Debugging synthesizeable VHDL Programs*

Franz Wotawa
Technische Universität Wien
Institut für Informationssysteme
Paniglgasse 16, A-1040 Wien, Austria,
Email: wotawa@dbai.tuwien.ac.at

Abstract

This paper describes the use of model-based diagnosis for locating bugs in hardware designs. We restrict our view to hardware designs written in a subset of the commonly used hardware description language VHDL. This subset includes all synthesizeable (register transfer level) programs. This are programs which can be automatically converted into a gate level description without changing their behavior. Therefore almost all VHDL programs are elements of this subset. We show the conversion of VHDL programs into a logical representation. Take this representation and apply model-based diagnosis. The resulting diagnoses are mapped back to the VHDL code fragments of the original program explaining misbehaviors. Finally, we specify some rules optimizing the obtained results. We further present some arguments showing that the proposed debugging technique scales up to large designs.

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 (Chung, Wang, & Hajj 1994; Friedrich, Stumptner, & Wotawa 1999; Stumptner & Wotawa 1998). In this paper we describe a new approach for diagnosing hardware designs. Similar to previous work we convert VHDL programs (see (Navabi 1993) 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 (Reiter 1987; de Kleer & Williams 1987)) 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 (Friedrich, Stumptner, & Wotawa 1999) 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 (Stumptner & Wotawa 1998)) 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 (VHDL IEEE Std 1076.6/D1.12 1998)) 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 ben-

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

efits: (1) A model used for diagnosis is automatically derived from the given program. (2) Diagnosis is done using standard model-based diagnosis techniques. (3) Because of the syntactical restrictions the approach can be applied even to large programs. (4) The approach is applicable to VHDL register transfer level (RTL) programs. Most of todays designs are written using the VHDL RTL subset. Such programs must not use loops and recursive functions because they provide no correspondence to hardware. The syntactical restrictions for synthesizeable programs are specified in (VHDL IEEE Std 1076.6/D1.12 1998). (5) There is no need for additional formal specifications and assertions or program annotations.

A VHDL Example

In this section we describe the behavior of a simple counter program written in VHDL. For an introduction into VHDL have a look at (Navabi 1993; VHDL IEEE Std 1076-1987 1988). 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 gives the VHDL program consisting of an entity COUNTER and and architecture BEHAV. The architecture describes the behavior of the counter. The signals CLK and RESETS are the inputs, and O1 and O2 are the outputs of the device. Two processes (MEM and COMB_IN) implement the desired functionality. The output of the counter program (assuming that the value changes on signals CLK and RESET are provided) is shown in figure 2 (a) in the form of a waveform trace. A waveform trace is a sequence of events, i.e., value changes, occurring on signals over time.

The VHDL semantics specify the behavior of a program and can be 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'.

To work as a running example we introduce a bug into the program. We change the program by replacing the **or** in line 25 by an **and**.

25*. D2 <= **not**((O1 **and** O2) **and** (V **and** **not**(O2)));

The resulting (faulty) architecture is named F5. The output of the COUNTER(F5) program is given in figure 2 (b).

```
1. entity COUNTER is
2. end COUNTER;
3. architecture BEHAV of COUNTER is
4.    signal CLK, RESET : BIT ;
5.    signal O1, O2 : BIT ;
6.    signal D1,D2 : BIT ;
7. begin
      :
8.    :   -- VHDL code for test case stimulating CLK and RESET
9.    mem: process ( CLK, RESET )
10.   begin
11.      if RESET = '1' then
12.         O1 <= '0';
13.         O2 <= '0';
14.      else if CLK = '1' and CLK'EVENT then
15.            O1 <= D1;
16.            O2 <= D2;
17.      end if;
18.   end if;
19.   end process mem;
20.   comb_in : process ( O1, O2 )
21.      variable V: BIT;
22.   begin
23.      V := not(O1);
24.      D1 <= V;
25.      D2 <= not((O1 and O2) or (V and not(O2)));
26.   end process comb_in;
27end BEHAV;
```

Figure 1: The VHDL COUNTER(BEHAV) program

Modeling programs

In this section we show the conversion of programs into diagnosis systems which 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 (Chung, Wang, & Hajj 1994).
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 the component connection model. A component is an object with ports and a specified behavior. Connections are associated with 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[1]. 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.

We further 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 mappings of signals are 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.

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[2]. Since only IEEE standard logic (see (VHDL IEEE Std 1076-1987 1988)) 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 conversion algorithm is given such as follows:

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$(St_i, comp, conn, signalConn, P)$
  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$(C, comp, conn)$
  convert$((St_1, \ldots, St_n), comp, conn, signalConn, C)$

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 oriented view.

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

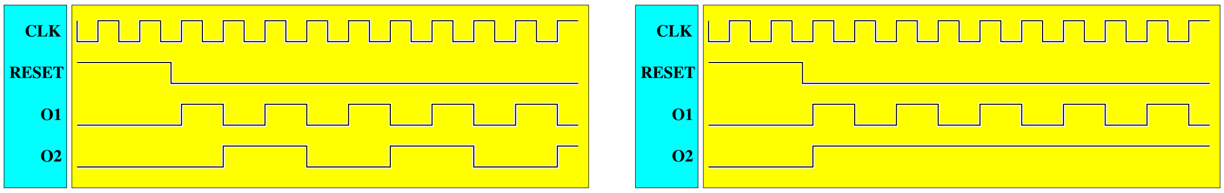
As stated previously the execution of the variable assignment immediately changes the variable value. We model this by changing the associated connection.

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 $targets$:

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

[2] 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.



(a) Behavior of COUNTER(BEHAV) (b) Behavior of COUNTER(F5)

Figure 2: The behavior of the two COUNTER programs

$STATEMENTS \mapsto 2^{SIGNALS \cup VARIABLES}$ 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 **copy** function creates new connections for every signal driver and variable. 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

$$\begin{array}{c} conditional(C) \wedge \neg ab(C) \Rightarrow \\ (cond(C) = true \rightarrow out_i(C) = in1_i(C)) \\ conditional(C) \wedge \neg ab(C) \Rightarrow \\ (cond(C) = false \rightarrow out_i(C) = in2_i(C)) \end{array} .$$

Additionally, and because we have no recursive functions and loops, we can check whether a condition is wrong or not, without any problems.

$$\begin{array}{c} conditional(C) \wedge wrong(C) \Rightarrow \\ (cond(C) = true \rightarrow out_i(C) = in2_i(C)) \\ conditional(C) \wedge wrong(C) \Rightarrow \\ (cond(C) = false \rightarrow out_i(C) = in1_i(C)) \end{array} .$$

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 connection model. Cycles in the structure of the model, which may occure, must be eliminated. To do this we convert the program into a functional dependency graph and remove as much signal vertices as necessary for eliminating all cycles. In (Friedrich, Stumptner, & Wotawa 1999) 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 3 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, $CLK'EVENT$, $RESET$, $O1(IN)$, $O2(IN)$ and the outputs $O1$ and $O2$. The connection $O1(IN)$ is associated with signal $O1$ and the connection $O2(IN)$ is associated with signal $O2$. The name is only used to distinguish between the input and the output value for $O1$ ($O2$). From the semantics point of view the value of $O1(IN)$ at time $t \neq 0$ is given by the value of $O1$ at the immediately preceding time point $t-1$. At time $t = 0$ the value of $O1(IN)$ is equal to the initial value.

Using the structural informations and the behavioral rules from above we can compute the system description for the COUNTER(BEHAV) design unit $SD_{COUNTER(BEHAV)}$. The following logical representation mainly are concerned about the part of $SD_{COUNTER(BEHAV)}$ associated with the variable assignment from line 23 *V := **not**(O1);* .

- *Components:* $not(NOT\_1), assign(ASSIGN\_1), \ldots$
- *Connections:*
  $O1(IN) = in(NOT\_1), out(NOT\_1) = in(ASSIGN\_1),$
  $out(ASSIGN\_1) = in_1(AND\_2),$
  $out(ASSIGN\_1) = in(ASSIGN_2), \ldots$
- *Behavior:*
  $\neg ab(NOT\_1) \rightarrow out(NOT\_1) = fnot(in(NOT\_1)),$
  $\neg ab(ASSIGN\_1) \rightarrow$
  $out(ASSIGN\_1) = in(ASSIGN\_1), \ldots$

Model-based Debugging

Before showing how model-based diagnosis can be used for debugging, we briefly recapitulate the basic definitions (see (Reiter 1987; Struss & Dressler 1989; Wotawa 1996)). A diagnosis problem consists of a system description (SD) describing the behavior of components (COMP) and their interconnections. The component behavior is associated with component modes. For every component at least the correct behavior ($\neg ab$ mode) must be specified. Let OBS be a set of observations. A diagnosis for $(SD, COMP, OBS)$ is a set of mode assignments Δ, mapping components to their actual mode, such that $SD \cup OBS \cup \{m(C) | m(C) \in \Delta\}$ does not lead to a contradiction.

Using model-based diagnosis for locating bugs in software needs the existence of SD, $COMP$, and OBS. In the previous section we have introduced the system description SD_D derived from a VHDL program D consisting of an entity and an architecture. Additionally, we now the set of components $COMP$ (together with their modes). In our case the set of observations OBS can be derived from waveform traces. While SD is computed from the buggy program, the facts stored in OBS describe the intended behavior, i.e., the specification. For example the observations OBS (table 1) are computed from the waveform trace of the correct COUNTER(BEHAV) program (figure 2 (a)). The columns of table form a test vector that are the observations for diagnosis.

We illustrate the application of model-based diagnosis to debugging using our buggy counter example COUNTER(F5). Because of the bug the component connection model of COUNTER(F5) is equal to the one depicted in figure 3 expect that the *or*-component OR_1 is replaced by an *and*-component. We assume that this and component has the same name, i.e., OR_1. The observations are given by test vector 7 (tc7). Using the system description $SD_{COUNTER(F5)}$ and the observations OBS_{tc7} and assuming that the program, i.e., all components behave correct, leads to a contradiction. This contradiction can be resolved by searching for faulty components. For example if we assume that IF_1 is faulty, i.e., $ab(IF\_1)$, and that all other components are correct, then it can be easily proved that the resulting sentence is consistent. So $\{ab(IF\_1)\}$ is a diagnosis. In summary we obtain 12 minimal diagnosis: $\{ab(\text{AND\_1})\}$, $\{ab(\text{OR\_1})\}$, $\{ab(\text{NOT\_3})\}$, $\{ab(\text{ASSING\_3})\}$, $\{ab(\text{ASSIGN\_7})\}$, $\{ab(\text{IF\_2})\}$, $\{ab(\text{CONST\_2})\}$, $\{ab(\text{EQUAL\_2})\}$, $\{ab(\text{AND\_3})\}$, $\{ab(\text{CONST\_1})\}$, $\{ab(\text{EQUAL\_1})\}$, $\{ab(\text{IF\_1})\}$.

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[3]. However, it does not improve the results obtained using much simpler approaches such as (Friedrich, Stumptner, & Wotawa 1999), where the two processes MEM and COMB_IN would be 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

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

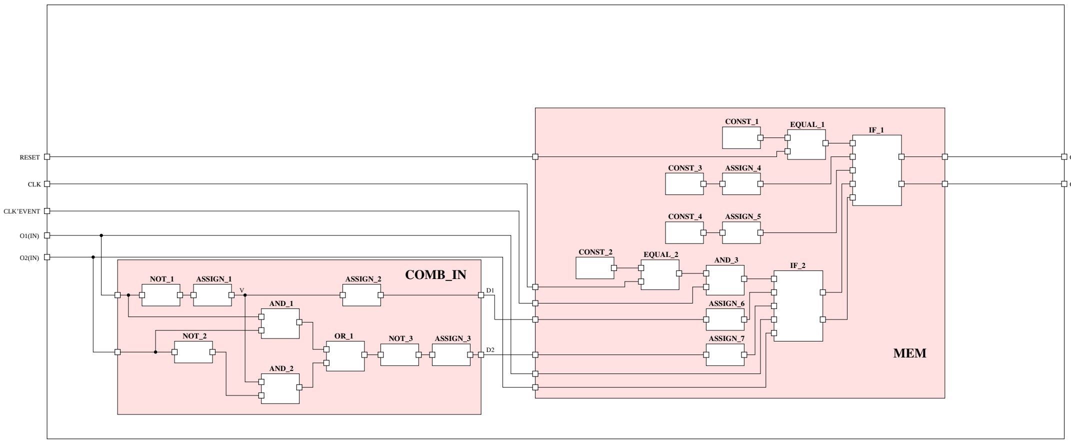


Figure 3: The final diagnosis system for COUNTER(BEHAV)

tc	$RESET$	CLK	$CLK'EVENT$	$O1(IN)$	$O2(IN)$	$O1$	$O2$	
1	'1'	'0'	true	'0'	'0'	'0'	'0'	
2	'1'	'1'	true	'0'	'0'	'0'	'0'	
..	....	....	....	....	....	....	....	
7	'0'	'1'	true	'0'	'0'	'1'	'0'	$*^1$
8	'0'	'0'	true	'1'	'0'	'1'	'0'	
..	....	....	....	....	....	....	....	
13	'0'	'1'	true	'1'	'1'	'0'	'0'	$*$
..	....	....	....	....	....	....	....	

Table 1: List of observations for diagnosis

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 that the expression contains the bug can be excluded, because wrong expressions are handled by the connected components associated with the expression. Possible corrections for a faulty assignment are to change the target signal, to add new statements before or after it, or to remove the statement. By assuming that only few changes are necessary to correct the fault and that structural faults are less likely, we can ignore assignments as diagnosis unless no other diagnosis can be computed.

- **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 (Stumptner & Wotawa 1999)).

The following rules summarizes the above principles.

1. Remove candidates where conditional statements C are assumed to behave wrong $ab(C)$ but where we know that the condition is not faulty ($\neg wrong(C)$).

2. Remove all diagnoses where components associated with a condition from a conditional statement are included, except when the conditional in the a diagnosis is mapped to the $wrong$ mode.

3. Eliminate diagnoses that include faulty assignments.

They work as a filter for the computed diagnosis candidates. The rules are only applicable while at least one diagnosis remains. They should be applied in the same order they are given above. If a rule eliminates all diagnoses the rule must not be applied. All previously removed diagnoses (using the rule) must be added to the set of resulting diagnoses. We illustrate the application of the rules by using our running example. After filtering only 3 diagnoses remain: $\{ab(\text{AND\_1})\}$, $\{ab(\text{OR\_1})\}$, $\{ab(\text{NOT\_3})\}$ including the expected diagnosis $\{ab(\text{OR\_1})\}$.

design	bug description
f1	[line 14] *CLK = '0'* instead of *CLK = '1'*
f2	[line 12] *O1 <= '1'* instead of *O1 <= '0'*
f3	[line 23] *O1* instead of ***not**(O1)*
f4	[line 15] ***not**(D1)* instead of *D1*
f5	[line 25] *. . .) **and** (. . .* instead of *. . .) **or** (. . .*
f6	[line 25] *D2 <= (O1 **and**. . .* instead of *D2 <= **not**(. . .*
f7	[line 14] *RESET = '1'* instead of *CLK = '1'*
f8	[line 25] *D1 <= . . .* instead of *D2 <= . . .*
f9	[line 16] *O1 <= D2* instead of *O2 <= D2*
f10	[line 16] *O2 <= D1* instead of *O2 <= D2*

Table 2: Wrong VHDL designs derived from the VHDL design COUNTER(BEHAV)

design	test	expected	diags
f1	tc7	true	7
f2	tc1	true	5
f3	tc7	true	8
f4	tc7	true	10
f5	tc7	true	12
f6	tc7	true	12
f7	tc7	true	7
f8	tc7	true	15
f9	tc7	true	15
f10	tc7	false	7

Table 3: Initial debugging results before applying the filtering rules Note: *expected* is true if the expected diagnosis is element of the computed diagnoses, and false, otherwise.

design	test	expected	diags
f1	tc7	true	4
f2	tc1	true	4
f3	tc7	true	1
f4	tc7	true	1
f5	tc7	true	3
f6	tc7	true	3
f7	tc7	true	4
f8	tc7	true	5
f9	tc7	false	5
f10	tc7	false	5

Table 4: Debugging results after applying the filtering rules Note: *expected* is true if the expected diagnosis is element of the computed diagnoses, and false, otherwise.

First Results

In this section we give first results. This includes an evaluation of the capability of debugging using 10 wrong variants *F1,. . . ,F10* of the counter example described herein. See table 2 for the introduced bugs. In addition, we argue about the scalability of the approach, i.e., that it can be either used for small and large VHDL designs.

The introduced bugs spans a wide area, from using wrong constants in expressions and conditions, and changing and removing functions used in expressions. 4 test cases (*F7,. . . ,F10*) deal with bugs related to the usage of the variable. Only missing are test cases with multiple bugs.The first results are very promising. Although, various additional improvements are still possible and required. This includes the use of knowledge about probabilities (Burnell & Horvitz 1995) and the use of multiple test cases (Stumptner & Wotawa 1999). Beside the debugging capability a debugging tool needs a carefully designed user-interface presenting the required informations and hints to the programmer.

Table 3 shows the results for the faulty designs without using the filtering rules. For all designs the search space from original 22 components is reduced to at most 15 components. The expected diagnosis is only excluded once. However, more diagnoses than necessary have been computed. Further improvements can be reported after applying the filtering criteria (see table 4). In the worst case 5 potential candidates remain. In the best case only one diagnosis is delivered as results. In one case the filtering rules remove the expected diagnosis. These first results indicate that in about 80 percent of all cases our approach provide substantial savings by focusing the programmers attention to the possible bug locations. The remaining cases can be handled by specialized models which have to be developed.

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 k-Bytes of design size. Table 5 (a) gives the performance results of the DRUM-2 algorithm (see (Fröhlich & Nejdl 1997)) for combinational circuits.

Assuming independence of involved random variables and equal probability, the expected diagnosis time per component is 0.007751 seconds. In hardware design, typical simulation runs last from several minutes to 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 k-Byte of source code. This number is taken from real-world examples (see figure 5 (b)). Assuming stochastical independence we get that about 20 components are created for 1 k-Byte of source code. Note, that we do not restrict components to be digital. Therefore, the given number of components in figure 5 (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 k-Byte of source code within 10 seconds. Although this estimation is very rough it provides a strong evidence for the usefulness of the proposed method.

Circuit	Number of Components	Diag. Time [sec]	Time per Comp. [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 (Fröhlich & Nejdl 1997))

Design	Size [k-Byte]	Comp.	Comp. per k-Byte
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 components for real-world designs

Table 5: Empirical informations of the performance of diagnosis algorithms and real-world designs

Conclusion and related research

Over the past years several approaches for debugging hardware designs have been proposed. (Chung, Wang, & Hajj 1994) 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. (Friedrich, Stumptner, & Wotawa 1999) 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 (Weiser 1984) but extents it in some directions. In (Stumptner & Wotawa 1998) 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 (Friedrich, Stumptner, & Wotawa 1999). 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: (1) Only the predefined data types including IEEE standard logic are allowed. (2) 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. (3) Delay times must not be specified. (4) 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 (if possible) recursive function calls.

References

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

Chung, P.-Y.; Wang, Y.-M.; and Hajj, I. N. 1994. Logic design error diagnosis and correction. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 2:320–332.

de Kleer, J., and Williams, B. C. 1987. Diagnosing multiple faults. *Artificial Intelligence* 32(1):97–130.

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

Fröhlich, P., and Nejdl, W. 1997. A Static Model-Based Engine for Model-Based Reasoning. In *Proceedings* 15^{th} *International Joint Conf. on Artificial Intelligence*.

Navabi, Z. 1993. *VHDL Analysis and Modeling of Digital Systems*. McGraw-Hill.

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

Struss, P., and Dressler, O. 1989. Physical negation — Integrating fault models into the general diagnostic engine. In *Proc.* 11^{th} *IJCAI*, 1318–1323.

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.

1988. IEEE Standard VHDL Language Reference Manual LRM Std 1076-1987.

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. 1996. *Applying Model-Based Diagnosis to Software Debugging of Concurrent and Sequential Imperative Programming Languages*. Ph.D. Dissertation, Technische Universität Wien.