Evaluating Models for Model-Based Debugging

Wolfgang Mayer
University of South Australia
Advanced Computing Research Centre
Mawson Lakes, SA 5095, Australia
Email: mayer@cs.unisa.edu.au

Markus Stumptner
University of South Australia
Advanced Computing Research Centre
Mawson Lakes, SA 5095, Australia
Email: mst@cs.unisa.edu.au

***Abstract*—Developing model-based automatic debugging strategies has been an active research area for several years, with the aim of locating defects in a program by utilising fully automated generation of a model of the program from its source code. We provide an overview of current techniques in model-based debugging and assess strengths and weaknesses of the individual approaches. An empirical comparison is presented that investigates the relative accuracy of different models on a set of test programs and fault assumptions, showing that our abstract interpretation based model provides high accuracy at significantly less computational effort than slightly more accurate techniques. We compare a range of model-based debugging techniques with other state-of-the-art automated debugging approaches and outline possible future developments in automatic debugging using model-based reasoning as the central unifying component in a comprehensive framework.**

I. Introduction

Developing tools to support the software engineer in locating bugs in programs has been an active research area during recent decades, as increasingly complex programs require more and more effort to comprehend and maintain them. Several different approaches have been developed, using syntactic and semantic properties of programs and languages. While a number of different approaches (slicing [1], mutation testing [2], model checking [3], Delta-debugging [4], trace-based analysis [5] and model-based debugging [6]) have been introduced, each addresses a particular deficiency of previous approaches and no systematic comparison of the strengths, weaknesses and relationships between the individual debugging methods has been undertaken.

In this paper, we begin the task by summarising the different approaches to *model-based debugging* that have been proposed in the literature up to the current state of the art presented by our abstraction-based model, and compare empirical results with those obtained from "traditional" debugging aids. Focusing on imperative and object-oriented languages, we outline the main modelling paradigms (Section II) and explore relationships between different techniques based on empirical results (Section III). After investigating different fault assumptions, we explore synergies between complementary modelling techniques to obtain a debugging framework that is effective for a variety of different programs and faults (Section IV).

II. Model-based Debugging

In recent decades, many different techniques and tools have been proposed to attack the software maintenance problem, in particular the debugging problem. The focus has largely been on addressing the technical complexity of this task by reducing the size of programs under consideration using heuristics, controlled experimentation and pruning techniques, and on helping developers to control and investigate program executions. However, the intrinsically difficult task to create, maintain and confirm (or refute) suitable fault hypotheses based on observed correct and faulty program executions has remained largely the user's responsibility.

To reduce the cognitive load when debugging requires capturing semantics and intent of program fragments and their interaction. Using different "models" reflecting different aspects of a program under consideration, it becomes possible to support users in managing fault hypotheses in a way that is aligned closely with their view of the program.

Here, model-based debugging techniques aim to fill the gap by providing a comprehensive framework that relieves a user from manually matching the observed program execution(s) to his/her intuition of how the program should behave. By using models as abstractions facilitating the matching of symptoms to expected behaviour, intelligent debugging tools can be devised that assist users in controlling experiments to discriminate between different fault possibilities. In the following, the ideas underlying the model-based debugging paradigm are outlined.

Model-based software debugging (MBSD) is an application of *Model-based Diagnosis (MBD)* [7] techniques to locating errors in computer programs. MBSD was first introduced by Console et. al. [8], with the goal of identifying incorrect clauses in logic programs;[1] the approach has since been extended to different programming languages, including VHDL [6] and Java [11]. One of the key aspects of the approach is that –unlike verification techniques– MBSD does not intrinsically require a separate formal specification, but is intended to operate solely on the outcome of test cases (test vectors); a formal representation of the program (a "model") is built automatically and used not to determine whether errors exist, but to try and locate them in the actual code. (On the other hand, the MBSD framework is flexible enough to turn

[1]Earlier work by Shapiro [9] on interactive debugging of Prolog programs pursued similar goals, but was not founded in the formal theory of model-based diagnosis to reason about multiple inconsistencies. Bond [10] later showed that Shapiro's technique can be seen as a special case of the model-based approach.

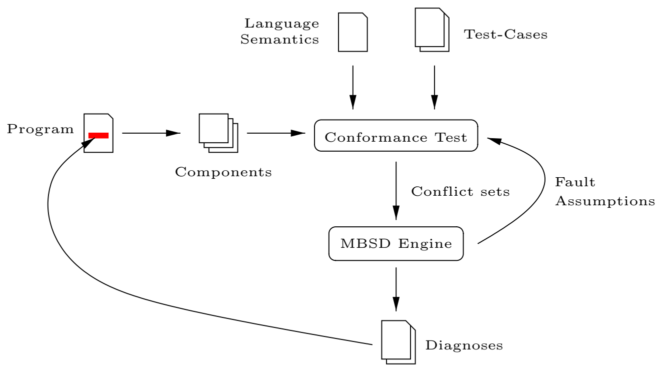

Fig. 1. MBSD cycle

this around and use external specifications to focus search, if available [12], [13].)

The basic principle of MBD (of physical systems) is to compare the *model*, a description of the correct behaviour of a system, to an *observed behaviour* of the system. Traditional MBD systems receive a 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 model 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 model and actual system: the model reflects the behaviour of the (incorrect) program, while test cases specify the correct result. Differences between the observed program execution and the result anticipated by test cases are used to compute model elements that, when assumed to behave differently, explain an 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*, where the component functions as specified in the program, or in one or more *abnormal modes*, with different behaviour. Intuitively, each component mode corresponds to a particular program modification. The main difference to Mutation Testing [2] is that MBSD typically does not assume a specific behaviour for the replacement program expression. Instead, a single fault mode typically subsumes an entire class of specific mutations. However, the MBSD framework is generic enough to allow to introduce fault modes reflecting specific mutations if desired. Let $P_{\{e_1/f_1,\dots,e_k/f_k\}}$ denote a program P modified such that expressions $e_1,\dots,e_k$ are modified to reflect fault assumption $f_1,\dots,f_k$. Initially, the set of fault assumptions F is empty ($k=0$).

The model components derived from P, a formal description of the semantics of the programming language and a set of test cases T are submitted to the conformance testing module to determine if the program reflecting the fault assumptions F is consistent with the test cases: $F=\{f_1,\dots,f_k\}$ is consistent with T if for all test cases $t\in T$ the program $P_{\{e_1/f_1,\dots,e_k/f_k\}}$ reflecting the fault assumptions does not contradict the test specification t. A program is found *consistent* with a test case specification if the program *possibly* satisfies the behaviour specified by the test case. Otherwise, if no trace in the modified program can lead to the anticipated test result, F alone is not a possible explanation and alternatives (or supersets of F) must be found.

In the case where 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 assess if an explanation is indeed valid. This is done by changing the fault assumptions associated with those program elements that are necessary to derive the conflicting result, as obtained from a simulation of $P_{\{e_1/f_1,\dots,e_k/f_k\}}$ on the input supplied by the test specification. This process iterates until one (or all) possible explanations have been found.

To illustrate the approach, consider the following code fragment and a test case that specifies that initially $a=1$, and after line 3, $x=2\wedge y=3\wedge z=10$ must hold:

$C_1:\ \ x = 2 * a;$
$C_2:\ \ y = x * 3;$
$C_3:\ \ z = 5 * x;$

Assuming that the program fragment is partitioned into three components C_1, C_2, C_3 such that each component corresponds to a single statement. The model of the program fragment is then derived from the source code, such that the model of each component simulates the effects of the corresponding statement. Here, the model is a representation of the (faulty) program that can be used to check consistency with a given test specification. In this example, an inconsistency between the model and the test specification is derived, since the model predicts that $y=6$ after line 3, while the test specification demands $y=3$. From this discrepancy the conflict $\{C_1,C_2\}$ is derived, which is used to refine the fault assumptions in the subsequent iterations of our framework. There are two cases to consider: if C_1 is faulty, the value of x is left undetermined by C_1, and if C_2 is incorrect, the model no longer predicts a precise value for y. It can be shown that a fault in component C_1 alone cannot explain the failed test, since $P_{\{C_1/x=?\}}$ still contradicts the test specification (the value for x is obtained by applying C_3 in backward direction). For C_2, the model is consistent with the test specification, indicating that the assignment to y may contain a fault. Indeed, if a is substituted for x in C_2 the test passes.

Formally our MBSD framework is based on Reiter's theory of diagnosis [7], extended to handle multiple fault modes for a component. MBSD relies on test case specifications to determine if a set of fault assumptions is a valid explanation, where each test case describes the anticipated result for an execution using specific input values.

While the overall process is intuitive, the precise test if a program variant conforms to a test specification is undecidable in general and must be approximated. Different abstractions have been proposed, ranging from purely dependency-based representations [6] to Predicate Abstraction [14]. In the following, the major modelling approaches are summarised briefly. A more detailed presentation is given in [15].

A. Dependency-based Models

Models derived from dependencies between program statements were among the first to be developed. Dependency models are constructed from dependencies between statements in a program, either by means of static analysis or through analysis of particular program executions [11]. An abstract model of the program is built that models the flow of correct and incorrect values through a program. Concrete values computed by a program are abstracted into a lattice where only *correct* and *incorrect* can be distinguished. Parametrised with boolean variables representing fault assumptions introduced by the diagnostic engine and transformed to a representation in propositional logic, the model is used to ascertain whether the program variant reflecting particular fault assumptions is possibly consistent with the test specification.

While the dependency-based models can be applied efficiently even to large programs ([6] successfully applied dependency-based models to VHDL-RTL programs of up to several MB of source code), object-oriented programs often result in many spurious explanations. This can be attributed to the coarse abstraction of the concrete semantics into abstract transitions computing only *correct* and *incorrect*. In particular for long chains of inter-dependent program fragments with no assertions or other observations in-between, the abstraction is too weak to determine that a particular candidate is inconsistent with the test specification and consistency must be assumed. Traditional purely dependency-based debugging techniques, such as Slicing, frequently lead to large result sets due to the same drawbacks. Indeed, it has been shown that MBSD based on control- and data-dependencies in a program has strong links to static and dynamic slicing [16].

B. Value-based Models (VBM)

The value-based approach [17] attempts to eliminate spurious explanations by strengthening the conformance test. The simple lattice used for dependency-based models is replaced with a version borrowed from constant propagation, where values that are precisely known are represented accurately; no information is derived if values are not uniquely determined given the fault assumptions and test specification.

Comparing values computed by a program reflecting fault assumptions with the values expected by a test specification, explanations that cannot fully account for all failed test cases can be eliminated. It can be shown that this approach is strictly more precise than the dependency-based formalism, leading to fewer spurious explanations [15].

While value-based conformance checking is more precise than dependency-based approximation, the models are computationally more expensive. The time required to check consistency in a dependency-based model is proportional to the program size; the complexity of the VBM additionally depends on the length of the execution trace, as in the worst case, the complete test execution must be simulated. Consequently, value-based models are only applicable for small programs (up to a few hundred components).

Concrete values allow to effectively diagnose programs where the control and data flow can be determined statically. For programs making heavy use of dynamically created data structures, loops and recursive methods, the approach is less effective, as a single fault assumption can render large parts of a program's control flow and program state unknown, collapsing the model and preventing the detection of inconsistencies, which again leads to very large candidate sets.

C. Abstraction-based Models

To overcome limitations inherent to previous approaches, the conformance tester can be amended to predict values of a program's state at an abstract level if the precise value cannot be ascertained. Two different frameworks have been proposed: (i) an extension to the value-based model using the Abstract Interpretation [18] framework, and (ii) a Predicate Abstraction based analysis [14].

On this basis we have defined the abstract interpretation based model (AIM; [15], [12]) to combine static analysis and test execution to approximate possible execution traces through a program. Instead of a unique execution trace, each combination of test case and fault assumption may induce a graph representing a set of consistent traces. Static analysis techniques are applied to those sections of an execution trace where precise values cannot be determined. Since fault assumptions in many cases affect only a small part of a program's state, the analysis is confined to small regions of a trace. Precise values derived in other parts of the execution trace can be exploited to replace complex relational abstractions with simpler non-relational approaches without suffering from degrading results. While traditional abstract interpretation is prohibitively expensive for many programs, combining concrete and abstract execution helps to keep the number of possible traces small. Results obtained by abstract interpretation (presented in the *Experimental Evaluation* section) demonstrate that results can be improved compared to the previous approaches.

The abstract interpretation approach has limited value for abstract execution traces where multiple related values are affected by a single fault assumption. In particular, fault assumptions affecting the dynamic allocation of data structures may have vast impact on the program execution. To eliminate such spurious explanations, additional information that cannot be derived from the program is required.

As the AIM may abstract away information that is vital to isolate faults in some programs, a different approach that incrementally refines the abstraction to suit the program has been proposed [14]. An additional analysis based on Predicate Abstraction and counter-example guided abstraction refinement [19] is performed if the value-based model cannot derive a conflict. We refer to the resulting model as *PAM*. Replacing the abstract interpretation engine with an iterative predicate abstraction process, model checking is able to improve results for selected programs [14]. While the initial results are encouraging, it is not clear whether the predicate refinement approach is effective in the presence of partially specified programs due to fault assumptions. Faults in loops and recursive programs are particularly difficult to debug using PAM, since the abstraction refinement process can be ineffective if the exact semantics of loops is not known due to fault assumptions introduced by the MBSD engine.

D. Debugging by Satisfiability Checking

A different approach using bounded model checking (BMC) has been presented in [20]. While not presented in terms of model-based debugging and diagnosis, the approach is quite similar in principle. The architecture of the system differs slightly from the one presented here, where the debugging engine and the conformance test are combined into a single model checking procedure based on propositional satisfiability. The program is transformed such that all execution paths up to a pre-set length can be represented without method calls and loops. The resulting program is encoded as a logic formula, where conditionals and free variables introduced in the compilation process model fault assumptions and their effects on program states. Values for these variables are not implied by the program behaviour, but are controlled by the model checker. An incremental search is performed to find consistent assignments to variables representing fault assumptions up to a given number of assumptions. Each set of variable assignments corresponds to a minimal diagnosis.

As a side-effect, this method is able to provide suggestions on how to modify the faulty program such that all test specifications are satisfied. While the repair algorithm is not complete in general, preliminary results suggest that useful hints can be obtained in many cases [20].

Comparing the BMC-based approach to PAM, it can be seen that the PAM may result in spurious explanations due to the abstraction used in the modelling process, while BMC does not share this undesirable property, provided that execution traces are explored in sufficient depth to cover the relevant program behaviour. If only a single failing test case is available and an explanation e returned by the BMC approach occurs only once in the trace, e is guaranteed to be non-spurious. In this case, BMC delivers results that are identical to the set of explanations obtained from the most precise consistency-based model (which cannot be constructed in general). Unfortunately, these conditions are not always satisfied in real programs, and it may be difficult to determine a suitable depth limit. Hence, further investigations in suitable approximations and tradeoffs between precision and model complexity are desired.

E. Worst-Case Analysis

Comparing the worst-case behaviour of different models, it can be seen that the models form a hierarchy, where the set of all executed program fragments resides at the top, and the most precise (but undecidable) model forms the least element. Slicing and dependency-based models are located near the top element (with models based on static approximation being less accurate than dependency-based models for simple faults), followed by the VBM. It can be shown that (static) dependency-based MBSD is equivalent to (static) Slicing when only single faults are investigated, and MBSD can lead to more precise results if multiple slicing criteria are used together. The VBM is more precise than dependency-based variants, since it can detect conflicts based on the semantics of the executed program rather than relying on control and data dependencies alone. However, this technique relies on simulation of the test runs and is thus slower than the dependency-based representations. Specialisations of the VBM using different heuristics and consistency assessment procedures form the hierarchy below, with the AIM lying between the VBM and the BMC model. While the BMC model can infer precise results if the path length of the correct program can be estimated from the faulty program and may require excessive computational resources, the AIM is more efficient and flexible but may suffer from over-approximation. The PAM and the other models specialising the VBM are mutually incomparable, due to different approximations being used. While abstraction refinement as used in the PAM can eliminate some spurious explanations derived by the other models, interferences between abstraction refinement and fault assumptions may limit its accuracy in the worst case. A more detailed presentation of the relationships between different MBSD approaches is given in [15].

The worst-case computational complexity of the various modelling techniques depends on the length of execution traces of the program and test cases under consideration and is difficult to quantify precisely. A detailed analysis of the complexity of different models can be found in [15]. The worst-case complexity of the hypothesis testing framework proposed by [7] is exponential in the number of components, but quadratic for single-fault explanations. Since we are predominantly interested in simple explanations, the complexity of managing hypotheses is dominated by the consistency test and is not a major concern in practice.

III. Experimental Evaluation

In this section, the performance of the individual models relative to each other is investigated empirically. While we aim at comparing a wide variety of models, the discussion in the remaining paper has been limited to models where either an implementation is available, or where results have been published using a common test set.

In the following, we show that the BMC-based approach provides the fewest false positives on average and is closely followed by the AIM. Dependency-based models generally perform poorly on our test set. Subsequently, we illustrate that while the BMC model is very effective for functional faults, that is, faults that do not imply assignments to incorrect variables, the AIM is shown to provide more general fault models that are effective for isolating structural faults (see Section III-D).

A. *Quality measure.*

To compare different debugging techniques, an unbiased measure of the quality of explanations is desired. The *Score* indicator has been proposed by [21] as an objective measure to compare the performance of different debugging approaches. Intuitively, it is defined as the fraction of a program that need *not* be investigated given the program fragments, R, implicated by a debugging tool such that the true cause, C, of a failure is guaranteed to be found [21]. The size of a program P is measured as the number of nodes in its *Program Dependence Graph* $pdg(P)$ that represents control and data dependencies between statements and methods. Starting at the set of implicated statements R, the program is traversed along the data and control dependencies in $pdg(P)$ in breadth-first order until an expression in C is reached:

$$score(P, R, C) = 1 - \frac{|BFS(pdg(P), R, C)|}{|pdg(P)|}.$$

$BFS(pdg(P), R, C)$ denotes the set of nodes in $pdg(P)$ that is traversed.

Similar to simpler indicators, for example the fraction of a program implicated by potential explanations, the score measure favours concise reports focusing on incorrect program elements, while large reports or reports not indicating a faulty program element are assigned a low score. In particular, if a report implicates only the nodes corresponding to the true cause, then $score(P, R, C)$ is close to one. Conversely, if the report implicates the entire PDG, the score is zero.

B. *Experimental Setup.*

A set of test programs has been used to evaluate the performance of different debugging approaches. The *TCAS* program was taken from the *Siemens Test Suite*[2], a test bench commonly used in the debugging community, and transcribed to Java. The remaining programs have been retained from earlier tests of the VBM and dependency-based models.

Table I summarises the experimental setup. For each program, n variants with different faults were created by seeding single and double faults, and programs were tested under different test specifications. The faults in the TCAS program were taken from the Siemens suite, while faults were randomly seeded in the remaining programs. The seeded faults were mainly mutations of expressions in assignment statements and conditional expressions, with infrequent occurrences of permuted statements and changed variables in the LHS of assignment statements.

All programs were analysed using between one and *Test* test cases. With the exception of *Adder* and *TCAS* (the former simulates a binary full adder circuit and the latter the resolution advisory component of a collision avoidance system used in aircraft), no significant difference between the single and the multiple test-case scenario was detected. This can be attributed to the structure of the selected programs, where a large fraction of statements are executed for all test cases.

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

C. *Experiment Results*

A summary of our results is given in Table I. The reported values are the average values over all program variants and available test cases, as well as the total average values. *LoC* denotes the number of non-comment, non-delimiter lines in the program's source code, *Tests* denotes the number of test cases available to compute explanations, *Comp* represents the number of diagnosis components used to construct explanations, *SSlice*, *DSlice* and *VBM* and *AIM* denote the scores obtained from single-fault explanations computed by the static and dynamic dependency models, the value-based model and the abstract interpretation-based approach, respectively. *Exec* contains the score obtained from the number of statements that are executed for a test run. Due to limitations of the implementation, some valid diagnoses listed for the AIM may not be included in the VBM's results, yielding a slight bias towards the VBM. The columns labelled *Time* denote the time in seconds required to compute explanations using the VBM and the AIM, respectively. Our prototype was implemented mainly to investigate the accuracy of different analysis techniques and has not been optimised for speed. We estimate that an advanced implementation that combines concrete execution with on-demand symbolic analysis akin to the one used in [22] could be applied to gain significant speedup and ensure scalability to larger programs. The reported results were obtained on an Intel P4 (1.2 GHz) with 512 Mb of RAM.

1) Static vs. Dynamic Dependency-Based Models: The results in Table I support the conclusion that that static slicing and dynamic slicing in many cases cannot improve much compared to the entire program and the statements executed in test runs. Similarly, dynamic slices often improve little compared to the executed statements, as all statements tend to contribute to the final result. Given the close relationship between dependency-based models and slicing [16], this result translates directly to dependency-based MBSD.

Dicing is not applicable for most programs, as correct test executions were not available. (*TCAS* is an exception and is accompanied with an extensive test suite; however, Dicing fails to implicate any program statement, as all incorrect program fragments are contained in both passing and failing test executions [21].)

The time required to compute explanations is well below 1 s for all dependency-based models and has been omitted from Table I.

2) VBM vs. AIM: Comparing VBM and AIM, it can be seen that the AIM improves over the VBM in most cases. In fact, cases where the VBM provides fewer explanations are due to explanations missed by the VBM.[3] Overall, the scores

[3]The implementation of the VBM makes quite strong assumptions on where in a program faults can be located; some faults in our test set violate those assumptions and cannot be detected adequately. This limitation is not shared by the other models.

for the AIM consistently outperform the VBM's, indicating that the improved approximation of the conformance test indeed helps to eliminate spurious explanations. The AIM provides significantly fewer explanations than slicing, but is computationally more demanding. The total time required to compute explanations using the AIM ranges from one to 3300 seconds.

3) AIM vs. BMC: Experiments presented in [20] indicate that the model checking approach derives slightly fewer explanations than the MBSD framework, but at the cost of scalability. The average number of potential faults for the 41 TCAS programs is 8 for the model checking approach, while the MBSD model returns 10.7 explanations on average. Translated to a score, the VBM results in 0.933, while BMC achieves scores exceeding 0.944 on average. This can be explained by the fact that the model checking approach simulates each path precisely, while the abstraction-based MBSD models rely on approximation. A more precise conformance test implies fewer explanations, but demands more resources.

For programs manipulating dynamic data structures, model checking is able to derive superior results due to the case distinctions performed by the model checker, but may exceed available resources even for simple programs. Suitable definitions of fault modes for expressions manipulating links between dynamically allocated data structures need to be devised to avoid implicating a large number of expressions. Griesmayer et. al [20] report that their model checker failed to deliver results on a program implementing a red-black tree with six nodes due to resource exhaustion.

Most of the BMC model's scores lie in the interval $[0.9, 1)$, which is slightly higher than typical AIM results (see Table I). This can partly be explained by the weaker abstraction we apply, but also by different model granularity. Comparing the diagnoses with the number of assumables instead of the size of the program, the two approaches lead to similar distributions. Comparing the execution times, it can be seen that the AIM is several times faster than BMC (which requires 1093 s on average) on the same processor, and as will be seen below, there are important problem classes where BMC was unable to compete due to memory exhaustion.

4) Difference-based Debugging: Table II compares the AIM to a spectra-based approach to fault localisation proposed in [21], where failing test executions are compared to "similar" correct executions to isolate potential faults. Columns two and three represent the distribution of scores computed for the AIM, whereas columns four and five represent scores derived using the spectra-based analysis (in particular, the nearest-neighbour-based analysis, which performed best in the cited study). For each TCAS program, a score was computed based on all available test cases. While a unique score for the model-based approach can be obtained by combining all available test cases in the analysis, the spectra-based approach derives a different score for each revealing test run. Different from the study presented in [21], column *NN* contains the *best* possible result that can be achieved using the nearest neighbour analysis rather than the distribution of average scores.

Model-based debugging achieves vastly better results than the spectra-based approach: where nearest neighbour based debugging typically cannot provide useful results for more than half of the considered programs, model-based fault localisation consistently achieves scores above 0.7. Although the MBSD technique delivers more consistent results, the maximum scores within the same category of the spectra-based approach are slightly higher. Therefore, if a large number of test cases is available that makes it easy to find a correct execution similar to a given faulty trace, difference-based approaches may provide equally good or better results than the model-based approach. Otherwise, model-based reasoning seems to provide better accuracy.

5) Cause Transitions: Another approach based on differences between passing and failing test cases is presented in [4]. The idea pursued in this work is to find "failure-inducing differences" between program states in a correct and a failing test. Relevant differences between states in different execution traces can be seen as vertices in a causal chain that may explain why one program execution failed whereas another execution succeeded. Bisection search techniques can be applied to refine causal chains and implicate additional expressions in a program. Faults are located by traversing dependencies between program statements starting at the last isolated cause transition. A comparison of cause transition based debugging and spectra-based fault localisation is presented in [4], identifying the cause transitions method as superior. Direct comparison with model-based techniques would be desirable, but individual results for the cause transitions method have not been published.

The distribution of scores obtained using model-based debugging is similar to the cause transition based approach *for the interval [0.9,1)* (the best cause transition method achieves such high scores for roughly 30% of all test programs). In approximately five percent of the test cases, cause transitions are able to pinpoint the location of a fault precisely – a result consistency-based approaches and spectra-based debugging cannot deliver. The remaining results obtained from cause transitions are mainly distributed in the interval $[0, 0.6)$; scores within $[0.6, 0.9)$ are distributed evenly and can be expected in roughly 20% of all cases. The comparison based on the distribution of results given here may not be fully accurate, as the wider scope of the analysis in [4] may lead to different results than TCAS only.

Overall, it seems that cause transitions are highly accurate for faults where program states close to the true fault can be inspected for both the correct and the failing program execution; otherwise, cause transitions seem to result in relatively low scoring explanations.

6) Distance Metrics: Another difference-based approach is presented in [23], where abstract traces derived by a model-checker are compared to isolate potential faults. The approach extends [24] such that execution traces similar to a known counter example are explored with the goal of isolating a minimal difference that induces a transition from a correct to a faulty execution trace. Column *explain* in Table III summarises

TABLE I
EXPERIMENTAL RESULTS

Name	n	LoC	Tests	Comp	SSlice	DSlice	Exec	VBM	Time (s)	AIM	Time (s)
Adder	5	49	8	31	0.510	0.551	0.408	0.837	10	0.865	16
Binomial	7	80	2	45	0.595	0.657	0.600	0.765	61	0.880	96
BinSearch	7	29	4	24.9	0.144	0.287	0.287	0.385	9	0.839	16
BubbleSort	5	29	4	11	0.621	0.655	0.655	0.655	7	0.869	12
Hamming	7	48	3	38	0.208	0.288	0.288	0.677	36	0.896	288
Permutation	5	54	2	25	0.537	0.548	0.548	0.578	10	0.859	104
Polynom	9	105	2	68.9	0.614	0.728	0.638	0.888	89	0.887	743
SumPowers	6	27	4	16	0.476	0.608	0.556	0.704	3	0.767	5
TCAS	41	78	131	42	0.000	0.857	0.811	n/a	n/a	0.933	377
Average	10.2	55.44	17.78	33.53	0.412	0.576	0.532	0.686	28	0.866	185

TABLE II
AIM VS. SPECTRA-BASED DEBUGGING

Score	AIM	%	NN	%
[0,0.1)	0	0	19	46
[0.1,0.2)	0	0	0	0
[0.2,0.3)	0	0	0	0
[0.3,0.4)	0	0	0	0
[0.4,0.5)	0	0	3	7
[0.5,0.6)	0	0	0	0
[0.6,0.7)	0	0	4	10
[0.7,0.8)	12	29	1	3
[0.8,0.9)	15	37	5	12
[0.9,1]	14	34	9	22

TABLE III
FAULT ISOLATION BY DISTANCE METRICS

TCAS	explain	Δ-slicing
v1	0.51	0.91
v11	0.36	0.93
v31	0.76	0.93
v40	0.75	–
v41	0.68	0.88

results for five TCAS programs obtained using the *explain* approach [24] and column Δ-*Slice* presents results obtained from the difference-based algorithm [23].

The AIM outperforms the *explain* approach on all five examples with equivalent results on version 31. Opposite results are obtained for the distance-based debugger, where the AIM consistently falls short by up to 16%. This is most likely a result of a more precise representation of the semantics of the underlying program, together with the requirement by the explain methods that an additional specification of the correct behaviour must be provided that is then used to search for correct executions that are close to a given counter example.

7) Finding Multiple Faults: Initial experiments with multiple seeded faults indicated that the number of explanations obtained from the VBM and AIM does not differ significantly from static slicing and does not warrant the additional overhead necessary for the complex models. Therefore, developing techniques that can deal effectively with multiple defects over a wide range of programs remains for future investigation.

D. Structural Faults

While the test suite described in the previous paragraphs occasionally contains assignment faults, missing or additional statements, a systematic evaluation of the models' ability to locate structural faults[4] is desired. In particular, since automated debugging techniques, such as Slicing, also tend to show low accuracy in localising faults that manifest through changed data dependencies between program fragments. We performed an experiment that specifically targeted such errors as compared to the more general evaluation in Section III-C.

For this purpose, 38 variants of the TCAS program were created, each containing a single assignment fault where the LHS of an assignment was changed randomly. The AIM was applied to each of the variants, using ten revealing test cases to drive the diagnostic process. In addition to the plain AIM applied in previous sections, we extended the model with additional fault models tailored to locate faults in assignment statements affecting variables and data structures reachable from local and globally visible variables. Specifically, we introduced the following fault modes:

RHS. Assume the right hand side expression of an assignment is incorrect. The value stored by an assignment is left unspecified, while the target variable remains unchanged. This is the basic fault model used in the evaluation described previously.

LHS. Each assignment statement assumed to exhibit the *LHS* fault may assign to any type-compatible local or global variable accessible at the assignment's label in the program. The right hand side expression of the assignment is assumed to be correct.

GSF (Global Structural Faults). In addition to the effects of mode *LHS*, the right hand side expression of each assignment in mode *GSF* is assumed incorrect.
To simulate complex structural faults in a method, each method call in mode *GSF* may assign a value to a variable visible at the call site. This fault model can be used to isolate a method as possible source of a structural fault, without modelling the method

[4]We use the term *structural fault* to refer to a fault that can only be explained by changing the target variable (its LHS) of an assignment statement, but cannot be explained by assuming that only the value of the assignment statement is changed.

TABLE IV
RESULT FOR ASSIGNMENT FAULTS

Model	Diagnoses	Located	Score	Time (s)
RHS	5	21 %	0.840	12
LHS	6	100 %	0.923	44
GSF	18	100 %	0.769	51

on a detailed level.

Table IV presents a summary of the results obtained under different fault assumptions. For each fault model, the median number of diagnoses per test case, the total percentage of test cases where the fault was among the implicated program elements, the average score and time in seconds for a test case are presented.

Using the *RHS* mode only, 8 out of 38 faults could be located; the results obtained with the remaining models included all seeded faults. While *RHS* returns fewer explanations, the overall score of the fault mode is lower, since seeded faults are not always correctly identified. On average, *RHS* and *LHS* derived between 5 and 6 explanations for each test case, while the *GSF* mode implicates roughly three times the number of statements. The time difference between *GSF* and the other modes can be attributed to the particular implementation of the model, which does not cache the set of accessible variables.

RHS and *LHS* produce a reasonable number of explanations, but *GSF* clearly is too general to be applied widely. However, the fault model may be useful when restricted to certain complex program elements, such as method calls. Then, faults that cover multiple statements or missing code within the contained program fragment may be isolated, which could not be achieved otherwise.

Initial experiments indicate that the BMC-based model is too precise to effectively deal with structural faults: we were unable to solve a single instance of the same debugging problem with BMC using the *LHS* fault mode. Given that the model checker has to explicitly examine all possible choices of target variables for each mode assignment, the state space becomes prohibitively large. None of the resulting SAT instances could be solved due to memory exhaustion.

The results indicate that specialised models are required to effectively locate faults inducing changes in a program's data flow. It is also apparent that unless the scope of a potential structural fault can be restricted to a narrow region of a program, precise models are of limited use. Instead, diagnoses computed using more coarse grained models on a higher level of abstraction can be used in a preliminary step to focus the modelling efforts. Furthermore, the abstraction level must be chosen carefully to ensure the search space remains small while not losing too much accuracy. This is important in particular for programs manipulating dynamic data structures, as then the set of target variables increases rapidly and explicit testing becomes infeasible.

IV. GENERALISED FAULT ASSUMPTIONS

Model-based approaches (like Model-Based Diagnosis of hardware) find the job hardest when dealing with faults that manifest as structural differences in the model, such as assignments to incorrect variables or missing program fragments. To improve upon previous MBSD tools, mechanisms to carefully select and apply fault modes to specific program fragments are required to obtain a practical framework. The most obvious way to do this, just as for hardware diagnosis, is to provide so-called *complementary models*. (In this section we use the term "model" to refer to a specification of the correct behaviour of a program and not to a formal representation of the original faulty program.) For example, in hardware diagnosis, bridge faults in an electrical circuit are diagnosed better when the spatial layout is known so that close-by wires can be identified.

In this section we discuss the incorporation of complementary models into the debugging process, in order to more precisely locate "difficult" faults that cannot be explained by changing the value in an assignment statement or in a conditional expression.

While tailored fault modes provide the means to isolate such faults, possible fault modes are numerous and must be applied in a controlled manner to avoid exponentially increasing the search space. Hence, additional information is required to do so. While formal specifications of certain aspects of the behaviour of a *correct* program, such as automata or modal logic formulae, are most beneficial, these are also difficult to obtain in practice in general. [5] In particular, in an ad-hoc debugging process, where relevant properties are not known beforehand or may change frequently, relying on (casual) debugger users to be able to provide any kind of formal specification is not realistic.

We propose to exploit certain invariants that in isolation provide little information about a program's behaviour, but may allow to focus on or exonerate program fragments that would otherwise be difficult to handle. Instead of forcing users to create strong formal descriptions, we rely on a pre-defined catalogue of abstractions that can be presented in a form suitable to casual users not trained in formal methods. For example, templates describing common properties of programs and data structures or simple dialog-based interaction may be used for presentation purposes.

Several ways have been studied to provide such complementary models, for example, explicit specification of automata [12] and dependency specifications [25]. Simple invariants derived from existing design documentation, such as type and cardinality constraints, UML sequence diagrams and OCL pre- and post conditions, can also be utilised to guide the search for structural faults [26]. An interactive refinement process using simple observations ("...this data structure should [not] have changed!") can further guide the debugger. Our aim is to develop a generic library of abstraction techniques designed to address specific shortcomings of existing MBSD models, focusing on dynamically allocated data structures.

Experiments using similar ideas based on automata theory have already demonstrated feasibility [13]. That approach

[5]This assumption holds for general-purpose programs. Safety-critical and embedded systems are exceptions where formal specification and model-driven development methods have gained momentum in recent years.

compares the behaviour specified as parametric state machines to the program implementing the specification. By applying generic mutations to the state machine models and matching the program behaviour to each model variant, specific indicators of potential faults can be isolated. Guided exploration of fault assumptions results in robust explanations that remain close to the original correct model. Efficient algorithms to compute explanations from state machines are available to isolate generic abstract fault modes that capture conceptual faults, such as missing updates. Overlaps between different fault assumptions have been shown to render the technique robust in the presence of unanticipated errors.

As the state machine based debugger operates mainly on the abstract model it does not necessarily directly support mapping the identified conceptual errors back to the source code level to pinpoint specific errors in the source code. Our approach aims to narrow this gap by focusing on the implementation level to isolate specific locations in the source code, while considering abstract models as a reference for a correct implementation. The state machine based debugger gains efficiency by operating on only a single type of model, whereas our framework gains versatility by combination of state machine based consistency assessment with static analysis methods and dynamic information obtained from concrete execution traces.

V. Conclusion

This paper has provided a overview of the state of the art in model-based software debugging representations: model-based diagnosis techniques are used to localise faults, starting out from the source code and test cases without an extra formal specification.[6] We provided the first multi-technique empirical comparison of source code fault localisation approaches. We have illustrated the relationship between dependency-based approaches that can be applied to large programs but can lead to coarse results to execution based and abstract frameworks, which lead to more refined approaches but are computationally more expensive.

The comparison shows that the MBSD approach outperforms all other approaches on the example suite in consistency, except the BMC approach which is prohibitively expensive even for very small structural errors. Spectra- and difference-based methods are fast but typically highly accurate only if many test cases are available, and do not provide the same level of consistency. While we do not claim that this study reflects the performance on arbitrary programs in all cases, the results obtained from our benchmark suite provide valuable insights into tradeoffs underlying the different approaches and may stimulate further research in automated debugging.

While MBSD has been shown to focus user attention to a small number of fault candidates, no single approach is likely to address arbitrary programs and faults effectively. Further research is needed to combine complementary techniques to obtain a debugging framework that can effectively handle a variety of programs.

We believe that the general debugging problem can only be solved by combination of different approaches. In this context, the MBSD approach has potential, with the MBSD engine as central component, unifying different debugging approaches through the common abstraction into components and fault modes, and its suitability for an iterative, interactive debugging session that reduces the number of diagnoses through incremental (abstract) specification of test cases [11].

Synergies between test execution and static analysis can be exploited to limit the scope of the analysis to relevant fragments of the execution trace, avoiding the state space explosion encountered for model checking while avoiding overly coarse approximation that can be encountered for static analysis based frameworks.

Our ongoing work aims at (i) developing specific modelling techniques to detect and locate specific faults, such as assignments to incorrect variables, missing or swapped statements, etc., with higher accuracy, (ii) integrating MBSD with spectra-based approaches to focus the debugging process [27], [28], and (iii) providing simple user interaction for incremental specification of complex program behaviour. Ultimately, extensions to existing approaches capable of isolating complex faults and missing code are desired to cope with general programs.

[6]While we agree that under certain conditions the source code of a system may not be available but an abstract specification will be [13], we do assume that in general the reverse will be the case.

References

[1] F. Tip, “A Survey of Program Slicing Techniques,” *Journal of Programming Languages*, vol. 3, no. 3, pp. 121–189, Sep. 1995.

[2] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf, “An experimental determination of sufficient mutant operators,” *ACM Transactions on Software Engineering and Methodology*, vol. 5, no. 2, pp. 99–118, 1996.

[3] S. Chaki, A. Groce, and O. Strichman, “Explaining abstract counterexamples,” in *SIGSOFT FSE*, R. N. Taylor and M. B. Dwyer, Eds. ACM Press, 2004, pp. 73–82.

[4] H. Cleve and A. Zeller, “Locating causes of program failures,” in *Proceedings of the International Conference on Software Engineering (ICSE)*, G.-C. Roman, W. G. Griswold, and B. Nuseibeh, Eds. ACM Press, 2005, pp. 342–351.

[5] H. He and N. Gupta, “Automated debugging using path-based weakest preconditions,” in *Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE)*, ser. Lecture Notes in Computer Science, M. Wermelinger and T. Margaria, Eds., vol. 2984. Springer-Verlag, 2004, pp. 267–280.

[6] G. Friedrich, M. Stumptner, and F. Wotawa, “Model-based diagnosis of hardware designs,” in *Proceedings of the European Conference on Artificial Intelligence (ECAI)*, W. Wahlster, Ed. Budapest: John Wiley & Sons, Aug. 1996, pp. 491–495.

[7] R. Reiter, “A theory of diagnosis from first principles,” *Artificial Intelligence*, vol. 32, no. 1, pp. 57–95, 1987.

[8] L. Console, G. Friedrich, and D. T. Dupré, “Model-based diagnosis meets error diagnosis in logic programs,” in *Proceedings of the* 13^{th} *International Joint Conference on Artificial Intelligence*, Chambery, Aug. 1993, pp. 1494–1499.

[9] E. Y. Shapiro, “Algorithmic program diagnosis,” in *Proceedings of the Symposium on Principles of Programming Languages*, 1982, pp. 299–308.

[10] G. W. Bond and B. Pagurek, “Declarative error diagnosis as consistency-based diagnosis,” in *Symposium on Logic Programming*, 1994, p. 673.

[11] D. Wieland, “Model-Based Debugging of Java Programs Using Dependencies,” Ph.D. dissertation, Technische Universität Wien, Nov. 2001.

[12] W. Mayer and M. Stumptner, "Model-based debugging using multiple abstract models," in *Proceedings of the* 5^{th} *International Workshop on Automated and Algorithmic Debugging, AADEBUG '03*, Ghent, Sep. 2003, pp. 55–70.

[13] C. Yilmaz and C. Williams, "An automated model-based debugging approach," in *ASE*, R. E. K. Stirewalt, A. Egyed, and B. Fischer, Eds. ACM Press, 2007, pp. 174–183.

[14] D. Köb, R. Chen, and F. Wotawa, "Abstract model refinement for model-based program debugging," in *Proceedings of the Sixteenth International Workshop on Principles of Diagnosis*, Monterey, California, USA, Jun. 2005, pp. 7–12.

[15] W. Mayer, "Static and hybrid analysis in model-based debugging," Ph.D. dissertation, School of Computer and Information Science, University of South Australia, Adelaide, Australia, Jul. 2007.

[16] F. Wotawa, "On the relationship between model-based debugging and program slicing." *Artificial Intelligence*, vol. 135, no. 1-2, pp. 125–143, 2002.

[17] W. Mayer and M. Stumptner, "Extending diagnosis to debug programs with exceptions," in *Proceedings* 18^{th} *International IEEE Conference on Automated Software Engineering*, Montreal, Canada, Oct. 2003, pp. 240–244.

[18] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixpoints," in *Proceedings of the Symposium on Principles of Programming Languages*, Los Angeles, 1977, pp. 238–252.

[19] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement for symbolic model checking." *Journal of the ACM*, vol. 50, no. 5, pp. 752–794, 2003.

[20] A. Griesmayer, S. Staber, and R. Bloem, "Automated fault localization for C programs," in *Workshop on Verification and Debugging (V&D'06)*, ser. Electronic Lecture Notes in Theoretical Computer Science. Seattle, USA: Springer-Verlag, Aug. 2006.

[21] M. Renieris and S. P. Reiss, "Fault localization with nearest neighbor queries." in *Proceedings* 18^{th} *International IEEE Conference on Automated Software Engineering*. IEEE Computer Society Press, 2003, pp. 30–39.

[22] D. R. Engler and D. Dunbar, "Under-constrained execution: making automatic code destruction easy and scalable," in *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, D. S. Rosenblum and S. G. Elbaum, Eds. ACM, 2007, pp. 1–4.

[23] A. Groce, "Error explanation with distance metrics," in *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004*, ser. Lecture Notes in Computer Science, K. Jensen and A. Podelski, Eds., vol. 2988. Springer-Verlag, 2004, pp. 108–122.

[24] A. Groce and W. Visser, "What went wrong: Explaining counterexamples," in *Proceedings of the 10th International Workshop on Model Checking Software (SPIN)*, ser. Lecture Notes in Computer Science, T. Ball and S. K. Rajamani, Eds., vol. 2648. Springer-Verlag, 2003, pp. 121–135.

[25] D. Jackson, "Aspect: Detecting Bugs with Abstract Dependences," *ACM Transactions on Software Engineering and Methodology*, vol. 4, no. 2, pp. 109–145, Apr. 1995.

[26] W. Mayer and M. Stumptner, "Model-based debugging with high-level observations," in *IFIP International Conference on Intelligent Information Processing (ICIIP)*, Beijing, Oct. 2004.

[27] P. Zoeteweij, R. Abreu, R. Golsteijn, and A. J. C. van Gemund, "Diagnosis of embedded software using program spectra," in *Proceedings of the 14th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS'07)*. IEEE Computer Society Press, Mar. 26 – 29 2007, tucson, AZ, USA.

[28] W. Mayer, R. Abreu, M. Stumptner, and A. J. van Gemund, "Prioritising model-based debugging diagnostic reports," in *Proceedings of the 19th International Workshop on Principles of Diagnosis*, Blue Mountains, Sydney, Australia, Sep. 2008.