Model-Based Program Debugging and Repair*

Markus Stumptner and Franz Wotawa
Christian Doppler Laboratory for Expert Systems
Institut für Informationssysteme, Technische Universität Wien
Paniglgasse 16, 1040 Wien, Austria, Europe
Email: {mst,wotawa}@dbai.tuwien.ac.at

Abstract

The current state of the art in integrated circuit design is based on the use of special hardware design languages such as VHDL. In the context of the development of an intelligent, knowledge-based debugging aid for VHDL programs, we are dealing with analysis and diagnosis of a subset of VHDL (which is similar to conventional concurrent programming languages). We present an adaptation of conventional model-based diagnosis methods to the debugging of VHDL expressions and signal assignments. The examination of possible faults in VHDL expressions leads to the use of fault models (i.e., a representation of typical errors) as an aid to focusing, and as a basis for proposing repair actions for small errors in these programs.

1 Introduction

Over the last decade, the model-based approach has achieved widespread use in hardware diagnosis [Rei87, dKW87]. While model-based techniques have also found recognition in the field of design in general [Wil90], their application to software design in particular has remained very limited. In more recent years, different authors have examined the use of model-based diagnosis for debugging logic programs [CFD93], but their efforts were limited to small examples. The work described in this paper attempts to extend this basis towards a more generally usable approach for diagnosing software, which is not limited to the particularly benevolent environment of pure logic programming languages, and also scales up to meaningful error search in programs of a more realistic size.

In the DDV (Design Diagnosis for VHDL) project, the authors have applied model-based diagnosis to the debugging of hardware designs written in the VHDL language [FSW95]. Hardware specification languages such as VHDL (an IEEE standard and the most widespread such language in current use) combine the syntax and expressiveness of a full conventional programming language with special parallel constructs (to simulate the concurrent internal workings of integrated circuits), data types (e.g., the multivalued logics used by circuit designers to describe different states of signal lines). VHDL development environments are delivered together with large libraries containing predefined standard circuits, such as the standard logical gate types. The main reason for the use of design languages is the attempt to avoid, by whatever means possible, the surfacing of design errors after a lot of money has been spent on production of the first prototypes of the board or chip. Circuit designs are successively refined to a degree where they can be automatically transformed ("synthesized") into the final gate-level representation. Large hardware designs (comprising multiple ASIC's and microprocessors) can reach dimensions of several 100.000 lines of VHDL code and thousands of components and signals at the top level. For such designs, typically written by a large team of designers, testing, fault detection and localization become very time-consuming activities, because they involve simulations of an actual circuit that can take hours or days, scanning of the resulting signal traces by hand, and subsequent identification of the error in a large program parts of which may have been produced by outside subcontractors. In effect, these designs present all the problems of software debugging with the additional problem that test runs and actual finding of misbehavior take a lot of manpower and real time. An automated assistant for fault-finding thus presents the possibility for major cost savings in the design process.

The tool developed in the DDV project is used for testing and diagnosing purposes. The trace files pro-

*This work was co-sponsored by the Christian Doppler Laboratory for Expert Systems and Siemens Austria

duced by a simulation run are compared with so-called specification traces, i.e., traces produced by the simulation of a less detailed, already tested version of the program. The output of the simulation run is a set of discrepancies, i.e., signals whose values differ between the two trace files. The set of discrepancies is passed to the diagnosis generator, which attempts to find the source of the error, with a model that is based on static analysis of the data dependencies inside the program. This approach has the significant advantage that the design can remain unaltered. No special code annotations are used, and separate axiomatizations (as commonly used by design verification systems) or complex test benches are unnecessary.

The current version of the tool (which is being tested by actual developers at the moment) uses a very abstract domain model which can optimally be used on the top level of VHDL programs but is not as good while diagnosing the lowest levels of the decomposition hierarchy making up the system. The model is, in effect, based on the topology of the data dependencies between different parts of the program (different components in the finished design). Detailed time values and signal values are ignored. One reason for this approach was the size of the systems involved. Diagnosing a 100,000 line program by reasoning about the detailed semantics of each individual expression in the program is effectively impossible for performance reasons. The abstract representation delivers an answer in minutes (which is a fraction of the time used for executing the simulation of the test run). However, in circuits with complex structure (i.e., with large feedback loops), the ability to discriminate between different possible diagnoses is limited.

The work described in this paper attempts to improve diagnosis quality in these cases, by introducing a more sophisticated model which uses detail information on signal values and the structure of assignments to achieve stronger focusing and discard more incorrect explanations for the fault. This requires analysis of the expressions that compute the values assigned to signals. Such expressions can contain arbitrary user-defined functions that are defined using the Ada-like sequential part of the VHDL language. Therefore, techniques for diagnosing faults not only in functional expressions, but also in imperative programs, may be ultimately required. Interestingly, the fact that the structure of a design may change over time (whereas the structure of a finished circuit is usually assumed to be unchanging), implies that considering repair actions may be a useful heuristic for focusing on the actual errors.

The paper is structured as follows: We first present the basic definitions of Model-Based Diagnosis (MBD) as they pertain to our particular domain. We then give (very small) example of the basic VHDL code, and discuss the various error types that could occur, as well as their representation by fault models in the MBD paradigm and the use of repair in focusing the search. Finally, we provide an overview of related research.

2 Basic Definitions

For the standard definition of model-based diagnosis, see [Rei87].

Definition 2.1 (Programming Language) *A programming language $\mathcal{L}$ is a tuple (L, S) consisting of a syntax L and a semantic description S.*

We assume that the description of the $\mathcal{L}$ semantics is given in first order logic. We use the term $COMP$ to denote the diagnosis components (statements, expressions, blocks) of a program.

Definition 2.2 (Program) *A program Π is an element of L^* where L^* denotes all syntactically correct programs of $\mathcal{L}$.*

Given a set of observations OBS describing the actual behavior of the program and a system description SD representing the semantics of the program, a debugging problem basically consists of fixing the program so the SD of the new fixed program will produce the predicted observations.

Definition 2.3 (Program Diagnosis) *Let Π be a program written in $\mathcal{L}$ and OBS be observations on values of variables (signals) used in Π. A diagnosis of Π is a function Δ associating a single mode with every (diagnosis) component such that the following proposition holds:*

$$OBS \cup S(\Pi) \cup \{\Delta(C) | \Delta(C) \in m(C) \wedge C \in COMP\}$$

The diagnosis task can be described as the search for an assignment of fault models to diagnosis components that explains the behavior given by the observations. It can be easily shown that the number of fault models for a program is infinite in general. However, use of a finite set of fault models will usually be sufficient. The usage of fault models can be guided by introducing additional information.

3 Problem Description

Consider the following simple VHDL program, implementing a three input and-gate with delay.

```
entity AND3_GATE is
     generic( delay:  time := 0 fs );
     port( signal I1, I2, I3:  in std_logic;
          signal O: out std_logic );
end AND3_GATE ;

architecture BEHAVIOR of AND3_GATE is
begin
     O <= I1 and I2 after delay;
-- Correct: O <= I1 and I2
--              and I3 after delay;
end BEHAVIOR;
```

Assume further that we have detected a discrepancy for the test case: $I1 = 1$, $I2 = 1$, $I3 = 0$.

In this case the fault in the program can be found by simply looking at the functional dependencies, i.e., which signals are involved in the computation of the incorrect signals. However, such a representation is limited in its expressibility because it gives no hint how the bug in the program should be corrected, although a number of simple alterations would produce the desired effect. For example, the correction given above is minimal in terms of the number of expressions changed in the program. A second possible (but not minimal) repair step consists of making two small program changes, resulting in

```
     ...  signal DUMMY: std_logic;
begin
     DUMMY <= I1 and I2;
     O <= I3 and DUMMY after delay;
     ...
```

Note that if an expression were syntactically restricted to contain one top-level operator (i.e., no multiple `and`'s), the second suggestion would actually be preferable. In general, it can be said that program repair is limited by

- The syntax and semantics of the programming language.
- The original program and a metric related to the complexity of repair actions.
- The given test cases.

The task of a diagnosis-repair algorithm must be to compute a repair plan using the restrictions above. These repair plans can be ordered using a metric which assigns a high cost if many changes are necessary and a low cost if this is not the case. By generating only plans with a bounded cost, the search space can be restricted. Note that for repairing a program, this original search space is infinite, since arbitrarily complex alterations can in principle be made to any program without detrimental effect on its correctness (e.g., adding variables that are never used in computing the result, and computations on them).

A restriction of the search space can also be achieved through assumptions about the possible types of faults. If a finite number of such assumptions, called *fault modes* in model-based diagnosis, is used, the search space is again restricted to a subset of the original one. Depending on the fault modes allowed, that subset can be finite or not. The assumption of finiteness can be made because we assume that the originally intended program can be produced by a small variation of the incorrect program. Fault modes for parts of a program can be viewed as repair proposals – a fault mode, by specifying a particular kind of error, also offers possible choices of how to correct it. Assuming a typical block structure as is usually the case with concurrent or imperative languages, fault modes can be associated with statements, expressions inside statements and statement blocks. We call the description of a fault mode a fault model.

For example, take a VHDL signal assignment (as shown above) that produces incorrect system behavior even though the values for the input signals are correct for that particular point in execution. The target signal on the left can be wrong, the expression can be incorrect (in which case one has to look inside the expression to find the error), or the statement itself should be replaced by another statement. The last fault mode can be expressed by a replacement of the statement with a null statement that has no effect on signal values in conjunction with the addition of another statement.

The addition of a new statement can be viewed as fault mode of a statement block. Since VHDL is a concurrent language, a statement can be added at the end of a block without changing the meaning of the other statements for a given point in time. This is not possible for sequential languages where the location of the statement has normally an immediate influence on the behavior, i.e., the execution of the subsequent code. Obviously, adding a statement is a complex change, due to the many degrees of freedom involved: statement type, argument signals, target signal, and the functions used to compute the output value must be determined. The only information available are the test case and the context, e.g., all other statements in the program. It would therefore be unrealistic to assume that adding statements results in a meaningful limitation on the search space except in very simple or restricted cases such as the one depicted above.

4 System Description

This subsection shows how behavior and fault models can be described using first order logic. We demonstrate the system description using only the VHDL signal assignment statement with a reduced syntax.

SignalAssignment ::=
Signal <= *Expression* `after` *TimeExpression*

Expression ::=
Signal | Constant |
(*Expression BinOp Expression*)

The semantics of this statement can be expressed as first order rules. In the system description, the correct behavior of a statement is associated with the *ok* mode of the statement. In other words, a statement S is assumed to work correctly if $ok(S)$ holds.

We introduce a predicate $type/2$ indicating the type of a concurrent statement, for example $type(SA, signalAssignment)$ holds if SA denotes a signal assignment. VHDL differentiates between five such types: *signalAssignment, processStatement, generateStatement, assertionStatement, componentInstantiation*.

The system description that is used for diagnosis is the union of the individual statement descriptions as given below. Note that this presentation is simplified and does not take the full VHDL behavior into account. A more sophisticated description is used in practice.

Every signal defined in a VHDL program has an associated initialization value (at time 0). This behavior is described by facts of the form

$value(S, V, 0)$, $event(S, V, 0)$

where V is either a value which is related to the type of S, or an explicitly given statement. This statement is part of the signal definition. The *event* predicate indicates that a signal has changed its value.

To define program execution we first must define what expressions evaluate to. This is specified via the predicate *evaluate* as follows.

$$
\begin{aligned}
& evaluate(E_1\ Op\ E_2, V, T) \leftrightarrow (evaluate(E_1, V_1, T) \wedge \\
& \quad evaluate(E_2, V_2, T) \wedge Op(V_1, V_2, V)) \\
& evaluate(C, C, T) \leftrightarrow constant(C) \\
& evaluate(S, V, T) \leftrightarrow signal(S) \wedge value(S, V, T)
\end{aligned}
$$

The following rule describes the semantics of a correctly working concurrent statement of the form $SA = S$ `<=` E `after` TE, where V is the assigned value, and T' the time the value appears on S (as opposed to the time when the statement is executed):

$$
\begin{aligned}
& mode(ok(SA)) \rightarrow \\
& \quad (active(SA, T) \wedge evaluate(E, V, T) \wedge \\
& \quad evaluate(TE, D, T) \wedge T' = T + D \\
& \qquad \leftrightarrow value(SA, S, V, T'))
\end{aligned}
$$

The *active* predicate is true if a statement can be executed at time T. This is the case when at least one input signal changes its value.

$$
\begin{aligned}
& value(S, V, T) \wedge value(S, V', T') \wedge \\
& \nexists T'' : T' < T'' < T \wedge value(S, V'', T'') \wedge V \neq V' \\
& \quad \rightarrow event(S, V, T) \\
& event(S, V, T) \wedge input(S, St) \rightarrow active(St, T)
\end{aligned}
$$

If a statement St is active is not true at a time point T, we assume that the value of the signals changed by St is equal to the value it had before T.

$$
\begin{aligned}
& \neg active(St, T) \wedge value(St, S, V, T') \wedge \\
& \nexists T'' : T' < T'' < T \wedge value(St, S, V'', T'') \rightarrow \\
& \quad value(St, S, V, T)
\end{aligned}
$$

If multiple concurrent statements attempt to assign values to the same signal, the signal needs to be *resolved* (see [VHD88] for more details). A concurrent statement that change a signal's value is called a driver for this signal. We introduce a predicate *drivers* defining all drivers for a signal.

$drivers(S, \{St_1, \ldots, St_n\})$

The value of the signal is now determined by the values of the signal drivers. Therefore add the following rule to the system description:

$$
\begin{aligned}
& drivers(S, D) \wedge valueD(S, V, T, D) \rightarrow value(S, V, T) \\
& value(St, S, V, T) \rightarrow valueD(S, V, T, \{St\}) \\
& (value(St_1, S, V_1, T) \wedge valueD(S, V_2, T, \{St_2, \ldots\}) \wedge \\
& \quad resolved(V_1, V_2, V)) \rightarrow valueD(S, V, T, \{St_1, St_2\})
\end{aligned}
$$

5 Fault modes

We now come to the description of the various possible fault modes. First, consider an assignment SA that assigns to the wrong target signal, with S' being the correct signal:

$$
\begin{aligned}
& mode(wrongTarget(SA, S')) \rightarrow \\
& \quad (active(SA, T) \wedge evaluate(E, V, T) \wedge \\
& \quad evaluate(TE, D, T) \wedge T' = T + D \\
& \qquad \leftrightarrow value(SA, S', V, T'))
\end{aligned}
$$

We can restrict the use of fault modes applicable to an incorrect signal assignment to those that deal with signals not already known to be correct due to observations. If we assume that the expression returning the value is incorrect, this is formalized as follows:

$$\begin{aligned}&mode(wrongExpression(SA, E')) \rightarrow\\&\quad (active(SA, T) \wedge evaluate(E', V, T) \wedge\\&\quad evaluate(TE, D, T) \wedge T' = T + D\\&\qquad \leftrightarrow value(SA, S, V, T'))\end{aligned}$$

where $SA' \equiv (S <= E'$ `after` $TE)$ is an alternative statement, and $timeExpression(SA, TE)$ holds. A fault model describing that a statement should be excised can be formalized by:

$$mode(wrongStatement(SA)) \rightarrow \top$$

Next, we describe the case where the behavior of the statement is not known, e.g. the *ab* mode usually used in model-based diagnosis. We denote this mode by $undef$.

$$\begin{aligned}&mode(undef(SA)) \rightarrow\\&\quad (active(SA, T) \wedge evaluate(TE, D, T) \wedge\\&\quad T' = T + D\\&\qquad \rightarrow value(SA, S, \epsilon, T'))\end{aligned}$$

ϵ is a value not used in any signal type indicating that the actual value is unknown. Operations on this value always cause as result the value ϵ.

Finally, a program could also be modified by adding complete statements to it and leave the rest unmodified. We do not discuss this (rather complex) operation due to space reasons.

Basically, the model-based diagnosis process works by attempting to find fault mode assignments that are consistent with the observations. Observations are represented by two facts for each observation.

$$value(S, V, T),\ event(S, V, T)$$

Contradictions occur if different values are assigned to the same signal:

$$\begin{aligned}&(value(S, V, T) \wedge value(S, V', T) \wedge V' \neq V \wedge\\&\quad V' \neq \epsilon \wedge V \neq \epsilon) \rightarrow \bot\end{aligned}$$

The aim of the repair task is to determine a new expression or statement explaining the faulty behavior, i.e., replacing the incorrect part of the program with the correct one. The search space for this task can be limited by the knowledge of the correct behavior, e.g. by knowing which signals depend functionally on other signals, and by the assumption that the new expression should be a small variant of the old one. This assumption explains the difference between the automatic programming community and our notion of correcting faults. While automatic programming deals with the question of automatically writing programs out of (a series of) examples or (logic) descriptions, we use an existing program and search for a small variant.

We do not discuss in this paper how temporal misbehavior can be found and corrected, but we claim this can also be achieved using appropriate fault models.

Note that the fault models as used above effectively induce a repair approach. If a diagnosis has been found, the associated fault models specify how to repair the program with regard to the test cases used. This is a significant difference to the standard hardware diagnosis approach, which is made possible by the malleability of software. In effect, finding the repair means that it can be immediately executed by altering the source code, whereas the opportunity for arbitrary modifications is usually very limited in hardware devices.

6 Related Research

The notion of using knowledge-based systems as a support for software and related design tasks is not new. Previous research efforts usually aimed at working with languages that had strictly limited expressiveness, such as telecommunications protocols [Rie93] or logic programming [CFD93]. The latter paper also presents arguments for the superiority of model-based reasoning over the algorithmic program debugging approach originating in [Sha83].

Much work has also been done in knowledge-based debugging support for software. We cite two examples. The PROUST system [Joh86] requires a separate description of the program to be manually developed and entered. Similarly, in [AC91], an axiomatization of the intended functionality is necessary. The **Talus** system [Mur88] attempts to match student solutions to example programs to find the source of divergent outputs. Both for PROUST and **Talus**, the goal is to aid students, i.e., novice programmers, in finding as many bugs as possible in small programs. In contrast, in our case, the goal is to help experts (at least, experienced users) orient themselves in very large programs of different structure and functional architecture.

Finally, we claim that our approach avoids many of the problems arising in automatic programming [WL69]. We do not deal with the idea of finding a program out of examples or a formal description. We only require that correct functioning of the program can be attained by small variations of the original, incorrect one. The search space for finding such a variant is much smaller and can be further restricted by making heuristic assumptions about fault models.

Of course restricting the number of fault models may exclude diagnoses. However, in this case, the user or the program itself still has the choice of relaxing the assumptions at the cost of a larger search space.

7 Conclusion

In this paper, we have introduced a model representing the semantics of the subset of the VHDL language (that deals with concurrent statements and expression evaluation) and its usage for diagnosing real-world VHDL programs and making limited repair proposals.

We plan to integrate the more sophisticated system description presented in this paper into the existing VHDL diagnosis tool. This can be achieved by moderate alterations and use of the new system description. The existing diagnosis tool consists of a theorem prover which is used to generate contradictions from SD and OBS, and a hitting set generator that chooses fault modes and uses the theorem prover for computing diagnoses. See [Rei87, GSW89] for an introduction about model-based diagnosis and the hitting set algorithm. VHDL programs are parsed and automatically converted into the SD representation.

The first version of our diagnosis tool, including a tool for automatic detection of misbehaviors by comparing waveform traces of VHDL programs, is currently being used in practice. Experience has shown that the current tool is well suited for automization of regression tests and additionally help the VHDL programmer to focus on the right diagnoses. However, more information about possible diagnoses and hints about how to repair a misbehavior will increase the effectiveness of the tools in many cases.

The fact that fault models (usually used in model-based diagnosis to improve the diagnosis result) are equal to possible repair actions in the software domain is crucial to this approach. We have introduced a number of ways for limiting the infinite space of possible repair actions. In particular, we assume that the corrected program is only a variant of the actual one that can be found with limited alterations. This is a very useful heuristic for ordering the search space, although more involved methods must be used to deal with fundamental structural defects in the program [Liv94]. Unfortunately, such methods have also never been used with regard to larger systems. This is an ongoing research effort.

References

[AC91] Dean Allemang and B. Chandrasekaran. Functional representation and program debugging. In *Proc. IEEE Knowledge-Based Software Engineering Conference*, pages 136 – 143. IEEE, 1991.

[CFD93] 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.

[dKW87] Johan de Kleer and Brian C. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32(1):97–130, 1987.

[FSW95] Gerhard Friedrich, Markus Stumptner, and Franz Wotawa. Model-based diagnosis of hardware designs. In *Proc. DX'95 Workshop*, Goslar, October 1995.

[GSW89] 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.

[Joh86] W. Lewis Johnson. *Intention-Based Diagnosis of Novice Programming Errors.* Pitman Publishing, 1986.

[Liv94] Beat Liver. Modeling software systems for diagnosis. In *Proceedings on the Fifth International Workshop on Principles of Diagnosis*, pages 179–184, New Paltz, NY, October 1994.

[Mur88] William R. Murray. *Automatic Program Debugging for Intelligent Tutoring Systems.* Pitman Publishing, 1988.

[Rei87] Raymond Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–95, 1987.

[Rie93] Marc Riese. *Model-based diagnosis of communication protocols.* PhD thesis, EPFL Lausanne, 1993.

[Sha83] Ehud Shapiro. *Algorithmic Program Debugging.* MIT Press, Cambridge, Massachusetts, 1983.

[VHD88] IEEE Standard VHDL Language Reference Manual LRM Std 1076-1987, 1988.

[Wil90] Brian C. Williams. Interaction-based invention: Constructing novel devices from first principles. In *Proceedings AAAI*, Boston, August 1990. The MIT Press.

[WL69] R. J. Waldinger and R. C. T. Lee. PROW: A step toward automatic program writing. In *Proceedings of the International Joint Conference on Artificial Intelligence*, Washington, DC, May 1969.