we do not assume a fixed program graph modelling all possible executions. Instead, the graph is constructed dynamically, using inputs I and assertions O from the test case specifications. Only the paths that may be executed when starting in a state I are created. Paths that contain $\perp$ are ignored. Once a program state is reached where multiple paths could be followed, we apply the conventional equation-based analysis to the remaining execution. Our framework can thus be placed in the middle grounds between purely static analysis [8] and dynamic analysis [9].

The algorithm for determining consistency for a program P with fault assumptions $\mathbf{\Delta}$ and test case T is outlined as follows:

1. Add the assertions in T to P and apply $\mathbf{\Delta}$ to P.
2. Construct the initial program graph using the forward abstract semantics $\widehat{[\![\cdot]\!]}$ and initial state $\hat{I}$.
3. Apply backward analysis to approximate the initial program states not implying assertion violations later in the execution.
4. Analyse program graph in forward direction using the new initial program states to eliminate values that cannot be realised.
5. Repeat from step 3 until a fixpoint is reached.
6. In case the initial program state is is $\perp$, no execution satisfying all test case specifications can be found and $\mathbf{\Delta}$ is not a valid explanation. Otherwise, $\mathbf{\Delta}$ is consistent and another test case is considered.

The benefits of the dynamic approach are as follows:

(i) The equation system is concise. This is a significant advantage when dealing with paths representing exceptions, as in Java runtime exceptions may be thrown by a large number of statements. However, typically only a few of these paths are actually followed in an execution.

(ii) Ambiguity in control flow is reduced compared to the purely static approach. This is an important advantage for the precision of the abstraction of dynamic data structures, which deteriorates dramatically in the presence of loops in the program graph.

(iii) Simple non-relational abstractions are sufficient to obtain good results if values for most related variables are known. Expressions can be partially evaluated, leading to tighter bounds for the remaining variables.

(iv) In case all values potentially accessed by a called method are known in a program state and the method does not include AB components, the fixpoint computation can be replaced with native execution.

The fixpoint analysis is limited to the regions of the execution where the values of abstract states critical to the region's execution are undetermined. As many fault assumptions affect only a minor part of a program state, execution can proceed normally in unaffected regions. Consequently, many alternative branches that must be considered by purely static Abstract Interpretation can be eliminated.

Example 5 *Assume that the MBSD examines $\mathbf{\Delta} = \{AB\,(C_4)\}$ for our running example. In this case, execution proceeds normally until line 4 is reached. The execution of $AB\,(C_4)$ leads to a state where $i \mapsto 1$ and x is undetermined. As i is known, the execution of statement 5 can proceed normally and no ambiguity in the control flow arises at line 3. In fact, the entire trace reduces to a linear chain. Backward analysis derives that the value of x must hold after statement 4 was last executed. The remaining occurrences of x have undetermined values. Since the initial state in the graph is not $\perp$, $\mathbf{\Delta}$ is consistent and is considered a valid explanation.*

3.3 Iterative refinement

The partially determined abstract states facilitate further iterative model refinement. Abstraction of the effects of loops and recursive method calls can be improved if the known states before and after a loop or method call contain conflicting values. In this case, the execution is either inconsistent, or the loop (or method) must be expanded further. In case the loop (call) has been shown to terminate (for example, through syntax-based methods presented in [16]) or a limit on the number of iterations (recursive calls) has been set by the user, the loop (call) can be expanded. As a result, values lost through widening and heap abstraction may be regained, potentially leading to a contradiction elsewhere. Similar improvements can be achieved through informed restructuring of the model based on automatically derived or user-specified properties of program regions. For space reasons we illustrate the refinement approach in the context of our running example and rely on [16] for more detailed discussion.

Example 6 *Assume the MBSD engine examines the candidate explanation $\mathbf{\Delta} = \{AB\,(C_3)\}$. As the loop condition is assumed incorrect, the loop condition cannot be evaluated uniquely. This implies that the number of loop iterations is not known and the program state S_6 after the loop is approximated using intervals: $S_6 : x \mapsto [1..\infty), i \mapsto [1..\infty)$. The upper bound of the intervals is due to the widening operator. The assertion x==24 may be satisfied in S_6, leading to $S_\omega : x \mapsto 24, i \mapsto [1..\infty)$. Backward analysis reveals that $x \mapsto 24$ must also hold after the loop. This state conflicts with the program state before the loop, which contains $x \mapsto 1$. Subsequently, the loop is expanded in an attempt to prove that $\mathbf{\Delta}$ does not represent the true fault. The process repeats and stops after six expansions, leading to a state $S_6 : i \mapsto [8..\infty], x \mapsto [29..\infty]$. At this point, it can be proved that the assertion is violated and $S_\omega : \perp$. A subsequent backward pass derives $S_\epsilon : \perp$ and eliminates $\mathbf{\Delta}$ from the set of potential explanations.*

3.4 Fault assumptions/structural flaws

Similar to model-based hardware diagnosis [1], previous MBSD approaches have great difficulty with faults that manifest as structural differences in the model. The dynamic modelling approach described here is more flexible in that fault assumptions modifying variables that were originally not affected by a component's behaviour can now be addressed appropriately. The modelling of dynamically allocated data structures benefits in particular, as the scope of fault assumptions can now be determined more precisely compared to the crude modelling in previous MBSD approaches, which led to a large fraction of undesirable explanations. Together with more expressive specifications of the correct behaviour [17], we believe that extended fault modes can deal effectively with complex faults.

4 Evaluation

This section presents experimental results obtained from the model described in the previous sections and compares the results with other MBSD approaches, as well as results obtained by Slicing [20]. Readers interested in a theoretical comparison of the different models are referred to [18].

4.1 Slicing

The classic principle of *Program Slicing* [20] is to eliminate all statements from a program that cannot influence the value of a distinguished variable at a given location in the program. The pair $\langle Variable, Location\rangle$ denotes the *slicing criterion*. The working hypothesis is that typically, only a small fraction of a program contributes to a variable's value; the rest can be pruned away to reduce debugging effort.

Static backward slicing starts at the variable and location mentioned in the slicing criterion and recursively collects all statements that may influence the variable's value in any program execution. Further, all statements that may cause the location to be skipped are also included. This process repeats until a fixpoint is reached. While static slices can be computed quickly even for large programs, the remaining program often remains large.

Dynamic slicing aims at reducing the number of statements by considering only dependencies that arise between statements executed for a particular test case execution. Similar to static slicing, dependencies for a variable are followed and all statements contributing to the variable's value are collected. While dynamic slices are typically much smaller than their static counterparts, for programs with many control dependencies or long data-flow chains, results remain similar.

Example 7 *The static slice of the program in Figure 2 w.r.t. the slicing criterion $\langle i, 6\rangle$ includes statements 1, 3 and 5. The* *remaining statements are ignored as they are not relevant for the computation of i. For this test case dynamic slicing computes the same result. The difference between the two approaches can be seen in case the loop is executed only once: The static slice for $\langle x, 6\rangle$ contains all statements, while the dynamic slice does not include statement 5.*

4.2 Dependency-based MBSD

A number of dependency-based MBSD approaches have been introduced in [21]. Wotawa [22] has shown that these approaches provide results equivalent to static or dynamic slicing in case only a single abnormal variable is observed. In the following, we limit our attention to results obtained through slicing without explicitly stating that the same results hold for dependency-based MBSD.

4.3 Value-based MBSD

Value-based Models (VBM) [14] extend beyond simple dependency tracking and model the effects of program statements explicitly. In contrast to the Abstract Interpretation-based approach discussed here, the VBM does not abstract from the concrete semantics and relies on statically constructed models. While the model is effective for programs where dependency-based representations do not provide accurate results, the poor abstraction causes the model to collapse when loop iterations or recursive method calls cannot be determined precisely.

4.4 Experimental results

A set of test programs have been used to evaluate the performance of different debugging approaches. Some programs were taken from the *Siemens Test Suite* [1] (transcribed to Java), a debugging test bench commonly used in the debugging community; others have been retained from earlier tests of the VBM and dependency-based models.

A summary of he results obtained is given in Figure 4. *LoC* denotes the number of non-comment lines in the program's source code, *Comp* represents the number of diagnosis components used to construct explanations, *SSlice*, *DSlice* and *Exec* denote the number of statements in the static slice, dynamic slice and the number of executed statements, respectively. *VBM* and *AIM* denote the number of statements returned as potential explanations for the VBM and the model described in this paper, and *Time* is the average diagnosis time in seconds (wall-clock time) required for the AIM. The results of the AIM and the VBM are limited to single fault explanations. Due to limitations of the implementation of the VBM, some diagnoses listed for the AIM may not be included in the VBM's results. Initial experiments with multiple faults indicated that the number of explanations does not differ significantly from static slicing and does not warrant the additional overhead necessary for the AIM and VBM.

For each program, n variants with different faults were created and tested with up to eight test cases per variant. The test results reported are the averages for each program over all variants and test cases. Slight improvements could be observed when using two rather than a single test case; no significant differences could be detected in any of the test programs (with the exception of *Adder*) when using more than two test cases. This can be attributed to the structure of the selected programs, where a large fraction of statements and test cases are executed for all test cases.

It can be seen that static slicing and dynamic slicing in many cases cannot improve much compared to the entire program and the statements executed in test cases. Similarly, dynamic slices often improve little compared to the executed statements, as all statements contribute to the final result. Comparing VBM and AIM, it can be seen that the AIM improves over the VBM in most cases. In fact, the case where the VBM provides fewer explanations is due to explanations missed by the VBM. Comparing AIM and slicing: the AIM provides significantly fewer explanations, but is computationally more demanding. (Slicing approaches typically compute solutions in a few milliseconds). Note that VBM and AIM are not currently optimised for speed and rely on a Java interpreter written in VisualWorks Smalltalk.

[1] `http://www-static.cc.gatech.edu/aristotle/Tools/subjects/`

5 Related Work

Delta Debugging [5] aims at isolating a root cause of a program failure by minimising differences between a run that exhibits a fault and a similar one that does not. Differences between program states at the same point in both executions are systematically explored and minimised, resulting in a single "root cause" explaining why the program fails.

Model checking has recently been applied to locate faults [11; 3] by comparing abstract execution traces leading to correct and erroneous program states. Likely causes for a misbehaviour can be identified by focussing on traces that deviate only slightly from passing and failing test cases.

Error traces have also been applied to synthesise potential corrections of faulty programs, given a specification of the program's correct behaviour [12]. Symbolic evaluation is used to compare symbolic representations of program states as computed by the program versus states necessary to satisfy the post condition of the program. Differences in the predicates allow to heuristically synthesise replacement expressions correcting single faults in the program. The approach is able to provide corrections automatically only if a formal specification is given, which is not required for MBSD.

6 Conclusion

We introduced the basic principle of model-based software debugging and illustrated a model centred around abstract simulation of programs. Differences to previous approaches were outlined and results obtained using different debugging strategies were compared. Notable improvements over other approaches have been achieved, in particular for "difficult" programs where traditional debugging techniques do not perform well.

The Abstract Interpretation based model has been shown to achieve considerable improvement compared to slicing and previous model-based debugging approaches, with a reduction of the average number of explanations from 81% for static slicing to roughly 26%. Conversely, the model is computationally more demanding and will require optimisation to be applicable to mid-sized or large programs in an interactive setting.

Name	n	LoC	Comp	SSlice	DSlice	Exec	VBM	AIM	Time (s)
Adder	5	49	31	24.3	22	29.2	9.8	7.6	10.9
BinSearch	6	29	24.9	24.9	20.5	20.5	6	3.6	12.2
Binomial	7	80	45	32.4	27.4	32	(9)	9.8	72.6
BubbleSort	5	29	11	11	10	10	6	4	8
Hamming	6	48	38	38	30.9	30.9	11	5.2	218.3
Permutation	5	54	25	25	24.4	24.4	10.3	8	79
Polynom	9	103	66.9	39.8	28	37.5	14	12	584.4
SumPowers	7	27	16	14.8	11.4	12.3	10	7.4	3.8
TCAS	8	78	42	42	35	35.5	n/a	20	34.5
Total	6.4	55.5	33.7	27.5	22.7	25.8	9.5	8.9	115.5
%				81.6	67.4	76.6	28.2	26.4	

Figure 4: Debugging results

Current ongoing and future work includes (i) broadening fault modes to deal with a wider range of complex faults, such as assignments to incorrect variables, missing or swapped statements, etc., and (ii) providing simple user interaction for incremental specification of complex program behaviour. Recent advances in automatic abstraction and program verification [4] could lead the way to an MBSD engine with a fast but powerful adaptive abstract conformance checker.

References

[1] Claudia Böttcher. No faults in structure? How to diagnose hidden interaction. In *Proc. IJCAI*, 1995.

[2] François Bourdoncle. Abstract debugging of higher-order imperative languages. In *Proc. SIGPLAN Conf. PLDI*, pages 46–55, 1993.

[3] Sagar Chaki, Alex Groce, and Ofer Strichman. Explaining abstract counterexamples. In Richard N. Taylor and Matthew B. Dwyer, editors, *SIGSOFT FSE*, pages 73–82. ACM, 2004.

[4] Edmund Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. Predicate abstraction of ANSI-C programs using SAT. *Formal Methods in System Design*, 25(2-3):105–127, 2004.

[5] Holger Cleve and Andreas Zeller. Locating causes of program failures. In Gruia-Catalin Roman, William G. Griswold, and Bashar Nuseibeh, editors, *ICSE*, pages 342–351. ACM, 2005.

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

[7] James Corbett, Matthew Dwyer, John Hatcliff, Corina Pasareanu, Robby, Shawn Laubach, and Hongjun Zheng. Bandera: Extracting finite-state models from Java source code. In *Proc.ICSE-00*, 2000.

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

[9] Michael D. Ernst, Adam Czeisler, William G. Griswold, and David Notkin. Quickly detecting relevant program invariants. In *Proc.ICSE-00*, pages 449–458, 2000.

[10] Gerhard Friedrich, Markus Stumptner, and Franz Wotawa. Model-based diagnosis of hardware designs. In Wolfgang Wahlster, editor, *ECAI*, pages 491–495. John Wiley and Sons, Chichester, 1996.

[11] Alex Groce and Willem Visser. What went wrong: Explaining counterexamples. In Thomas Ball and Sriram K. Rajamani, editors, *SPIN*, volume 2648 of *Lecture Notes in Computer Science*, pages 121–135. Springer-Verlag, 2003.

[12] Haifeng He and Neelam Gupta. Automated debugging using path-based weakest preconditions. In Michel Wermelinger and Tiziana Margaria, editors, *FASE*, volume 2984 of *Lecture Notes in Computer Science*, pages 267–280. Springer-Verlag, 2004.

[13] Daniel Köb, Rong Chen, and Franz Wotawa. Abstract model refinement for model-based program debugging. In *Proc. DX'05*, pages 7–12, 2005.

[14] Cristinel Mateis, Markus Stumptner, and Franz Wotawa. A Value-Based Diagnosis Model for Java Programs. In *Proc. DX'00 Workshop*, 2000.

[15] Wolfgang Mayer and Markus Stumptner. Model-based debugging using multiple abstract models. In *Proc. AADEBUG '03 Workshop*, pages 55–70, 2003.

[16] Wolfgang Mayer and Markus Stumptner. Debugging program loops using approximate modeling. In *Proc. ECAI*, 2004.

[17] Wolfgang Mayer and Markus Stumptner. High level observations in java debugging. In *Proc. ECAI*, 2004.

[18] Wolfgang Mayer and Markus Stumptner. Model-based debugging – state of the art and future challenges. In *Workshop on Verification and Debugging*, 2006.

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

[20] Frank Tip. A Survey of Program Slicing Techniques. *Journal of Programming Languages*, 3(3):121–189, 1995.

[21] Dominik Wieland. *Model-Based Debugging of Java Programs Using Dependencies*. PhD thesis, Technische Universität Wien, 2001.

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