Detecting and Locating Faults in Hardware Designs*

Markus Stumptner and Franz Wotawa
Technische Universität Wien,
Institut für Informationssysteme and
Ludwig Wittgenstein Laboratory for Information Systems,
Paniglgasse 16, A-1040 Wien,
Email: {mst,wotawa}@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. Designs are programmed in VHDL and 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. The task of circuit design therefore becomes primarily one of software development. 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 debugging costs when compared to the current manpower-intensive approach. A prototype is currently being tested as an integral part of the standard computer-aided VHDL development environment. Discrimination between diagnoses can be improved by use of multiple test cases (as well as interactive input by the developer).

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 procedure 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 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, the earlier stages of IC design nowadays effectively constitute a specialized software development process, and the debugging, i.e., search for faults in the programs that describe the designs tends to absorb a significant part of the design effort in these stages, all the more so since the code for 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 developed during the DDV (Design Diagnosis for VHDL) project. VHDLDIAG is used as a debugging aid in the development of hardware descriptions in VHDL (Very High Speed Integrated Circuit Hardware Description Language), which is probably the most widely used such language. We use techniques of model-based diagnosis for creating a simple internal representation of the program, 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 programmer (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. Also included in this paper are two extensions regarding the used internal model. The first one uses the abstract representation for locating the faulty process statement within a program and an exact representation for locating the fault within the process. The second extension converts the whole VHDL programs into one single flat representation. While the first extension is currently imple-

*This work was partially supported by the Austrian Science Fund project N Z29-INF and Siemens Austria research grant DDV GR 21/96106/4.

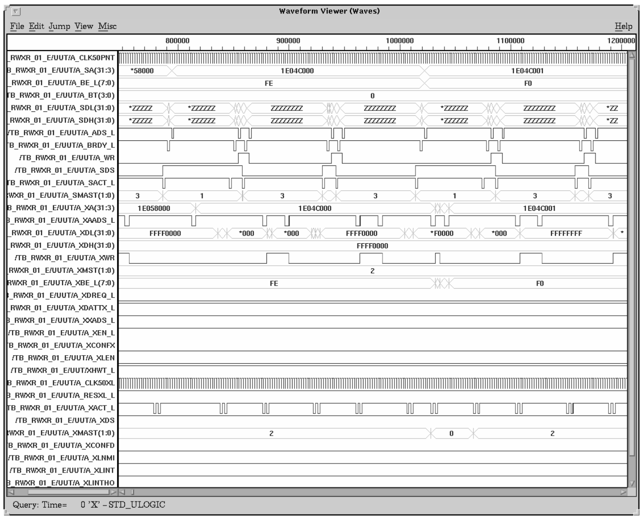

Figure 1: A typical waveform trace

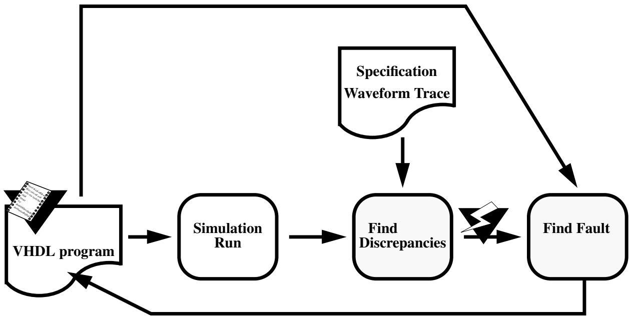

Figure 2: The VHDL design subcycle

mented the second is not.

Knowledge-Based VHDL Design Support

The original goal of the DDV project was to develop a tool that would reduce the overall development effort without engendering significant changes in the overall structure of the design process (which is codified and fixed by the funding company). 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 such cycles. Figure 2 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 development progresses). During the design subcycle the hardware designer writes a VHDL program and simulates it, receiving a so-called waveform trace as a result which lists value changes for selected signals over time (see Figure 1), usually comprising several 10.000 signal changes. The designer then compares the resulting waveform trace with a verbal specification or another trace, referred to as the *specification waveform trace*. In case of discrepancies, the error has to be located and corrected and the cycle starts again.

A major requirement of the project was that any tools should not result in the need to alter or embellish the design cycle or the designs/programs 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 (such as a separate formal specification) for every design was out of the question. This is important since large systems often involve integration of VHDL code coming from different sources which may or may not be using the same tools (e.g., code 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, and with the diagnosis process afterwards.

Detecting Errors

Currently error detection is achieved manually by the programmer (hardware developer) with the help of test benches which can be used to format the output in an appropriate manner. Ultimately, a major part of the detection effort is spent by the designer in scrolling along an implementation waveform trace such as in figure 1, trying to spot timing or signal value differences with regard to the matching signal from the specification trace (or a verbal specification). VHDLDIAG eliminates the need for comparing waveform traces manually by executing an automatic comparison between implementation and specification trace. Instead of checking the waveforms manually, the designer now merely needs to provide: (1) the specification of which signals are to be simulated, and (2) the choice of the exact comparison operator. VHDLDIAG provides several different comparision operators, e.g., compare at sample time points, compare on identity, and compare event sequences ignoring time.

All the traced signals can be compared in a single comparison run after the simulation is finished. 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, e.g., a nonterminating loop inside a user-defined function, which will simply cause the simulation to hang.

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.

After finding the symptoms (signals showing a discrepancy), we use model-based diagnosis to find their source (as mentioned, typically only a few percent of the signals in a system are observed). We now examine this process in more detail.

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 (Stroulia *et al.* 1992), 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 (Reiter 1987), 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).

Given the system description, diagnoses are computed by applying the standard hitting set DAG method as described in (Reiter 1987; Greiner, Smith, & Wilkerson 1989; de Kleer 1991). 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.

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 (Kloos & Breuer 1995), 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. However, it is feasible for small parts of the VHDL program or for programs where some syntactical restrictions apply.

Therefore, we use two models in our VHDLDIAG tool: A strongly abstracted view of the design for debugging the whole program and an exact model for locating faults within small program fragments, i.e., process statements. Accordingly, the first representation abstracts over values and time points (Hamscher 1991), but retains the capability to distinguish between the initialization phase and operating mode of a circuit, a requirement for handling feedback loops. The second representation maps expressions and statements directly to diagnosis components while retaining the semantics.

In the following we describe both representations using the small VHDL program from figure 3 implementing a counter. The specification of the counter program COUNTER(BEHAV) is given as follows: If both inputs E1, E2 are set two '0' the circuit counts up. If E1 is set to '1' and E2 to '0' the count is decreased. If E1 is set to '0' and E2 to '1' the counter value is set to '0', and in the last case where both inputs are set to '1' the counter is '1'. The output is coded using a 1-from-4 decoder. A counter value '0' is represented by the output vector A1='1', A2='0', A3='0', A4='0', and 3 is represented by A1='0', A2='0', A3='0', A4='1'.

From the correct COUNTER(BEHAV) program we derive a buggy variant by introducing a fault in line 24 which is changed to:

```
1.  entity COUNTER is
2.    port(
3.      E1,E2,CLK: in BIT;
4.      A1,A2,A3,A4: out BIT);
5.  end COUNTER;
6.  architecture BEHAV of COUNTER is
7.    signal D1,D2,Q1,Q2,NQ1,NQ2: BIT;
8.  begin
9.  -- Input combinational block
10. comb_in: process (Q1,Q2,E1,E2)
11.   variable I1,I2: BIT;
12. begin
13.   I1 := not((Q1 and Q2) or (not(Q1) and not(Q2)));
14.   I2 := (I1 and E1) or (not(I1) and not(E1));
15.   D1 <= (E1 and E2) or (E2 nor I2);
16.   D2 <= (E1 and E2) or (E2 nor Q2);
17. end process comb_in;
18. -- Output combinational block
19. comb_out: process(Q1,Q2,NQ1,NQ2)
20. begin
21.   A1 <= NQ2 and NQ1;
22.   A2 <= Q2 and NQ1;
23.   A3 <= NQ2 and Q1;
24.   A4 <= Q1 and Q2;
25. end process comb_out;
26. NQ1 <= not(Q1);
27. NQ2 <= not(Q2);
28. -- Memory block
29. dff: process(CLK)
30. begin
31.   if (CLK = '1') then
32.     Q1 <= D1;
33.     Q2 <= D2;
34.   end if;
35. end process dff;
36. end BEHAV;
```

Figure 3: The VHDL program COUNTER(BEHAV)

```
24*.  A4 <= Q1 or Q2;
```

To distinguish between the two program variants we refer to the faulty one as COUNTER(FAULTY). During the rest of this paper we always refer to the incorrect program when talking about the counter example.

The Abstract System Description

The abstract system description is used to debug very large VHDL designs. Diagnosis happens on the level of concurrent statements (processes), i.e., the concurrent statements are mapped to diagnosis components. Bugs within concurrent statements, cannot be directly distinguished. 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.

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.

Instead of presenting the model in a formal way, we give an overview of the principles. For more details about the model and various improvements see (Friedrich, Stumptner, & Wotawa 1996; 1999). As stated above, the abstract model uses concurrent statements as diagnosis components. In our example the set of components $COMP$ is $\{comb\_in, comb\_out, dff, assign1, assign2\}$ where $assign1$ ($assign2$) denotes the assignment *NQ1 <=* ***not****(Q1)* (*NQ2 <=* ***not****(Q2)*) and the other the processes.

The abstract behavior of the components is given by their associated functional dependencies. We say that a signal S depends on another signal X if a value change on X may cause a change of S's value. The dependencies for component $assign1$ are $\{(NQ1, \{Q1\})\}$. In this case the signal $Q1$ is be seen as input and $NQ1$ as output for the component. If we assume that the component is working as expected and that all inputs have correct values then we can conclude that the output values have to be correct. For a component C an its functional dependency $(O, \{I1, \ldots, I_n\})$ we can specify a rule expressing this basic principle:

$$ok(C) \wedge ok(I_1) \wedge \ldots \wedge ok(I_n) \rightarrow ok(O)$$

To model other aspects of VHDL, namely driving signals and the occurence of cycles, the system description used in VHDLDIAG is slightly more complex. We refer to (Friedrich, Stumptner, & Wotawa 1996) for more details.

Using the abstract system description for our running example COUNTER(FAULTY) we can compute the diagnoses. We know that the inputs E1, E2, CLK and the outputs except A4 are correct. The incorrect value arises because if Q1 is '1' and Q2 is '0' the wrong value '1' is assigned to A4. Diagnoses computed are $\{comb\_in\}$, $\{comb\_out\}$ and $\{dff\}$. A further discrimination can be done by using VHDLDIAG's measurement selection algorithm returning that the classification of the signals Q1 and Q2 as beeing correct has to be done. In our case Q1 and Q2 are correct and only the diagnosis $\{comb\_out\}$ remains.

The advantage of the abstract models is that the user (hardware designer) is focused only on the relevant parts of the program, i.e., those parts possible causing a detected misbehavior. In our example the statements $assign1$ and $assign2$ are excluded from the list of candidates after the first diagnosis run. After the measurement selection step, the user is ask to classify two signals. All others are not of interest. The diagnosis process can now be continued inside the processes using a more detailed model.

The Exact System Description

In addition to the abstract model we have added a model for the sequential statements used in VHDL process statements. This model represents each statement as diagnosis component. The behavior of the diagnosis component formalized in first order logic corresponds exactly to the semantics of the associated statement. This model has been introduced in (Stumptner & Wotawa 1998). Some ideas used there were taken from (Stumptner & Wotawa 1999).

Given that what we want to diagnose are process statements and the expressions contained in them, the set of components in the program is the set of these process statements and expressions.

- A *variable assignment* has the form *V* **:=** *Expr*. The value of the variable *V* after execution is equal to the value of the expression *Expr*.
- A *signal assignment* of the form *S* <= *Expr* **after** *TExpr* attempts to change the history of signal *S* using the value of expression *Expr* at the time *TExpr* according to the VHDL semantics, i.e., the transaction $(Expr, Texpr)$ is placed on the driver of S.
- An *if-statement* is given by **if** *Cond* **then** *St1* **else** *St2* **end if**, where the **else** part is optional. If the Expression *Cond* evaluates to *true*, then *St1* is executed. Otherwise, the sequence of statements *St2* is executed.

We now describe the mapping from statements to components. A variable assignment is converted into a component having three inputs and one output. The inputs denote the input environment (*env_in*), the variable identifier (*var*), and the expression (*expr*). The output (*env_out*) returns the new, altered environment. Through the output *env_out* the new environment is delivered. The mapping for signal assignments is similar, except we have history ports *history-out* and *history-in* instead of the environment ports *env_out* and *env_in*, to explicitly mark the fact that they carry different information. We assume that the signal values from before the process is called are stored in the initial process environment, which is propagated through the statements during execution. In addition, there is a port representing the temporal expression (*texpr*). Conditional statements are converted into a component with inputs *cond*, *env_in*, *history-in*, *st1*, *st2*, *history-1*, *history-2* and two outputs *env_out* and *history-out*. The *cond* port is connected to the condition expression, whereas *st1* (resp. *st2*) is connected to the output environment of the corresponding statement part *St1* (*St2*). The history inputs *history-1* and *history-2* are connected to the history outputs of the corresponding converted statement parts.

Just as with statements, the expressions contained in these statements can be mapped to a component oriented view. We do not discuss this here in detail. See (Stumptner & Wotawa 1999) for details of such a representation and its use in diagnosing expressions. For example, the process $comb\_out$ is mapped to a diagnosis system depicted in figure 4. The behavior of the *OR* component and the assignments are given as follows:
$orcomp(C) \rightarrow or(out(C), in_1(C), in_2(C))$

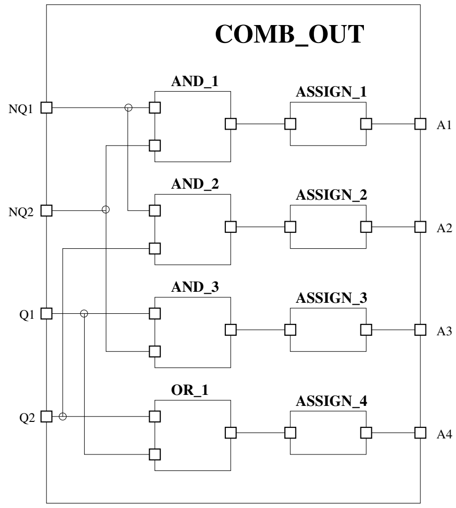

Figure 4: The converted *comb_out* process of the COUNTER(FAULTY) program

where the predicate or is defined as expected, i.e. $or('1','1',X), or('1',X,'1'), or('0','0','0')$.
$assign(C) \Rightarrow (ok(C) \rightarrow equal(in(C), out(C)))$
with $equal(X, X)$.
Assuming the values Q1='1', Q2='0' and A4='0' which are taken from the specification of the counter example we get one diagnosis $\{ASSIGN\_4\}$ which is the expected one.

A System Description for Synthesizeable Programs

In this section, instead of using a hierarchical diagnosis approach, we convert the program to a flat structure, and the resulting statements and expressions to (diagnosis) components. This is similar to the synthesis process by which a (VHDL) program is directly converted into hardware that implements the same functionality (VHDL IEEE Std 1076.6/D1.12 1998). Here, however, integers and other data types are not mapped into boolean values in the usual fashion of synthesis tools. Instead, special diagnosis components are used.

The mapping from VHDL programs to a logical representation is done as follows: In the first step the program D is converted into a component connection model M, where components represent program fragments, e.g., expressions and statements, and connections represent signals and variables. This step is mainly concerned with programming language syntactical issues. However, conversion also has to consider some semantical issues related to VHDL. This include the handling of driving signals together with resolution functions, and semantical differences between variables and signals. The next step involves the removal of cycles in M to reduce the computational complexity of the diagnosis. Finally, we convert the resulting component connection model M into a logical representation by using the component behavior and the information about connectivity. The component behavior is given by logical sentences derived from the VHDL semantics of the related program fragment.

In (Wotawa 1999b; 1999a) a formal description of the conversion process, the removal of cycles, and a discussion about the usefulness of the model together with filtering rules for improving the diagnosis results are given. In this paper we only give an overview of the model and its application to software debugging using the example program COUNTER(FAULTY). The graphical representation of the program after eliminating cycles is depicted in figure 5. The connections labeled by *E1*, *E2*, *Q1(0)*, *Q2(0)*, *CLK* are inputs, and *Q1(1)*, *Q2(1)*, *A1*, *A2*, *A3*, *A4* are outputs. *Q1(0)* (*Q2(0)*) and *Q1(1)* (*Q2(1)*) represent the signal *Q1* (*Q2*) before and after program execution. Note, that the computed value for *Q1(1)* at time t works as input *Q1(0)* for the execution of the program at the immediately next point in time $t + 1$.

For example using the inputs *E1=0*, *E2=1*, *CLK=1* leads to the computation of *Q1(1)=0*, *Q2(1)=0*, *A1=1*, *A2=A3=A4=0*. Changing *E2* to 0 and using the computed values for *Q1*, *Q2* as inputs we can derive new output values. The computed results correspond to the results obtained by executing the VHDL program using a test-bench with the following behavior. First, set the counter to zero. Afterwards increment the counter. The clock value is initially set to 1 and changes at every predefined time point. Table 1 shows the computed values. The row marked with (*) contradicts the expected (specified) behavior.

After detecting an inconsistency we are interested in locating the misbehavior. We do this by using a similar model as used for VHDL sequential statements. The following list gives the behavior of the single components described in first order logic (FOL).

Assignments For assignments $assign(X)$ only the correct behavior can be specified, where the value of the expression must be the same as the value of the variable or signal after executing the statements. Formally, we write
$\neg ab(X) \rightarrow out(X) = in(X)$
where $in(X)$ denotes the input port (connected with the components representing the expression) and $out(X)$ the output port (associated with a variable or signal).

Conditionals A conditional X, written $cond(X)$, has two determined behaviors. In the case the conditional behaves correctly, i.e., $\neg ab(X)$ the THEN-branch is executed whenever the condition evaluates to true. Otherwise, the execution of the ELSE-branch is performed. If we assume the condition is the source of a bug, i.e., $wrong(X)$, then these two are exchanged. In the model M, a component representing a conditional statement has several inputs and outputs, one for every signal or variable Y used in a signal assignment $'Y <= \ldots '$ occurring either in the THEN- or the ELSE-branch. Those inputs and outputs are indexed by integers. For example, the ports $then_1$, $else_1$, out_1 are all associated to the same variable or signal. Formally, the behavior of the conditional is given by:
$\neg ab(X) \wedge cond(X) = true \rightarrow out_i(X) = then_i(X)$
$\neg ab(X) \wedge cond(X) = false \rightarrow out_i(X) = else_i(X)$
$wrong(X) \wedge cond(X) = false \rightarrow out_i(X) = then_i(X)$
$wrong(X) \wedge cond(X) = true \rightarrow out_i(X) = else_i(X)$
where $cond(X)$ denotes the port connected with the condition and i the index.

Functions For functions $Func(X)$ where $Func \in \{and, or, not, equal, \ldots\}$ used in expressions and conditions we only are able to specify the correct behavior. Components associated with unary functions like **not** have two ports (*in*, *out*), while binary functions have three (in_1, in_2, *out*). Formally, the correct behavior is given by the rules:
$\neg ab(X) \rightarrow out(X) = fnot(in(X))$
$\neg ab(X) \rightarrow out(X) = fand(in_1(X), in_2(X))$
$\neg ab(X) \rightarrow out(X) = for(in_1(X), in_2(X))$
$\neg ab(X) \rightarrow out(X) = fequal(in_1(X), in_2(X))$
where $fnot$, $fand$, for, and $fequal$ are defined as expected.

In addition to the behavior of a diagnosis system, i.e., in our case the representation of a program, we need observations for computing diagnoses. The observations are $\{E1 = 0, E2 = 0, CLK = 1, Q1(0) = 0, Q2(0) = 0, A1 = 0, A2 = 1, A3 = 0, A4 = 0\}$. They are derived from previous computations (Q1,Q2) and from the given specification. Using model-based diagnosis we receive only two single diagnoses $\{OR\_1\}$, $\{ASSIGN\_4\}$ and several multiple diagnoses all containing at least one of AND_2 or $ASSIGN\_2$ as result. Since we usually assume that single diagnoses are more likely than diagnoses containing several components, a debugging tool would first present the two single diagnoses to the user (ignoring the rest). In our case both single diagnoses are correctly pointing to the bug within the VHDL program. This result is better than the one obtained by using the functional dependency model. The abstract models allows to derive 3 processes as potential diagnosis candidates while using the model for synthesizeable VHDL programs delivers only one process as source of the bug.

Comparing the abstract model together with the model of the sequential statements SD_{ABSTR} with this model SD_{SYNTH} we obtain the following results. While a hierarchical diagnosis approach is utilized by the abstract model this is not the case for SD_{SYNTH} representing a flat structural model. SD_{SYNTH} can only be used for debugging a superset of register transfer level (RTL) programs but not full VHDL. RTL programs are synthesizeable where syntactical restrictions apply. On the other side the abstract model can be used for debugging almost all VHDL programs except those using file access and pointers (which are very rare because they do not implement hardware). Other advantages of the abstract models are that discrepancies can be derived quickly and that the number of possible diagnoses candidates is reduced. Hence, SD_{ABSTR} can be used for debugging very large VHDL designs. Since SD_{SYNTH} uses an exact model, diagnosis takes more time. However, it can be argued that SD_{SYNTH} can be at least be used for medium size designs (see (Wotawa 1999b)). Compared to SD_{ABSTR}, the diagnosis capability is improved. Using SD_{SYNTH} minimizes the set of diagnosis candidates whenever possible. However, both models should be seen as complementary. The abstract model should be used for large de-

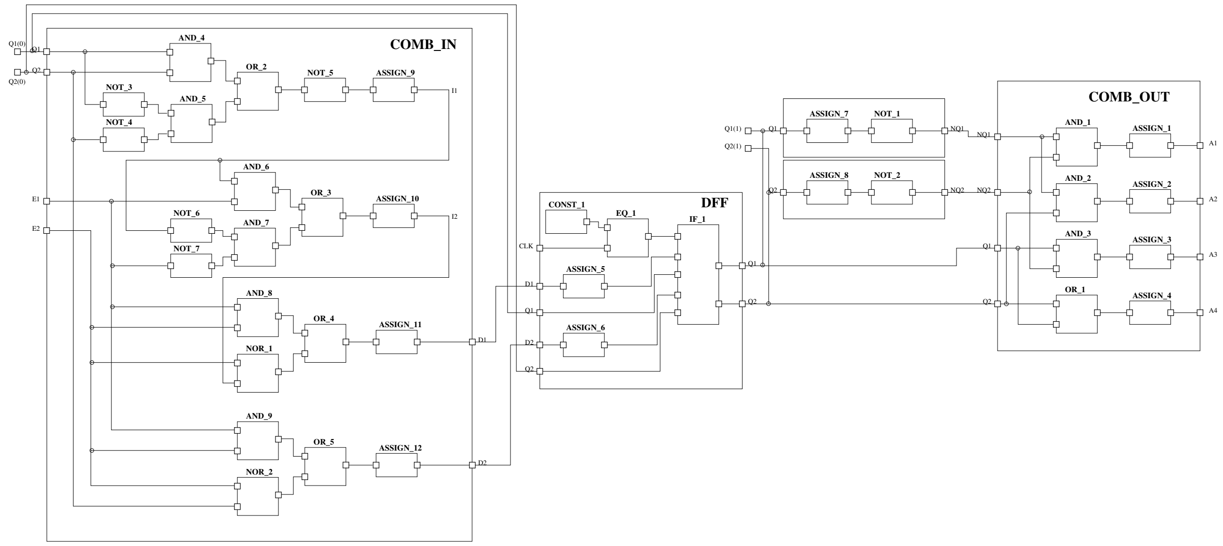

Figure 5: The converted COUNTER(FAULTY) program

E1	E2	CLK	Q1(0)	Q2(0)	Q1(1)	Q2(1)	A1	A2	A3	A4	
0	1	1	X	X	0	0	1	0	0	0	
0	1	0	0	0	0	0	1	0	0	0	
0	0	1	0	0	0	1	0	1	0	1	(*)
...											

Table 1: The behavior of COUNTER(FAULTY)

signs to compute a focus of attention for SD_{SYNTH}, and in the cases where the other model cannot be applied.

Discussion

Diagnosing hardware designs results in a number of differences to the usual model-based paradigm.

The Assumption of Model Correctness In a certain respect the problem of diagnosing software 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 VHDL and other programs, 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 of the VHDL program 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.

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 (Davis 1984). Such faults, mainly excluded from consideration in subsequent work on diagnosis, are very relevant when diagnosis is applied to software. 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.

Related Work Formal verification techniques are a powerful technique in VHDL design. One of the reasons for the development of the VHDL debugging tool is though that the requirement of a separate formal specification can only be achieved for small parts of systems (and with restricted semantics) in practice, whereas VHDLDIAG works from the VHDL source code only. The Aspect system (Jackson 1995) uses functional dependencies between program variables for checking a less restrictive form of program correctness, but still requires explicit program annotations. The work presented in (Burnell & Horvitz 1995), combines path analysis with probabilistic reasoning on a large assembler application (using Bayesian nets developed in interviews with experts on that application). Program Slicing (Weiser 1984)is a well-known technique and active research area for analyzing dependencies in programs similar to our abstract dependency model, but examines mainly individual variable influences.

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. In addition we have shown a model applicable for synthesizeable VHDL programs providing better discrimination between diagnosis candidates.

The tool has been successfully used for finding faults in full-scale, real world ASIC designs: up to 6MB of source code. Diagnosis times are in the region of below 10 seconds per run, but with several runs typically required to isolate a fault. The system 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 (Allemang & Chandrasekaran 1991). 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.

References

Allemang, D., and Chandrasekaran, B. 1991. Functional representation and program debugging. In *Proc. IEEE Knowledge-Based Software Engineering Conference*.

Burnell, L., and Horvitz, E. 1995. Structure and Chance: Melding Logic and Probability for Software Debugging. *Communications of the ACM* 31 – 41.

Davis, R. 1984. Diagnostic reasoning based on structure and behavior. *Artificial Intelligence* 24:347–410.

de Kleer, J. 1991. Focusing on probable diagnoses. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, 842–848.

Friedrich, G.; Stumptner, M.; and Wotawa, F. 1996. Model-based diagnosis of hardware designs. In *Proceedings of the European Conference on Artificial Intelligence*.

Friedrich, G.; Stumptner, M.; and Wotawa, F. 1999. Model-based diagnosis of hardware designs. *Artificial Intelligence*. To appear.

Greiner, R.; Smith, B. A.; and Wilkerson, R. W. 1989. A correction to the algorithm in Reiter's theory of diagnosis. *Artificial Intelligence* 41(1):79–88.

Hamscher, W. C. 1991. Modeling digital circuits for troubleshooting. *Artificial Intelligence* 51(1-3):223–271.

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

Kloos, C. D., and Breuer, P. T., eds. 1995. *Formal Semantics for VHDL* . Kluwer Academic Publishers.

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

Stroulia, E.; Shankar, M.; Goel, A.; and Penberthy, L. 1992. A model-based approach to blame-assignment in design. In *Proceedings Artificial Intelligence in Design*.

Stumptner, M., and Wotawa, F. 1998. VHDLDIAG+:Value-level Diagnosis of VHDL Programs. In *Proceedings of the Ninth International Workshop on Principles of Diagnosis*.

Stumptner, M., and Wotawa, F. 1999. Debugging Functional Programs. In *Proceedings* 16^{th} *International Joint Conf. on Artificial Intelligence*. To appear.

1998. IEEE P1076.6/D1.12 Draft Standard For VHDL Register Transfer Level Synthesis.

Weiser, M. 1984. Program slicing. *IEEE Transactions on Software Engineering* 10(4):352–357.

Wotawa, F. 1999a. Debugging synthesizeable VHDL Programs. In *Proceedings of the Tenth International Workshop on Principles of Diagnosis*.

Wotawa, F. 1999b. New Directions in Debugging Hardware Designs. In *Proceedings of the International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems*.