Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
Electronic Notes in Theoretical Computer Science 174 (2007) 61–82
www.elsevier.com/locate/entcs

Model-Based Debugging – State of the Art And Future Challenges

Wolfgang Mayer[1] Markus Stumptner[2]

Advanced Computing Research Centre
University of South Australia
Adelaide, Australia

Abstract

A considerable body of work on model-based software debugging (MBSD) has been published in the past decade. We summarise the underlying ideas and present the different approaches as abstractions of the concrete semantics of the programming language. We compare the model-based framework with other well-known Automated Debugging approaches and present open issues, challenges and potential future directions of MBSD.

Keywords: Automated Debugging, Model-based Reasoning

1 Introduction

Model-based software debugging (MBSD) is an application of *Model-based Diagnosis (MBD)* techniques to debugging computer programs. Model-based diagnosis was first introduced by [14] and subsequently refined by [34]. Diagnosis was initially focussed on locating faults in physical systems, in particular faulty gates in electronic circuits. MBSD was first introduced by [11,5], with the goal of identifying incorrect clauses in logic programs; the approach has since been extended to different programming languages, including VHDL [16] and Java [27]. Before describing the MBSD approach in detail, the underlying principles of MBD are summarised.

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

[1] Email: mayer@cs.unisa.edu.au
[2] Email: mst@cs.unisa.edu.au

doi:10.1016/j.entcs.2006.12.030

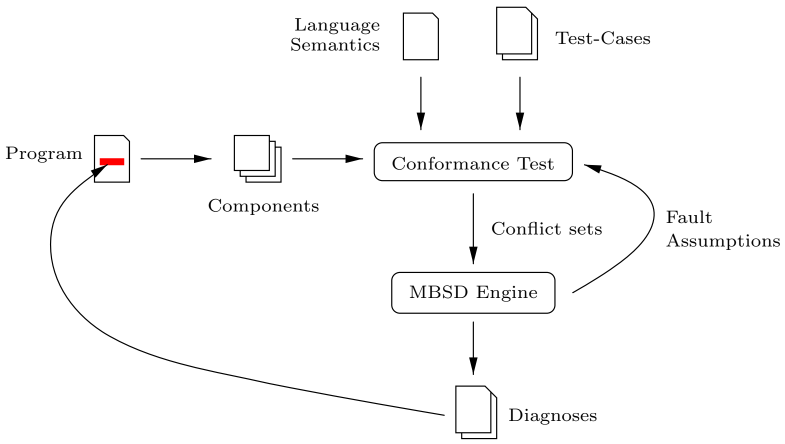


Fig. 1. MBSD cycle

to identify components that, when assumed to deviate from their normal behaviour, may explain the observed behaviour.

Translated to the software domain, substituting the program for the concrete system and observing its behaviour on a set of test cases seems possible. This turns out to be difficult in practice, as a formal description of the correct program is required to detect discrepancies. Current practice in software engineering shows that formal models are rarely provided and if available, they often suffer from maintenance problems where changes in the desired functionality of the program are not reflected in the formal models. Further, formal models typically do not cover the complete behaviour of the system, but are restricted to a particular property of the program.

The following section gives a brief introduction to the MBSD principles and basic definitions. Sections 2.4–4 introduce different models taken from the literature. Section 5 analyses the relationships between the models, followed by a discussion of related work in Section 6. A number of potential future developments of MBSD are raised in Section 7.

2 Model-based Software Debugging (MBSD)

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 result anticipated result. Differences between the values computed by the program and the specified 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 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.[3] 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 determines 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 is found inconsistent, a set of components necessary to derive the inconsistency is computed and passed to the MBSD engine. The MBSD engine computes possible explanations in terms of mode assignments to components and invokes the conformance testing module to determine if the explanation is indeed valid. This process iterates until one (or all) possible explanations have been found.

2.1 What is a Debugging Problem?

MBSD relies on test case specifications to determine if a set of fault represents is a valid explanation. It is sufficient to assume that a nonempty set of test cases is given, each test case describing the anticipated result for a test run using specific input values.

Definition 2.1 A *test case* for a program P is a pair $\langle In, Out\rangle$ where In and Out specify the input values and expected result of P.

Throughout this work, it is assumed that the set In completely specifies the program's initial state, whereas Out may be partially specified.[4] In the following, In and Out denote both the assertions provided by a test case and the set of states satisfying the assertions. Test cases can be generalised to allow assertions at arbitrary labels.

Definition 2.2 A *Debugging Problem* is a tuple $\langle P, \mathbf{T}, \mathbf{C}\rangle$ 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 are 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.

Example 2.3 The program in Figure 2 computes a linked list containing the first n elements of the well-known *Fibonacci* sequence. The program is partitioned into ten components, each representing a statement. The expected result at label *end* when run with input $n = 5$ is list $\mapsto [1, 1, 2, 3, 5]$. Represented as an assertion, the desired result is

```
assert@end list.value==1 && list.next.value==1
      && list.next.next.value==2 && list.next.next.next.value==3
      && list.next.next.next.next.value==5 && list.next.next.next.next.next==nil
```

The program contains a fault at label 9: a ←a−b is computed, causing the incorrect result list $\mapsto [1, 1, 0, -1, -1]$.

[3] The main difference to Mutation Testing [32] is that our modifications to the program are not necessarily executable, but may be at an abstract level subsuming multiple concrete expressions.

[4] The assumption may not be necessary for all models discussed herein. For the models based on execution of the program, the initial state must be completely specified to obtain a unique execution trace.

```
class FibList {
  int number;
  FibList next;
  FibList init(int n, FibList fl) {
    number ← n;
    next ← fl;
    return this;
  }
}
```

```
int a ← 1;
int b ← 0;
int list ← nil;
while (n > 0) {
    b ← a + b;
    a ← a - b;  “correct is a ← b − a;”
    list ← new FibList();
    FibList ignore ← list.init(b, list);
    n ← n − 1;
}
end
```

Fig. 2. Fibonacci program

2.2 The MBSD engine

To compute explanations once failing test execution has been detected, a version of Reiter's consistency-based diagnosis framework [34,17] is employed. A set of fault assumptions $\Delta \hat{=} \{C_1, \ldots, C_k\}$ 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.

Each fault assumption generated by the MBSD engine corresponds to a modification of the program. A component C representing part of the original program's source code is replaced with a component C' that specifies a relaxed form of C or does not specify any specific behaviour. [5]

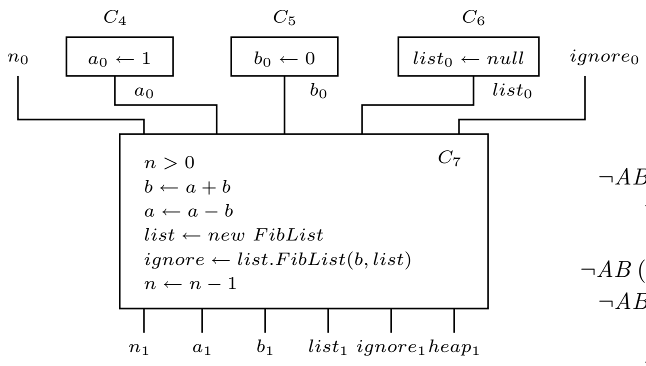
In case a failing test case is encountered, the MBSD engine determines a set of program components ("conflict set") necessary to imply the failure. In the simplest form, a variant of Slicing [37] can be applied to compute the set. In general algorithms such as the Resolution calculus or constraint-based systems [23], are utilised to compute small conflicts. Using Reiter's algorithm, explanations are computed from conflicts.

2.3 Conformance testing

The conformance testing module decides whether a variant, P', of program P, conforms to the behaviour anticipated by the test case specifications. P' is derived from P by applying fault assumption, Δ, obtained from the MBSD engine. Transformations of P into P' are model specific and are presented in the following sections.

Definition 2.4 Fault assumption Δ is *consistent* with a set **T** of test case specifications if and only if for all test cases, it cannot be derived that all program executions satisfying the model of the program (altered to reflect Δ) violate the test case.

[5] Different models apply different strategies to determine the variables and fields affected by an abnormal component. Here, it is implicitly assumed that only variables present in C are affected. This restriction will be relaxed in Section 3.9.



$$\neg AB\,(C_4) \rightarrow ok(a_0)$$
$$\neg AB\,(C_5) \rightarrow ok(b_0)$$
$$\neg AB\,(C_6) \rightarrow ok(list_0)$$
$$\begin{gathered}\neg AB\,(C_7) \wedge \neg AB\,(C_8) \wedge \neg AB\,(C_9) \wedge \\ \neg AB\,(C_{12}) \wedge ok(a_0) \wedge ok(b_0) \wedge \\ ok(n_0) \rightarrow ok(v_1) \quad v_i \in \{a_1, b_1\}\end{gathered}$$
$$\neg AB\,(C_7) \wedge \neg AB\,(C_{12}) \wedge ok(n_0) \rightarrow ok(n_1)$$
$$\begin{gathered}\neg AB\,(C_7) \wedge \neg AB\,(C_8) \wedge \neg AB\,(C_9) \wedge \\ \neg AB\,(C_{10}) \wedge \neg AB\,(C_{11}) \wedge \\ \neg AB\,(C_{12}) \wedge ok(a_0) \wedge ok(b_0) \wedge \\ ok(n_0) \wedge ok(list_0) \rightarrow ok(w_i) \\ w_i \in \{list_1, ignore_1, heap_1\}\end{gathered}$$

Fig. 3. Dependency model of the FibList program

2.4 An optimal consistency-based model

Using symbolic execution of the program to decide if a program satisfies all test specifications yields an optimal MBSD model. Unfortunately, this model is not computable in general and approximations have to be introduced.

Example 2.5 The program in Figure 2 clearly violates the test case given in example 2.3: the program computes value 0 for list.next.next.value, while the assertion requires value 2 to be satisfied. From the execution trace it is determined that all components except [**return** this]$_3$ are necessary to derive the incorrect value for the instance variables.

The MBSD engine subsequently creates fault candidates for each component in the conflict and re-examines the test case. Assume component $[a \leftarrow a - b]_9$ is selected and line 9 in the program is replaced with the more general variant $[a \leftarrow \Box]_9$. The placeholder $\Box$ is not part of the original program syntax, but is introduced by the conformance tester and represents an unknown expression.

The execution proceeds as before until label 9 is reached, and the value of variable a is set to an (at this point unknown) value denoted by ξ_1. The execution continues and ξ_1 is stored in the newly created list node ($[list \leftarrow \ldots]_{10}$). The loop iterates until the condition becomes false and program terminates in a final state where list $\mapsto [\xi_1, \ldots, \xi_5]$, with all ξ_i undetermined.

It is easy to see that the final state of the trace in example 2.5 is consistent with the assertion given in example 2.3 if the ξ_i are assigned the values specified in the assertion. Therefore, component 9 is a potential cause of the failing test execution. Repeating this process for the remaining candidate explanations $[\cdot]_1$ to $[\cdot]_{12}$, all but $[\text{number} \leftarrow \text{n}]_1$, $[\text{b} \leftarrow \text{a} + \text{b}]_8$, $[\text{a} \leftarrow \text{a} - \text{b}]_9$, $[\text{list} \leftarrow \textbf{new}\ \text{FibList}]_{10}$ and $[\text{ignore} \leftarrow \text{init(b,list)}]_{11}$ are exonerated.

3 Dependency-based modelling

Models derived from dependencies between program statements were among the first to be developed. Although the term "model-based software debugging" had not been invented, the approach presented in Kuper's thesis [26] was based on similar ideas and could be considered model-based. Later, MBSD was applied to Prolog programs [11,5] and to programs written in the hardware description language VHDL [16,39], knowledge bases for automatic configuration systems [15] and imperative and object oriented languages [27,38]. In the following, the focus is on the work using Java.

Wieland's thesis [38] presents three models for Java programs, each using the same modelling language and reasoner but applying different model building strategies: The *Execution Trace based Dependency Model (ETDM)*,[6] the *Detailed Dependency Model (DDM)* and the *Summarised Dependency Model (SDM)*. The dependency models utilise a common model representation, while differences between the models are found in the approximations of dependencies and heap data structures.

3.1 Structural abstraction

Dependency models are constructed from dependencies between statements in a program P, which has been transformed into *Static Single Assignment Form (SSA)* [13]. The SSA form is a program representation where each variable is assigned exactly once and computing dependencies between statements becomes trivial.

A statement S_i depends directly on a statement S_j if there exists an execution such that S_j precedes S_i and the computation of the effect of S_i requires a value computed by S_j ("data dependency"), or the outcome of S_j may cause S_i to be (un)reachable ("control dependency"). This is sufficient for detecting faults not involving the use of incorrect variables and will be relaxed in Section 3.9. The model of an entire method is obtained by composing dependencies of the method's statements. The resulting dependencies are essentially the same as dependencies obtained by applying a slicing algorithm [37,41].

Dependencies are transformed into a component-connection model, where components correspond to program statements and connections correspond to dependencies between the statements. Each components $C \in \mathbf{C}$ has a set of inputs $in(C)$ and a set of outputs $out(C)$, corresponding to all variables and locations potentially used and modified by statements represented by C.

Example 3.1 Figure 3 depicts a graphical representation of the components and connections created for the program in Figure 2. The loop structure has been collapsed into a single component, with different dependencies between input and output variables. For example, variable a_1 depends on the values of n_0, a_0, b_0 and the fault assumptions of C_7, C_8 and C_9. The computation of these dependencies is discussed in more detail in the following section.

[6] Throughout [38] the term "Functional Dependency" is used to refer to data flow and control dependencies between statement. We prefer to use the single word "Dependency" instead.

As the exact modelling of heap data structures depends on the models used, the connections representing objects are omitted and represented as a single connection, *heap*, instead.

3.2 Dependency representation

The component-connection model is compiled into sentences in propositional logic, abstracting from the concrete semantics and concrete values. The model only expresses whether the value of a variable v is correct, $ok(v)$, or incorrect, $\neg ok(v)$. The behaviour of a primitive component C is reduced to preserve correctness if all its inputs $in(C) = \{v_{i_1}, \ldots, v_{i_m}\}$ provide correct values and C is itself correct. In this case, the component's outputs $out(C) = \{v_{j_1}, \ldots, v_{j_n}\}$ are also correct:

$$\neg AB\,(C) \wedge ok(v_{i_1}) \wedge \ldots \wedge ok(v_{i_m}) \rightarrow ok(v_{j_1}) \wedge \ldots \wedge ok(v_{j_n}).$$

Otherwise, C's effect is potentially incorrect. Rules are formed such that $ok(v_k)$ is not predicted by C for any potentially affected variable v_k.

The model of the entire program is obtained by forming the conjunction of all sentences. Abstractions of heap data structures, *"locations"*, are treated similarly to variables. Further discussion of heap locations is provided in sections 3.4–3.6.

3.3 Conflict extraction

To obtain the test outcome for a test $T \hat{=} \langle I, O\rangle$, the program is executed and the values computed are compared with the ones specified in O. For each variable v_k, if the value obtained from the execution agrees with the value specified in O, the corresponding model proposition $ok(v_k)$ is set to *true*, otherwise the negated proposition is asserted. Assertions $ok(v_k)$ are added for all variables v_k specified in I, denoting that all inputs provided by $\mathbf{T}$ are correct. The fault assumptions Δ are introduced into the model though literals $AB(C)$, $C \in \mathbf{C}$. For all remaining components $C' \in \mathbf{C} \setminus \Delta$, $\neg AB(C')$ is asserted.

To identify components explaining failed test assertions, a linear-time unit resolution prover (LTUR) is applied by to derive inconsistencies between the logic representation of the model and the facts representing the test outcome. A conflict has been found if there is a variable v where both $ok(v)$ and $\neg ok(v)$ can be derived. The components contributing to the derivation of the two conflicting literals form a conflict.

3.4 ETDM

The *Execution Trace based Model (ETDM)* is constructed from dependencies between statements in a *single execution* of the (faulty) program P on the inputs, I, specified by a test case. The program is run starting in state I and the execution trace is subsequently transformed into SSA form and used as the basis for the dependency computation. As only the single executed path is considered, the model

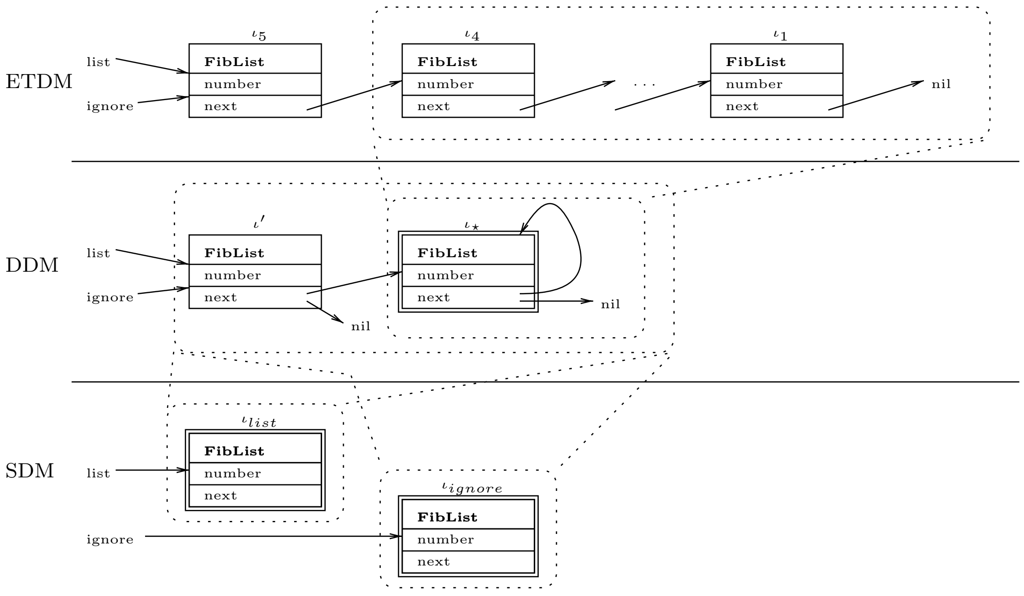


Fig. 4. Heap abstractions for ETDM, DDM and SDM

represents variables and dynamic data structures precisely and ignores dependencies between statements that have not been executed. The model is free of spurious dependencies, as only dependencies which actually arise during execution are considered. Heap data structures are treated as normal variables and do not require special consideration. The drawback is that executing the program adds additional overhead for long-running programs, as the program must be executed once for each test case.

Example 3.2 Figure 4 presents the heap abstractions at statement labelled 12 obtained for different models from the program and test case given in Section 2.1. For each loop iteration, a separate location ι_i, $i \in \{1, \dots, 5\}$, representing an instance of type FibList is created. The values of all variables of reference type are known precisely and no approximation is necessary.

3.5 DDM

In contrast to the ETDM, the *Detailed Dependency Model (DDM)* does not rely on program execution to build the model representation. Instead, static analysis techniques are applied to analyse the program with respect to control and data flow, including dynamically created data structures. All possible executions must be considered.

The program is analysed in a preliminary step, partitioning the dynamically allocated data structures into separate abstract *locations*.

Definition 3.3 A location represents one or more objects that are potentially allocated by the program at runtime. A location potentially representing more than a single object is a *summary location*. Each location contains type and instance variables of the represented objects.

In Figure 4, locations potentially representing more than one object are depicted with double borders.

Example 3.4 Using the simple heap analysis presented in [12] the following heap partitions can be obtained from the program in Figure 2 at label 12 (Figure 4): both variables list and ignore reference the same unique object (an instance of FibList). The value of the object's next instance variable may be nil, or may reference one of a number of different objects, represented as a summary location. However, it is not known which object is referenced, nor if the structure reachable through the next instance variable is cyclic or even different from the object containing the reference. [7]

In case a location corresponds to a single object, there is no difference to the modelling of a regular variable. For summary locations, the logic representation must be extended to account for the fact that it is not known which object is being referenced. For a location $\iota_\star$ representing multiple concrete objects, $ok(\iota_\star)$ and $\neg ok(\iota_\star)$ may only be asserted if it is guaranteed that this fact holds for *all* concrete objects represented by $\iota_\star$. The model may contain spurious dependencies due to ambiguity in control flow and due to approximation of heap data structures during the preliminary analysis phase.

Example 3.5 The heap partitioning used in the DDM in Figure 4 is more concise than the one used in the ETDM, as heap locations $\{\iota_1, \dots, \iota_4\}$ are no longer represented explicitly, but are summarised into a single proposition $\iota_\star$.

The model is less demanding than the ETDM in memory and runtime, but may lead to a larger number of potential explanations.

3.6 SDM

Large programs (several MB of source code) require even more abstract models to make debugging feasible. The *Summarised Dependency Model (SDM)* further abstracts from the DDM and represents heap data structures as abstract locations corresponding to the variables pointing to that location. The model creates a single location that represents the *entire* data structure referenced by each program variable. The coarse approximation of heap structures allows more efficient reasoning, but introduces imprecision caused by aliasing and summary locations. To provide a safe approximation, dependencies must be added to ensure the model provides a safe abstraction.

Example 3.6 The heap abstraction of the SDM depicted in Figure 4 represents the dynamic data structures referred to by the two variables list and ignore as two locations, ι_{list} and ι_{ignore}. Note that two abstract locations have been created,

[7] Advanced shape analysis approaches such as [35] exist that allow to deduce more information, such as the shape of data structures and whether nodes are being referenced from multiple locations.

both representing *the same* concrete objects. If the program included an instruction modifying a location pointed-to by one of the two variables, the dependency representation of the statement would have to be amended to reflect the same modification to the other location.

3.7 Modelling hardware descriptions

[39] present a dependency-based model designed to locate faults in a subset of the VHDL programming language. The semantics of VHDL are based on the notion *concurrent processes* which may be triggered by changing activations of *signals* and may in turn change the activation of other signals. Dependency models of a VHDL program express concurrent processes as components and the signals used and changed by each process as connections. Similar to the DDM and the SDM, cyclic dependency graphs are collapsed into a single node.

Starting at signals observed to be incorrect (either through manual inspection or by using an automatic comparator tool [16]), the signals and processes potentially contributing to an incorrect result are identified. Explanations can be isolated effectively through combination of conflicts and through fault models expressing faults commonly found in VHDL programs.

3.8 Plan-based modelling (PBM)

Kuper [26] introduces an interactive debugger for LISP programs, Debussi, which is also based on dependencies between expressions. The model constructs a simple plan (essentially a dependency graph), representing the expressions and subexpressions computed during program execution and their interdependencies. The reasoning strategy used to identify potentially incorrect expressions is based on constraint suspension [14] to exonerate conditional expressions that cannot be responsible for a fault. Simple heuristics to exclude unlikely explanations are used to filter candidate explanations.

Kuper follows a hierarchic, interactive process where the user is prompted to judge if a particular expression obtained at runtime is correct or not. Based on the outcome the system eliminates explanations that conflict with the user's answer until a single explanation has been isolated. In case the identified expression represents a function application, a new debugging problem using the function's body is spawned.

3.9 The Abstract Dependency-based Model (ADM)

The models discussed previously are able to locate faults that involve incorrect expressions, but do not provide the means necessary to locate faults involving missing statements or assignments to wrong variables. To locate such faults, complementary models not derived from the program are necessary. Ideas introduced in [36,33] provide a first attempt at exploiting simple specifications [21] defining the relationships between input variables and result variables of a method.

Indications of potential faults are obtained by comparing dependencies induced by the program with dependencies obtained from the test case specification. These hints are subsequently used to modify the program to contain $\Box$ tokens in the left hand side of assignment expressions. [33] use a constraint-based approach to compute suitable sets of variables. The algorithm limits the ADM to detecting and diagnosing *missing* dependencies.

4 The family of value-based models

While the dependency-based models presented in the previous section are lightweight tools that can be applied efficiently even to large programs [16], for object-oriented programs they often return many spurious diagnoses. Closer analysis revealed that insufficient reasoning capabilities in the conformance tester are the major contributor to false positives. In particular, the coarse abstraction of the concrete semantics into abstract transitions computing $ok(\cdot)$ and $\neg ok(\cdot)$ is too weak to determine that a particular candidate is inconsistent and consistency must be assumed.

A possible remedy is to strengthen the model representation such that conflicts can be derived in some of these cases, excluding spurious explanations.This section surveys some of the approaches to strengthen the model representations and conflict extraction procedures.For brevity, only differences to the dependency models are described. As before, our focus is on Java models, glossing over similar developments for VHDL programs [40].

4.1 The Value-based Model (VBM)

A direct extension to the dependency-based models was proposed in [27], replacing the $(\neg)ok(\cdot)$ literals with concrete values computed by the program. The model computes concrete values (or no value in case not all required input values are received). The model essentially simulates the program in case all values necessary to compute a new value are known and does not predict any value otherwise. Inconsistencies are derived when two differing values are derived for the same model variable.

Example 4.1 Figure 5 presents the logical model derived from the program in Figure 2. The loop is no longer represented as a single component, but consists of a hierarchical structure containing models of the loop's condition and body. These in turn contain models describing the individual statements and called methods. A literal $a.b.c$ denotes instance variable c of object b in a program state a. The model enforces the $=$ operator only if the at least one of the two expressions evaluates to a concrete value. Otherwise, the model does not predict any value and thus assumes consistency.

Copies of the models of the loop's body and condition are instantiated dynamically, depending on the outcome of a simulation of the condition's model for the preceding iteration. This process repeats until the model of the condition does not imply *true*. If the condition implies *false*, the values of the variables derived by the preceding model are unified with the loop component's outputs. Otherwise, the number of iterations cannot be determined and the loop cannot predict values on

$$\begin{aligned}
\neg AB\,(C_4) &\rightarrow a_0 = 1\\
\neg AB\,(C_5) &\rightarrow b_0 = 0\\
\neg AB\,(C_6) &\rightarrow list_0 = \text{nil}\\
\neg AB\,(C7) \wedge \neg AB\,(C_8) &\rightarrow b_i = a_{i-1} + b_{i-1}\\
\neg AB\,(C7) \wedge \neg AB\,(C_9) &\rightarrow a_i = a_{i-1} - b_i\\
\neg AB\,(C7) \wedge \neg AB\,(C_{10}) &\rightarrow list_i = \iota_i \wedge \qquad (\iota_i \text{ fresh in } heap_{i-1})\\
\forall_{\iota \neq \iota_i, k} heap'_i.\iota.k = heap_{i-1}.\iota.k &\wedge heap'_i.\iota_i.value = 0 \wedge heap'_i.\iota_i.next \mapsto \text{nil}\\
\neg AB\,(C7) \wedge \neg AB\,(C_{11}) \wedge \neg AB\,(C_1) &\rightarrow heap_i.list_i.value = b_i \wedge\\
&\qquad \forall_{\iota \neq \iota_i, k} heap_i.\iota.k = heap'_i.\iota.k\\
\neg AB\,(C7) \wedge \neg AB\,(C_{11}) \wedge \neg AB\,(C_2) &\rightarrow \forall_{\iota \neq \iota_i, k} heap_i.\iota.k = heap'_i.\iota.k \wedge\\
&\qquad heap_i.list_i.next = list_{i-1}\\
\neg AB\,(C7) \wedge \neg AB\,(C_{12}) &\rightarrow n_i = n_{i-1} + 1
\end{aligned}$$

Fig. 5. Logical model of the FibList program for the VBM

its outputs. A formal description of this propagation process is given in [27].

The VBM can effectively handle a variety of programs for which the dependency-based models provide little advantage compared to slicing. For data structures where different instance variables are processed by different sections of the program, the heap-partitioned model provides much improved explanations. However, soundness of the results depend on the absence of variable faults on the left hand side of assignments. The key advantage of the VBM compared to dependency-based models is its ability to approximate the control flow of programs more precisely and to derive contradictions even for branch-free executions.

4.2 The Exception Model (EM)

The VBM is valid for programs with simple control flow, but is not expressive enough to deal with arbitrary control flow such as structured exception handling and non-local branches.

The EM removes this limitation by changing the model construction to use the *Static Single Information Form* (SSI) [3], a bidirectional representation designed to support forward and backward reasoning. The translation of the component model is extended to incorporate the elements introduced by the SSI form, but remain otherwise unchanged compared to the plain VBM.

The conflict extractor can be described as follows: The model is represented as a flow-graph and is partitioned into regions with a single entry point. In case an inconsistency is detected in a region, the entire region is marked inconsistent and a different path must be followed. Once the region containing the entry point of the program is marked inconsistent, there is no consistent execution and the set of components associated with the outermost region is returned as conflict.

4.3 The Abstract Interpretation-based Model (AIM)

An inherent problem common to most previously presented models is the fact that the model must represent *all* possible executions of the program. For object-oriented programs featuring polymorphism and side-effects, a conservative approximation of the call graph and data flow must be computed, leading to potentially large models with tightly interconnected components.

The *Abstract Interpretation-based Model* AIM [28] shifts the modelling approach from a static to a dynamic one, integrating the structural modelling phase with the conflict extraction. The concrete semantics of the program is replaced with an interval lattice to approximate program states that cannot be determined precisely. Fault assumptions are applied to P and a model is generated dynamically, constructing only the feasible paths. A sequence of forward and backward analyses [6] is applied to eliminate paths that do not lead to the results specified by the test cases. A conflict is detected if no feasible path remains.

The modelling process is more efficient as only feasible execution paths are generated. Faults involving assignments to the wrong variables can reliably be detected and located.

4.4 The Predicate Abstraction-based Model (PAM)

[24] introduce a synthesis between Predicate Abstraction [4] and the VBM. Whenever the plain VBM cannot derive a conflict, the PAM is applied to the regions of the model where no values could be predicted. A conflict is returned if a set of predicates can be derived that are sufficient to prove that the model is inconsistent.

While the abstraction refinement approach has been shown to perform well for the purpose of verifying programs [9], the impact of under-specified program elements introduced by fault assumptions on the refinement process remains to be analysed in more detail.

4.5 The Heap Invariant Model (HIM)

The *Heap Invariant Model (HIM)* [8] utilises predicates representing invariants of heap data structures. For example, a particular tree data structure manipulated by the program should be acyclic at all times.

The HIM uses the plain VBM to simulate the program, keeping track of the heap invariants. If an invariant is found to be violated, the model is traced backward to find the component C which first introduces the violation. The components implying that C is reached and the components implying C's inputs are returned as conflict. Unfortunately, the description of the precise algorithm proposed in [8] is rather vague, but it seems that the HIM can be seen as a variant of the VBM model enhanced with simple predicate abstraction and heuristics for conflict minimisation.

4.6 The High-level Observation Model (HOM)

Ideas similar to those presented in the HIM and the AIM have been introduced in [29]. The model provides improved precision and better conflict detection by combining the interval lattice used in the AIM with a fixed set of predicates modelling certain properties of program executions. The predicates together with the

AIM allow the conformance tester to build refined models that can detect conflicts when the plain AIM or the abstract properties alone do not provide useful information.

The HOM encompasses a catalogue of abstract properties, each associated with a plain text description for interacting with the user. Properties represented by the HOM include among others: (i) variables and data structures should (not) be modified between two points in the execution, (ii) data structures should always be (a)cyclic, or (iii) a loop should iterate over all elements of a data structure. The benefit of such high-level specifications is twofold: (i) any fault candidate violating the property is eliminated, and (ii) the conformance tester can exploit those properties to build more precise models and to better approximate the consistency test. The properties have been confirmed to exclude a number of spurious but difficult to eliminate explanations on a set of toy programs. Thorough evaluation on a larger set of programs remains future work.

5 Comparing models

The models summarised in this work can be compared according to multiple criteria. The following aspects may be of interest to determine if a particular model suits a given program:

- Precision: the fraction of spurious results returned as valid explanations.
- Completeness: is it guaranteed that the true fault is always included in the explanations returned by the MBSD engine?

The first aspect is important from a practitioner's point of view, as it is well-known that users quickly lose confidence if many false explanations are reported [43]. The relation $\subseteq_\Delta$ between models is used to compare results obtained from different modelling approaches. Specifically, $A \subseteq_\Delta B$ denotes that the set of program statements returned as possible explanations among all diagnoses Δ^A obtained from approach A is a subset of the program statements implicated by diagnoses Δ^B obtained from B:

Definition 5.1

$$A \subseteq_\Delta B \longleftrightarrow \bigcup_{S \in \Delta^A} S \subseteq \bigcup_{S' \in \Delta^B} S'$$

Model A implicates a subset (or the same set) of the statements model B considers valid explanations and potentially returns fewer spurious explanations than B. Throughout this section, it is assumed that models use the same test case specifications, heap abstraction and program components.

5.1 *The dependency-model hierarchy*

The following relationships hold between the models presented in Section 3:

$$ETDM \subseteq_\Delta DDM \subseteq_\Delta SDM$$

Proof. Given that the models apply the same reasoning strategy and model representation, it is sufficient to examine the heap abstraction and approximation of dependencies applied by each model. While the ETDM represents only a single execution, the *DDM* safely approximates dependencies in *all* possible executions. All dependencies modelled in the ETDM must be contained in the DDM. It can be seen that the representation of dynamic data structures in the *DDM* (*SDM*) is derived from the *ETDM* (DDM) by aggregating heap locations and adding additional dependencies for summary locations. It follows that the set of dependencies derived for the precise models are all included in the abstract models.The more abstract model is a safe approximation of the more precise model. It follows that whenever the precise model is consistent, so is the abstract model. □

Extending Kuper's dependency-based model (*PBM*) with support for dynamic data structures leads to a representation that is equivalent to either the *DDM* or the *SDM*, depending on the heap abstraction. Extending Hunt's model the same way leads to a debugger giving similar results as program dicing [1].

The definitions of static and dynamic slices [37] give rise to the following inclusions:

$$\begin{gathered} \mathit{DICE} \subseteq_\Delta \mathit{SSLICE} \subseteq_\Delta \mathit{INSTR} \\ \mathit{DSLICE} \subseteq_\Delta \mathit{EXEC} \subseteq_\Delta \mathit{INSTR} \\ \mathit{DSLICE} \subseteq_\Delta \mathit{SSLICE} \end{gathered}$$

INSTR and *EXEC* denote the set of all instructions and the set of executed instructions in the program, respectively, and *DICE*, *DSLICE* and *SSLICE* denote the instructions obtained from program dicing and dynamic and static slicing, respectively.

It was shown in [41] that conflicts in dependency-based models are equivalent to slices if no structural faults are present.

$$\mathit{ETDM} \subseteq_\Delta \mathit{DSLICE} \text{ and } \mathit{DDM} \subseteq_\Delta \mathit{SSLICE}$$

where for the second inclusion it is required that the heap abstraction used to compute *SSLICE* is not more precise than the one used in *DDM*.

If only a single variable is observed to be incorrect, dependency-modes lead to the same results as slicing. When restricted to single fault explanations, the explanations are precisely the ones contained in the intersection of the individual slices for each incorrect variable [41]. Otherwise, the model-based approaches can improve the results compared to purely slicing-based strategies through the application of specific fault models for components.

Neither Slicing nor the dependency-based models are complete for general faults. Both guarantee that a fault is included in the set of explanations in case the fault is not masked, does not involve missing or additional statements or assignments to wrong variables.

5.2 The VBM hierarchy

It is easy to see that the *VBM* is less precise than both the *Loop-free model (LFM)* [30] and the *PAM*, as both models are specialisations of the VBM:

$$LFM, PAM \subseteq_{\Delta} VBM$$

Proof. Both the *LFM* and the *PAM* extend the *VBM* with additional constraints that restrict behavioural models of components.Extensions are applied when the *VBM* alone is consistent to refine the approximation of the consistency test. It follows that the specialised models derive a superset of all conflicts obtained from the plain *VBM*. Therefore, the *VBM* is consistent whenever the specialised models are consistent. □

$$AIM \subseteq_{\Delta} EM \subseteq_{\Delta} VBM$$

Proof. The *AIM* can be seen as a dynamic unfolding of the *EM*, specialising the control flow graph to a subset of the paths in case some paths cannot be realised in a program state. The interval lattice is strictly more expressive than the simple lattice used in the *VBM*. Therefore, the *AIM* has a potentially larger set of control locations, each more specialised than the *EM* and annotated with at least the amount of information as the *EM* provides. It follows that consistency of the *AIM* model implies consistency of the *EM* model.

The *EM* also provides fewer explanations than the VBM: while the conformance tester applies the same lattice for both models, the *EM* can derive inconsistencies more often due to superior region-based reasoning used for conditionals. The formal proof is lengthier but can be established by induction on the structure of the models. □

$$HOM \subseteq_{\Delta} AIM$$

Proof. The *HOM* is obtained from the *AIM* by replacing the interval lattice with a reduced product [31] of the interval lattice and the abstract lattices representing the abstractions modelled by the HOM. This implies that the conformance tester of the *HOM* derives at least the same information for each program state as the *AIM*. Therefore, all conflicts derived by the *AIM* can be derived by the *HOM* and the *AIM* is consistent whenever the *HOM* is consistent. According to Reiter's hitting set algorithm [34], the diagnoses of the *HOM* must be a subset of the *AIM*'s result.□

The *ADM* alone is not directly comparable to any of the previous models, as dependencies are computed rather than used for modelling and no values are being propagated.

$$ADM \subseteq_{\Delta} INSTR$$

Proof. Trivial All explanations must consist of statements in the program. □

$$VBM \subseteq^1_\Delta DSLICE$$

if the *VBM* is restricted to single fault diagnoses.

Proof. The *VBM* and all its variants precisely simulate the program behaviour when no fault assumptions are applied. A conflict $\Theta \subseteq DSLICE$ can be derived by computing the dynamic slice of an incorrect variable. According to Reiter's theory of diagnosis [34], all single fault diagnoses are elements of Θ. For explanations comprising multiple components this result does not holds, because the *VBM* may compute conflicts consisting of components not in *DSLICE*. □

The *VBM* provides better results than the *ETDM* when restricted to single-component explanations:

$$VBM \subseteq^1_\Delta ETDM.$$

Proof. *VBM's* single component explanations must be contained in *DSLICE*. *DSLICE* contains precisely the statements necessary to compute an (incorrect) value, and both models simulate the program when no fault assumptions are present. The initial conflict derived by the *VBM* must be included in *DSLICE*; for the *ETDM*, the conflict is equivalent to *DSLICE*.

The conflict extractors used in the *ETDM* and the *VBM* both operate on the same set of candidates in *DSLICE*. It can be shown that whenever the *ETDM* eliminates an candidate, the *VBM* also eliminates the candidate: conflicts for the *ETDM* correspond to model paths where all input variables are represented as $ok(\cdot)$. In this case, the representation of the *VBM* can also simulate the execution, as all required input values are known. Thus, whenever the *ETDM* derives a conflict, the *VBM* can derive the same conflict. □

Results for models using heuristics and unsafe approximations vary and cannot be compared to most other models. It cannot be guaranteed that the results obtained from *ADM*, *DICE*, *HM*, *LFM* or *HIM* are a superset if the results obtained from *PRECISE*.

The remaining models use a safe approximation of the concrete semantics. It follows that none of these models can provide better results than *PRECISE* (which is not computable in general):

Theorem 5.2

$$PRECISE \subseteq_\Delta M, \quad M \in \{HOM, PAM\}$$

The result for the other models follows by transitivity of $\subseteq_\Delta$.

Proof. Trivial, as models are safe approximations of *PRECISE*. □

Figure 6 summarises the relationships between the different models.

All models except program dicing, the LFM, the HIM and Hunt's models are guaranteed to locate faults not involving structural faults. Most models cannot reliably detect or locate faults involving missing or additional statements, the use of wrong variables and other structural differences. The *ADM*, *AIM* and *HOM* are

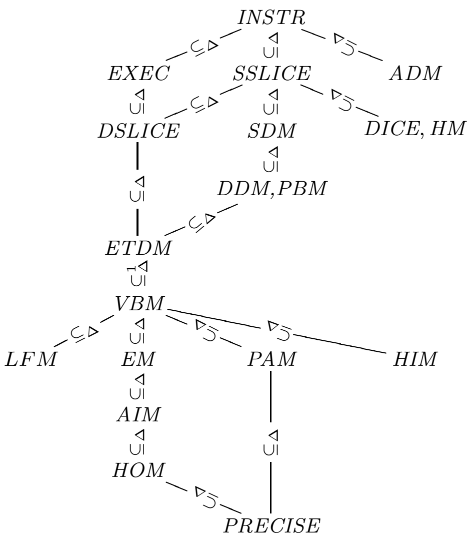


Fig. 6. Relationships between different models

guaranteed to locate and detect faults given a suitable test case specification. Faults not manifested as failing test cases cannot be detected by any MBSD approach.

6 Related work

An approach for debugging VHDL that in principle corresponds to dependency-based MBSD is presented in [2]. Fault models are represented as multiplexer components inserted into the original design, where the channel selector signals representing fault assumptions. Additional constraints limit the number of faults permitted in valid explanations. The entire model is subsequently transformed into propositional logic and solved using a SAT solver. The number of faults is increased and the process is repeated in case no solution is found.

A fault simulation based approach is described in [20], where potential explanations are identified by replacing signals with constants 0 or 1 to determine if a signal can potentially correct the circuit with respect to a given set of test vectors. The number of possible combinations of signals to be tested is reduced by exploiting the structural composition of the circuit. The constants introduced for fault simulation can be seen as a strong fault models that predict a constant instead of no value at all. Consequently, [20] require two simulation runs, while the model-based approach requires only one. The fault simulation based approach relies on intelligent pruning techniques to reduce the number of fault combinations to simulate instead of conflicts to build explanations.

Delta Debugging (DD) [10] aims at isolating a root cause of a program failure by minimising differences between a run that exhibits a fault ("failing run") and a similar one that does not ("passing run"). 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.

Injecting different values in the programs state can be seen as a special case of fault models of a program statement. While DD uses values taken from a different program execution, MBSD does not specify exact values. Instead, generic placeholders (□) are used, causing the program simulator to follow all possible paths that are consistent with the under-specified program state.

The causes presented by DD depend on the subsets of values that are exchanged, and different subsets may lead to different causes. Also, not all differences are equally interesting. It would be interesting to compare a generalisation of DD with MBSD algorithms to see how the output relates to MBSD in case a larger number of explanations is returned. Conversely, MBSD suffers from relatively high false positive rates, which does not seem to be the case with DD.

Well-known verification techniques have recently been applied to not only verify correctness, but also locate a fault [18]. The basic principle is to relate abstract execution traces leading to correct and erroneous program states. By focussing the search on traces that deviate only slightly from passing and failing test cases, likely causes for a misbehaviour can be identified. [25] compare the error trace-based strategy to MBSD and conclude that the former is sensitive to variations of the search depth limit used to restrict the search. Conversely, counterexamples may provide more information to the developer provided the trace is short.

Another approach based on bounded model checking is presented in [7], where a constraint solver is used to derive faulty *abstract* traces that differ minimally from correct executions. In contrast to the approach outlined in Section 4.4, the result is presented as the difference between traces and not as locations within the program.

Error traces have also been applied to synthesise potential corrections of faulty programs, given a specification of the program's correct behaviour [19]. 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 goes beyond what current approaches in MBSD can achieve, not only pinpointing possible faults but also providing corrections automatically. The downside is that a formal specification of the program's behaviour is required.

7 Challenges

Despite considerable progress in MBSD, many challenging issues remain to be solved. Some of the aspects presented below are particular to the MBSD approach, while others are instances of problems common to many different automatic debugging techniques. We do not claim that the list is complete; however, it reflects a number of issues we feel are important for the further development of MBSD and automated debugging in general.

- How to avoid false positives? The main problem inherent to current MBSD approaches seems to be a significant number of theoretically valid explanations, which would be considered absurd by any reasonable developer. For example, replacing the last instruction of a program with a function computing the correct result and ignoring the rest of the program. A potential remedy to filter such undesirable candidates is to combine MBSD with symbolic approaches such as [19] to filter spurious explanations and estimate the "size" of the replacement required for valid explanations. Filtering and ranking techniques, such as [43] and [22], may also be applicable.

- How to select appropriate models and fault assumption given a program and test cases? Both the complexity of the debugging process and the quality of the result directly depend on the selected model. [42] provides a set of heuristics as a first step towards automatic model selection. It remains an open issue if techniques developed for semi-automatic verification of programs can be adapted and extended to suit the MBSD framework.

8 Conclusion

This work briefly introduces the idea of Model-based Software Debugging and compares individual models. Relations between different models have been studied, leading to a hierarchy of models with different diagnostic characteristics. The comparison was focused on diagnostic strength of different models, a more in-depth analysis quantifying the differences between models remains for future work.

References

[1] Hiralal Agrawal. *Towards Automatic Debugging of Computer Programs.* PhD thesis, Purdue University, 1991.

[2] Moayad Fahim Ali, Andreas G. Veneris, Alexander Smith, Sean Safarpour, Rolf Drechsler, and Magdy S. Abadir. Debugging sequential circuits using boolean satisfiability. In *ICCAD*, pages 204–209. IEEE Computer Society / ACM, 2004.

[3] Scott Ananian. The static single information form. Master's thesis, Department of Electrical and Computer Science, Princeton University, 1999.

[4] Thomas Ball, Rupak Majumdar, Todd D. Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In *PLDI*, pages 203–213, 2001.

[5] Gregory W. Bond. *Logic Programs for Consistency-Based Diagnosis.* PhD thesis, Carleton University, Faculty of Engineering, 1994.

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

[7] Sagar Chaki, Alex Groce, and Ofer Strichman. Explaining abstract counterexamples. In *SIGSOFT FSE*, pages 73–82. ACM, 2004.

[8] Rong Chen and Franz Wotawa. An object store model for diagnosing Java programs. In *Aust. AI*, volume 3809 of *LNCS*, pages 865–870. 2005.

[9] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model Checking and Abstraction. *ACM TOPLAS*, 16(5):1512–1542, 1994.

[10] Holger Cleve and Andreas Zeller. Locating causes of program failures. In *Proc. ICSE*, pages 342–351. ACM, 2005.

[11] 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.

[12] James C. Corbett. Using shape analysis to reduce finite-state models of concurrent Java programs. Technical report, Department of Information and Computer Science, University of Hawaii, 1998.

[13] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM TOPLAS*, 13(4):451–490, 1991.

[14] Randall Davis. Diagnostic reasoning based on structure and behavior. *Artificial Intelligence*, 24:347–410, 1984.

[15] Alexander Felfernig, Gerhard Friedrich, Dietmar Jannach, and Markus Stumptner. Consistency-based diagnosis of configuration knowledge bases. *Artificial Intelligence*, 152(2):213–234, 2004.

[16] Gerhard Friedrich, Markus Stumptner, and Franz Wotawa. Model-based diagnosis of hardware designs. In *Proc. ECAI*, pages 491–495. 1996.

[17] Russell Greiner, Barbara A. Smith, and Ralph W. Wilkerson. A correction to the algorithm in Reiter's theory of diagnosis. *Artificial Intelligence*, 41(1):79–88, 1989.

[18] Alex Groce and Willem Visser. What went wrong: Explaining counterexamples. In *Proc. SPIN*, volume 2648 of *LNCS*, pages 121–135. 2003.

[19] Haifeng He and Neelam Gupta. Automated debugging using path-based weakest preconditions. In *Proc. FASE*, volume 2984 of *LNCS*, pages 267–280. 2004.

[20] Shi-Yu Huang and Kwang-Ting Cheng. Errortracer: design error diagnosis based on fault simulation techniques. *TCAD*, 18(9):1341–1352, 1999.

[21] Daniel Jackson. Aspect: Detecting Bugs with Abstract Dependences. *ACM TOSEM*, 4(2):109–145, 1995.

[22] James A. Jones, Mary Jean Harrold, and John T. Stasko. Visualization of test information to assist fault localization. In *ICSE*, pages 467–477. ACM, 2002.

[23] Ulrich Junker. QUICKXPLAIN: Conflict detection for arbitrary constraint propagation algorithms. In *IJCAI'01 Workshop on Modelling and Solving problems with constraints*, 2001.

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

[25] Daniel Köb and Franz Wotawa. A comparison of fault explanation and localization. In *Proc. DX'05*, pages 157–162, 2005.

[26] Ron I. Kuper. Dependency-directed localization of software bugs. Technical Report AI-TR 1053, MIT AI Lab, 1989.

[27] Cristinel Mateis, Markus Stumptner, and Franz Wotawa. Debugging of Java programs using a model-based approach. In *Proc. DX'99 Workshop*, 1999.

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

[29] Wolfgang Mayer and Markus Stumptner. Model-based debugging with high-level observations. In *ICIIP*, 2004.

[30] Wolfgang Mayer, Markus Stumptner, and Franz Wotawa. Model-based Debugging or How to Diagnose Programs Automatically. In *Proc. IEA/AIE*, LNAI, pages 746–757, 2002.

[31] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis.* 1999.

[32] Jefferson A. Offutt and Stephen D. Lee. An empirical evaluation of weak mutation. *IEEE TSE*, 20(5):337–344, 1994.

[33] Bernhard Peischl, Saffeeullah Soomro, and Franz Wotawa. Dependence in verification and debugging. In *Proc. DX'06*, 2006.

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

[35] Thomas W. Reps, Shmuel Sagiv, and Reinhard Wilhelm. Static program analysis via 3-valued logic. In *CAV*, *LNCS 3114*, pg. 15–30. 2004.

[36] Markus Stumptner. Using design information to identify structural software faults. In *Aust. AI*, volume 2256 of *LNCS*, pages 473–486, 2001.

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

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

[39] Franz Wotawa. *Applying Model-Based Diagnosis to Software Debugging of Concurrent and Sequential Imperative Programming Languages.* PhD thesis, Technische Universität Wien, 1996.

[40] Franz Wotawa. Debugging hardware designs using a value-based model. *Applied Intelligence*, 16(1):71–92, 2002.

[41] Franz Wotawa. On the relationship between model-based debugging and program slicing. *Artificial Intelligence*, 135(1-2):125–143, 2002.

[42] Franz Wotawa. Debugging VHDL designs: Introducing multiple models and first empirical results. *Applied Intelligence*, 21(2):159–172, 2004.

[43] Yichen Xie and Dawson R. Engler. Using redundancies to find errors. *IEEE TSE*, 29(10):915–928, 2003.