DBAI

TECHNICAL REPORT

INSTITUT FÜR INFORMATIONSSYSTEME

ABTEILUNG DATENBANKEN UND EXPERTENSYSTEME

New Directions in Debugging Hardware Designs

DBAI-TR-99-24

Franz Wotawa

DBAI TECHNICAL REPORT

FEBRUARY 1999

Institut für Informationssysteme
Abteilung Datenbanken und
Expertensysteme
Technische Universität Wien
Paniglgasse 16
A-1040 Wien, Austria
Tel: +43-1-58801-18403
Fax: +43-1-58801-18492
sek@dbai.tuwien.ac.at
www.dbai.tuwien.ac.at

TU

TECHNISCHE UNIVERSITÄT WIEN

DBAI TECHNICAL REPORT

DBAI TECHNICAL REPORT DBAI-TR-99-24, FEBRUARY 1999

New Directions in Debugging Hardware Designs

Franz Wotawa[1]

Abstract. This paper introduces a new approach in the debugging of hardware designs. The design is given as a VHDL program and converted in a component connection model. The conversion is similar to the synthesis of register transfer into gate level programs. The resulting model is directly used for locating faults within the design. To do this, we propose the application of model-based diagnosis. The advantage of this approach is its degree of automation and that it can be applied even on today's mid-size to large size programs.

[1]Technische Universität Wien, Institut für Informationssysteme, Paniglgasse 16, A-1040 Wien, Austria, Tel. +43 (1) 58801 18432, E-Mail: wotawa@dbai.tuwien.ac.at

Acknowledgements: The work presented in this paper was partially funded by Siemens Austria under research grant DDV GR 21/96106/4.

A short version of this paper appears in the Proceedings of the Twelfth International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems IEA/AIE-99.

Copyright © 1999 by the authors

1 Introduction

Since hardware designs continue to become larger and more complex the verification task becomes a crucial factor. In order to reduce time to market and the overall design costs, formal verification techniques and extensive simulations are currently used during the design cycle. However, currently there is almost no help for the designer in fixing the faults thus found. Only traditional debuggers allowing to go step by step through the code are available, making fault localization and correction a time consuming and difficult task.

Because a large amount of time is spent in fault localization and correction, automation of this task helps to improve the design process. Work has recently been done in this direction include [CWH94, FSW96, SW98]. In this paper we describe a new approach for diagnosing hardware designs. Similar to previous work we convert VHDL programs (see [Nav93, VHD88] for an introduction into VHDL) into a declarative model describing the programs behavior. Such a model consists out of component models, associated with statements or expressions, and connections, associated with signals and variables used in the program. By applying standard diagnosis techniques (see [Dav84, Rei87, dKW87]) we can locate faulty components in the model and map them back to the associated statements or expressions within the program. This debugging approach is not restricted to VHDL.

To be general applicable the model has to be automatically derived from the program. The main part of the conversion process is to map parts of the programs into components. Which parts are onsidered will influence the granularity of the diagnosis results. If statements are directly mapped to components, bugs can only be localized on the statement level. Faults inside a statement can not be diagnosed with such a model. This must not be a drawback since diagnosis time depends on the number of diagnosis components. On the other hand, if a converted program consists only out of one component, diagnosis does not provide any discrimination. Therefore the granularity should be carefully chosen.

In [FSW96] VHDL concurrent statements were mapped to diagnosis components. Since the number of concurrent statements varies from 1 to about 1,000 with an average value of about 20, this diagnosis granularity is good enough to provide substantial informations in most cases. Additionally, an extented model (see [SW98]) allows to apply hierarchical diagnosis on the contents of a concurrent process statement. The hierarchical diagnosis process is done as follows. First, the bug is located at the level of concurrent statements. If one VHDL process statement can be identified as source of an error, a model using the sequential statements in a process as components, allows to locate the bug within the process.

The approach used in this paper differs from the previous in the following point. Instead of using a hierarchical diagnosis approach, we convert the program into a flat structure. Statements and expressions are converted to (diagnosis) components. This is similar to the synthesis process (see [VHD98]) where (VHDL) programs are directly converted into hardware, implementing the same functionality. However, integers and other data types are not mapped into boolean values in the manner usual for synthesis tools. One advantage of this approach is that the resulting system does not need to take account of the VHDL operational semantics speeding up computation. Of course it is guaranteed that the diagnosis system implements the same functionality as the program.

In summary the proposed approach has the following benefits:

- A model used for diagnosis is automatically derived from the given program.
- Diagnosis is done using standard model-based diagnosis techniques.
- Because of the syntactical restrictions the approach can be applied even to large size designs.

```
entity COUNTER is
  port (
    CLK, RESET : in STD_LOGIC;
    O1, O2 : out STD_LOGIC );
end COUNTER;
architecture BEHAV of COUNTER is
  signal D1,D2 : STD_LOGIC ;
begin
  mem: process ( CLK, RESET )
  begin
    if RESET = '1' then
      O1 <= '0';
      O2 <= '0';
    else if CLK = '1' and CLK'EVENT then
        O1 <= D1;
        O2 <= D2;
    end if ; end if ;
  end process mem;
  comb_in : process ( O1, O2 )
    variable V: STD_LOGIC;
  begin
    V := not (O1);
    D1 <= V;
    D2 <= not ((O1 and O2) or (V and not (O2)));
  end process comb_in;
end BEHAV;
```

Figure 1: The VHDL COUNTER(BEHAV) program

- The approach is applicable to VHDL register transfer programs. Most of todays designs are written using the VHDL register transfer subset.

This paper is organized as follows. In section 2 the approach is introduced using a small example. It is followed by a description of the conversion procedure. Section 4 shows how the model can be used for debugging. A section about related research and future directions concludes the paper.

2 A basic example

We illustrate our approach using a small VHDL program implementing a 2 bit counter. The counter is reseted by a positive value on the RESET input and counts up every time a positive edge of the clock input CLK is detected. Figure 1 shows the VHDL program consisting of an entity COUNTER and describes the inputs (CLK,RESET) and the outputs (O1,O2) of the device. The behavior of the counter is given by the architecture BEHAV. Two processes (MEM and COMB_IN) implement the desired functionality. The behavior of the counter is shown in figure 3 (a).

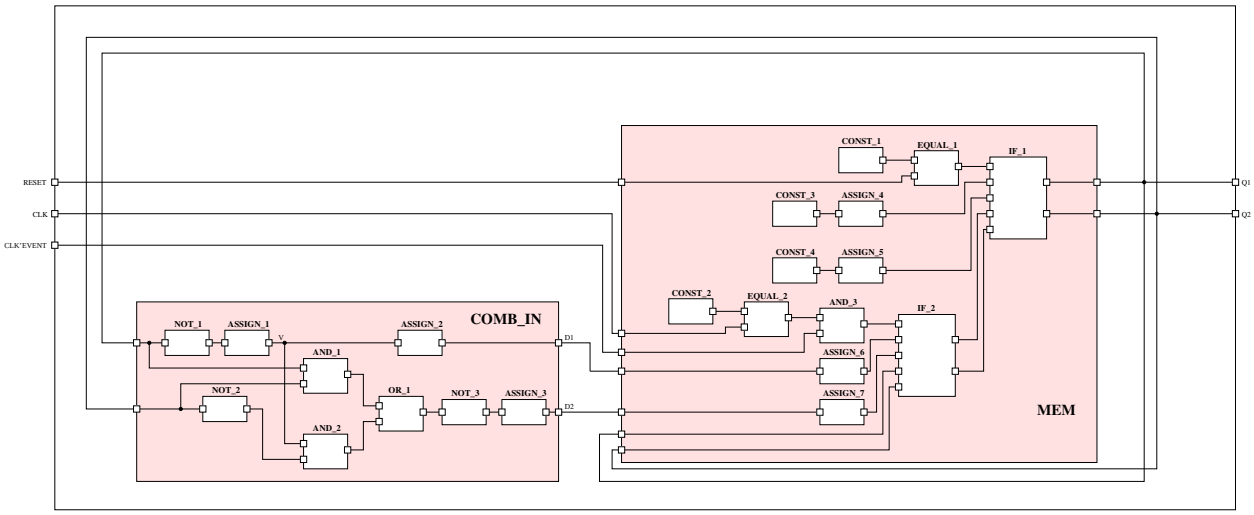


Figure 2: The converted COUNTER(BEHAV) program

The program behavior (given by the VHDL semantics) is described as follows: Processes are only executed after detecting an event on a input signal at the current simulation time. In our case an event occurs if at least one signal of the process sensitivity list changes the value. The sensitivity list of the MEM process has two signals: CLK and RESET. So if RESET is set to '1' the process is executed. The signals O1 and O2 are set to '0'. Since both signals occur in the sensitivity list of COMB_IN, it is executed and D1 is set to '1' and D2 to '0'. If RESET is set to '0' the counter starts counting up every time a positive edge on CLK is detected. In the first step O1 is set to '1' and O2 to '0'. Then the process COMB_IN is executed leading to D1 = '0' and D2 = '1'.

The same functionality as described above can be implemented using hardware. [VHD98] describes a VHDL subset together with a conversion algorithm for mapping programs into a gate level representation. The gate level view of the COUNTER(BEHAV) program corresponds directly to hardware. An advantage of the gate level implementation is that it can be simulated much faster than ordinary RTL programs, since VHDL specific signal update cycle can be completely ignored for gate level simulation.

In our approach we use the basic idea of RTL synthesis. However, we do not restrict ourself to such a small class of RTL programs as synthesis does. See [VHD98] for a description of how VHDL process must be organized to be mapped to flip flops, latches and other digital gates. Instead the only restriction we currently apply is that wait statements are not allowed within processes[1] and that delay times are ignored. Expressions, conditionals, assignments, loops, etc. are converted to simple components while signals and variables are converted to connections. Figure 2 shows the initial diagnosis representation of the COUNTER(BEHAV) program.

3 Modeling programs

In this section we show the conversion of programs into diagnosis systems. For presentation purposes we restrict VHDL to a subset. However, the used techniques can be easily applied to other VHDL constructs

[1] Usually the wait statement can be replaced by a corresponding process sensitivity list.

namely component instantiation statements, loops, and others. The overall conversion process can be summarized as follows:

1. Convert the program into a component connection model. This representation may contain cycles.

2. Make the model acyclic. In the hardware context this step creates a combinational circuit from the sequential circuit. Other approaches uses similar techniques [CWH94].

3. Use the resulting model together with observations for diagnosis. In the debugging domain, observations come from an oracle, i.e., a formal specification or the programmer herself.

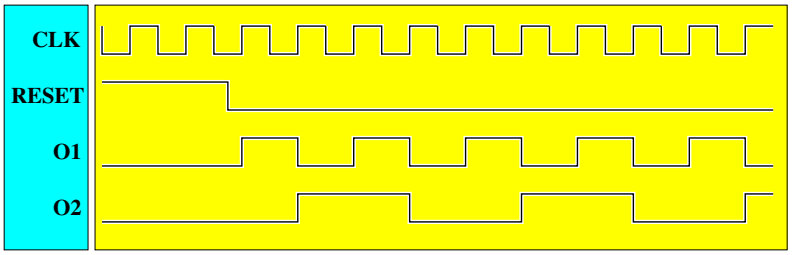
Before introducing an algorithm for converting programs we define our component connection model. A component is an object with ports and a specified behavior. Connections are associated to ports. To have access to the model we introduce a function p for every port of the given component returning the associated connection, i.e., $p : COMPS \mapsto CONNS$. For example, given a *NOT* component C with two ports in and out we assume the existence of the functions $in(C)$ and $out(C)$. We further introduce a function $ports : COMPS \mapsto 2^{PORTS}$ returning a set of ports for a given component. For component C we have $ports(C) = \{in, out\}$. The behavior of a component can be obtained using the $behavior$ function. For simplicity we introduce a function $connPorts$ returning all ports connected with the given signal. The component connection model for a program is given by a set of components. The connections can be easily computed from the underlying information.

The component connection model forms a system description that can be used in model-based diagnosis (MBD) to determine faulty components. In our approach we convert programs into system descriptions, i.e., models, compute diagnoses, and map the diagnoses back to the associated parts of the programs. To allow the last step we assume a function $part$ associating the components to the parts of the program, i.e., statements and expressions, and a function $signal$ mapping a connection to the underlying signal (or driving signal explained later in this section).

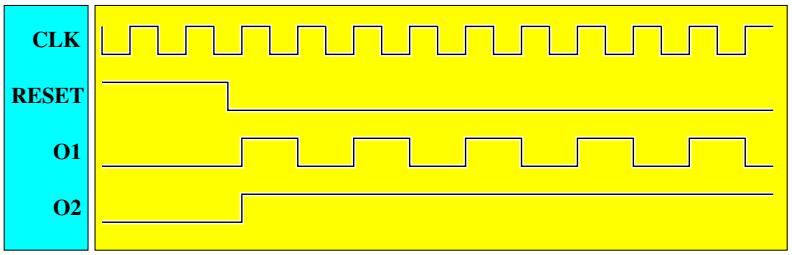
VHDL programs are given by an entity and an architecture[2]. While entities specify the interface, i.e., those signals accessible from the outside, architectures implement the (maybe faulty) behavior. Therefore only architectures must be converted into diagnosable systems. For simplicity we assume that an architecture consists out of processes. That is no semantic restriction because every concurrent statement relevant for simulation that is not a single process can be rewritten in this manner.

Let $SIGNALS$ be the set of signals declared in an entity and architecture and $STATEMENTS$ be the set of processes. We assume a function $conn : (SIGNALS \times STATEMENTS) \cup VARIABLES \mapsto CONNS$ mapping driving signals and variables to connections, and a function $signalConn : SIGNALS \mapsto CONNS$ associating signals to connection. Note, that a driver of a signal is generated whenever a process assigns a value to a signal. The driver itself is a signal that determines the signal value according to the VHDL semantics. The mapping of signals to connections is never changed during conversion. They are initially given by associating a new connection to every signal driver occurring in the VHDL program (for $conn$) and by associating a connection to every signal(for $signalConn$). For every variable we also initialize $conn$ with a new connection. However, the associated connection may change during conversion. This distinction is necessary because of the semantically different handling of signals and variables in a process.

[2]Configurations are ignored for our purposes because it is always possible to automatically generate a program with an equal behavior without using configurations.



(a) Behavior of COUNTER(BEHAV)



(b) Behavior of COUNTER(FAULTY)

Figure 3: The behavior of the two COUNTER programs

As stated the signal value is determined by its driving signals. Therefore we must also define a way for representing this behavior in the semantics of VHDL. In VHDL a resolution function maps the driving values to a signal value for those data types where a resolution function is defined[3]. Since only IEEE standard logic (see [VHD88]) is used in practice, we only consider the resolution function for IEEE standard logic for debugging. Whenever a signal S has multiple driving signals $D_1, \ldots, D_m$ ($m > 1$) we introduce a component implementing the behavior of the resolution function. The inputs $in_1, \ldots, in_m$ are connected with $D_1, \ldots, D_m$ and the output out is connected with signal S. We assume that such a driving component will never fail. If a signal has only one driving signal the driving component is removed from the component connection model and the driving signal is replaced by the signal itself.

The following algorithm **convert** defines the mapping from VHDL programs into diagnosis systems. Note, that this conversion is only correct for the synthesis subset of VHDL.

- **Architectures** are converted by successively converting their concurrent statements. As stated above we assume that the functions $conn$ and $signalConn$ are initialized before calling the conversion algorithm. The algorithm returns a set of components defining the diagnosis system. Let " **architecture** A **of** E **is** SignalDef **begin** $St_1 \ldots St_n$ **end** " be the architecture.

  convert $((St_1, \ldots, St_n), conn, signalConn)$
  $comp = \emptyset$
  for $i = 1$ **to** n **do**
  convert $(St_i, comp, conn, signalConn, P)$

[3]For most of the standard data types, e.g., bit, boolean, integer, and char, no resolution function is defined. A signal of a type with no resolution function has exactly one driving signal.

end for
return $comp$

- **Processes** are converted by converting the statement part. Let C = " **process** $P\ (S_1, \ldots, S_n)$ **is** $VarDef$ **begin** $St_1, \ldots, St_n$ **end process** " be a process.

  convert $(C, comp, conn)$
  conn' = **copy** conn
  convert $((St_1, \ldots, St_n), comp, conn, signalConn, C)$
  for $S \in SIGNALS$ **do**
  replace $conn'((S, C))$ **by** $conn((S, C))$ **in** $comp$
  end for

  The **copy** function creates new connections for every signal driver. Connections for variables are not changed. Let $conn'$ = **copy** $conn$ and x be a variable or signal driver, then $conn'$ is defined as follows:

$$conn'(x) = \begin{cases} \textbf{new connection} & \text{if } x \text{ is a signal driver} \\ conn(x) & \text{otherwise} \end{cases}$$

  The **replace-by-in** function replaces the given connection by the other specified connection within the given diagnosis system. This operation is necessary because connections associated with signals must be finally replaced by the connections associated with signal drivers (defined once for every process and signal).

- **Sequential Statements** are converted statement by statement. Since signal assignments change the value after leaving a process and our syntactical assumption that no time delays occur we have to consider the last statement. This is not the case if variable assignments are involved. Every variable read must use the immediately preceding variable assignment.

  convert $((St_1, \ldots, St_n), comp, conn, signalConn, P)$
  for $i = 1$ **to** n **do**
  convert $(St_i, comp, conn, signalConn, P)$
  end for

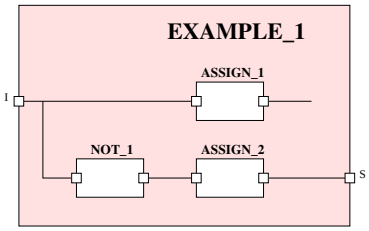
- **Signal, Variable Assignments** are converted to a component $C_{assignment}$ with one input in and one output out. The behavior is simple. If the assignment is assumed to be correct the input value is converted to the output and vice versa, i.e., $assignment(C) \wedge \neg ab(C) \Rightarrow in(C) = out(C)$. The following algorithm converts a signal assignment to a component. Although signal values changed only after exectuting a process, we have to introduce a new connection for the signal. After conversion the remaining connection stored in $conn$ is replaced by the connection associated with the driver.

  convert $(T <= E, comp, conn, signalConn, P)$
  c = **new** $C_{assignment}$
  $comp = comp \cup \{c\}$
  $in(c)$ = **convert** $(E, comp, conn, signalConn)$
  $conn((T, P))$ = **new connection**
  $out(c) = conn((T, P))$

For example the program fragment

```
EXAMPLE_1: process (I)
begin
  S <= I ;
  S <= not I ;
end process EXAMPLE_1;
```

is mapped to the following component connection model:



We see that the first assignment has no effect on the value of signal *S*. Note, that this kind of conversion is only correct if that no delay time is specified.

As stated previously the execution of the variable assignment immediately changes the variable value. We model this by changing the associated connection.

convert $(T := E, comp, conn, signalConn, P)$
  c = **new** $C_{assignment}$
  $comp = comp \cup \{c\}$
  $in(c)$ = **convertExpr** $(E, comp, conn, signalConn)$
  $conn(T)$ = **new connection**
  $out(c)$ = $conn(T)$

- **Conditionals** are mapped to a new instance of a generic conditional component C_{cond}. The number of inputs and outputs varies depending on the signal and variables used as target of an assignment. Assume C = " **if** $Cond$ **then** $Stmnts_1$ **else** $Stmnts_2$ **end if** " and a function $targets : STATEMENTS \mapsto 2^{SIGNALS \cup VARIABLES}$ associating statements to the used target signals and variables. If no *ELSE* branch exists we assume a default statement part $Stmnts_2$ that consists out of an assignment $T <= T$ $(T := T)$ for every signal (variable) T from $targets(Stmnts_1)$. We further assume that $targets(Stmnts_1) = targets(Stmnts_2)$. If both branches exists and $targets(Stmnts_1) \neq targets(Stmnts_2)$ we add an assignment $T <= T$ $(T := T)$ for every signal (variable) T from $targets(Stmnts_2) \setminus targets(Stmnts_1)$ to the *THEN* branch an an assignment $T' <= T'$ $(T' := T')$ for every signal (variable) T' from $targets(Stmnts_1) \setminus targets(Stmnts_2)$ to the *ELSE* branch. After applying this modification rules both $targets$ sets must be equal.

convert $(C, comp, conn, signalConn, P)$
  c = **new** $C_{cond}(|targets(Stmnts_1)|)$
  $comp = comp \cup \{c\}$
  $cond(c)$ = **convertExpr** $(Cond, comp, conn)$
  $conn'$ = **copy** $conn$

convert $(Stmnts_1, comp, conn', signalConn, P)$
$conn''$ = **copy** $conn$
convert $(Stmnts_2, comp, conn'', signalConn, P)$
for $x \in targets(Stmnts_1)$ **do**
  $i = index(x)$
  $in1_i(c)$ = $conn'(x)$
  $in2_i(c)$ = $conn''(x)$
end for

The behavior of the conditional statements is defined as expected. For every element x of $target(Stmnts_1)$ ($i = index(x)$) we introduce the rules
$conditional(C) \wedge \neg ab(C) \Rightarrow (cond(C) = true \rightarrow out_i(C) = in1_i(C))$
$conditional(C) \wedge \neg ab(C) \Rightarrow (cond(C) = false \rightarrow out_i(C) = in2_i(C))$.

Additionally, and because we have no recursive functions and loops, we can check whether a condition is wrong or not. This is done by the rules
$conditional(C) \wedge wrong(C) \Rightarrow (cond(C) = true \rightarrow out_i(C) = in2_i(C))$
$conditional(C) \wedge wrong(C) \Rightarrow (cond(C) = false \rightarrow out_i(C) = in1_i(C))$.

- **Expressions** used in the right hand side of an assignment statement or as a condition in an *IF* statement are mapped to components and connections as follows:

  Constants are mapped to a diagnosis component $C_{constant}$ with one output port out and an attribute $const$ storing the constants value. The behavior of the component is given by
  $constant(C) \wedge \neg ab(C) \Rightarrow out(C) = const(C)$.

  The conversion of the constant C is given by:

  convertExpr $(C, comp, conn, signalConn)$
    c = **new** $C_{constant}$
    $comp = comp \cup \{c\}$
    $conn(C)$ = **new connection**
    $out(c) = conn(C)$
    return $conn(C)$

  Variables and Signals are not mapped to components but to connections. Therefore no behavior is given.

  The conversion of a signal S uses the $signalConn$ function rather than the $conn$ function as for variables.

  convertExpr $(S, comp, conn, signalConn)$
    return $signalConn(X)$

  The conversion of a variable V is given by:

  convertExpr $(V, comp, conn, signalConn)$
    return $conn(V)$

Functions are converted to a diagnosis component $C_{function}$ with input and output ports and an associated behavior. For simplicity we restrict the use of functions to *NOT*, *AND*, *OR*, *XOR*, *NOR*, *NAND*, =, <, >, <=, >=, <>. For the *AND* function for example, we have a component C_{AND} with two inputs in_1, in_2 and one output out. The behavior of the *AND* function is given by
$and(C) \wedge \neg ab(C) \Rightarrow fand(out(C), in_1(C), in_2(C))$
where $fand('1','1','1')$, $fand('0','0',X)$, $fand('0',X,'0')$ are facts.

The conversion of functions $F(X_1, \ldots, X_n)$ with n parameters X_i (expression) is defined by:

convertExpr $(F(X_1, \ldots, X_n), comp, conn, signalConn)$
  c = **new** $C_{function}(F)$
  $comp = comp \cup \{c\}$
  $conn(F(X_1, \ldots, X_n))$ = **new connection**
  $out(c) = conn(F(X_1, \ldots, X_n))$
  for $i = 1$ **to** n **do**
    $in_i(c)$ = **convertExpr** $(X_i, comp, conn, signalConn)$
  end for
  return $conn(F(X_1, \ldots, X_n))$

The **convert** algorithm defines the conversion of programs into a component oriented view. Have a look at figure 2 for the graphical representation of the component connection model for program COUNTER(BEHAV). Since this representation may contain cycles we must eliminate them. Therefore we first convert the program into a functional dependency graph and remove the signal vertices eliminating all cycles. In [SW95] the conversion of VHDL programs into a functional dependency graph is shown.

Let $\{S_1, \ldots, S_n\}$ be a minimal set of signals that, when removed, eliminates all cycles for a given program. We adapt the component connection model described above by introducing new connections $C_1, \ldots, C_n$ (associated with $S_1, \ldots, S_n$) and replacing every connection $conn(S_i)$ connected with a port p with the new connection C_i. This final model represents the system description used for diagnosis. Figure 4 shows the final system description for the example program COUNTER(BEHAV).

It remains to give a condition specifying those signals and variables that must be observed in order to compute a diagnosis. This set is given by all signals and variables where their associated connection is only used in exactly one port. All such signals and variables are either inputs or outputs.

In our case we have to give the values for the inputs CLK, $CLK'EVENT$, $RESET$, $Q1(IN)$, $Q2(IN)$ and the outputs $Q1$ and $Q2$. The connection $Q1(IN)$ is associated with signal $Q1$ and the connection $Q2(IN)$ is associated with signal $Q2$. The name is only used to distinguish between the input and the output value for $Q1$ ($Q2$). From the semantics point of view the value of $Q1(IN)$ at time $t \neq 0$ is given by the value of $Q1$ at the immediately preceding time point $t - 1$. At time $t = 0$ the value of $Q1(IN)$ is equal to the initial value.

4 Using the model for program debugging

Until now we have shown the conversion of VHDL programs into system descriptions for model-based diagnosis and specified the set of signals and variables whose values must be known to find the location of a bug. In this section, we use the example from section 2 to demonstrate how bugs can be located. For this purpose we introduce a single bug into the COUNTER(BEHAV) program. Instead of the line " D2 <=

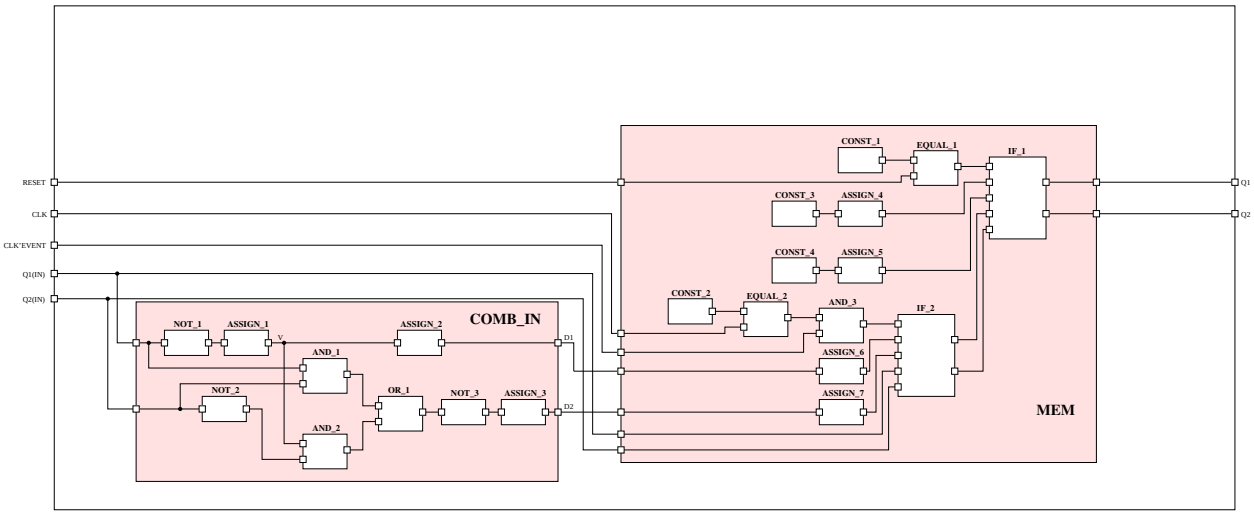


Figure 4: The final diagnosis system for COUNTER(BEHAV)

not ((O1 **and** O2) **or** (V **and** **not** (O2))); " in the correct program we assume that the faulty architecture (named FAULTY) has the line " D2 <= **not** ((O1 **and** O2) **and** (V **and** **not** (O2))); ".

The faulty program COUNTER(FAULTY) now produces a behavior as given in figure 3 (b). The Comparison of this waveform trace with the original one (figure 3 (a)) leads to the detection of discrepancies. In the next step we use the specified behavior together with the computed system description. Because of the kind of bug the structure of the system description of both programs is equal. Therefore we can use the schematics depicted in figure 4 for diagnosis. We only have to assume that component OR_1 behaves like an *AND* component. The specified behavior (given in figure 3 (a)) can be viewed as observations for the diagnosis system. Table 1 shows the list of observations. The columns of the table are the test vectors for diagnosis. Vectors leading to contradictions are denoted with a star (*) on the right side of the table.

At this point we are interested in locating the bug. Since we use standard model-based diagnosis techniques this means that we are searching for a subset of the component set, that when assumed to behave incorrect, does not anymore lead to a contradiction. See [Rei87] for a formal definition, [SD89] for an extension to handle different behaviors, and [FGN90] for introducing the concept of physical impossibilities. In the context of model-based diagnosis, debugging is viewed as searching for component behaviors not leading to a contradiction. For example if we assume that IF_1 is faulty, i.e., ab(IF_1), and that all other components $C \in COMPS$ behave correct, i.e., $\neg ab(C)$ then it can be easily proven that no contradiction occurs anymore. Hence we have found a diagnosis $\{ab$(IF_1)$\}$ which can be mapped back to the first conditional statement of the process MEM.

In summary we get 12 minimal diagnoses: $\{ab$(AND_1)$\}$, $\{ab$(OR_1)$\}$, $\{ab$(NOT_3)$\}$, $\{ab$(ASSING_3)$\}$, $\{ab$(ASSIGN_7)$\}$, $\{ab$(IF_2)$\}$, $\{ab$(CONST_2)$\}$, $\{ab$(EQUAL_2)$\}$, $\{ab$(AND_3)$\}$, $\{ab$(CONST_1)$\}$, $\{ab$(EQUAL_1)$\}$, $\{ab$(IF_1)$\}$.

This initial result is good in the sense that the number of possible fault locations is reduced. Only 12 of original 22 components can cause the faulty behavior[4]. However, it does not improve the results obtained using much simpler approaches such as [FSW96], where the two processes MEM and COMB_IN would be

[4]Note, that the number of computed diagnoses depends on the system and the observations. In some examples the result can be much better.

Table 1: List of observations for diagnosis

test	$RESET$	CLK	$CLK'EVENT$	$Q1(IN)$	$Q2(IN)$	$Q1$	$Q2$	
1	'1'	'0'	true	'0'	'0'	'0'	'0'	
2	'1'	'1'	true	'0'	'0'	'0'	'0'	
3	'1'	'0'	true	'0'	'0'	'0'	'0'	
4	'1'	'1'	true	'0'	'0'	'0'	'0'	
5	'1'	'0'	true	'0'	'0'	'0'	'0'	
6	'0'	'0'	false	'0'	'0'	'0'	'0'	
7	'0'	'1'	true	'0'	'0'	'1'	'0'	$*^1$
8	'0'	'0'	true	'1'	'0'	'1'	'0'	
9	'0'	'1'	true	'1'	'0'	'0'	'1'	
10	'0'	'0'	true	'0'	'1'	'0'	'1'	
11	'0'	'1'	true	'0'	'1'	'1'	'1'	
12	'0'	'0'	true	'1'	'1'	'1'	'1'	
13	'0'	'1'	true	'1'	'1'	'0'	'0'	*
...	.....					.....		
...	.....					.....		

diagnosis candidates. Although the result of the new model is more accurate it does not distinguish between the two processes. In order to improve diagnosis using the component connection model we have to take a closer look at the involved components, their behavior modes, and the possible corrections.

- **Conditional statements** have three behavior modes: $\neg ab(C)$, $ab(C)$, and $wrong(C)$ to indicate correct behavior, incorrect behavior, and a fault within the condition causing the execution of the wrong path. The $ab(C)$ mode is the most general one. Nothing about the real behavior is specified leading to the conclusion that everything is possible. Therefore this mode should only be taken into account if nothing else (e.g., $wrong(C)$) explains to a wrong behavior. Correcting a faulty conditional statement with an $ab(C)$ mode is difficult. Several corrections are possible including removing the conditional, adding lines of code before or after the conditional statement, among others. Only a bug in the used condition justified by $wrong(C)$ is not caught. The $wrong(C)$ mode gives more information about the bug location than the $ab(C)$ mode. In the first diagnosis step we can assume that all components forming a condition work as expected, i.e., are correct. In a next step after verifying $wrong(C)$ we concentrate on the condition for finding the bug. The correction of a faulty conditional statement with a $wrong(C)$ behavior has to be done by correcting the condition.

- **Assignments** have two modes, one for the correct behavior ($\neg ab(C)$) and one indicating a fault ($ab(C)$). A $ab(C)$ gives no exact hint about the kind of fault. Only two possible faults can be excluded, a faulty expression and a wrong target signal. The wrong expression is handled by the connected components associated with the expression. The wrong target signal can only be handled by assuming a structural fault in the converted system. A possible correction for a faulty assignment is to add new statements before or after it. By assuming that only few changes are necessary to correct the fault we can ignore assignments as diagnosis unless nothing else is a diagnosis.

- **Constants, Functions** can be correct ($ab(C)$) or faulty ($\neg ab(C)$). Faulty constants or functions can be corrected by replacing them with another constant, function or variable (see [SW98]).

Filtering diagnosis candidates is done as follows:

1. Remove all diagnoses where components mapped to a condition are included.
2. Remove candidates where conditional statements C are assumed to behave incorrectly $ab(C)$ but where we know that the condition is not faulty ($\neg wrong(C)$).
3. Eliminate diagnoses that includes faulty assignments.

Note, that the rules 2, 3 are only applicable while at least one diagnosis remains. Rule 1 is not allowed to be used if diagnosis focus on finding the bug in a conditional. We illustrate the filtering criteria by using our running example where only 3 diagnoses remain: $\{ab(\text{AND_1})\}$, $\{ab(\text{OR_1})\}$, $\{ab(\text{NOT_3})\}$.

This result can be further improved by using a technique from [SW98] where multiple testcases are used. Assume for example that diagnosis $\{ab(\text{NOT_3})\}$ is correct. In this case we can compute the following input/output table:

test	in(NOT_3)	out(NOT_3)
7	'0'	'0'
9	'0'	'1'
11	'0'	'1'
13	'0'	'0'

We see that with the same input value, i.e., '0', different output values should be returned. Assuming the working hypothesis, that the correct program is a small variant of the wrong one, leads to the conclusion that NOT_3 can not be a single diagnosis. Note, that if this working hypothesis is not used, then replacing a *NOT* by an *AND* (with one additional input) might be applicable and we cannot reject this diagnosis.

With this technique diagnosis $\{ab(\text{AND_1})\}$ can also be rejected. Hence, finally only the diagnosis $\{ab(\text{OR_1})\}$ remains. Because the described diagnosis and filtering steps are performed automatically using the new model, the bug is found automatically.

Although, the techniques described in the paper are not general applicable, they can be used in a wide variety of examples with similar results. In some examples where the simplifying assumptions can not be used, we can find the bug locations by using measurement selection (see [dKW87]) and oracle calls, i.e., user calls or a formal specification. However, the results computed remain correct and provide a reduction of the initial search space. However, there is a problem with the model. In the current version it can not handle bugs related to the use of a wrong variable. Finding such a fault is related with a structural change in the model. Similar problems occur in hardware diagnosis, e.g., bridge faults. Some solutions that may also be useful for software debugging has been proposed elsewhere [Dav84, Böt95].

In the remainder of this section we argue that the model can be used for debugging of medium-sized and large VHDL programs. Because we have not yet implemented the proposed approach we give an estimation for the diagnosis of single bugs. This is done by comparing state of the art diagnosis algorithms with the expected number of generated components per KBytes of design size. Table 2 (a) gives the performance results of the DRUM-2 algorithm (see [FN97]) for combinational circuits.

Assuming independence of involved random variables and equal probability, the expected diagnosis time per gate is 0.007751 seconds. In hardware design, typical simulation runs last from several minutes to

Table 2: Empirical informations of the performance of diagnosis algorithms and real-world designs

Circuit	Gates	Diag. Time [sec]	Time per Gate [sec]
c499	202	0.13	0.000644
c880	383	0.02	0.000052
c1355	547	1.78	0.003254
c2670	1193	6.43	0.005390
c3540	1669	1.42	0.000851
c5315	2307	0.31	0.000134
c6288	2406	118.59	0.049289
c7552	3512	8.4	0.002392

(a) Performance of DRUM-2 (from [FN97])

Design	Size [kByte]	Gates	Gates per kByte
D-01	2,553	17,188	6.73
D-02	952	1,030	1.08
D-03	48	1,335	27.81
D-04	18	809	44.94
D-05	13	220	16.92
D-06	12	284	23.67
D-07	8	156	19.50

(b) Approximated number of gates for real-world designs

several days. If we want to restrict diagnosis time by an upper limit of 10 minutes (that is convenience to hardware designers) we can diagnose systems with up to 77,400 components. Within 10 seconds, smaller programs with up to 1,300 components can be diagnosed in the average case. Note, that this estimation is based on the assumption that the time for computing single diagnoses is linear in the number of components.

Finally, we combine the performance results with the expected number of components per kByte of source code. This number is taken from real-world examples (see figure 2 (b)). Assuming statistical independence we get that about 20 components are created for 1 kByte of source code. Note, that we do not restrict components (gates) to be digital. Therefore, the given number of components in figure 2 (b) is smaller than the number of digital components obtained after applying synthesis, i.e., converting VHDL programs into hardware.

Hence, we can expect to diagnose programs with up to 3.8 MB of source code within 10 minutes and programs with up to 65 kByte of source code within 10 seconds. Although this estimation is very rough it provides a strong evidence for the usefulness of the proposed method.

5 Conclusion and related research

Over the past years several approaches for debugging hardware designs have been proposed. [CWH94] introduces algorithms for locating and correcting faults in gate level designs where only boolean values and functions were used. Our approach on the other hand is not restricted to boolean values. Instead arbitrary data types and their functions can be used. Applying model-based diagnosis to software debugging especially for the hardware description language VHDL is not new. [FSW96] has proposed an abstract model in order to handle very large programs with up to 10 MB source code. This approach is similar to the program slicing [Wei84] but extents it in some directions. In [SW98] a program description for VHDL process statements handling the full (sequential) VHDL semantics was introduced. This paper extents previous work in the VHDL domain by using a concise model for the concurrent statements. Since not all parts of VHDL are utilized it is not general usable such as [FSW96]. However, we can handle those parts of VHDL that are mostly used in todays hardware designs.

In summary we use the following restrictions for VHDL programs:

- Only the predefined data types including IEEE standard logic are allowed.
- Neither loops nor recursive function calls are allowed. Code with loops that can be translated into a loop free version can be supported with the proposed technique.
- No delay times must be specified.
- Only one resolution function (for standard logic) is supported.

For RTL synthesis the above restrictions must be also applied. But some synthesis restrictions are not used because a possible hardware implementation must not be taken into account. Future work and research in this area should include: (1) A proof that the above model really does not change the VHDL semantics for the used subset. (2) An implementation together with a detailed analysis. Performance and an evaluation of the diagnosis results should be done. (3) Extending the VHDL subset to the more general case including user defined data types and recursive function calls.

References

[Böt95] Claudia Böttcher. No faults in structure? How to diagnose hidden interaction. In *Proc. IJCAI*, Montreal, August 1995.

[CWH94] Pi-Yu Chung, Yi-Min Wang, and Ibrahim N. Hajj. Logic design error diagnosis and correction. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2:320–332, 1994.

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

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

[FGN90] Gerhard Friedrich, Georg Gottlob, and Wolfgang Nejdl. Physical impossibility instead of fault models. In *Proceedings AAAI*, pages 331–336, Boston, August 1990. Also appears in Readings in Model-Based Diagnosis (Morgan Kaufmann, 1992).

[FN97] Peter Fröhlich and Wolfgang Nejdl. A Static Model-Based Engine for Model-Based Reasoning. In *Proceedings* 15^{th} *International Joint Conf. on Artificial Intelligence*, Nagoya, Japan, August 1997.

[FSW96] Gerhard Friedrich, Markus Stumptner, and Franz Wotawa. Model-based diagnosis of hardware designs. In *Proc. ECAI*, Budapest, August 1996.

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

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

[SD89] Peter Struss and Oskar Dressler. Physical negation — Integrating fault models into the general diagnostic engine. In *Proc.* 11^{th} *IJCAI*, pages 1318–1323, Detroit, August 1989.

[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.

[SW98] Markus Stumptner and Franz Wotawa. Model-based debugging of functional programs. In *Proc. DX'98 Workshop*, Cape Cod, May 1998.

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

[VHD98] IEEE P1076.6/D1.12 Draft Standard For VHDL Register Transfer Level Synthesis, 1998.

[Wei84] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.