Model-Based Diagnosis of Hardware Designs

Gerhard Friedrich

Siemens AG Österreich
Abteilung PSE EZE TNA 1
Erdberger Lände 26, A-1030 Wien, Austria
phone: (+43) - 1 - 717 11–5030
fax: (+43) - 1 - 717 11–5425
gfr@garwein.hai.siemens.co.at

Markus Stumptner
Franz Wotawa

Institut für Informationssysteme
Technische Universität Wien
Paniglgasse 16, A-1040 Wien, Austria
phone: (+43) - 1 - 588 01–6127
fax: (+43) - 1 - 505 53 04
{mst, wotawa}@vexpert.dbai.tuwien.ac.at

Abstract

The state of the art in hardware design is the use of hardware description languages such as VHDL. The designs given by VHDL programs are refined up to the point where the physical realization of the new circuit or board can be created automatically. The designs are tested by simulating them and comparing their output to that prescribed by the specification. A significant part of the design effort is taken up by detection of unacceptable deviations from this specification and the subsequent localization of the sources of such faults. In this paper, we describe an approach to employ model-based diagnosis for fault detection and localization in very large VHDL programs (several 100.000 lines of code). In order to achieve sufficient performance for practical applicability, we have developed a representation that allows a highly abstracted view of programs and faults, but is sufficiently detailed to yield substantial reductions in the fault localization costs when compared to the current manpower-intensive approach. The implementation in conjunction with the knowledge representation is designed with openness in mind in order to allow use of the highly optimized simulation tools available. Discrimination between diagnoses is achieved by use of multiple test cases and measurement selection. Test results show sufficient performance for practical use.

1 Introduction

Recently, a number of authors have proposed the use of model-based diagnosis for localizing faults in the designs of artifacts, e.g., bugs in software systems or communications protocols [CFD93; Rie93]. In this paper, we describe the use of diagnosis techniques according to the consistency-based approach [Rei87; dKW87] for identifying faults in hardware design descriptions. Since a substantial part of the overall effort in hardware design is spent trying to discover misbehavior and to identify the design errors resulting in this misbehavior, the use of diagnosis utilities will produce a significant return of investment.

Figure 1 is an overview of the hardware design cycle using a language such as VHDL (Very High Speed Integrated Circuit Hardware Description Language). We focus on VHDL because it 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 a number of test cases. The simulations produce traces of the values occurring over time on the signal lines in the simulated hardware (so-called waveform traces; see Figure 2 for an example). The waveform traces of the program to be tested, usually comprising several 10.000 signal changes, are compared to the traces generated by the specification on the same input values. A discrepancy indicates the existence of a fault in the design. The average simulation runtime increases strongly throughout the design cycle as the design becomes more complex and detailed.

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, fault detection and localization becomes a very time-consuming activity.

The usual model-based system representation in diagnosis (consisting of a set of components, a system description, and observations) can be adapted to VHDL programs without much trouble. The system description is the VHDL program together with part of the semantics of VHDL, and the observations are either based on the traces produced by simulating the specification for a given test case, or on indications of incorrect signal values entered by the user. Components refer to instances of specific VHDL constructs occurring in VHDL programs. For a detailed exposition of basic model-based diagnosis terminology, see [Rei87].

However, given that even an individual simulation run takes from hours to days of real time on a high-end workstation, diagnosing a complete logical representation of the full VHDL program and its semantics is not feasible. A strongly abstracted view of the design must be used for diagnosis. We abstract over values and time points [Ham91], but retain the capability to distinguish between the initialization phase and operating mode of a circuit, a requirement for handling feedback loops. Sufficient discrimination between diagnoses is achieved by applying multiple test cases and measurement selection. Further discrimination can be achieved by requesting the user to evaluate the correctness of particular signals.

The diagnosis system as described in this article was implemented to fit into the existing VHDL design environment and is intended for use in all design phases where VHDL is used.

The paper is structured as follows: Section 2 introduces the example we will use throughout the paper and defines the basic terminology required to apply model-based diagnosis to VHDL. In Section 3, we discuss the abstracted knowledge representation used in our tool and its complexity. This is followed by an overview of the implementation and test results of the diagnosis system (Section 4) and finally a comparison with related work.

2 A VHDL example

To give an example of how to design models and find faults in VHDL, we introduce a simple VHDL program which represents a device we have called D75. The D75 is a sequential device with five input ports a, b, c, d, e, three outputs f, g, h and the following functional specification.

$$f = \begin{cases} state_t + a * b & \text{if } b > 0 \\ state_t & otherwise \end{cases}$$

$$g = \begin{cases} -b * c & \text{if } b \leq 0 \\ 0 & otherwise \end{cases}$$

$$h = state_t$$

where

$$state_t = \begin{cases} (d + e + state_{t-1}) \text{ after 5 ns}^1 & \text{if } t > 0 \\ 0 & otherwise \end{cases}$$

[1]The after keyword indicates that the new value is assigned with a delay of 5 ns.

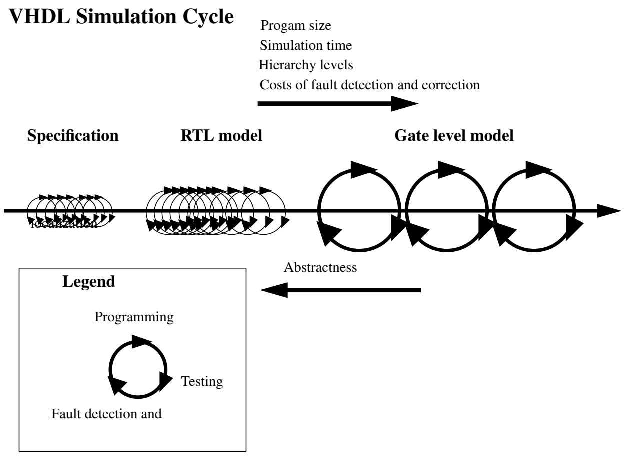

Figure 1: The design cycle using VHDL

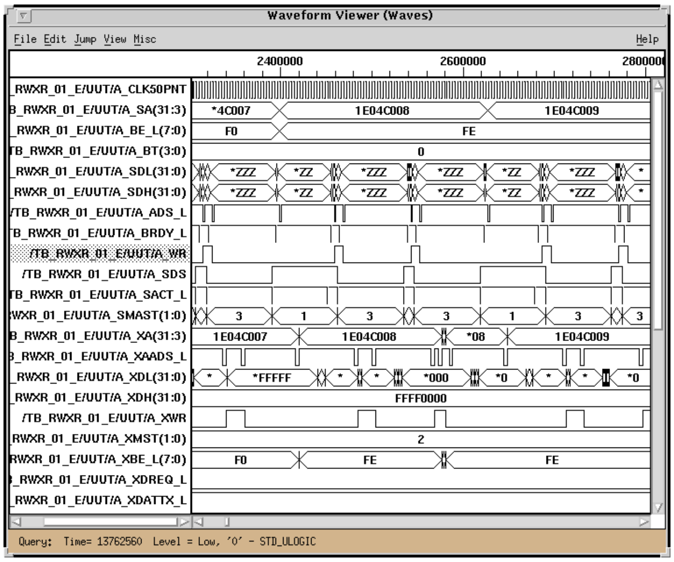

Figure 2: A typical waveform trace

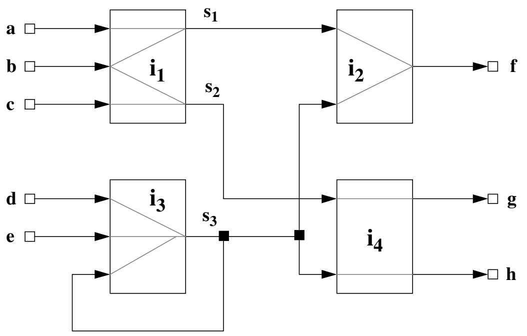

Figure 3: The D75 - Diagnosis components

This formal specification can be easily transformed to a specification written in VHDL, which allows us to simulate the specification and produce a waveform trace.

Figure 3 shows a possible design of the D75 consisting of four components. A designer who uses VHDL must now implement a program for every component to get the same behavior as the specification. The behavior can be tested by checking the waveform traces for every test case specified by the designer with the specification traces. If this check leads to discrepancies we want to find the component that causes the misbehavior. The dotted lines in Figure 3 show the functional dependencies between inputs and outputs of the components.

The following VHDL code fragment describes the internal topology of the D75 as shown in figure 3.

```
entity d75_e is
    port( signal a,b,c,d,e: in integer;
          signal f,g,h: out integer);
end d75_e;
```

```
architecture a2 of d75_e is
    ...
    signal s1,s2,s3: integer := 0;
    ...
begin
    i1: c1 port map (a,b,c,s1,s2);
    i2: c2 port map (s1,s3,f);
    i3: c3 port map (d,e,s3,s3);
    i4: c4 port map (s2,s3,g,h);
end a2;
```

This VHDL top level implementation of the D75 shows the most important VHDL components, *entities* and *architectures*. Entities are interface declarations, whereas architectures are behavior definitions, i.e., they associate a behavior with an entity. An architecture consists of a set of *concurrent statements*, so called because they are executed in parallel (`i1` ... `i4` in the example above). Communication between concurrent statements is achieved by *signals*. Signals are defined in the declaration part of the architecture or entity, or in the interface declaration (port ...) of the entity. VHDL is strongly typed. For further details on VHDL syntax and semantics see, e.g., [VHD88], [Nav93], or [Bha92].

Definition 2.1 *Let A be an architecture and E an entity. $ConcSt(A, E)$ denotes the set of concurrent statements defined in A. $Signals(A, E)$ denotes the set of signals defined in A and E.*

Example:
$ConcSt(a2, cc2\_e) =$

$$\left\{ \begin{array}{l} \text{i1: c1 port map (a,b,c,s1,s2);} \\ \text{i2: c2 port map (s1,s3,f);} \\ \text{i3: c3 port map (d,e,s3,s3);} \\ \text{i4: c4 port map (s2,s3,g,h);} \end{array} \right\}$$

$Signals(a2, cc2\_e) = \{a, \ldots, h, s1, s2, s3\}$.

For simplicity we will write down only the label of the concurrent statement (e.g. $i1 \ldots i4$) instead of the whole statement in our paper. $i1$ thus stands for "$i1$: $c1$ $port$ $map(a, b, c, s1, s2)$" and so on.

Until now we have not defined the behavior of the concurrent statements in our D75 example. To do this we write down signal assignments that are supposed to describe the behavior for every concurrent statement $i1 \ldots i4$. We write `in`i for the i-th input and `out`i for the i-th output of the concurrent statement. We order the signals of the port maps so that input signals come first and outputs last.

- $i1$ port map(in1,in2,in3,out1,out2)

  ```
  out1 <= in1 * in2;
  out2 <= (-1)*(in2 * in3);
  ```

- $i2$ port map(in1,in2,out1)

  ```
  out1 <= in1 + in2;
  ```

- $i3$ port map(in1,in2,in3,out1)

  ```
  out1 <= in1 + in2 + in3 after 5 ns;
  ```

- $i4$ port map(in1,in2,out1,out2)

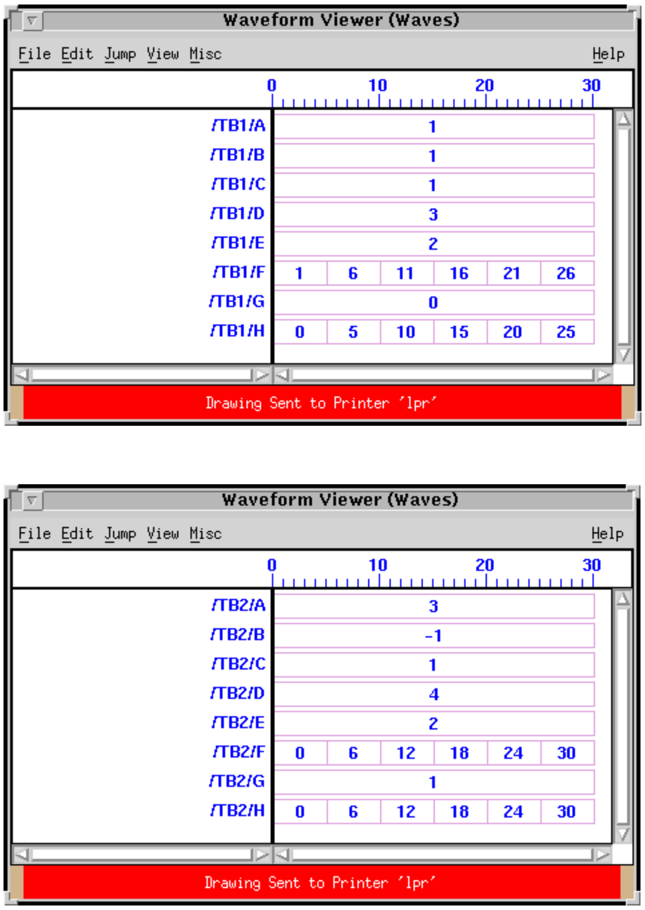

Figure 4: Specification Waveform Trace - Test Case 1 and 2 of the D75

```
out1 <= in1;
out2 <= in2;
```

The VHDL program can now be tested. In our example, we have two test cases which stimulate the D75 with the following initial input vectors.

Signal	a	b	c	d	e
Test Case 1	1	1	1	3	2
Test Case 2	3	-1	1	4	2

The VHDL simulator produces the waveform trace for test case 2 shown in figure 4, which can be checked for correctness with regard to our specification or an older (correct) version of the program. In our D75 example we have a formal specification that can be transformed to VHDL. We can thus show the correctness of a VHDL program *for a given input pattern and a certain time interval* by comparing (automatically) the trace from the specification execution with the trace of the VHDL program. The discrepancies are then used as a starting point for computing diagnoses. Figure 4 shows the specification trace of our D75 example for both test cases.

More formally, a trace is defined as follows:

Definition 2.2 (Waveform Trace) *Let A be an architecture and E an entity. The waveform trace*

$Trace(A, E)$ is a set of tuples (called events*) (s, v, t) representing that a certain signal s (defined in A and E) changes to value v at time t.*
$Trace(A, E) = \{\langle s, v, t\rangle | s \in Signals(A, E) \wedge v \in Type(s) \wedge t \in Time\}$ with

1. *$Type(s)$ ... represents the domain of values for signal s.*
2. *$Time$ ... represents a discrete set of time values ≥ 0.*

The value of each signal at an arbitrary point in time can be observed from the waveform trace:

Definition 2.3 (Value) *Let $Trace(A, E)$ be a waveform trace for A and E. The value of each signal $s \in Signals(A, E)$ at time t, denoted by $Value(s, t)$, is the value that s has taken at the last event before t that changed s, i.e. $Value(s, t) = v_0$ if $\exists\langle s, v_0, t'\rangle \in Trace(A, E) \wedge t' \leq t \wedge \nexists\langle s, v_1, t''\rangle \in Trace(A, E) : t' < t'' \leq t \wedge v_0 \neq v_1$.*

Now we are able to formally define discrepancies.

Definition 2.4 (Discrepancy) *The discrepancies of A and E, written as $\mathcal{D}(A, E)$, are the set of tuples $\langle s, v, t\rangle$ where $s \in Signals(A, E)$, $t \in Time$, $v \in Type(s)$ is the expected value of the signal s, and $Value(s, t) \neq v$. We write $Discrepancy(s) \leftrightarrow \exists\langle s, v, t\rangle \in \mathcal{D}(A, E)$*

A signal discrepancy can be found automatically by comparing waveform traces or as result of an oracle, e.g., input by the designer of the VHDL program. A signal discrepancy and the description of the VHDL program (e.g., its concurrent statements) are the input for the diagnosis engine. Figure 5 shows the program waveform trace for test case 1 and 2, where discrepancies for signals g and f, respectively can be detected confirming the existence of a misbehavior.

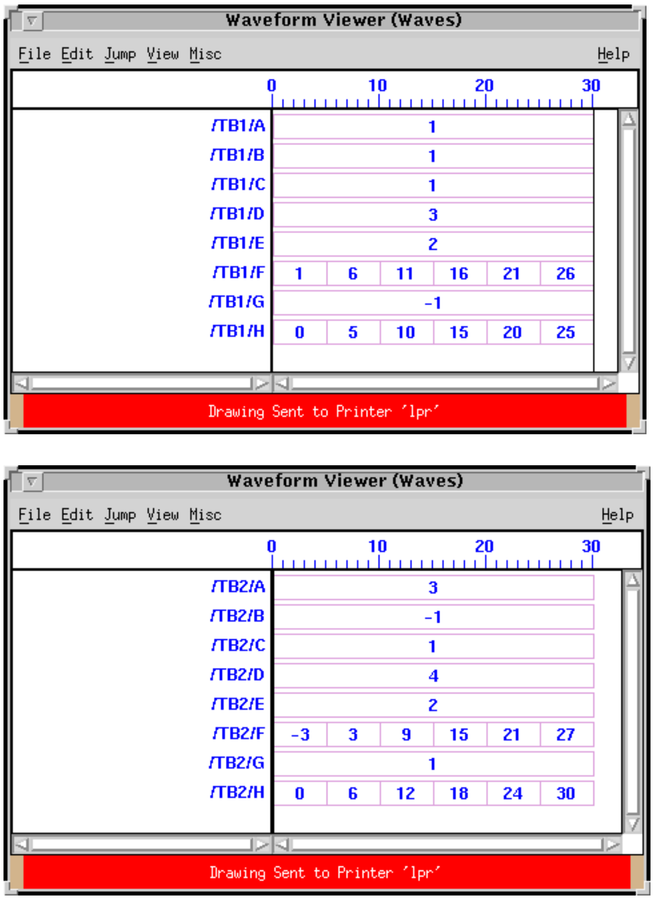

Figure 5: Waveform Trace - Test Case 1 and 2 of the D75

3 Computing Diagnoses

In this section we formalize the diagnosis knowledge used in our VHDL-diagnosis system. The complexity of the VHDL programs to be diagnosed precludes analysis down to the last detail. The goal, rather, must be to compute results that will enable the designer to pinpoint the actual fault and find it faster.

Because of the complexity of the problem, we have to be content with a significantly abstracted representation, governed by these criteria:

- No diagnoses may be excluded due to abstractions.
- Integration with available commercial simulation packages.
- Computational costs must be minimized by requirering very few additional simulation runs.

The principal idea is to abstract as much as possible over time and values on the one hand, while preserving the capability to discriminate between substantial parts of the VHDL-code on the other hand.

3.1 Components

VHDL code is partitioned into hierarchies of concurrent statements which usually bottom out in a sequence of sequential statements. Such a sequence contained in one concurrent statement typically comprises only a small fraction of the complete VHDL-program, and as a first approximation it is not worth the effort needed to pinpoint individual sequential statements as the source of a fault. Therefore, identifying faulty concurrent statements is our main goal.

Definition 3.1 (Components) *The set of components $COMPS$ of an entity E and an associated architecture A consists of all concurrent statements of A; i.e.*
$COMPS(A, E) = ConcSt(A, E)$

Example (cont.): In our D75 example, $COMPS(a2, D75) = \{i1, i2, i3, i4\}$.

Components are connected by signals which are defined in the VHDL code. In principle, VHDL distinguishes between in, out, and inout signals. Each component defines how it uses signals which means that we are able to specify three different sets of ports for a component.

Definition 3.2 (Input, Output, Input/output) *Let C be a component of entity E and architecture A. For each components C, the set of all signals that are used in C and declared as inputs, is called the set of input signals of C denoted by $in(C)$. The sets of output/inout signals, called $out(C)$ and $inout(C)$, are defined in the same manner.*

Example (cont.): $in(i1) = \{a, b, c\}$, $out(i1) = \{s_1, s_2\}$

For here on we represent each inout signal as a combination of one input and one output signal.

3.2 System description

Since the simulation is executed by commercial simulation tools we are not able to compute the dependencies between ports dynamically (except for the trace, the state of the simulation is not visible). Therefore, we analyze the VHDL code statically and compute the functional dependencies. The functional dependencies (encoded in rules) are then used to find the components which are responsible for the discrepancies.

Definition 3.3 (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 $t' \geq t$.*

Example (cont.): s_1 depends functionally on signals a and b.

The main idea of our abstraction is to ignore time and values of the signals. In our system description we only talk about correct or incorrect signals which means that we don't care when the discrepancy occurred and what the expected value would be. In particular, the output of a component is correct if the component is correct and the input signals (on which the output depends) are correct.

On one hand this abstraction has the advantage that we do not need to reason about thousands of time points for several hundreds of signals. On the other hand it has the obvious drawback that we can exclude fewer diagnoses than with a detailed model. However, programmers use this abstraction during debugging very effectively to identify the faulty concurrent statement.

So far we have completely abstracted over time and signal values in order to gain an efficient diagnosis system. However, in order to handle feedback cycles and the initialization phase of the circuit, at least a simple representation of time must be included in the model.

Definition 3.4 (SD part 1: components) *Let A be an architecture and E an entity. The system description modeling the component behavior of the VHDL program given by E and A is a set of sentences for every component $C \in COMPS$ and $out_i \in out(C)$ 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 the initialization
$ok(C) \rightarrow ok(C, out_i)_0$
where out_i depends functionally on the input signals $in_1 \ldots in_n$.

The correct behavior of a single VHDL concurrent statement is described by multiple rules. Our running example results in a separate rule for each signal assignment:
$ok(i1) \wedge ok(a)_T \wedge ok(b)_T \rightarrow ok(i1, s1)_{T+1}$
$ok(i1) \wedge ok(b)_T \wedge ok(c)_T \rightarrow ok(i1, s2)_{T+1}$
$ok(i2) \wedge ok(s1)_T \wedge ok(s3)_T \rightarrow ok(i2, f)_{T+1}$
$ok(i3) \wedge ok(d)_T \wedge ok(e)_T \wedge ok(s3)_T \rightarrow ok(i3, s3)_{T+1}$
$ok(i4) \wedge ok(s2)_T \rightarrow ok(i4, g)_{T+1}$
$ok(i4) \wedge ok(s3)_T \rightarrow ok(i4, h)_{T+1}$

The same signal can be declared as output in different concurrent statements. In this case a simulator uses a so-called resolution function to deduce the final value of a signal at a given time point. This value depends on all the components which use the signal as output.

Definition 3.5 (SD part 2: connections) *Let A be an architecture, E an entity, and $Driver(s) = \{c | c \in COMPS(A, E) \wedge s \in out(c)\}$ where s is a signal.*

The system description modeling the connections of the VHDL program given by E and A is the set of sentences which includes for each signal $s \in Signals(A, E) \wedge s \in out(c) \wedge c \in COMPS$ the sentence
$\bigwedge_{c_j \in Driver(s)} ok(c_j, s)_T \rightarrow ok(s)_T$

Given component $i3$ from our example the rule
$ok(i3, s3)_T \rightarrow ok(s3)_T$
must be element of the system description.

3.3 Observations

After the programming phase, tests, i.e., simulation runs, are conducted, possibly leading to the discovery of a misbehavior of the system. Misbehavior is detected by comparing selected outputs of the program with the outputs of the specification. We will call these selected outputs together with the stimulated inputs *traced signals*.

Definition 3.6 (Traced Signals) *The set of traced signals $TSIG$ consists of all signals that are defined in the entity, associated architecture and libraries and that are included in the waveform trace.*
$TSIG = \{s | s \in Signals(A, E) \wedge \langle s, v, t \rangle \in Trace(A, E)\}$

Example (cont.): $TSIG = \{a, \ldots, h, \}$

The traces of the signals in $TSIG$ are used to deduce observations in the sense of consistency-based diagnosis. If there are no discrepancies in the trace for a signal s, the $ok(s)$ literal is element of the set of observations. If a discrepancy has been found, then $\neg ok(s) \in OBS$.

Signals that are not element of the trace are not observed, but may be included in an additional simulation, depending on their information content and their observability. A signal is observable if either a specification for this signal exists or the user is willing to perform manual inspection.

Definition 3.7 (Observation) *Let $TSIG$ be a set of traced signals. The set of observations OBS is defined as*

$$OBS = \{\neg ok(s) | s \in TSIG \wedge Discrepancy(s)\} \cup \{ok(s) | s \in TSIG \wedge \neg Discrepancy(s)\}$$

Example (cont.):
Test Case 1:
$OBS = \{ok(a), \ldots, ok(f), \neg ok(g), ok(h)\}$
Test Case 2:
$OBS = \{ok(a), \ldots, ok(e), \neg ok(f), ok(g), ok(h)\}$

Next, we need to assign observations to particular points in time since we have associated time points to the $ok/1$ predicate declaring a signal to be correct. Since we ignore the exact time point when a signal becomes incorrect we have chosen an assignment of the observation to a given time point such that no diagnoses are excluded. Therefore every conflict in our abstraction has to be a superset of (or equal to) a conflict in a "hypothetical most accurate model". In other words, we have to make sure that conflicts are not too small.

Lemma 3.1 *Let A be an architecture and E an entity. Because of the structure of our system description we are able to conclude that for $n = |COMPS|$ and each signal $s \in Signals(A, E)$ and $t \geq n$ holds: $ok(s)_n \leftrightarrow ok(s)_t$.*

Therefore, it is sufficient to consider n time points where n is the number of components and to assert $\neg ok(s) \leftrightarrow \neg ok(s)_n$ for each $s \in TSIG \wedge Discrepancy(s)$.

For each $s \in TSIG \wedge \neg Discrepancy(s)$ we have to assert $ok(s) \leftrightarrow \bigwedge_{i=0 \ldots n} ok(s)_i$

Example (cont.):

Let us consider test case 2 and only signals f and g. We have to assert

$ok(f) \leftrightarrow ok(f)_4 \qquad ok(g) \leftrightarrow \bigwedge_{i=0 \ldots 4} ok(g)_i$.

3.4 Diagnosis

For the definition of diagnoses we will apply the consistency-based view as defined by Reiter [Rei87].

Definition 3.8 (Diagnosis) *A diagnosis Δ for a VHDL program $\langle A, E \rangle$ is a subset of $COMPS$ with the following property:*

$$SD \cup OBS \cup \{ok(c) | c \notin \Delta\} \cup \{\neg ok(c) | c \in \Delta\} \not\models \perp$$

Since in our domain in almost all cases of predefined tests are applied to a VHDL program, we incorporate tests in the definition of diagnosis. Note that because of our abstraction we do not need to transfer knowledge across tests as in [RdKSS91].

Definition 3.9 (Diagnosis wrt Tests) *Let $TOBS$ be a set of OBS_i where each OBS_i represents the observations for a particular test.*

A diagnosis Δ for a VHDL program $\langle A, E \rangle$ is a subset of $COMPS$ iff for each $OBS_i \in TOBS$ holds that
$SD \cup OBS_i \cup \{ok(c) | c \notin \Delta\} \cup \{\neg ok(c) | c \in \Delta\} \not\models \perp$

Consider again our example. For test case 1 there exist two minimal single fault diagnoses: $[i1]$ and $[i4]$ (the conflict is $\langle i1, i4 \rangle$). For test case 2 there exist two minimal single fault diagnoses: $[i1]$, $[i2]$ and $[i3]$ (the conflict is $\langle i1, i2, i3 \rangle$).

Therefore, there is one minimal single fault diagnosis $[i1]$. Inspecting the definition of $i1$, we note that the if statement concerning the value of signal b is missing. As a result, we achieved a fault localization with two test cases.

3.5 Consequences of this definition

As noted, we designed our knowledge representation according to the safety principle that no diagnosis may be excluded. On one hand, this may result in too many diagnoses. However, this is not a serious problem if multiple test cases are applied, and in addition the user can be asked to judge the correctness of a signal.

On the other hand, the calculation of diagnoses is simple and fast enough for computing diagnoses even for large programs. This is exactly the case where assistance for fault localization has a very high return on investment.

Note that if we were willing to accept lower performance for the sake of more detailed reasoning about the VHDL program, we still would need to cope with a modeling problem since often physical hardware components (like microprocessors) are used to speed up the simulation. For such components a VHDL model or some other formal description would have to be specified, which would increase the design development costs significantly.

3.6 Complexity issues

Unfortunately the need to introducing discrete time points implies that the size of the system description does not increase linearly with the size of the system.

Theorem 3.2 *Let E be an entity, A be an architecture and $COMP$ be the set of diagnosis components computed from E and A. SIG is the set of signals defined in E and A. Then the following propositions hold:*

1. *The number of rules of the system description SD for $COMP$ and SIG is of order $o(|SIG| \cdot |COMP|^2)$.*
2. *The number of literals of the system description SD for $COMP$ and SIG is of order $o(|SIG|^2 \cdot |COMP|^2)$.*

For programs with hundreds of components, the nonlinear size of the system description implies very long runtimes for the consistency checks required during the search for diagnoses. Since usually multiple consistency checks are required, a straightforward implementation of

	with user interaction	automatic
programming, testing	$\surd$	
regression tests	$\surd$	$\surd$

Figure 6: VHDLDIAG: application domains

this system description results in completely unsatisfactory performance. Therefore, for implementation purposes this representation is transformed into a system description that employs a simplified approach for subsystems which are not involved in the kind of cyclic dependencies that led us to introducing distinct time values in the first case. The modified approach produces system descriptions that grow linearly in terms of system size. For space reasons we do not present the transformation in this paper. For a detailed description, see [SW95].

4 Implementation and Result

VHDLDIAG parses VHDL source files to generate a set of diagnosis components and their interconnections. Diagnoses are computated with standard techniques described in [Rei87; GSW89; dK91]. The diagnosis generator uses the component description and a set of discrepancies to build a hitting set dag and then all minimal diagnoses.

To prune the hitting set dag we allow only the generation of the dag up to a fixed size or depth. Only diagnoses with size up to the dag depth are computed. To restrict the set of diagnoses to a relevant focus, we also allow the insertions of fault probabilities for each diagnosis component computed by a simple heuristic based on its complexity.

Additional measurements for discrimination are selected using the minimum entropy approach introduced in [dKW87]. In our context, making additional measurements means specifying additional signals to be traced in the next simulation cycle.

Our diagnosis approach has been tested on a number of examples, including actual full-size ASIC designs. The results indicate a successful application of VHDLDIAG in hardware design, especially by utilizing regression tests, i.e., comparison of test results for two subsequent versions of a program. Figure 6 shows the domains where VHDLDIAG can be optimally used for finding faults.

The following data points describe one of the ASICS used for testing of the VHDLDIAG system (which has been manufactured and used in real world applications).

Components at top level	379
Number of interface signals at top level	210
Number of signals at top level	412
Maximum number of levels	5
Average number of levels	~ 2
Size of source code	6 MB
Number of gates in produced circuit	$> 100{,}000$

To show the capability of our diagnosis approach, we developed two test situations. In the first case, we used two different version of the ASIC implementation, which are structurally equivalent, and compared them by applying test cases. Discrepancies are detected using our waveform compare tool. From them we are able to find the faults in the program. The following table shows the measured runtimes (on a SPARC 10/30) for locating a fault.

Simulation time	10 minutes
Compare time	40 minutes
Diagnosis time	2 minutes
Diagnosis cycle	52 minutes

Because of the knowledge of all signal values over time, VHDLDIAG computes exactly one minimal diagnosis per level. This diagnosis was always the expected one. A complete regression test lasts about one hour per test case, whereas manual examination of test results lasts several hours for only the top level of an ASIC. VHDLDIAG can diagnose faults at all levels of the ASIC design due to the ability to incorporate all traced signals in the solution (which is far beyond the ability of a human studying a trace).

In the second case, we used only one ASIC implementation and user input about signal correctness to find the faults. Every question VHDLDIAG presents to the user reduces the number of diagnoses but increases the overall diagnosis time, due to the need to recompute diagnoses. The effect of VHDLDIAG in this domain is not as good as it is in the regression test domain, but because of the ability of VHDLDIAG to focus the designer's attention on the relevant signals by using automatic measurement selection, debugging time can again be saved.

All other ASIC examples required at most 3 minutes to compute all minimal diagnoses at the top level. The number of components and signals decreases for lower levels in the ASIC design. At the lowest level we have measured diagnosis times below one second.

Diagnosis results strongly vary depending on the number of signals which could be classified as *ok* or $\neg ok$. If this number increases, the expected diagnosis can be computed with more accuracy.

VHDLDIAG has shown that a simple model used for model-based diagnosis can be good enough for utilization in the VHDL design process. Especially in the domain of regression tests, fault-finding time can be reduced and quality can be increased. Using VHDLDIAG while implementing a new program helps the designer by focusing on relevant signals and components. The diagnosis runtimes achieved by the VHDLDIAG prototype are short enough to allow interactive diagnosis even for very large VHDL programs.

5 Related Work and Conclusion

A number of previous publications dealt with issues related to model-based software diagnosis, influenced by [CFD93]. However, in our case we had to exploit abstractions in order to handle large VHDL programs.

An abstraction over quantitative signal values (distinguishing between correct and faulty values) in the domain of electronic devices is also proposed in [BvSHM89], but there is no analysis of functional dependencies. In contrast, we can generate these dependencies automatically and use them to construct the model without user interaction. The use of functional dependencies reduces the set of inputs influencing a given output, resulting in stronger restrictions on the set of diagnoses.

A different approach to diagnosis and repair of hardware designs is introduced by [CWH94]. Due to restrictions to single faults and gate level designs (nowadays normally automatically synthesized from high-level designs) its applicability is limited. Another approach, with a long research history in the hardware design community is formal design verification [AFK88]. Implementations of these algorithms have been used with success on small circuits with limited functionality, but usually fail due to unacceptable performance when applied to large designs.

A number of authors have presented methods and algorithms for error diagnosis in logic programs [Sha83]. Their approach is based on exploiting proof properties of prolog programs and is therefore not directly applicable to imperative programming systems. Relative advantages of the model-base approach are pointed out in [CFD93].

In this paper, we have presented a computationally feasible approach to diagnosing very large hardware design descriptions. We used the VHDL language as the basis for our implementation, but the results can also be used with other hardware description languages. The model is based on an abstract representation of signal dependencies which are computed automatically by analyzing the VHDL source code. The representation is simple but effective to reason with and allows the user to focus the search for failure locations on a small fraction of the total VHDL code, resulting in significant savings in terms of overall design effort.

References

[AFK88] Magdy S. Abadir, Jack Ferguson, and Thomas E. Kirkland. Logic design verification via test generation. *IEEE Transactions on Computer-Aided Design*, 7(1), 1988.

[Bha92] Jayaram Bhasker. *A VHDL Primer*. Prentice Hall Inc., 1992.

[BvSHM89] R.R. Bakker, D.C. van Soest, P.A. Hogenhuis, and N.J.I. Mars. Fault models in structural diagnosis. In *International Model-Based Diagnosis Workshop*, Paris, July 1989.

[CFD93] Luca Console, Gerhard Friedrich, and Daniele Theseider Dupré. Model-based diagnosis meets error diagnosis in logic programs. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1494–1499, Chambery, August 1993. Morgan Kaufmann Publishers, Inc.

[CWH94] Pi-Yu Chung, Yi-Min Wang, and Ibrahim N Hajj. Logic design error diagnosis and correction. *IEEE Transactions on VLSI Systems*, 2(3), 1994.

[dK91] Johan de Kleer. Focusing on probable diagnoses. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 842–848, Anaheim, July 1991. Morgan Kaufmann Publishers, Inc.

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

[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.

[Ham91] Walter C. Hamscher. Modeling digital circuits for troubleshooting. *Artificial Intelligence*, 51(1-3):223–271, October 1991.

[Nav93] Zainalabedin Navabi. *VHDL Analysis and Modeling of Digital Systems*. McGraw-Hill, 1993.

[RdKSS91] Olivier Raiman, Johan de Kleer, Vijay Saraswat, and Mark Shirley. Characterizing non-intermittent faults. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 849–854, Anaheim, July 1991. Morgan Kaufmann Publishers, Inc.

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

[SW95] Markus Stumptner and Franz Wotawa. Modeling VHDL Programs for Diagnosis with Linear Computational Complexity. Technical Report DBAI-MBD-TR-95-03, Technische Universität Wien, June 1995.

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