A Model-Based Tool for Finding Faults in Hardware Designs*

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
email: {mst, wotawa}@vexpert.dbai.tuwien.ac.at

Abstract

The state of the art in integrated circuit design is the use of special hardware description languages such as VHDL. The designs programmed in VHDL are refined up to the point where the physical realization of the new circuit or board can be created automatically. Before that stage is reached, 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 correction of such faults. This paper deals with the development of VHDLDIAG, a knowledge-based design aid for VHDL programs, with the goal of reducing time spent in fault detection and localization in very large designs (hundreds of thousands of lines of code). Size and variability of these programs makes it infeasible in practice to use techniques based on a detailed representation of program semantics. Instead, the functional topology of the program is derived from the source code. Model-based Diagnosis is then applied to find or at least focus in on the component(s) in the program that caused the behavioral divergence. The support given to the developer is sufficiently detailed to yield substantial reductions in the fault localization costs when compared to the current manpower-intensive approach. A prototype is currently being tested as an integral part of the standard computer-aided design environment. Discrimination between diagnoses can be improved by use of multiple test cases (as well as interactive input by the developer).

1 Introduction

The current state of the art in the design of integrated circuits is based on heavy use of hardware specification languages. Starting from a specification, a new ASIC or circuit board is designed by developing a description in such a language, which can then be executed to simulate the functionality of the circuit. This significantly increases the chance that errors in the design can be found and corrected before the physical circuit is produced, thus reducing the costs of the overall design process (throwing away and redoing the masks and tooling for a circuit design that was found to be defective is an extremely expensive proposition). As the design process continues, the design is continually

*This paper originally appeared in the *Proceedings of the International Conference on Artificial Intelligence in Design AID-96, 1996*

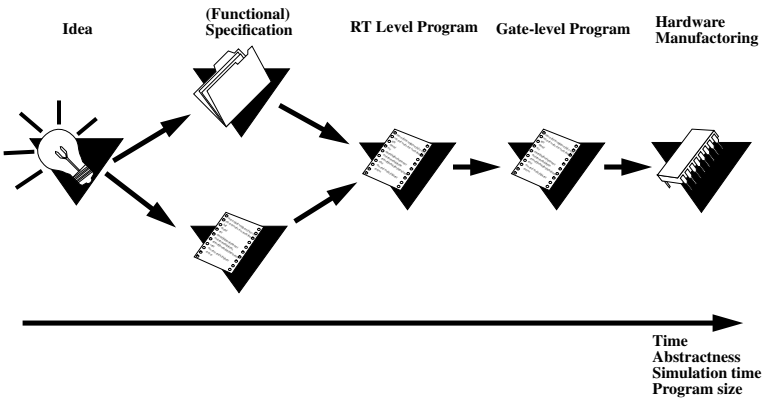

Figure 1: The hardware design cycle

refined, until a level of detail has been achieved where it can be transformed automatically into a representation at the logic gate level. The gate level description is used as the basis for layouting and the production of the physical circuit, a process which also requires little human interference.

As a result, IC design is increasingly getting similar to the software design process, and the search for faults in the programs that describe the designs tends to absorb a significant part of the design effort in the earlier stages of the design process, all the more so since 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 large design teams (or multiple teams at different physical locations), fault detection and localization becomes a very time-consuming activity.

This paper describes the principles behind the VHDLDIAG tool that is used as a design aid in the development of hardware descriptions in VHDL (Very High Speed Integrated Circuit Hardware Description Language), which is probably the most widely used of these languages. We use techniques of model-based diagnosis for creating a simple internal representation of the design, checking test runs for errors, and locating the source of the errors. If unique identification is not possible, the tool helps at least in focusing the attention of the user (i.e., the hardware designer) on those parts of the system where the problem originates, proposes signals whose observation will reduce the set of diagnoses, and can also continue analysis on the basis of observations entered interactively by the user. The system is used in conjunction with existing commercial design support tools (e.g., simulators and graphical design tools) and is intended for use in all design phases where VHDL is used.

The paper is structured as follows: Section 2 takes a closer look at the hardware design cycle using VHDL and the requirements for a knowledge-base debugging tool. Sections 3 and 4 describe the representation and reasoning techniques on the basis of a simple example. Finally, we discuss experiences with implementation and test use of the system and compare our approach with other work related to knowledge-based software design support.

2 The Hardware Design Cycle

Figure 1 gives an overview over the hardware design cycle using VHDL. Typically, the design of an integrated circuit (either an ASIC or a circuit board with multiple ASICs mounted on it) starts with a specification delineating the functional requirements.

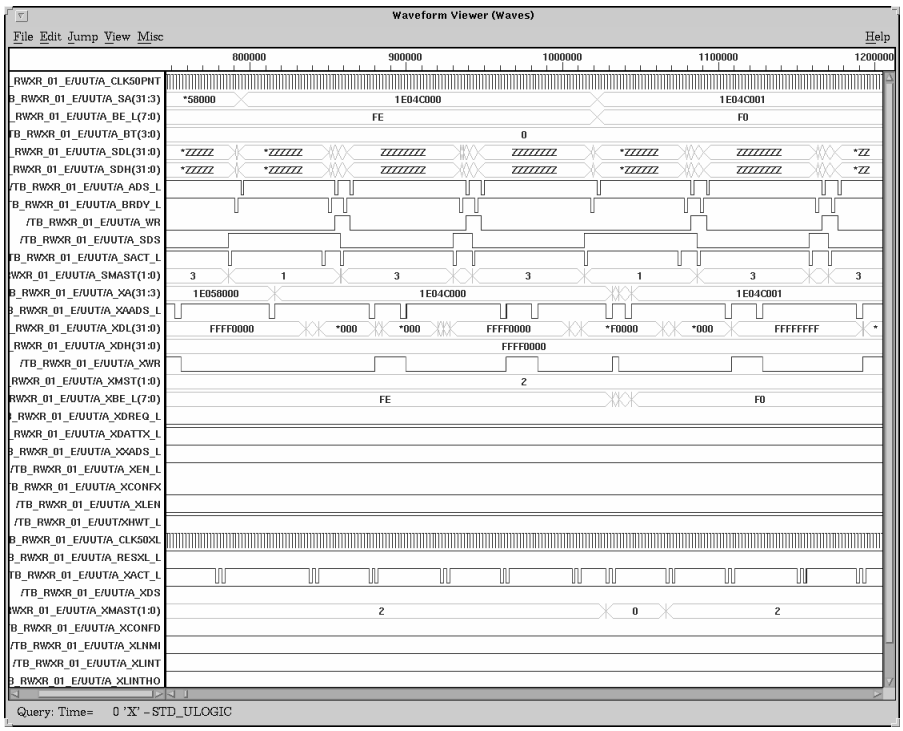

Figure 2: A typical waveform trace

This specification document can be given as either a pure textual document or as a VHDL program. It usually consists of a coarse description of the intended architecture of the design, i.e., a partitioning into components, the functional description of all parts and their interactions, and other design information such as expected fault probabilities, used technology, timing constraints and physical constraints (i.e., working temperature range). Finally, a testing guide will be supplied. If the basic partitioning and the functional descriptions are already written in VHDL, one gains the big advantage of being able to test or verify the specification using the mental model of the expected hardware device. Otherwise, a very simple VHDL design will be created initially to reproduce as closely as possible the specified behavior without paying attention to architectural or structural notions.

From here on, design progresses by the addition of more detailed substructure to the design, i.e., the description of the design in terms of components and their subcomponents. The functional behavior of the design is distributed across the components so their interaction will reproduce the originally specified behavior. Testing of the new versions of the design is achieved through simulating its behavior and having the designer observe the data values occurring on the signal lines between components. The graphs of signals over time are called waveforms (see Figure 2). 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. From here on, we will refer to the former as the *implementation waveform* and the latter as the *specification waveform*. A discrepancy between the two may indicate the existence of a fault in the design.

The next abstraction level reached after the functional specification is the so-called register transfer level (RTL). The RTL design will be much larger than the original specification, increasing simulation and therefore testing time. A typical RT programs could have about 6 MB source code and produce simulation runs lasting from one hour to several days, depending on the simulated amount of real time.

```
...
signal S,A,B: integer := 0 ;
...
S <= A + B ;
...
```

(1) Behavior

```
...
signal S,A,B: bit_vector( 1 to 4 ) := "0000" ;
...
c1: carryadder(A(1),B(1),'0',S(1),QC1);
c2: carryadder(A(2),B(2),QC1,S(2),QC2);
c3: carryadder(A(3),B(3),QC2,S(3),QC3);
c4: carryadder(A(4),B(4),QC3,S(4),Q);
...
```

(2) Gate-level

Figure 3: VHDL behavior vs. gate-level: Adding two numbers

After finishing the RT program implementation, a synthesis tool converts it to a gate-level program with a limited amount of user interaction (which deals with specification of certain parameters such as, say, the width of a bus). Gate-level programs are then simulated as well. The resulting waveform traces are checked for correctness. Because of the automatic conversion, a detected misbehavior of a gate-level program (that performed correctly at RTL level) can only be caused by timing problems and virtually never by a functional fault. Timing faults are mainly corrected using the synthesis tool and trying different parameter values. The gate-level VHDL code is not modified by hand. In contrast, faults in RT programs or specifications are directly corrected. While the assumption usually holds that the more abstract (already tested) program is the correct one, the designer also has to check for the possibility that this is not the case.

Figure 3 shows the difference between the abstraction levels using a small part of a VHDL behavior and a gate-level program.

Next, we take a brief look at the testing considerations for the various design phases.

Functional Specification In addition to preliminary information about structure and behavior, the functional specification als contains information about the intended gate technology, environmental considerations, etc. It can only be verified by comparison to the underlying mental model or older versions of itself, possibly also written in VHDL (in the case of regression tests).

Register Transfer Program Multiple versions of the RTL program will usually be created until the program meets the requirements (i.e., matches the behavior of the specification). Every version must be tested using most of the test cases. Usually up to 20 versions are shipped out for testing. Comparing test results and finding the faults is a major task while programming in register transfer level. A study has shown that up to 60 percent of the effort in this design stage are used for these tasks.

Gate-level Because simulation time and compare time are very high compared to RTL programs, automatic detection of discrepancies is also recommended for the gate-level phase. A simulation run is typically compared with the simulation run of the associated RT program. Subsequent gate-level versions are not directly compared to each other.

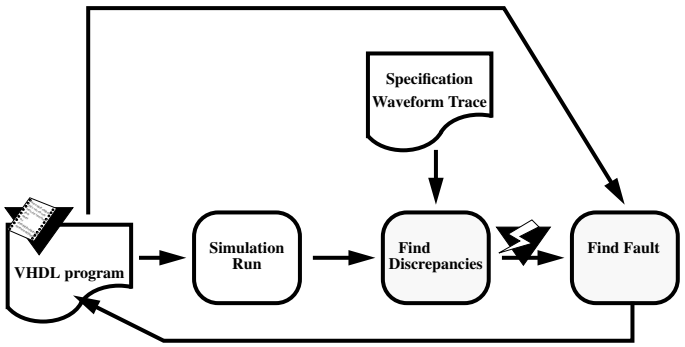

Figure 4: The VHDL design subcycle

3 Knowledge-Based VHDL Design Support

The original goal of the project was to develop a tool that would reduce the overall design effort, without engendering significant changes in the overall structure of the design process (which is codified and fixed by the funding institution). The main interest lay in reducing the amount of time for each individual simulation/fault detection/fault correction cycle, as well as (by improving the quality of the detection and correction stages) possibly reducing the number of cycles. Figure 4 shows the design subcycle. The time involved in a single iteration depends strongly on the complexity and size of the program (i.e., the time increases as the design progresses).

Of the three stages, simulation is already being handled by highly optimized commercial tools. Detection and correction are largely manual tasks open to improvement.

Of major importance is the requirement that any tools generated should not result in the need to alter or embellish the design cycle or the designs themselves. The effort of using the tools should be kept to a minimum, i.e., while entering numerical parameters would be considered adequate, for example, developing a separate representation for every design was out of the question. This is important since designs often involve integration of VHDL code coming from different sources which may or may not be using the same tools (e.g., circuits from subcontractors or from the extensive standard libraries supplied by the simulator companies).

What is required therefore is a generic approach that will allow the mapping of the semantics of a VHDL program to the somewhat abstracted internal representation. Given a discrepancy, the resulting model of the program should then be analyzed to find or at least limit the area of the program where the fault can have originated. Therefore, the model must represent the structure and the functional and causal relationships between the signals in the VHDL program. This can be achieved with an adaptation of the representation and reasoning mechanisms commonly used in model-based diagnosis. However, apart from the model, we also need to derive the observations that describe the actual fault which occurred. Since virtually all the information available about a simulation run is contained in the resulting waveforms, the comparison of specification and implementation waveforms is a crucial prerequisite to checking the correctness of the program and an integrated part of the tool. Therefore we deal with this issue first. Section 4 deals with the diagnosis process.

3.1 Finding faults in the design

Currently, comparison is done manually by the user (hardware developer) with the help of test benches which can be used to format the output in an appropriate manner. However, the test benches themselves require significant development effort and are by their nature both specific to a particular circuit and abstraction level. Ultimately, a major part of the detection effort is spent by the designer in scrolling along an implementation waveform such as in figure 2, trying to spot timing or signal value differences with regard to the matching signal from the other trace.

The VHDLDIAG diagnosis tool eliminates the need to write testbenches by executing an automatic comparison between implementation and specification trace. Internally, a waveform trace is represented as a set of *events*, each of which is a tuple (s, v, t) representing the fact that a certain signal s changed to value v at time t. The value of each signal at an arbitrary point in time is the value produced by the last event up to that time. Instead of checking the waveforms manually, the designer now merely needs to provide

- the specification of which signals are to be simulated.
- the choice of the exact comparison operator.

All the traced signals can be compared in a single comparison run after the simulation is finished. [1] Note that programs have to be syntactically correct to be simulated. Also, the approach obviously cannot deal with errors that do not produce divergent signal values, i.e., a nonterminating loop inside a user-defined function, which will simply cause the simulation to hang.

The need to provide a choice of compare operators arises from the fact that every program manifestation may have a (slightly) modified behavior that is expressed by subtle waveform trace differences without impeding the correct function of the design. These differences can occur, for example, because abstract hardware descriptions do not take delay times into account while gate-level descriptions do, or because a simulation run contains minor errors (i.e., double initialization cycles) that do not influence the overall outcome. Testing that both traces are identical is therefore not always an useful way to detect errors, despite the deterministic nature of the simulation. Another (real world) example would be that of two otherwise identical simulation runs using different clock cycles. In such a case an identity comparison of signal value behaviors would deliver literally thousands of errors. The different compare operators can be used to introduce tolerances into the comparison process.

Identity (I) Comparing for identity means that an error occurs if at some time t a signal s has different values in both traces.

Identity plus Tolerance (IT) The introduction of a tolerance limit allows a small temporal difference δ between two waveforms. An event that occurs at time t in one waveform trace must occur in the interval $[t - \delta, t + \delta]$ within the second trace.

Functional (F) A functional comparison ignores the absolute time values in both traces. Instead, the events in both waveform traces are grouped according to the partial order defined by their time values. All events occuring at the same time are summarized in a so-called state. Two waveforms are said to be equal if their sequences of states are equal.

[1] The number of signals to be traced and compared is currently not restricted by the diagnosis tool but by the simulators which do not have the capacity to trace a large number of signals, as the ability to consider more than about a hundred signals per run (out of, say, 10,000 in a large ASIC) was beyond the capability of any human observer.

Functional plus Tolerance (FT) This mode allows an event a to change its position with regard to an event b in the functional ordering, as long as the relative position of a to b is not displaced by more than δ along the original time axis.

The following is an estimate of the utility of the different compare modes based on use with actual trace files (0 – never used, 1 – always used). Several modes can be used for comparing the same pair of traces. Interactive tuning of the compare parameters is then used to improve the hit rate beyond the values in the table.

Compare Mode	Degree of Utilization	Fault detected but no fault	No fault detected but real fault
I	0.7	0.5	0
IT	0.5	0.4	0.1
F	0.5	0.2	0.1
FT	0.2	0.2	0.2

From a more theoretical viewpoint we can order the compare modes by their relative strictness. In a two dimensional space, each dimension consisting of possible waveform traces, we denote by R_m the number discrepancies reported by each mode $m \in \{I, IT, F, FT\}$. The following relations hold for all possible traces:

$$R_I \subset R_F \subset R_{FT}$$

$$R_I \subset R_{IT} \subset R_{FT}$$

The comparison process also allows for sampling, i.e., the restriction of the comparison to specific, possibly periodic time points or intervals during the simulation.

The need to provide tolerance intervals for values instead of event times did not arise as floating-point signals occur rarely in the designs. It is, however, possible for different data types to be used for a pair of matching signals in the two traces. Gate-level programs, for example, typically use a four-valued logic and bit vectors where behavior and register transfer programs use nine-valued logics and integers, respectively. A simple value comparison would result in many spurious errors. Therefore, the designer can specify equivalence classes for data values to be used during the comparison.

In summary, the compare functionality implemented in the diagnosis tool is generic and not tied to a particular commercial simulation environment. It makes test bench generation unnecessary and the required parameters can usually be chosen without problems by the designer. The result is a set of observations that are delivered to the actual model-based diagnosis engine, of the form $(Signal, \{(Value, Time)|\ldots\})$ for correct signal events and $(Signal, Value, Expected_Value, Time)$ for discrepancies, i.e. deviations from the correct behavior. These observations are the basis for deducing which (probably unobserved) parts of the system may contain the fault.

4 Locating design defects

After finding the symptoms (signals showing a discrepancy), we have to find their source (as mentioned, typically only a few percent of the signals in a system are observed).

4.1 Adapting Model-Based Diagnosis to Design Problems

The model-based approach is based on the notion of providing a representation of the interactions underlying the correct behavior of a device. By describing the structure of a system and the function of its components, it is in principle possible both to reason about the way to achieve desired behavior [SSGP92], i.e., to synthesize a design (although this requires a very detailed model), as well as to ask for the possible reasons why the desired behavior was not achieved. In the diagnosis community, the model-based approach has achieved wide recognition due to the advantages already mentioned: once an adequate model has been developed for a particular domain, it can be used to diagnose different actual systems from that domain. In addition, the model can be used to search for single or multiple faults in the system without alteration.

The usual model-based system representation in diagnosis can be adapted to the design of VHDL programs without much trouble. A system is assumed to consist of a set of components $COMP$, whose correct behavior is described by a logical theory called *system description* (SD). The assumption that a component C behaves correctly is expressed by the fact $ok(C)$. The set of observations OBS contains statements about the actual, observed behavior of the system.

Using the standard consistency-based view as defined by Reiter [Rei87], a diagnosis Δ for a VHDL program is a subset of $COMPS$ such that the assumption of incorrectness for exactly the components in Δ is consistent with the observations:

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

The basis for this is that an incorrect output value (where the incorrectness could be observed directly or derived from observations of other signals) cannot be produced by a correctly functioning component with correct inputs. Therefore, to make the system consistent and avoid a contradiction, the component must be assumed to work incorrectly. In practical terms, one is interested in finding minimal diagnoses, i.e., a minimal set of components whose malfunction explains the misbehavior of the system (otherwise, one could explain every error by simply assume every component to be malfunctioning). We will return to the special features involved in diagnosing designs instead of finished artifacts, after discussing the representation.

Developing a model-based representation for VHDL programs faced two problems. First, the definition of formal semantics for VHDL is an open research topic [KB95], although the definition of the VHDL languages as an IEEE standard means that the existing commercial VHDL environments are reasonably compatible. Second, the size of the programs involved precludes the use of a more intricate representation. Merely executing (i.e., simulating) a VHDL program using a highly optimized commercial simulator takes from hours to days of real time on a high-end workstation. Therefore, 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. Accordingly, our representation abstracts over values and time points [Ham91], but retains the capability to distinguish between the initialization phase and operating mode of a circuit, a requirement for handling feedback loops.

Additional criteria for the choice of representation were:

- No diagnoses may be excluded due to abstractions. In other words, misleading the designer is worse than offering him an answer that does not uniquely identify the component involved.
- Integration with available commercial simulation packages.
- Computational costs must be minimized by requiring very few additional simulation runs.

Figure 5: The D75 - Diagnosis components

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. Stronger discrimination between diagnoses can be achieved by applying multiple test cases and measurement selection (i.e., specifying signals that offer good chances of discriminating between diagnoses when included in a trace). Further discrimination can be achieved by requesting the user to evaluate the correctness of particular signals.

4.2 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 and three outputs f, g, and h.

A possible initial design of the D75, consisting of four components, is shown in Figure 5. 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. If this check leads to discrepancies we want to find the component that causes the misbehavior. The dotted lines in Figure 5 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 5. (A full VHDL program would include additional declarations; they are not required for understanding the example and hence omitted.)

```
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. Each entity has a type associated with it (this means that multiple entities of the same type can exist). For brevity, we have omitted the type definitions `c1, c2, c3`, and `c4`. For details on VHDL syntax and semantics see, e.g., [VHD88].

The behavior of the concurrent statements in the D75 is described by signal assignments for every concurrent statement. 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)
    out1 <= in1;
    out2 <= in2;
```

A concurrent statement represents a parallel process associated with a particular component in the system. Due to the hierarchical structure of entities (each of which has a set of concurrent statements associated with it), VHDL code is partitioned into hierarchies of concurrent statements which usually bottom out in a sequence of sequential statements, since the computation of a new signal value (which in our case only uses standard arithmetic operators and a simple delay) can be arbitrarily complex, using library functions and user-defined functions that use the full expressiveness of the imperative language parts of VHDL (these bear a strong resemblance to the language Ada). The current version of the VHDLDIAG system does not analyze the semantics of sequential statements, its granularity is at the level of concurrent statements.

4.3 System Description

Since the simulation is executed by commercial simulation tools, the internal state of the system as it is simulated is not visible, apart from the signal values that are contained in the trace. Therefore, it is not possible to determine by observation in which manner exactly a particular (e.g., incorrect) data value was computed. VHDLDIAG analyzes the VHDL code statically and computes the *functional dependencies* between signals (i.e., which signals are possibly involved in the computation of which other signals) regardless of whether the actual branch of the program that contains the signal assignment in question is actually executed at runtime. The system model based on functional dependencies is then used to search for the components which are responsible for the discrepancies. For example, in the D75 circuit, 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 do not care when the discrepancy occurred and what the expected value would be. A simple ordinal time axis is retained for the handling of feedback loops. The length of the time axis is equal to the number of components (since we are ignoring actual time values, n time steps are sufficient to propagate any observation through all components in the system).

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.

4.3.1 Components

The system description modeling the component behavior of a VHDL program is a set of sentences (one for each value of T) on the time axis) for every concurrent statement $C \in COMPS$ 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 $in_1 \ldots in_n$. In effect these rules state that the output of a component is correct if the component is correct and the input signals on which the output depends are correct. For example, the following describes the concurrent statement $i4$ in the D75:

$ok(i4) \wedge ok(s2)_T \rightarrow ok(i4, g)_{T+1}$
$ok(i4) \wedge ok(s3)_T \rightarrow ok(i4, h)_{T+1}$
where $T \in \{1, \ldots, 4\}$

4.3.2 Connections

Let $Driver(s) = \{c | c \in COMPS \wedge s \in out(c)\}$ 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

$\bigwedge_{c_j \in Driver(s)} ok(c_j, s)_T \rightarrow ok(s)_T$

i.e., the output of a signal is correct if all signal values being placed on it are correct. For example, given component $i3$ from our example the rule

$ok(i3, s3)_T \rightarrow ok(s3)_T$

must be element of the system description.

For a more detailed discussion of the representational issues, see [FSW95].

4.4 Computing Diagnoses

Given the system description, Diagnoses are computed by applying the standard hitting set DAG method as described in [Rei87, GSW89, dK91]. In principle, the method is based on computing the so-called *conflict set*. In MBD terminology, a conflict is a disjunction of abnormality assumptions for individual components that is implied by $SD \cup OBS$. In other words, at least one of the components in C must be abnormal for SD to be consistent with OBS. Computing a minimal hitting set for the set of minimal conflicts yields a diagnosis. The system description basically has the effect of propagating correctness assumption claims along signals as long as components are assumed correct. If an inconsistency is derived because the correctness is propagated to a signal that was observed to be incorrect, the components involved in the propagation are part of the conflict. E.g., in Figure 5, if signals $s1$ and $s3$ are known to be ok, but $\neg ok(f) \in OBS$ (f is known to be not ok), and $ok(i_2)$ is assumed, then $ok(f)$ will be derived, yielding a contradiction that results in the conflict $\{i_2\}$ (i.e., i_2 must be part of any diagnosis given these particular observations). Various heuristics are used to restrict the size of the hitting set DAG and improve performance. For more detail, see [FSW95].

The diagnosis tool (called VHDLDIAG) has been implemented in Smalltalk (Visualworks) and is currently used in actual production work by selected designers. Below are typical performance figures, produced with an actual ASIC:

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	$> 100{,}000$
Simulation and Compare time	30 minutes
Actual Diagnosis time	2 minutes

4.4.1 The Assumption of Model Correctness

In a certain respect the problem of diagnosing a design is unique in the realm of model-based reasoning. 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 design, however, the assumption that the specification will be a complete representation of the structure of the artifact is obviously invalid. In our case, the internal structure and the way in which the behavior is described will differ widely between a functional specification and its RTL implementation – the implementation will usually contain many internal components and signals which have no counterpart at all in the functional specification. The only part of the specification that is directly usable is the waveform trace generated by the specification. We are therefore forced to base our model of the VHDL implementation on analysis of the code of the implementation itself. That implies, however, that it is the model that reflects the incorrectness of the design and whose output (the implementation trace) is confronted with observations that are correct (the specification trace), 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 design defect may manifest itself in the model leads us to the related issue of so-called structural faults.

4.4.2 The Assumption of Structural Correctness

Structural faults are faults that do not occur because a component is functioning incorrectly, but because there is 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 [Dav84] and mainly excluded from consideration in subsequent work on diagnosis. However, it is very relevant when diagnosis is applied to designs, and in particular, software designs. 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. This is an open research issue.

5 Discussion

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]. In addition they usually dealt with relatively small problems so that employing a representation that expressed the complete semantics of the problem domain was feasible. In our case, an adequate subset of the language had to be identified so that the representation would be abstract enough to deal with large programs, but would work even if the full expressiveness of VHDL were used in the programs. The result was a representation that focuses on the functional structure of the program. Even if full discrimination (i.e., finding a unique minimal diagnosis) is not achieved in a particular case, the tool still aids the designer in understanding the functional dependencies in the circuit. Information flow is described and important areas are identified so the designer can start debugging with a more detailed focus. Other salient points of the problem domain include:

- the existence of hidden structural differences between specification and implementation – signals may not match or have no counterpart, components may have no counterpart, the internals of components may have been changed.
- functional differences between specification and implementation – e.g., program control structures may be replaced by interacting subcomponents
- parts of the system may be unrepresentable – for example, the simulator may be coupled with actual existing hardware that is too complex to be integrated into the design (e.g., a commercial microprocessor to be mounted on a circuit board that is being simulated).
- No changes to the design
- Requirements for external axiomatizations are inacceptable, as their preparation is time-consuming and requires additional training for designers who already have to cope with the complexity of VHDL. Finally, in case of designs coming from different teams, (sometimes subcontractors in different companies using different development methodologies), or in case of designs that incorporate complex existing hardware (i.e., the abovementioned board designs which include a microprocessor and therefore require interfacing to an actual, physical processor for simulation) external axiomatizations are often impossible to obtain.

The last point bears some elaboration. First, it effectively disqualifies the verification-based systems which are available on the market and often applied successfully in similar circumstances. These systems do require a separate axiomatization of the properties to be verified (for each circuit), and even the best available are very limited in the size ($\sim$ 1 K lines of actual VHDL code) as well as complexity of the systems they can be applied to (e.g. [Bar83, Bur94]). In addition, while they can be used to verify individual properties of a program, they do not try to pinpoint the location of the error. Thus they complement the simulation/debugging approach rather than avoiding it.

In design, model-based techniques have been used to guide the redesign of a component or system [GC89, SSGP92]. Like the verification approach, this concept requires the existence of an independent functional specification which describes (qualitatively or quantitatively) the variables and causal relations in the system. In the VHDL design domain, such a specification unfortunately does not exist. Instead, waveform traces provide specifications about particular aspects of the system behavior (test cases) on the basis of limited observations of the system. The only description about the composition and detailed workings of the design is the possibly incorrect program itself. A description of the functionality of larger program components is therefore not available. What is available as the basis of a system model are the semantics of individual VHDL statements.

With our current abstract representation, the guidance for redesign apart from pointing out the violating statement is limited. Consider the statement `S<= f(E1,...,En)`. If it is assumed to behave incorrectly, this might be due to a number of reasons which would lead to different redesign approaches, such as changing one of the expressions `Ei`, changing the function `f`, changing the target `S` or eliminating the whole statement. Providing capabilities for distinguishing between these possibilities requires detailed reasoning about sequential statement semantics and variable/signal values. The local integration of such mechanisms in a manner that does not significantly degrade overall performance (and again limit practical usability) is an open topic. One possible approach would be the use of fault modes, which are commonly used in hardware diagnosis to discern between different types of faults and prune the search space. Software diagnosis must in principle deal with an unlimited number of fault models including structural faults. The possibility of expressing structural faults through fault modes has been recently examined in [Böt94].

Finally, much work has 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 so that assumptions about the intentions of the programmer can be gleaned from comparison with the description. Similarly, in [AC91], it is necessary to provide an axiomatization of the intended functionality. The **Talus** system [Mur88] attempts to match student solutions to example programs to find the the source of divergent outputs. While structural variations between the example and student programs are allowed, the system still requires a direct match between the subresults produced inside a function and also uses the full semantics of the language used (a functional subset of LISP) to reason about the program, an approach that is not acceptable as a general method in our case, where the computational effort would be too high. 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 or at least experienced users orient themselves in very large programs of different structure and functional architecture.

6 Conclusion

In this paper, we have described the VHDLDIAG tool which provides design support by using model-based reasoning for determining the source of errors in hardware designs that are written in the VHDL specification language. One of the basic requirements was that the tool should fit into the standard design process used. The tool parses the standard VHDL source code written by the designers, and derives observations about execution correctness by automatically comparing the waveform traces produced by specification and more detailed implementation versions of the VHDL design.

The system uses a model of the functional structure of the design to identify components that are responsible for incorrect behavior. If a test case does not allow complete discrimination of the components involved, multiple test cases, automatically generated proposals for measurement selection, and finally interactive input from the designer can be used for restricting search further.

The tool has been implemented in Visualworks/Smalltalk and has been successfully used for finding faults in full-scale, actual ASIC designs. It is currently being tested in its future production environment. Results so far indicate savings of up to 10 % of the whole design cycle. Possible future improvements include a more complete representation of VHDL semantics. In particular, we will investigate the representational issues of the sequential parts of the language along the lines of design for imperative languages as described in [AC91, Liv94]. While computationally more expensive, this representation could be used (strictly locally) to increase the discriminatory power if the standard representation produces too many diagnosis candidates. In the vein of the tutoring environments discussed in the previous section, we also intend to utilize this for providing limited repair capability for designs.

Acknowledgement

This work was supported by Siemens Austria under project grant DDV GR 21/96106/4).

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.

[Bar83] Henry Barrow. Proving the correctness of digital hardware designs. In *Proceedings AAAI*, pages 17–21, August 1983.

[Böt94] Claudia Böttcher. No faults in structure? How to diagnose hidden interaction. In *Proc. IJCAI*, Montreal, August 1994.

[Bur94] J. R. Burch. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Circuits and Systems*, 13(4):401–423, 1994.

[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.

[Dav84] Randall Davis. Diagnostic reasoning based on structure and behavior. *Artificial Intelligence*, 24:347–410, 1984.

[dK91] Johan de Kleer. Focusing on probable diagnoses. In *Proceedings AAAI*, pages 842–848, Anaheim, July 1991. Morgan Kaufmann Publishers, Inc.

[FSW95] Gerhard Friedrich, Markus Stumptner, and Franz Wotawa. Model-based diagnosis of hardware designs. In *Proceedings on the Sixth International Workshop on Principles of Diagnosis*, Goslar, October 1995.

[GC89] Ashok Goel and B. Chandrasekaran. Functional representation of designs and redesign problem solving. In *Proc. IJCAI*, pages 1388–1394, Detroit, August 1989. Morgan Kaufmann Publishers, Inc.

[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.

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

[KB95] Carlos Delgado Kloos and Peter T. Breuer, editors. *Formal Semantics for VHDL* . Kluwer Academic Publishers, 1995.

[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.

[SSGP92] E. Stroulia, M. Shankar, A. Goel, and L. Penberthy. A model-based approach to blame-assignment in design. In *Artificial Intelligence in Design*, pages 519–537, Pittsburgh, 1992.

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