A Survey of Intelligent Debugging

Markus Stumptner and Franz Wotawa
Technische Universität Wien,
Institut für Informationssysteme,
A-1040 Vienna, Paniglgasse 16, Austria, Europe
{*mst,wotawa*}*@dbai.tuwien.ac.at*

Automated debugging systems have a long history with interesting results produced by research prototypes and deployed applications. We present an overview of Artificial Intelligence approaches to the development of intelligent debugging systems. These systems range from tutoring systems that possess detailed knowledge about the individual programs as well as about typical programmer errors occurring in exactly these programs, over Bayesian Net formalisms that employ statistical results about error reports, to traditional debugging approaches such as Algorithmic (or Declarative) Debugging. Finally, we examine the more recent use of model-based diagnosis principles as a basis for software debugging research. We illustrate the potential of model-based reasoning by discussing several models (differing in expressivity and assumptions on language semantics) that are currently in various stages of realization, from prototype implementations to test use in an industrial environment.

Keywords: Debugging, Validation, Model-based Diagnosis

1. Introduction

Automated support for software debugging has long been an active research area. It derives its importance from the (past and present) central role of debugging in the software development process. Work on debugging has involved and involves the development of improved debugging environments that provide better visualization, more language-specific context-dependent editing support, and computer support for cooperative debugging sessions. Unlike these auxiliary technologies, automated debugging systems attempt to attack the problem at the root, by examining the semantics of programs and attempting to derive the cause of errors from certain aspects of these semantics. Among the automated debugging techniques proposed, knowledge-based debugging systems have so far shown a lot of appeal in dealing with different programming language paradigms. However, despite an impressive array of prototypes, general applicability still seems to be a remote goal. A recent special journal issue on advances in debugging technology did not mention automated debugging at all [19]. At the same time, fault diagnosis in other technical domains has been tackled successfully for a long time, with diagnosis research over the last decade focusing on model-based reasoning as a promising approach. The goal of this paper is to examine recent and current debugging approaches that promise to have the potential of going a step further along the road towards automatic program debugging.

Traditionally, automated debuggers in the literature have been classified according to a large variety of criteria. For example, Shamehri et.al. [31] use the timing (but interestingly, not the amount) of user interaction, the kind of additional inputs required by the debugger (intended I/O behavior, intended implementation, intended program behavior), generality (i.e., limitations to programming style or bug types), and dependency on the degree of user expertise (i.e., whether the system is intended for expert or novice programmers). The last criterion is mirrored in [14], where a distinction is made between *tutoring* (for novices) and *diagnosis* systems (for experienced programmers). The latter work also groups systems according to basic strategies, namely *verification* with respect to a formal specification, *checking* based on explicit knowledge about a particular programming language, and *filtering*, i.e., the exclusion of program parts that cannot be responsible for a misbehavior. Thirteen special techniques are identified as being used in the systems reviewed, and grouped according to the three strategies. The only approach listed in these two surveys that was picked up by multiple researchers was Shapiro's Algorithmic Debugging [30], originally specified in a Logic Programming context, which was refined in various directions and also experimentally extended to other languages (also known in the logic programming literature as Declarative Debugging or Declarative Diagnosis [25]).

Using a fundamentally different approach to software debugging, Burnell et al. [2, 3] introduce a system that uses fault probabilities and information about

AI Communications 0 () 0
ISSN 0921-7126 / $8.00 © , IOS Press

the program's data flow to focus on the faulty program parts. The fault probabilities have been obtained from experts using empirical data available from one (very large and long living) program. In [3], it is stated that in 75 percent of all debugging cases presented to the system, the fault was correctly identified.

Recently, researchers have begun to examine model-based diagnosis as a way of describing the search for program bugs [10, 6, 15]. The concept of model-based reasoning consists of providing a model that describes a certain domain and then using it to diagnose different systems from that domain. The advantages of the approach appear particularly enticing in the software domain due to the ability to produce the description of the system automatically from the program code, but the hard part of the task is to find effective models, heuristics and focusing techniques.

The rest of this paper is structured as follows. We begin with a discussion of the basic issues of intelligent program debugging and the different challenges facing tutoring systems and debugging systems or debugging aids aimed at a more general audience. We then start the overview of specific systems with a look at two tutoring systems, followed by a presentation of the Bayesian approach chosen by [3]. In the next section, we briefly describe the basic tenets of Model-Based Diagnosis, and then describe in Section 6 how model-based diagnosis can be used for the diagnosis of errors in software (i.e., debugging). Section 7 gives an overview of different debugging approaches based on model-based reasoning. The paper closes with a comparison of the model-based approach to other knowledge-based debugging approaches.

2. Intelligent Programming Tutors

As mentioned above, this section will summarize the properties of two examples of intelligent tutoring programs that use knowledge about the intended solution of a particular student assignment, together with knowledge about expected student errors to aid the students in debugging their programs. PROUST [18] and **Talus** [24]. A comparison of contemporary systems from this field is contained in [24].

The PROUST system attempts to analyze the intent of the student programmer by matching his PASCAL program against a elements from a plan library. The idea is that by matching against a particular plan, the intent of the student can be discerned. The analysis is done in a top-down manner. A simple plan example is the following which describes one particular choice of processing what PROUST calls a Sentinel-Controlled Input Sequence, i.e., a stream of input values ended by a specific end value. This particular plan represents the programmer's intent to use a while loop to solve the problem. It specifies the variables used within the loop, the notion that occurrence of a specific constant is used to end the loop, and the operations which switch to the next input value. The goal of the system is to match the four parts of the plan specified in the template to actual parts of the program (and, if successful, perhaps continue matching these parts to lower level plans).

SENTINEL PROCESS-READ WHILE PLAN
Constants: `?Stop`
Variables: `?New`
Template:

```
InitInput: subgoal Input(?New)
MainLoop: while ?New <> ?Stop do
          begin
Process:     ?*
Next: subgoal Input(?New)
                    end
```

To be effective for debugging, the system has to have the ability to match plans even if only part of the plan is recognizable (because it is distorted by a bug). So-called *plan-difference rules* are used to express specific variations from a particular plan. As noted in [24], the effectiveness of PROUST is highly dependent on its ability to recognize all parts of a program via these mechanisms, and the difficulty of this task rises sharply with the complexity of the example programs.

The **Talus** system [24], intended to debug programs written in a subset of LISP, attempts what is called a *semantics-based* approach. The student program is first simplified in a manner that leaves its semantics unchanged (e.g., by rewriting conditional expressions using COND as IF statements). The program, represented as a tree, is then matched against a set of reference implementations of the function to provide recognition of the algorithm used. The Boyer-Moore theorem prover is then used to attempt to construct an induction proof of the equivalence of the student function with the reference function. The base step for induction is the case distinction provided by conditionals, while the induction step corresponds to recursive calls. If the proof succeeds, the student function is correct, if not, then the error can be traced to an induction condition that does not hold (which is why the proof failed). For example, given the student program

```
(defun in-tree (x tree)
    (if(atom tree)
      nil
      (or(in-tree x (car tree))
         (in-tree x (cdr tree)))))
```

and the reference function

```
(defun memtr (x tr)
    (if(atom tr)
      (eq x tr)
      (or(memtr x (car tr))
         (memtr x (cdr tr)))))
```

the conjecture

```
(implies (atom tr)
      (iff (eq x tr) nil))
```

is violated, and making the appropriate replacement in the student program will result in a correct function.

While both systems can be quite effective depending on the library of plans (or reference functions) provided, such reference libraries would of course not be available in other software development contexts beyond novice tutoring.

3. Functional Debugging

This approach is based on the Functional Reasoning principle [29] used for the design and diagnosis of hardware devices. It uses a functional model as an extensive specification of the components of a sequential program. The function of a segment of the code is expressed in terms of *input state*, *output state*, and *proviso*. Given a particular input state (i.e., a partial specification of a state of the program), the function will provide a particular output state (i.e., another partially specified state) under the assumption that the proviso is true. For example, the ZD representation [20] of an operator that returns the next element from an array of integers for processing in a loop would take the following shape (with the **If** property omitted):

GET-NEXT-INTEGER
If ...
ToMake $x = array[index]$
$\quad \wedge i.last = x \wedge \ldots$
Provided $i.last = s$

The variable i specifies the current state of the program, and the attribute $last$ is supposed to contain the value of the last iteration. The **ToMake** property describes the desired output state, in this case referring to the fact that the variable x is supposed to be assigned the next value of the array, and that this is assigned to the variable $i.last$. The proviso $i.last = x$ will check on the next iteration that this condition still holds.

The implementation for such a specification is defined in terms of subfunctions connected in a state transition graph (the functions correspond to state transitions in the graph). If during use the specification is violated, then the diagnosis uses the specification to decide what kind of violation has occurred. For example, a *faulty context of use* error has occurred when the proviso of a function is not satisfied. A *faulty intended use* has occurred if a subfunction does not produce the required output state.

The ZD representation is descendant of the DUDU (Debugging Using Device Understanding) system [1], which uses a plan-based approach similar to the PROUST system. However, since plans are expressed in functional representation, the system is more flexible in fitting different actual code fragments into the overall plan.

4. Probabilistic Debugging

In [2, 3], another technique for debugging programs is introduced that combines logic with probabilistic inference methods. The DAACS debugging system described in these papers was developed for analyzing runtime errors of the Sabre airline reservation system. Sabre is a large legacy program running under the IBM TPF operating system and written in Assembler. Runtime errors are a frequent occurrence and are protocolled by a large dump-file which has to be analyzed to fix the bug. A typical dump-file contains a snapshot of relevant parts of memory at the time the illegal operation, e.g., an illegal memory reference, occurred.

The goal driving the development of the DAACS system was to assist the program analysts to determine the root error in a program segment. A root error is defined as an initial instruction (or sequence of instructions) that led to an erroneous situation. To reach this goal the DAACS system performs two task:

1. A prediagnosis task fetches all necessary informations from the dump-file, e.g., the type of the operating system violation, or other relevant symptoms.

2. The diagnosis task itself is divided into two phases. In the first phase a logical inference procedure is used to determine feasible execution paths using informations gained by the prediagnosis. The second phase uses probabilistic inference to rank the paths delivered from the logical phase.

We explain the DAACS diagnosis process using an illegal memory reference example from [2]. A pointer variable, VAR1, has been pointing to a memory cell allocated by another process and has been acccessed after the block was deallocated (this is called a PASTHELD error). To fix the root cause of this error, we first examine the structure of the execution paths. Consider the following (partial) program listing:

```
Block A:
     1.  set var3 = var4
     2.  set var1 = var2
     3.  if var3 = 0 goto label y
Block B:
     4.  label x:
⋮
     5.  set var1 = var1 + 80
⋮
     6.  set var3 = var3 - 1
     7.  if var3 = 0 goto label x
Block C:
     8.  label y:  set var2 = a value
     9.  set var3 = a value
     10. set var4 = a value
     11. print var1 << *** ERROR ***
```

This program part consists of 3 basic blocks. A basic block is a set of statements executed in a sequence. An execution path is a sequence of such basic blocks that could be possible executed during program execution. For this example 2 execution paths can be obtained. One path is A–B–C, where B is executed at least once, and the other is A–C. To distinguish between the two paths DAACS uses Bayesian networks for computing a ranking. Belief networks are created for every type of error, i.e., in our example the PASTHELD error.

The goal is to compute the probability that a path is faulty, i.e., contains the bug, under the condition that a fault has been detected. We denote this probability by $p(P = i|F = true)$ where P is the random variable associated with the selected path and F a random variable indicating the occurrence of an error. This probability on can be calculated according to the Bayesian rule.

$$p(P = i|F = true) = \frac{p(F = true|P = i) \cdot p(P = i)}{P(F = true)}$$

As we know that a fault has occurred, it holds that $P(F = true) = 1$. If we assume that the probability of the bug location is uniformly distributed, then $p(P = i) = 1/(\text{Number of paths})$, i.e., every path is equally likely to contain the buggy statement.

We now have to compute the probability that a specific (selected) path contains the bug. In [2, 3], this is done by specifying conditional probabilities for specific path properties when a specific path error has occurred. The probabilities $p(path\_findings|path\_error\_class)$ were determined by expert engineers. The system now computes $p(path\_error\_class|path\_findings)$. By summing the conditional probabilities of all error classes, the failure probability of the selected path is established. The example given in [2] distinguishes between the error classes

- *BAD SET*: Instructions have improperly initialized the erroneous variable.
- *BAD ADJUST*: Instructions have improperly modified a value stored in the variable.
- *ENTERED BAD*: The variable is not modified on the path.
- *BAD LOOP*:There are errors regarding loops causing the erroneous value of the variable.

Analysis of program paths may result in additional information.

- *SYNTACTIC STRUCTURE*: The structure of the path causes an error, e.g., a *SET FAIL* rules out *BAD ADJUST* because no adjustment instructions exist on the path.
- *CLOSE REGS*: The proximity of values of variables to the value assigned to the erroneous variable. The appearance of close values is an indication of *BAD SET* errors.
- *NEG REGS*: The presence of large negative values, which suggest *BAD ADJUST* and *BAD LOOP* errors.

Continuing with the example, the path properties *set bad* (because var1 is set to var2) and *adjust bad* (because of the statement var1 = var1 + 80) are observed. They are within the domain of the random variable *SYNTACTIC STRUCTURE*, among others.

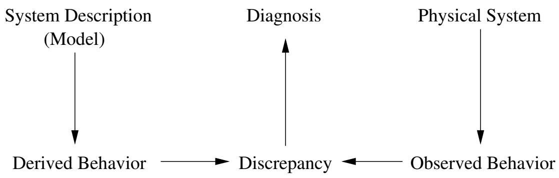
This approach to debugging uses a (probability-based) model obtained from experts. Although the described system is used only for one large legacy system, the approach is obviously viable for large program systems and should be applicable to a wider domain under the assumption that appropriate random variables and probabilities are available. The DAACS system is different from most of the other systems in this survey because it specifically deals with runtime errors that cause a dump file to be written. The other approaches are intended to be used for locating bugs during the test phase and also attempt to address general functional errors (i.e., incorrect results that do not lead to a program crash).

5. Model-based diagnosis

The concept of model-based diagnosis originates in research intended improve on the ”brittle” rule-based diagnosis expert systems that contain (painstakingly hardcoded) rules. The goal was to produce generally more capable system than allowed by the narrowly focused description of individual symptom-cause chains of first generation expert systems, and the means to that end was the employment of *reasoning from first principles* [13]. This approach has been applied to other areas apart from diagnosis, and attempts to provide fundamental knowledge about the relationships between dentities in a particular domain. From the point of view of diagnosis, given a particular application domain, these first principles are embodied in the description of

- the structure of the system to be diagnosed, as described by a set of components and their connections
- the behavior of the system, described in terms of the values produced by the system on its outputs for certain input values.

Establishing a domain model for a particular application domain (e.g., digital circuits, power distribution systems, or hydrodynamic processes) then makes it possible for the model to be re-used for different systems from the same domain by adapting the structural description of the system. The fundamental way in which the system description produced from the model is used to find the cause of a fault through model-based diagnosis is illustrated in the following figure:



Given an individual set of inputs to the system, the system description is used to derive the expected (correct) behavior of the system, i.e., the resulting correct outputs. If the system to be diagnosed is not working correctly, then the outputs produced by it will not agree with these expected values. The task of the diagnosis system is to conclude, from the discrepancies between expected and observed values, which components in the system must have malfunctioned to produce the observed outputs.[1] These principles were subsequently formalized by Reiter [28] as follows:

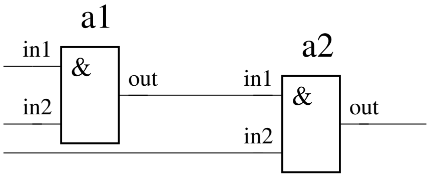
Definition 5.1 (Diagnosis Problem)
A diagnosis problem is a triple $(SD, COMP, OBS)$ *where* SD *(a set of logical sentences) is the system description of the system to be diagnosed ,* $COMP$ *is the set of components in the system, and* OBS *is a finite set of observations. A set* $\Delta \subseteq COMP$*,* Δ *is a diagnosis iff* $SD \cup OBS \cup \{ab(c)|c \in \Delta\} \cup \{\neg ab(c)|c \in COMP \setminus \Delta\}$ *is consistent. A diagnosis is minimal if no subset is itself a diagnosis.*

The set of ab literals specify assumptions on the correctness of components in the system. If $ab(c)$ holds, the component c is assumed to be abnormal, i.e., not working correctly. For example, the following clause describes part of the behavior of a logical *and* gate:

$$and\_gate(x) \wedge \neg ab(x) \supset out(x) = and(in1(x), in2(x))$$

[1] It should be noted that the distinction between inputs and outputs as used here does not necessarily correspond to the actual flow of events inside the system. Rather, in this paragraph we refer as *inputs* to those aspects of the system which are known beforehand and as *outputs* to those which result from operation of the system (derived from the system description or produced by the actual system).

This general behavior model that is applicable to all and gates could, e.g., be used for diagnosing an individual the following circuit composed from such gates:



The system description consists of the *and* gate behavior and a set of formulae that describe the interconnections of the components (in this case, of the gates in the circuit):

$$and\_gate(a1) \wedge and\_gate(a2) \wedge out(a1) = in1(a2)$$

Now consider the following set of observations:
$in1(a1) = 1$. $in2(a1) = 1$.
$in2(a2) = 0$. $out(a2) = 1$.

The assumption that all components behave correctly would lead to a contradiction, since then, according to the behavior model, the correct value of $out(a2)$ would be 0. Therefore, some non-empty set of components must exist whose malfunction that explains the misbehavior. In this case, $ab(a2)$ is the only minimal diagnosis, since assuming that $ab(a1)$ and $\neg ab(a2)$ still results in a contradiction.

Apart from decoupling the semantics of the diagnosis definition from the implementation (multiple algorithms exist for computing diagnoses as specified above efficiently), model-based diagnosis also has the property that detailed prespecified knowledge about particular symptom-fault connections is not required for diagnosis. Instead, what is used is the system description that describes the correct functioning of the system, and faults of individual components are then derived by finding out which correctness assumptions for components contradict the observations. A consequence is that the diagnosis of multiple faults is also covered by the same formal definition.

In model-based diagnosis research, a distinction is usually made between *abductive* and *consistency-based* diagnosis. Abductive diagnosis (e.g., [26]) is based on the concept of finding a set of assumptions that implies the erroneous observations, whereas consistency-based diagnosis merely assumes that compliance with the observations marks a diagnosis. The above definition, naturally, is a consistency-based one. The main difference is that abductive diagnosis requires the presence of more detailed fault models (but consistency-based diagnosis will work just as well if such fault models are present). We will not deal with the distinction in detail in this paper. See [12] for an extensive discussion of the relationship between abductive and consistency-based diagnosis.

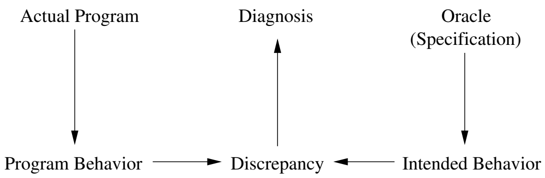
6. Model-based Diagnosis of Software

With the successful use of model-based diagnosis for hardware problems, eventually attempts were made to apply the same methods to software - in other words, to use the model-based approach for debugging. As it turns out, the basic approach remains the same, but a number of differences have to be taken into account. Most of these differences derive from one central issue - the role of the system description.

6.1. Adapting Diagnosis to Debugging

The system description, as defined by Reiter, defines the structure and behavior of the system. Since the system to be diagnosed is a program, the system description will be a description of the behavior of the program, derived from the program itself and the semantics of the language in which the program is written. Therefore, as was only to be expected, the model chosen will usually vary with the language involved (because different language have different semantics), and given a fixed choice of how the semantics should be represented, the system description can be automatically derived from the code of the program.

The model-based view of software debugging is represented in the following figure.



Just as with hardware diagnosis, it is the task of the diagnosis system to derive which parts of the program were responsible for a particular misbehavior. However, the role of some of the parts of the diagram has changed significantly, as will be described below.

P

6.2. Diagnosis vs Tutoring

We return here for a moment to the distinction between tutoring and diagnosis systems as made in the automated debugging literature. The main distinction between the two purposes is that these two classes of debuggers run in completely different organizational environments, deal with different types of programs, and deal with somewhat different types of errors and explanation requirements. Specifically, novice programs will be small and most importantly, their purpose will be prespecified and their design and implementation will be repetitive. In a tutoring environment, the same task may be repeatedly given to dozens of students, possibly over a timespan of several years. It is therefore possible to write a clear and very detailed specification or a reference solution (i.e., a correct program) for the problem, and the knowledge gained by starting from this basis will allow the debugger to engage in significantly more complex reasoning about the behavior of the (incorrect) student programs. This area of automated debugging is exemplified by powerful systems such as PROUST [18] and **Talus** [24].

Most software development, however, occurs under conditions where the new program is in many aspects unique, large and complex, and there is no time to provide anything even remotely resembling a full formal specification for the program. The correctness of the program will be checked almost exclusively by testing, i.e., by comparing output generated by the program with desired output values. It is this kind of environment that is the goal of model-based debugging. A separate formal specification of the system is not needed, since the system description is automatically derived from the code of the program. The desired output values provided as parameters for the testing process correspond to the observations in model-based diagnosis.

6.3. Components

The notion of what is a component and what isn't is less clearcut than in hardware diagnosis. Even in model-based hardware diagnosis, of course, the choice of components depends on the desired level of abstraction chosen for the representation. The result of a representation that is too detailed may be a system that is too slow to be usable in practice. However, the physical relationships between subsystems make it relatively easy to come up with an appropriate decomposition into components (e.g., gates, pumps, valves) and connections (e.g., wires, pipes, physical forces), and ten years of research in model-based diagnosis have produced, if not a consensus, then a certain vocabulary for modelling phenomena in specific physical domains. The first work that examined the model-based diagnosis of software [10] chose a path that kept close to the hardware-related representations. The domain chosen was that of pure logic programs and the components were the individual clauses of the program. This allowed an extremely simple translation from the programs to the system description (effectively, the latter was syntactically identical to the former). For other languages (e.g., imperative languages), the issue is more complex and we will discuss it again in the next section.

On the other hand, once the choice of model has been made (what parts of the program are represented as components, and what parts of the semantics of the language are to be represented), converting a program to a model for a diagnosis system is straightforward and technically trivial since the program must be available in machine-readable form to run at all. This brings up an interesting issue.

6.4. Model and System

In conventional model-based diagnosis, the system description is an exact specification not only of the overall behavior of the system, but of its individual parts. For example, when diagnosing the hardware implementation of a 16-bit adder, the adder's system description will describe the behavior of the logical gates from which the adder is composed. A fault is assumed to occur because one of the components does not act according to its specification.

In diagnosing a program, the system description will typically (at the appropriate level of abstraction chosen) be an exact representation of the program. However, unlike in hardware diagnosis, that means that the system description mirrors the bugs in the program - it is not an independent, correct specification, rather it contains the representation of the errors the program has in the first place.

6.5. Observations and system

A dual change occurs with respect to the role of observations. In hardware diagnosis, observations describe the behavior of the actual system, and if the system is faulty, the observations will differ from the correct behavior predicted by the system description.

In debugging, the roles have been reversed. It is the system description (derived from the program) that reflects the incorrectness of the design and whose output (incorrect in places) is confronted with observations that are correct (the correct output to be produced by the test data), whereas in traditional diagnosis problems, the model is correct and it is the observations, made from the behavior of the actual system, that reflect on the incorrect behavior. In addition, the question of how a programming error may manifest itself in the model leads us to the related issue of structural faults.

6.6. Structure

Structural faults are faults that are not caused by an incorrectly functioning component, but by a missing or *additional* connection between two components, as in a bridge fault in electrical engineering. The bridge fault problem was first noted by Davis [13] and mainly excluded from consideration in subsequent work on diagnosis (except [7]). However, it is very relevant when diagnosis is applied to software, since many typical bugs are structural in nature. The use of an incorrect argument in an expression (e.g., by using a different variable name, switching the ordering of arguments), or the omission of part of a complex expression constitute typical examples of such faults.

The usual way for dealing with structural faults is to assume the existence of a different, complementary model that allows to reason about the likelihood of such faults (i.e., modelling of spatial neighbourhood in the case of bridge faults). In software, such models could take the shape of considering name misspellings, variable switchings, or attempts to repair expressions (i.e., synthesize missing parts) to provide correct functionality. The work described in Section 7.4 provides a foundation for such activities, but the appropriate gauging and tuning of heuristics to control the process to make it usable in real-world problems is an open research issue.

7. Model-based Debugging approaches

This section describes different debugging approaches using different abstraction levels for debugging.

7.1. Model-based debugging of logic programs

The first to describe the application of model-based diagnosis to software debugging were Console et al. [10] . In this work, they introduced definitions and an algorithm for debugging definite logic programs. They argued that MBD provides a better framework for debugging because it minimized the queries necessary to localize a bug and could provide all solutions to a debugging problem at once. In this section we give an example from [10] that motivates the use of MBD techniques and present the basic definitions and propositions.

Consider the following program (which, as we will see, is incorrect):

(c_0) $p(0)$
(c_1) $q_1(X) \leftarrow q_2(X)$
...
(c_{n-1}) $q_{n-1}(X) \leftarrow q_n(X)$
(c_n) $q_n(X) \leftarrow p(X)$
(c_{n+1}) $r_1(X) \leftarrow r_2(X)$
...
(c_{2n-1}) $r_{n-1}(X) \leftarrow r_n(X)$
(c_{2n}) $r_n(X) \leftarrow p(X)$
(c_{2n+1}) $s_1(X) \leftarrow s_2(X)$
...
(c_{3n-1}) $s_{n-1}(X) \leftarrow s_n(X)$
(c_{3n}) $s_n(X) \leftarrow p(X)$

Assume that the following is the intended behavior of the program: neither $q_1(0)$ nor $r_1(0)$ nor $s_1(0)$ should be entailed by the program, while $q_1(1)$, $r_1(1)$, and $s_1(1)$ should be. The above program does not meet its specification. A declarative debugger as described in [30, 25] would query the user about the validity of at least $O(\log n)$ facts. Instead, a MBD system would return one single fault $c_0(p(0))$ and other diagnosis candidates with at least three clauses, e.g., c_1, c_{n+1}, c_{2n+1}. Ordering the diagnoses by complexity would lead to c_0 being the most likely diagnosis. Measurement selection could then be used to determine additional clauses to ask about.

In the above example we have seen that the debugging problem consists of a program and a (partial) specification consisting of two clause sets. One of these contains clauses which need not to be entailed by the program. The other contains clauses which must be entailed. Console et al. give the following formal definition:

Definition 7.1 (Error diagnosis problem) *An error diagnosis problem is a triple* (P, S^+, S^-) *where:*

– *P is a definite logic program;*
– *S^+ and S^- are sets of queries, where variables in S^+ are intended as universal quantified and variables in S^- are intended as existentially quantified.*

Given the example above, $P = \{c_0, \ldots, c_{3n}\}$, $S^+ = \{q_1(1), r_1(1), s_1(1)\}$, and finally $S^- = \{q_1(0), r_1(0), s_1(0)\}$. The task of the diagnosis process is now to determine a set of clauses that must be removed from P, and a set of clauses that must be added to P, according to the specification S^- (S^+, respectively).

Definition 7.2 (Diagnosis) *A pair (C, E) of sets of clauses, where $C \subseteq P$, is a diagnosis iff:*

– *$(P \setminus C) \cup E \models a$ for all $a \in S^+$*
– *$(P \setminus C) \cup E \not\models a$ for all $a \in S^-$*

The set C can be viewed as core of the diagnosis whereas E is highly underconstrained. One possible solution is provided by the following property.

Property 7.1 *Given an error diagnosis problem and a set C of clauses, then the following are equivalent:*

1. *There exists an E such that (C, E) is a diagnosis;*
2. *(S, C^+) is a diagnosis.*

This proposition states that is sufficient to find a solution for C such that (S, C^+) is a diagnosis. However, it seems to be better to provide a more informative solution E to the user, giving a hint on how to correct the program. In principle E has the potential to provide enough information on how to correct the program. Because of the computational complexity of the program synthesis task, Console et al. do not try to search for a program repair solution E.

To rank diagnoses, the authors introduce the following concepts:

Definition 7.3 (Ordering among diagnoses) *$(C, E) < (C', E')$ iff $|C| < |C'|$ or ($|C| = |C'|$ and the number of predicates in the heads of clauses in E is less than the number of predicates in the heads of clauses in E').*

Definition 7.4 (Preferred diagnosis) *A diagnosis is preferred iff it is minimal with respect to $<$.*

From the above definitions it follows that there are several possible sets E fulfilling the preference criteria for a given set C. Therefore Console et al. introduce a syntactic rule that all solutions must obey. Any solution E is limited to clauses in the language of P and S^+ that are of the form:

$$p_i(X) \leftarrow X \neq a_1, \ldots, X \neq a_n$$

such that they give the strongest definition of p_i which is consistent with $S-$. Such clauses are called *exception clauses*. Diagnoses using exception clauses are called *diagnoses in exception form*.

For our example, the preferred diagnosis in exception form is
$(\{c_0\}, \{p(X) \leftarrow X \neq 0\})$.

Console et al. surmise that the exception form is not restrictive, i.e., allows computing every diagnosis. However, Bond et al [8] showed that the exception form is not canonical (and therefore the algorithm described in [10] is incomplete). Consider for example the following program

$$a(X) \leftarrow b(X, Y), c(Y)$$

with the specification $S^+ = \{b(1,1), c(2)\}$ and $S^- = \{a(1)\}$. In this case $(\emptyset, \{b(1,1), c(2)\})$ is a diagnosis and there exists no set of exception clauses E such that $(\emptyset, E)$ is a diagnosis. Therefore the restriction to exception clauses is not canonical and the proposed diagnosis computation is incomplete. Variants of the algorithm that solve this problem have been given in [5, 6].

Based on this work, the application of MBD techniques has been extended to debug programs written in concurrent, sequential, and functional languages.

7.2. Dependency-based debugging

The state of the art in hardware design is the use of hardware description languages such as VHDL (Very High Speed Integrated Circuit Hardware Description Language). VHDL is one of the most widely used hardware description languages and an IEEE standard. A specification (written in VHDL) delineates the functional requirements for the hardware to be designed. This is followed by a detailed design on the register-transfer (RT) level, which, once it has passed all tests, is automatically converted ("synthesized") into a gate-level representation. The design is tested by running simulations of the VHDL code for multiple test cases. The simulations produce traces of the values occurring over time on the signal lines in the simulated hardware. A significant part of the design effort is spent on detecting unacceptable deviations of these traces from the specification and subsequently localizing the sources of the faults. The main task in the course of this particular project was the development of a diagnosis system for supporting hardware designers in debugging their designs.

Lest the use of the term "hardware description language" cause misleading assumptions, note that despite its theoretically restricted goals, VHDL encompasses virtually all features of a 1980s general-purpose programming language in terms of control structures, data types, and modularization constructs. (Although this is not explicitly acknowledged, VHDL bears a strong resemblance to Ada-82.) Despite the slightly unusual terminology used, VHDL designers are in effect programmers that write programs of up to several 100.000 lines of code in an imperative programming language with a parallel processing layer added (for simulating the parallel execution of the finished hardware).

A VHDL program is divided into so-called entities (corresponding to abstract data types or Ada packages) whose implementation (called architecture) is decomposed into smaller entities. The interfaces between entities are defined in terms of signals. The bodies of entities (at every decomposition level) are made up out of processes that can assign values to signals in parallel. The values are computed from function expressions in terms of variables and other signal values. A process body can also consist of a conventional sequence of statements, contain user-defined function calls etc. A VHDL program can therefore be considered as a set of parallel processes, composed of signal assignments inside of which conventional imperative programs are executed. The imperative programs and functional expressions contained therein compute the actual values.

In theory, an actual executable specification exists, since the output of the early-stage (functional) specification is used to test the correctness of the late-stage (more strongly decomposed) implementation. However, due to the storage and organizational overhead that would result, the source of earlier versions is not available later in the design process, although the results of their test runs are, providing the requisite observations for the output signals of the design. The absence of a formal (or executable) specification also limits the use of formal verification techniques.

In order to achieve sufficient performance for practical applicability, a highly abstract model was used for the system description of VHDL programs in the initial prototype. The set of components was chosen to be the set of concurrent statements, i.e., basically the signal assignments. The system description is expressed purely in terms of whether the value of a given signal is correct or not, and does not refer to a particular value, i.e., it expresses the dependency of one signal on another, but not the exact relationship between the values.

Definition 7.5 (Functional Dependency) *Let C be a component of entity E and architecture A. An output $out_i \in out(C)$ depends functionally on a set $I \subseteq in(C)$ if a change of an input signal $in_j \in I$ at time t changes the output signal out_i at some time $t' \geq t$.*

When implementing the system, it was decided that a sufficient criterion for dependency was acceptable, and computing a minimal set of functional dependencies was postponed. An additional restriction was that pointer types (while present in the VHDL definition) could be ignored, since they would not be used in the designs handled by the system, so that aliasing situations .

Consider for example the following VHDL process (which is a small part of a VHDL program depicted in figure 1):

```
sync_fsm_start:  PROCESS(sclk,reset_n)
BEGIN
     IF (reset_n='0') THEN
          start_fsmi <= '0';
          start_fsm <= '0';
     ELSIF(sclk'event AND sclk='1') THEN
          start_fsmi <= or_start_fsm;
          start_fsm <= start_fsmi;
     END IF;
END PROCESS sync_fsm_start;
```

The functional dependencies are given by:
$\{sclk, reset\_n, or\_start\_fsmi\} \rightarrow start\_fsmi$
$\{sclk, reset\_n, start\_fsmi\} \rightarrow start\_fsm$
denoting the direction of the propagation of value changes. This information now can be used for diagnosis.

Definition 7.6 (System description) *The system description is given by*

– *For every component $C \in COMPS$ and $out_i \in out(C)$ where out_i depends functionally on the input signals $in_1 \ldots in_n$, a clause of the form*
  $ok(C) \wedge ok(in_1)_T \wedge \ldots \wedge ok(in_n)_T \rightarrow ok(C, out_i)_{T+1}$,
  and (for initialization purposes)
  $ok(C) \rightarrow ok(C, out_i)_0$.

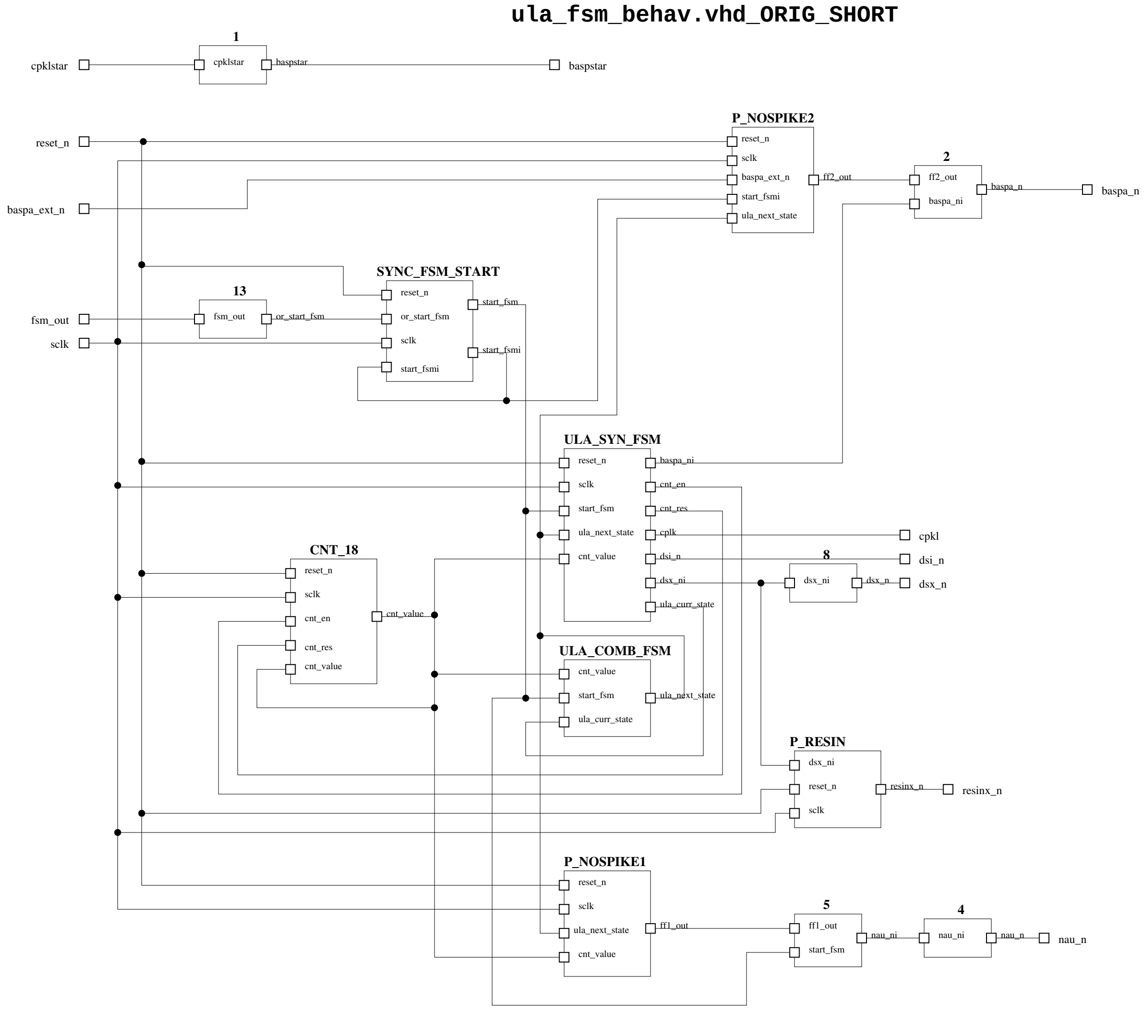


Fig. 1. The graphical representation of a (small) VHDL program

– *for each signal* $s \in Signals(A, E) \wedge s \in out(c) \wedge c \in COMPS$ *a clause of the form* $\bigwedge_{c_j \in Driver(s)} ok(c_j, s)_T \rightarrow ok(s)_T$

Note that this system description can be further reduced by using graph-theoretic methods. This technique first computes all strongly-connected components. The resulting graph then is used to compute a minimal system description and preserving correctness. With this system description diagnoses for even large programs can be computed on the order of minutes. For our example (source code size 585 lines of code, i.e., about 10 printed pages), diagnosis took less than 100ms (including conversion time) and reduces the search space to less than 1/3 of the original program.

The function *Driver* denotes the set of processes trying to assign values to s (this is necessary since multiple value assignments may be attempted in parallel). Colloquially phrased, the description for components specifies that a component which is correct will also produce correct output if its inputs are correct, as well as that initially, all signals are correctly initialized. The description for signals specifies that a signal is correct if no process attempts to assign an incorrect value to it. Observations correspond to statements that a particular signal is ok or not at some point in time.

The system is described in more detail in [15]. It is currently in test use, and has been shown to be fast enough for interactive use for diagnosing bugs in full-sized designs. At the moment, the main strategy decisions to be made are whether the system is supposed to diagnose a complete system or whether the search for the cause of a fault is supposed to descend hierarchically from the top level. Observations are derived from an automatic comparison of signal trace-files, but the user can also interactively specify signals as correct or incorrect. The system will use standard measurement selection algorithms to propose additional signals to be traced if an error cannot be localized with sufficient precision.

7.3. A value-oriented representation

It will not surprise the reader that the ability of the purely dependency-oriented system description in the previous section to distinguish between multiple possible statements as sources for a bug can be quite limited. In particular, if no observations (positive or negative) exist for any signal inside a feedback loop that produces an incorrect value, then any component inside that loop could be responsible. The number of signals traced typically is only a fraction of the signals in the system, so this scenario is not improbable.

Given that outside knowledge is not typically available, there are two ways of handling this problem. First, there is the possibility of using a more detailed representation. While a more detailed representation will not be practicable for systems of the size described in the previous section, the strategy of following the decomposition boundaries of a large system might eventually result in subsystems small enough for a detailed representation. We therefore defined a small, typed functional language, called EXP, with the goal of using it to study the effects of using a value-based system description [33]. The program below is a recursive multiplication function written in EXP (type specifications are external to the declaration of the function body and not shown).

```
myMult(x1,x2) =
    cond(ge(x1,1),
        plus(x2, myMult( minus(x1,1), x2))
        0)
```

The system description used for diagnosing an EXP program is no longer fixed solely on the basis of the program alone. Rather, since the individual values achieved during computation are important, and since (unlike for the parallel layer of VHDL processes) there is no concept of explicit time involved in the execution of an EXP function. As a result, this execution must be explicitly represented in terms of an evaluation trace graph, with each vertex in the graph corresponding to an expression, and arcs corresponding to the flow of data during evaluation. Figure 2 gives the graphical representation of the *myMult* example.

A vertex v representing an operator $\circ$ in the graph, with arcs leading from x and y to z will result in rules of the form

$$ok(x) \wedge ok(y) \wedge ok(v) \Rightarrow z = apply(\circ, x, y)$$

$$ok(x) \wedge ok(y) \wedge ok(v) \Rightarrow ok(z)$$

(The description has been simplified here for space reasons; in reality, the definition is different for the different types of EXP expressions, i.e., conditional expressions, user-defined and built-in function calls, etc.)

This model is somewhat better than using the purely dependency-based representation. For example, consider the expression $x * y$, with both arguments being integers, x known to be (correctly) 2 and the expected value 5. We then know that the operator $(*)$ is incorrect, since no choice of value can produce the correct (expected) output. The dependency-based approach, applied to such an expression, would only have told us that either y or the operator are incorrect. But the improvements are limited. For example, diagnosing recursive functions leads to the observation that all components may have caused the fault if only one expected input/output combination is given, mostly due to the hierarchical nature of function evaluation. If several measurement selections have been done, the approach can be more effective even for recursive functions. However, the approach of course does not explain the fault, it merely points out the location of the problem. Its main advantage is that it is fast enough for practical use even for larger examples.

7.4. A Repair-Based Representation

The approach presented in the previous section is still limited in its capability to judge the quality of diagnoses. To achieve better discrimination between diagnosis candidates, we search for a replacement that explains the faulty behavior, i.e., if the replacement is applied, the program will compute a correct result. Using a simple search and test algorithm would result in an effectively infinite search space (in principle, the whole program could be replaced by a program computing the same function using a different algorithm),

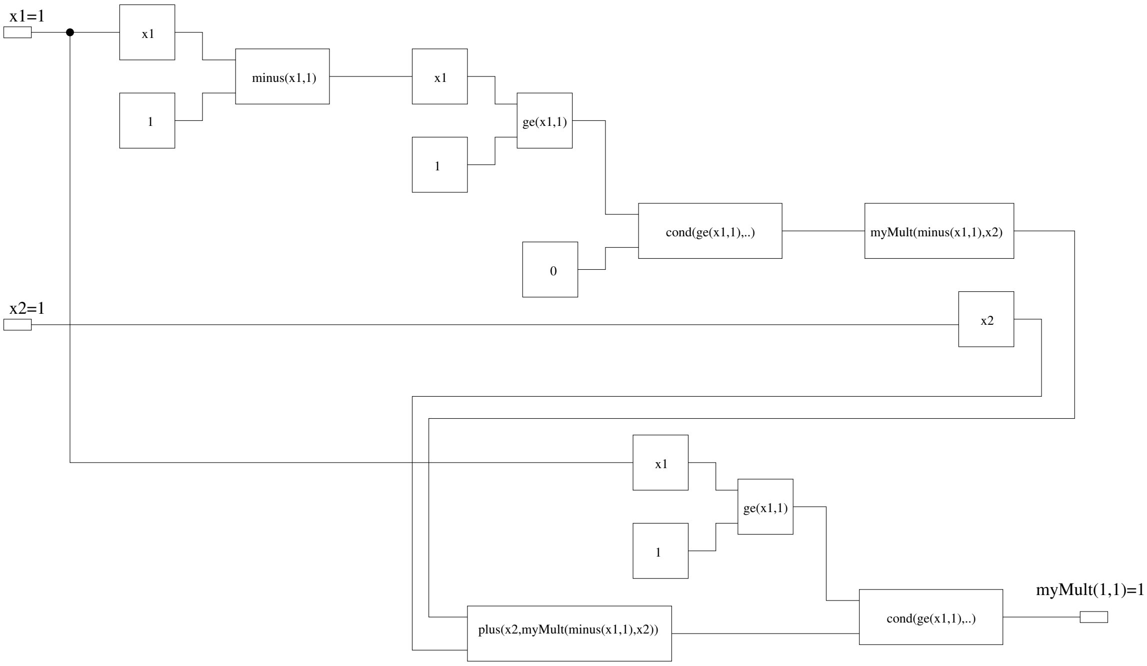


Fig. 2. Evaluation trace graph for the *myMult* example

therefore effective pruning methods are required. This section describes a way of selecting possible expression replacements which works well in practice for certain classes of errors. In general, it is possible to exclude classes of replacements that cannot lead to a fault explanation. As an example, assuming the type-correctness of a program (which can be guaranteed by conventional means), it is worse than useless to replace the function `plus` by a boolean `and` without further changes in the program. Other, similar restrictions can be applied for user-defined functions. Consider a function `test` and assume that other functions such as `test2, test3, myAnd, ...` are defined. In this case, a programmer would focus his attention on the functions `test2` and `test3` because of their similar names. Likewise, we assume that variables will often be replaced by variables and constants by constants. Using infinite domains (e.g., numbers) leads also to the question of how to focus on the optimal set of constants for a replacement, since trying them all is impossible. Here, a number of heuristics are possible, mostly based on syntactical similarity. For example, it is more likely that a programmer will mistype a `5` as a `6` instead of a `9`, or `5000` as `50000` instead of, say, `25325`. We therefore try to restrict the search for constant replacement to the nearest constants.

Expression replacement also introduces a new kind of problem to the diagnosis task. Since the program itself is changed, a terminating program might be changed to a non-terminating one. Theoretically, we would have to prove that a given replacement will not change termination by code inspection. To avoid problems regarding termination proofs we introduce an evaluation limit. Essentially, we stop evaluation if the replaced expression causes a much longer evaluation time (typically specified by the user) than the original (incorrect) one.

We write $\Pi|\Theta$ to denote the application of a set of expression replacements Θ to a program Π. We also define an ordering on expression replacements according to the number of changes implied by them. The distance $diff$ between two expressions is defined as number of different subexpressions, e.g., $diff(x1, x1) = 0$, $diff(x1, f(x1)) = 1$, and so on.

Definition 7.7 (Replacement Diagnosis) *Let* Π *be an* EXP *program* (*i.e., a set of user-defined functions*) *with* δ_Π *its function environment,* q *a query and* E *the* *expected value of the evaluation of q in the environment δ_{Π}. A set of expression replacements Δ is a diagnosis if the the expected value E' of the evaluation of q in the environment $\delta_{\Pi|\Delta}$ is consistent with OBS.*

We incorporate replacements in the consistency-based diagnosis approach by expressing replacements as fault models. To be exact, we introduce a fault mode $wrongexpr(x)$ which has the replacement expression x as parameter. Proving whether a replacement causes a contradiction is simple. Create a new body with the given replacements and call the evaluator. If the result is not equal to the expected value, a contradiction has been detected.

We use the function $modes : \mathsf{EXP} \mapsto 2^{MODES}$ to denote the set of all fault modes to be considered relative to an expression, where $MODES$ is the set of all fault models. Remember that since this set specifies all possible replacements, it is infinite in general and can be described by $MODES = \{wrongexpr(e)|e \in \mathsf{EXP}\} \cup \{ok\}$. As mentioned above, we are only interested in searching replacements that do not change too much of the original expression. In other words, we assume that the correct expression to be searched for is only a small variant of the wrong one.

So far, the repair-based approach has been implemented for primitive data types (i.e., logical and numeric types) only [32], an implementation for structured types (e.g., lists) is under construction. It should be noted that logical and numeric types are the most common types to be encountered in the VHDL domain which is supposed to serve as our testbed for the large-scale viability of these methods. The experiments have shown that using the value-based approach first and the repair-based approach for discriminating within the focus defined by the value-based approach can be advantageous, but this depends on the types and operators involved. In any case the repair-based approach alone will require significantly more time due to the large search space and the requirement to re-evaluate the function [32].

The embedding of the two value-based functional approaches into a general VHDL debugging system can be achieved in the same manner as the integration of the repair-based approach with the standard value-based approach. First, diagnosis candidates are computed using the dependency-based approach, and then re-evaluated (pruned) using the functional approaches.

7.5. Top-Down Consistency Based Diagnosis

A different model-based approach, referred to as top-down consistency based diagnosis, is the one taken by Bond [6]. It is based on the top-down declarative error diagnosis procedure of Lloyd [21]. The designation top-down is used because it descends successively deeper into the component hierarchy of the system to be diagnosed. Operationally, the approach is different from the declarative debugging approach in that complete diagnoses (instead of individual faults) are rejected when an oracle responses (a user query) indicates one of the faults included in such a diagnosis is not an acceptable explanation. In addition, the system does not require all user queries to be answered, instead an abductive theorem prover is used to derive assertions if the user cannot reply to a query. As a result, the number of queries that is answered by the oracle can be varied from all (in which case the system performs like a declarative debugger) to as few as possible, down to none at all (in which case the system performs like a model-based diagnosis system). Like a declarative debugging approach, and different from most other model-based approaches, the procedure generates minimal diagnoses sequentially instead of generating all minimal diagnoses at once.

Apart from operational differences, the work is highly interesting since it describes the underlying connection between the two different approaches when using a logic program as system description. Such a description can be the logic program itself that is to be diagnosed, but it could also be a logic program that describes the behavior of a hardware system (the paper includes the example of a full adder composed from logical gates). Formally, it is shown in the paper that if a system description is given in the form of a logic program, then the fault notions of consistency-based diagnosis and declarative debugging coincide.

7.6. Trial-based software diagnosis

The work of Hunt [16] transplants Raiman's trial approach for model-based diagnosis [27] to the software domain. The approach is likened to the incrimination and exoneration of suspects in a justice trial. The system generates candidate diagnoses, with search being focused by removing suspects which have an alibi from the candidate list. The concept is to generate the system description for diagnosis by generating causal models from the source code. The causal graphs for different parts of the resulting program state are matched to provide an overall representation of the immediately preceding actions in the program, and variables with correct values provide alibis for the statements assigning them. However, the paper does not discuss the system's implementation, or performance or discrimination issues for real-world programs.

8. Discussion

As mentioned in the introduction, the distinction between tutoring and diagnosis systems is one of the main watersheds in knowledge-based debugger design. Systems like PROUST, **Talus**, the Functional Representation [1], and Pudsy [22] gain their strong diagnostic power at the price of requiring the existence of an external specification, which is not available in a general software development environment. In contrast, the other approaches presented here were introduced as a way of supporting designers and implementors of large software artifacts, and are based on the realization that external specifications will not be forthcoming. A certain loss of perspicacity is thus unavoidable, but must and can be accepted. This property is shared with the Algorithmic Debugging approach, which has enjoyed a certain longevity at least among researchers (and was recently shown to be formally a special case of model-based software diagnosis [6]).

The dependency-based representation could be considered a weak form of program slicing [34, 35]. Weak, because the dependency representation is more general than individual slices; as noted by Weiser, slices need not be unique, nor must the existence of a unique minimal slice be guaranteed. The dependency network, on the other hand, covers all possible slices. This makes it weaker in terms of discrimination, but guarantees that no diagnoses will be lost. However, it should be noted that program slicing is a particular technique; it is not a framework for debugging in general. Likewise, the dependency-based representation is one of several representations usable in the model-based diagnosis framework, and the flexibility that makes the smooth incorporation of diverse representations possible that is part of the power of the model-based approach.

In [14] verification is described as an debugging technique. The verification task itself is to compare a program with its specification and answering true if both are equal and false otherwise. Verification techniques includes *symbolic execution*, i.e., the formal equivalence proof, *program transformation*, *algorithm recognition*, *symbolic derivation of assertions* and others. Most of the current available verification enginees are not intended to be used as debugger. Therefore fault localization is hardly utilized and if available do not provide enough informations for doing so. However, there are some debugging systems which are strongly related to formal verification, e.g., *algorithmic program debugging* mentioned earlier [30]. In such systems the behavior is verified.

Most verification systems comparing formal specifications with the actual program use theorem provers (e.g., [4]). The specification of the properties to be verified must be explicitly written in the theorem prover's logic whereas the program is automatically converted into the formal system. The theorem prover then proves whether a contradiction arises or not. More recent work in this area [9, 23, 11] uses a model checking approach, i.e., a formal specification (written in a branching-time logic, e.g., CTL) and the converted program are used to prove whether a model for the resulting logical sentence exists or not. This approach is currently used for verifying VHDL programs. In [17], a verification system is described that uses (functional) dependencies associated with program parts. Instead of verifying the concrete behavior, only the intended dependencies are compared, which is much easier computationally.

While debugging systems and verification systems support different tasks, i.e., localizing bugs in the program versus finding misbehavior, both can probably be effectively combined. Most verification systems provide at least one input-output vector for which the concrete program does fail its specification. This test case could be used directly in a model-based debugging system.

8.1. User interaction

Generally, the assumption is made that all required information is available at the start of the diagnosis process. If observations or SD are too weak to determine a unique diagnosis, resulting in a large set of potential diagnosis candidates, attempts are made to gain additional observations by testing. Since for hardware diagnosis, testing is usually a fairly cumbersome process compared to diagnosis (involving technicians taking measurements and possibly having to gain access to other parts of the diagnosed artifact), there is a strong desire to minimize the number of tests required. The process of *measurement selection* attempts to determine the locations most likely to provide better discrimination between diagnoses. The measurements taken at these locations will then be added to the observations for another run of the diagnosis system. In addition, whole new *test cases* can also be added to the specification of the diagnosis problem.

In automated debugging systems, typically a strong distinction is made between systems that have all knowledge before they are started and those which consult the user for additional information as debugging progresses (called *static* and *interactive*, respectively, in [31]). While the model-based diagnosis process as described above is closer to the first type, given sufficiently high performance, an MBD system can be employed in what amounts to the second way by successive cycles of diagnosis activity and (if necessary) requests for additional user input.

9. Conclusion

In this paper, we have given an overview of AI approaches to the intelligent debugging of software, starting with intelligent tutoring systems that can operate on a detailed understanding of the purpose of the programs to be diagnosed. The PROUST system does this by explicit representation of potential programmer intentions in terms of plans, the **Talus** system uses a theorem prover to find differences to the semantics of an explicitly specified reference function.

The next group is presented by functional approaches (ZD and DUDU) which also attempt to provide a detailed additional specification at multiple levels of a program that is intended to show how smaller functions interact to produce the total function of the system as a whole. While a detailed specification is necessary, the intent has moved beyond the novice programmer to functionwise specification of general applications.

In the probabilistic diagnosis of the DAACS system, this individual specification is replaced by expert analysis of example cases for specific categories of runtime errors. Additional focusing is provided by logical analysis of execution paths that remove parts of the program from consideration as possible sources of the error.

For the model-based representations described in the final section, while the representations are quite different in nature, they share some basic characteristics: the system description is expected to specify the semantics of the program when executed, albeit at a (sometimes very) abstract level, and the provided observations specify the expected output values (or, abstractly, whether these output values were correct). Also, none of these approaches require a formal specification external to the program. In some cases, this is facilitated further by the fact that they operate directly (and exclusively) on logic programs so that the translation from program to model is trivial, and analysis to any desired level is possible by exploiting the declarative and logic characteristics of the logic program. For the dependency-based, value-based, and repair-based representations, a separate level of translation is necessary. These three representations correspond to a move from abstract to detailed representation. With the most abstract representation being suitable for diagnosing even large VHDL programs, the other representations can be introduced if desired to provide clearer discrimination between diagnoses where the abstract dependency-based representation fails. As mentioned in [31], this progress from a start at a more abstract level to a more detailed level of fault-searching is also the strategy used by expert programmers.

References

[1] Dean Allemang and B. Chandrasekaran. Functional representation and program debugging. In *Proc. IEEE Knowledge-Based Software Engineering Conference*, pages 136 – 143. IEEE, 1991.

[2] Lisa Burnell and Eric Horvitz. A Synthesis of Logical and Probabilistic Reasoning for Program Understanding and Debugging. In *Proceedings of the International Conference on Uncertainty in Artificial Intelligence*, pages 285 –291, 1993.

[3] Lisa Burnell and Eric Horvitz. Structure and Chance: Melding Logic and Probability for Software Debugging. *Communications of the ACM*, pages 31 – 41, 1995.

[4] R.S. Boyer and J.S. Moore. *A computational logic handbook*. Academic Press, 1988.

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

[6] Gregory W. Bond. Top-down consistency based diagnosis. In *Proceedings of the Seventh International Workshop on Principles of Diagnosis*, Val Morin, Canada, 1996.

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

[8] G. W. Bond and B. Pagurek. A Critical Analysis of "Model-Based Diagnosis Meets Error Diagnosis in Logic Programs". Technical Report SCE-94-15, Carleton University, Dept. of Systems and Computer Engineering, Ottawa, Canada, 1994.

[9] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Transactions on Database Systems*, 8(2):244–263, April 1986.

[10] Luca Console, Gerhard Friedrich, and Daniele Theseider Dupré. Model-based diagnosis meets error diagnosis in logic programs. In *Proc. IJCAI*, pages 1494–1499, Chambery, August 1993. Morgan Kaufmann Publishers, Inc.

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

[12] Luca Console and Pietro Torasso. Integrating models of correct behavior into abductive diagnosis. In *Proc. ECAI*, pages 160–166, Stockholm, August 1990. Pitman Publishing.

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

[14] Mireille Ducassé. A pragmatic survey of automatic debugging. In *Proceedings of the 1st International Workshop on Automated and Algorithmic Debugging, AADEBUG '93*, Springer LNCS 749, pages 1–15, May 1993.

[15] Gerhard Friedrich, Markus Stumptner, and Franz Wotawa. Model-based diagnosis of hardware designs. In *Proc. ECAI*, Budapest, August 1996.

[16] John Hunt. Model-based software diagnosis. Technical report, University of Wales, Aberystwyth, 1996.

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

[18] W. Lewis Johnson. *Intention-Based Diagnosis of Novice Programming Errors*. Pitman Publishing, 1986.

[19] Henry Lieberman. The debugging scandal and what to do about it. *Communications of the ACM*, 40(4), 1997.

[20] Beat Liver. Modeling software systems for diagnosis. In *Proceedings of the Fifth International Workshop on Principles of Diagnosis*, pages 179–184, New Paltz, NY, October 1994.

[21] J. W. Lloyd. Declarative Error Diagnosis. *New Generation Computing*, 5:133–154, 1987.

[22] F. J. Lukey. Understanding and debugging programs. *International Journal on Man Machine Studies*, 12(2):270–279, 1978.

[23] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993. ISBN 0-7923-9380-5.

[24] William R. Murray. *Automatic Program Debugging for Intelligent Tutoring Systems*. Pitman Publishing, 1988.

[25] Lee Naish. Declarative diagnosis of missing answers. *New Generation Computing*, 10:225–285, 1992.

[26] David Poole. Explanation and prediction: an architecture for default and abductive reasoning. *Computational Intelligence*, 5(2):97–110, 1989.

[27] Olivier Raiman. Diagnosis as trial - the alibi principle. In *International Model-Based Diagnosis Workshop*, Paris, July 1989.

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

[29] V. Sembugamoorthy and B. Chandrasekaran. Functional representation of devices and compilation of diagnostic problem-solving systems. In Janet Kolodner and Christopher Riesbeck, editors, *Experience, Memory, and Reasoning*, pages 47–73. Lawrence Erlbaum Associates, 1986.

[30] Ehud Shapiro. *Algorithmic Program Debugging*. MIT Press, Cambridge, Massachusetts, 1983.

[31] Nahid Shahmeri, Mariam Kamkar, and Peter Fritzson. Usability criteria for automated debugging systems. In *Proceedings of the 1st International Workshop on Automated and Algorithmic Debugging, AADEBUG '93*, Springer LNCS 749, pages 16–39, May 1993.

[32] Markus Stumptner and Franz Wotawa. A model-based approach to software debugging. In *Proceedings of the Seventh International Workshop on Principles of Diagnosis*, Val Morin, Canada, 1996.

[33] Markus Stumptner and Franz Wotawa. Model-based program debugging and repair. In *Proceedings of the International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems*, Fukuoka, 1996.

[34] Mark Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–452, July 1982.

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