

TOWARDS A FRAMEWORK FOR AUTOMATED DEBUGGING: ABSTRACTING THE TEMPORAL BEHAVIOR OF VHDL-RTL PROGRAMS

Bernhard Peischl and Franz Wotawa ¹

*Technische Universität Graz
Institut for Software Technology
8010 Graz, Inffeldgasse 16b/2, Austria
{peischl,wotawa}@ist.tugraz.at*

Abstract: In this article we propose a formal framework that allows for a precise description of automated debugging of hardware designs. Unlike to previous work, that deals with localizing faults by considering the state of the system at a single point in time, we outline an approach that exploits the temporal behavior of hardware designs employing a standard diagnosis engine by unfolding the program with respect to time.

Keywords: software debugging, debugging of hardware designs, debugging framework

1. INTRODUCTION

During the last decades, hardware description languages such as Verilog and VHDL have become more and more popular. Almost every company in the area of digital circuit design uses some sort of hardware-description language in order to improve the design process. The aim of the design process is to write a program that describes the behavior and structure of the circuit and can be automatically synthesized into a gate-level representation. This process allows for detecting and correcting faults in an early stage of the design cycle and thus design costs can be significantly decreased. However, searching for faults in a design tends to be a very difficult and time consuming process since complex hardware designs reach dimensions of several 100.000 lines of code and may be written by a team of designers that are located at different physical locations. Automated debugging tools that support fault localization and correction within hardware designs may thus provide considerable aid in decreasing the time to market and may contribute to the reduction of overall costs in today's fast paced economy.

Locating bugs in hardware designs has thus been in the focus of several contributions in literature. In this

paper we focus on the use of model-based diagnosis (Reiter, 1987; de Kleer and Williams, 1987) for locating bugs in programs. Although, this paper is based on these underlying principles it is different to previous approaches in this area with respect to the employed model. Friedrich and colleagues (Friedrich *et al.*, 1999) use a dependency-based model that leads to similar results than traditional approaches, e.g., program slicing (Weiser, 1982). Wotawa (Wotawa, 2002) introduced a model that captures the semantics of a large subset of VHDL. This model, however, does not consider time. Hence, diagnosis of VHDL programs is limited to a setting where the input state of the program and the corresponding output state has to be provided as an observation. In order to overcome this limitation, the approach described in this paper allows for representing programs as a sequence of process executions and state changes. This sequence can be compiled into a logical description by using Wotawa's (Wotawa, 2002) logical models for processes, sequential statements, and expressions.

In the following we outline the basic idea behind the model. For this purpose we use the following small VHDL program:

1. A1: S <= I1 or CLK;
2. A2: O <= not S;

¹ Authors are listed in alphabetical order. The work was partially supported by the Austrian Science Fund (FWF) under project grant P15163-INF.

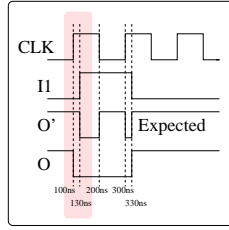


Fig. 1. The signals' values over time

This program comprises two concurrent signal assignments $A1$, and $A2$. The signals $I1$ and CLK are the input signals from which values for signals S and O are computed. Figure 1 depicts the computed value and the expected value of O . The two values are different at some points in time. Hence, there must be a bug in the program. The behavior of the program can be explained as follows. At time 100ns the input values are set to $CLK = '1'$ and $I1 = '0'$. Hence, statement $A1$ causes the value of S to be set to '1'. From this statement $A2$ computes $O = '0'$. During this computation step no simulation time elapses but the computation corresponds to a sequence of states and active processes, e.g., signal assignment statements.

In the following sequence, states are denoted by brackets \llbracket_T^i where i stands for the number of the state at a specific time point T , and active processes are surrounded by braces $\{\}$.

$$\begin{aligned} & \llbracket_{100ns}^0 \{ CLK = '1', I1 = '0', S = '0', O = '1' \} \\ & \quad \{ A1 \} \\ & \llbracket_{100ns}^1 \{ CLK = '1', I1 = '0', S = '1', O = '1' \} \\ & \quad \{ A2 \} \\ & \llbracket_{100ns}^2 \{ CLK = '1', I1 = '0', S = '1', O = '0' \} \end{aligned}$$

Having reached state 2 no further signal changes occur and thus the state of computation at time point 100ns is stable. This view on the VHDL behavior naturally reflects the semantics of VHDL and can be easily compiled into a diagnosis system. The diagnosis system comprises all temporal instances of processes which occur in the sequence. As an advantage we get a loop-free diagnosis system which is a requirement to model statements and expression as introduced in (Wotawa, 2002).

The rest of the paper is organized as follows. Section 2 introduces some basic notions and proposes a formal framework that allows for a precise description of debugging VHDL designs. Afterwards, in Section 3 the framework is applied to line out our basic approach towards exploiting the temporal behavior of a VHDL program in order to localize the causes of an observed misbehavior. Finally, the paper is concluded in Section 4.

2. A FRAMEWORK FOR DEBUGGING VHDL-RTL PROGRAMS

VHDL designs consist of entities, architectures and statements, to only mention the most important program constructs. Entities represent the physical parts or subdivisions of the circuit to be designed and architectures are used to describe the internal structure of an entity. In the common terminology of programming

languages an entity corresponds to an abstract data type and an architecture describes its implementation. The behavior of an entity is described by its body that usually consists of several processes. A process itself is composed of statements, such as signal assignments or conditionals. Whenever at least one signal from the sensitivity list of a process changes its value, the statements in the process are executed and the resulting changes to its output signals are propagated to other processes, possibly causing them to be executed in turn. Thus, the communication between processes is done by means of signals. The parallel execution of the processes of different entities, resulting in the simulated behavior of the hardware unit to be designed, is performed by executing the VHDL program and recording the signal changes over time.

An overview of the VHDL language features and a definition of syntax and semantics can be found in (IEEE, 1988). Furthermore (Navabi, 1993) provides an introduction into designing circuits with VHDL.

For debugging VHDL programs it is reasonable to require some additional restrictions. These restrictions have proven to be meaningful in the context of debugging VHDL-RTL designs and thus are subsumed in the following by the term RRTL (Reduced Register Transfer Level). Below a definition of the RRTL subset of VHDL is given.

Definition 1. (RRTL). The Reduced Register Transfer Level (RRTL) program is a VHDL-RTL program where the following conditions are valid:

- (1) No signal assignment has an VHDL after clause.
- (2) Processes have a sensitivity list but no wait statements.
- (3) The program halts on the given test cases.

A VHDL program consists of processes that are assumed to be executed in parallel. As outlined above, the different processes communicate by means of signals. Moreover, each process may use variables in order to represent its state. This state is encapsulated, that is, only statements of the corresponding process can change the value of a variable whereas the state of the signals can be modified globally. However, there is a difference in the semantics of a signal and a variable assignment. Whereas the variable-assignment statement may change the value of the target variable immediately, the value of the target of the signal-assignment statement is modified after process execution has completed. Note that a signal-assignment statement does not directly change the value of a signal, rather it changes the value of a so called signal driver. This is due to VHDL semantics where multiple signal-assignment statements may influence the same signal. If the same signal is declared as an target in different processes a special function, called resolution function, is used to deduce the final value of that signal at a given point in time. The resolution function thereby specifies the dependency between the signal and the drivers of a specific signal and takes all the drivers of a signal as arguments and returns a unique value which is stored as the value of the signal at a

specific point in time. In the following we introduce interpretations for signals and variables as well as for signal drivers:

Definition 2. (Interpretation). Let SIG be signal and VAR be a variable of a RRTL program. Let $PROC$ be a processes of the program. VAL denotes a domain of values. The functions $E_T^I : SIG \cup VAR \mapsto VAL$ and $E_T^I : SIG \times PROC \mapsto VAL$ represent the value of a signal/variable respectively a signal driver at a given point in time T and a specific state I .

We use these definitions to formalize the semantics of simulating a RRTL program. To provide a compact description we also have to introduce the notion of an event on a specific signal according to VHDL semantics. Thus, in the following, each signal S is associated with another signal $S'EVENT$ that is true, if and only if the signal S has changed its value in the current state of simulation, otherwise it is false.

Simulating a RRTL program corresponds to computing all the values of the signals, variables and driver values for a predefined ordered set of times. In the following the simulation of a program is decomposed into atomic steps also referred to as basic simulation cycles (BSC). A BSC at a specific point in time T and a given state I of a RRTL program comprises of two steps:

- (1) Execute all processes that have at least one event signal S in their sensitivity list.
- (2) Compute the signal values from their driver values. If two processes are writing the same signal, use a resolution function to determine an unique value.

After performing such a BSC the point in time of simulation remains the same but the state of the program in general may have changed. The modification of the state is described by enumerating the bindings of the signals and variables to their associated values before and after process execution. In the following the structure enumerating those bindings is referred to as environment, which from a structural point of view, is a set of ordered pairs.

Definition 3. (Environment). The environment E_T^I at time T and state I is a set of ordered pairs (X, V) where $X \in SIG \cup VAR \cup (SIG \times PROC)$ and $V \in VAL$. The domain $dom : ENV \mapsto NAMES$, where ENV denotes an environment and $NAMES$ is a set of variables/signals respectively signal drivers, is a function that returns the signals/variables/drivers of the given environment ENV .

Note that events are defined in the form of signals rather than as a function and therefore they are also part of the environment. In the following we give a formal definition of a BSC of a RRTL program at a specific time T in state I .

Definition 4. (Basic Simulation Cycle (BSC)). A basic simulation cycle BSC_T^I of a RRTL program is

a tuple $(E_T^I, P_T^I, E_T^{I+1})$ where I is the current state at time T , and E_T^I, E_T^{I+1} denotes the environments before respectively after executing the processes in P_T^I in state I at time T .

Note, that according to the VHDL semantics, carrying out $BSC_{T_0}^0$ executes all the processes $P_{T_0}^0$ that built up the RRTL program. The simulation for time T_0 is continued by performing the next simulation cycle $BSC_{T_0}^1$ by executing those processes whose sensitivity list enumerates a signal on which an event has occurred in the previous cycle $BSC_{T_0}^0$. In this way, a process that enumerates specific signals in the sensitivity list is invoked as long as there occurs an event on at least one of these signals in the previous BSC. The whole process of performing BSC 's finishes if there is no event triggering any of the processes. Referring back to assumption (3) of Definition 1 such a state has to be reached and thus only a finite number of BSC 's are carried out. The environment after performing the last BSC represents the bindings recorded for the signal and variable values of time T_0 . However, composing a finite number of BSC 's to define the values of the signals and variables at a specific point in time can be stated in a natural way by introducing the notion of a simulation sequence.

Definition 5. (Simulation sequence). A simulation sequence Σ_T at time T of a RRTL program is a sequence of simulation cycles $BSC_T^0, BSC_T^1, \dots, BSC_T^{len}$. The state number of the last element of the simulation sequence is called the length of the simulation sequence len .

Recalling the definition of a BSC, the output environment of BSC_T^i is the input environment of BSC_T^{i+1} . Note, that the starting environment $E_{T_0}^0$ has to be defined appropriately in order to compute meaningful values for the variables at time T_0 . In the terminology of circuit design establishing an appropriate environment corresponds to providing input values at time T_0 . Rather than describing the effects of applying input values at a specific point in time our framework allows for computing the impact on an input stimulus, i.e. a sequence of input values.

Definition 6. (Input Stimulus). An input stimulus for a RRTL program is a sequence of the form $(E_{T_0}^0, T_0), (E_{T_1}^0, T_1), \dots, (E_{T_k}^0, T_k)$ where each $E_{T_i}^0$ denotes an environment representing the input at time T_i .

In our framework the simulation of a program can formally be expressed by computing the bindings of the signals and variables at discrete moments in time. However, there are two points that are worth noting. First, there is an ordering relation on the time points that has to be reflected in our framework. Second, conforming to that ordering relation, the stable environment at the end of a simulation sequence is the input environment of the succeeding simulation sequence. In formal terms this can be expressed as follows:

Definition 7. (Simulation). A simulation of a RRTL program on a given input stimulus $(E_{T_0}^0, T_0), (E_{T_1}^0, T_1), \dots, (E_{T_k}^0, T_k)$ is a sequence of simulation sequences $\Sigma_{T_0}, \Sigma_{T_1}, \dots, \Sigma_{T_m}$ where the time points are absolutely ordered, i.e., $0 < T_1 < \dots < T_m$. The initial input values of a simulation sequence Σ_{T_j} is defined as follows. For each signal or variable X :

$$E_{T_j}^0(X) = \begin{cases} E_{T_j}^0(X) & j=0 \vee X \in \text{dom}(E_{T_j}^0) \\ E_{T_{j-1}}^{len}(X) & \text{otherwise} \end{cases}$$

The formula given above, states that the state at time point T_{j-1} is used for performing the simulation at time point T_j except from those variables and signals that are explicitly given in the input environment $E_{T_j}^0$. Note that the formalization requires an initial binding for every signal or variable X in $E_{T_0}^0$.

Now our framework allows for expressing the simulation of a program with respect to a given input stimulus in formal terms. The result of a simulation is a mapping from variables and signals to values at a given point in time. In order to allow for a specification of the expected behavior, the definition of an input stimulus has to be extended.

In addition to the input stimulus given in Definition 6, a test stimulus also contains the corresponding expected results. Thus, a test stimulus consists of a sequence of test cases. Each test case in turn extends an input environment with the expected output environment. In formal terms this can be expressed as follows:

Definition 8. (Test Stimulus). A test stimulus for a RRTL program is a sequence of tuples of the form $(E_{T_0}^0, E_{T_0}^{out}, T_0), (E_{T_1}^0, E_{T_1}^{out}, T_1), \dots, (E_{T_k}^0, E_{T_k}^{out}, T_k)$ where each $E_{T_i}^0$ and $E_{T_i}^{out}$, $i \in \{0..k\}$, denotes the input and output environment, respectively.

At a technical level a faulty program is one that does not produce the specified results for a specific stimulus. In terms of the definitions given above, a RRTL program is considered to be faulty with respect to a given stimulus, if there exists at least a single test case that contradicts the expected behavior. Within our formal framework this can be expressed as follows:

Definition 9. (Faulty program). Given a test stimulus $(E_{T_0}^0, E_{T_0}^{out}, T_0), (E_{T_1}^0, E_{T_1}^{out}, T_1), \dots, (E_{T_k}^0, E_{T_k}^{out}, T_k)$ for a RRTL program. The program is faulty with respect to the given stimulus if there exists an element $(X, V) \in E_{T_i}^{out}$ for an arbitrary time point T_i where $V \neq E_{T_i}^{len}(X)$.

Note that a test is defined with respect to a given stimulus. In this context a test is said to be successful if it identifies a faulty program, i.e., the associated stimulus does not pass the test. However, if a test is not successful, in general, we cannot conclude that the program is correct. We only know that it is correct with respect to a given stimulus.

3. FAULT LOCALIZATION IN RRTL-DESIGNS USING TEMPORAL ABSTRACTION

In automated software debugging we use a model of the VHDL program and a successful test for localizing faults in a given faulty program. In order to employ model-based diagnosis, we have to provide a component-connection model that reflects the structure and the behavior of the diagnosis problem in terms of logical sentences. In our case, for every program artefact (e.g. a statement, process or function) we have to provide a model of the correct behavior by means of a propositional theory. For example, in the following a logical description of a process component is given. A process p is executed if at least one of the signals that occur in its sensitivity list has changed its value immediately before. If this is the case, the values of the signals used as a target in the sequential statement part of p are computed according to the VHDL semantics. This computation is formally represented by components that are associated with the sequential statements of process p . Hence, the value of the signals are given by the values computed by the sub block connected to the input of process p . If none of the signals within the sensitivity list has changed its value, then the original input values before executing the sequential statement block are propagated to the output of the process component. Formally, this can be written as follows:

$$\begin{aligned} &\forall y \in \text{inputs}(p) \cdot \exists x \in \text{sensitivity-list}(p) \cdot \\ &\quad s_x(p) = \text{true} \rightarrow \text{out}_y(p) = \text{in}_y(p) \\ &\forall y \in \text{inputs}(p) \cdot \forall x \in \text{sensitivity-list}(p) \cdot \\ &\quad s_x(p) = \text{false} \rightarrow \text{out}_y(p) = \text{def}_y(p) \\ &\forall y \cdot \text{out}_y(p) = \text{def}_y(p) \leftrightarrow \text{out}_{y, \text{EVENT}}(p) = \text{false} \\ &\exists y \cdot \text{out}_y(p) \neq \text{def}_y(p) \leftrightarrow \text{out}_{y, \text{EVENT}}(p) = \text{true} \end{aligned}$$

Thereby $\text{inputs}(p)$ denotes the set of inputs of the process p and $\text{sensitivity-list}(p)$ represents the sensitivity list of the process p . The predicate $s_x(p)$ becomes true if any signal that is enumerated in the sensitivity list has changed its value. The signal value after executing the sequential statement y is represented by $\text{in}_y(p)$ whereas the corresponding unmodified input of the process p is represented in formal terms by $\text{def}_y(p)$. If an output signal is changed, the corresponding event signal is set to true. A detailed discussion on modeling other VHDL-RTL artefacts is out of scope of this article and can for example be found in (Wotawa, 2002).

The modeling process, e.g., the computation of a system description SD can be expressed by the system descriptions that corresponds to the simulation sequence for a given test-case. Formally, the system description for a simulation cycle $S_{T_0}, S_{T_1}, \dots, S_{T_k}$ where each S_{T_i} is a sequence of BSCs $BSC_{T_0}^0, \dots, BSC_{T_i}^{len}$ is given by the following logical sentences:

$$SD = \bigcup_T SD(S_T) \cup \text{CONN}, \quad SD(S_T) = \bigcup_{i=1}^{\text{len}(S_T)} SD(BSC_T^i)$$

where $SD(BSC_T^i)$ is composed from the system description that corresponds to the active processes together with information about the handling of the sen-

sitivity list described previously. Moreover, *CONN* denotes sentences which describe both, the connection between the BSC sequences, and between each BSC of a sequence. This connectivity is given by the signals a temporal process instance uses likewise as input and output.

Since we know the number of process activations in advance a temporal instance of the process can be created for every activation. By connecting the output of the i^{th} process instance $p^{(i)}$, associated with state i , to the input of process instance $p^{(i+1)}$, that is correspondingly associated with state $i + 1$, a chain of processes is built that allows for an explicit representation of the transformation on the state. In addition, we have to provide some additional logic that decides whether a specific process instance is to be activated or simply forwards the input values to the next process instance without doing any transformations on the state.

The system description which is associated to the temporal process instance can for example be found in (Wotawa, 2002). Hence, the system description *SD* we described herein is based on (Wotawa, 2002) and can be seen as its extension that captures the semantics of VHDL in two aspects. First, the model is able to use an input stimulus rather than a single test case by unfolding the signal values over time. Second, the treatment of processes is much closer to the semantics of the VHDL language than in previous models.

Beside the system description *SD* we have to define the observations *OBS* and the set of components *COMP* before we are able to use the model-based diagnosis approach. The set of observations *OBS* for our model is equal to the test stimulus of the original program. The set of components is given by the set of temporal process instances, i.e., $COMP = \{Stmt(p_T^i) | p_T^i \in BSC_T^i\}$ where *Stmt* returns all diagnosis components that corresponds to the process instance. Diagnosis components that represent the same statement in the code but belong to different process instances are treated as different components. A multiple fault diagnosis thus may exclusively contain components corresponding to the same statement in the source code.

The system that is obtained by using the approach outlined above can be applied to compute diagnosis directly. Unlike to our previous models, in which a simple test case is used to localize the fault, a test stimulus can be used in order to compute diagnoses. However, since temporal instances of a process are handled completely independent, the number of components that may account for a certain misbehavior may increase in comparison to the original model. Thus, the obtained diagnoses are mapped back to the unfolded system by reducing the instances to their corresponding components in accordance to the following procedure.

$$\begin{aligned} \varphi : \Delta^* \subseteq COMP^* &\mapsto \Delta \subseteq COMP \\ \varphi(\Delta^*) &= \{C_i | C_i^{(j)} \in \Delta^*\} \end{aligned}$$

In the mapping given above *COMP** denotes the instances of the components whereas *COMP* refers directly to the components. In similar fashion, Δ^* stands

```

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
15.       if CLK = '1' and CLK'EVENT then
16.         O1 <= D1;
17.         O2 <= D2;
18.       end if;
19.     end if;
20.   end process mem;
21.
22.   comb_in : process ( O1, O2 )
23.     variable V: BIT;
24.   begin
25.     V := not(O1);
26.     D1 <= V;
27.     D2 <= not((O1 and O2) or (V and not(O2)));
28.   end process comb_in;
29. end BEHAV;

```

Fig. 2. The VHDL COUNTER(BEHAV) program

for the diagnoses that are computed using temporal instances and Δ simply denotes the corresponding components. Moreover, $C_i^{(j)}$ denotes the instance j of component C_i .

In order to demonstrate the applicability of our approach we use a small but realistic VHDL program. We first show how the simulation is done leading to the unfolded diagnosis system and than discuss the results that were obtained from our prototype implementation. The example program, which is outlined in Figure 2, is taken from (Wotawa, 2002) and implements a 2-bit counter. The signals *CLK* and *RESET* denote the inputs whereas *O1* and *O2* denote the output signals of the device. The program consists of two processes, *mem* and *comb_in*. Process *mem* is executed whenever an event on *CLK* or *RESET* occurs. Likewise, process *comb_in* is triggered by events on the signals *O1* and *O2*. Signals *D1* and *D2* are used to store future values of the counter. Whenever *CLK* changes from '0' to '1', the future values are transferred to *O1* and *O2* by process *mem* and new values are computed by process *comb_in*. We assume that all signal values (according to the VHDL semantics) are set to '0' and their corresponding EVENT signals are set to true at the beginning of the simulation which causes all processes to be executed immediately after starting the simulation. By using our formal framework this can be expressed as follows (where the corresponding EVENT signals at time T_0 and state 0 have been omitted for sake of brevity):

$$\left[\begin{array}{c} (CLK, '0') \\ (RESET, '0') \\ (D1, '0') \\ (D2, '0') \\ (O1, '0') \\ (O2, '0') \end{array} \right]_{T_0}^0 \left\{ \begin{array}{c} mem \\ comb_in \end{array} \right\} \left[\begin{array}{c} (CLK, '0') \\ (RESET, '0') \\ (D1, '1') \\ (D2, '0') \\ (O1, '0') \\ (O2, '0') \\ (V, '1') \\ (D1'EVENT, true) \end{array} \right]_{T_0}^1$$

Since there is only an event on signal *D1* and *D2*, this does not trigger process *mem* nor *comb_in* and the process execution stops. The bindings for time T_0

can be obtained from the environment $E_{T_0}^1$. In the next time point T_1 the clock signal is assumed to change its value from '0' to '1'. In our framework, the impact of the rising edge of the clock signal is represented by the following BSC:

$$\left[\begin{array}{c} (CLK, '1') \\ (CLK'EVENT, true) \\ (RESET, '0') \\ (D1, '1') \\ (D2, '0') \\ (O1, '0') \\ (O2, '0') \\ (V, '1') \\ (D1'EVENT, true) \end{array} \right]_{T_1}^0 \{ mem \} \left[\begin{array}{c} (CLK, '1') \\ (RESET, '0') \\ (D1, '1') \\ (D2, '0') \\ (O1, '1') \\ (O2, '0') \\ (V, '1') \\ (O1'EVENT, true) \end{array} \right]_{T_1}^1$$

$$\{ comb_in \} \left[\begin{array}{c} (CLK, '1') \\ (D1, '0') \\ (D2, '1') \\ (O1, '1') \\ (O2, '0') \\ (V, '0') \\ (D1'EVENT, true) \\ (D2'EVENT, true) \end{array} \right]_{T_1}^2$$

Again, neither $D1$ nor $D2$ are listed in the sensitivity list of the processes of our counter example and thus the simulation can be stopped. The bindings for the signal values at time T_1 can be retrieved from the environment $E_{T_1}^2$ by applying the interpretation function given in Definition 2.

For evaluating our model we intentionally introduced a bug in line 27 of our example. Before converting the source code to the model, we altered it in the following way:

27. $D2 \leq \text{not}((O1 \text{ and } O2) \text{ and } (V \text{ and } \text{not}(O2)))$;

Furthermore, we computed diagnoses by using two rising edges of the CLK signal, thus taking into account 4 cycles in time. In order to specify our arrangement exhaustively, the initial values of the signals and variables, $E_{T_0}^0$, as well as the expected values $E_{T_3}^2$ are given below:

$$\left[\begin{array}{c} (CLK, false) \\ (RESET, false) \\ (D1, false) \\ (D2, false) \\ (O1, false) \\ (O2, false) \\ (V, false) \end{array} \right]_{T_0}^0, \left[\begin{array}{c} (CLK, true) \\ (RESET, false) \\ (D1, true) \\ (D2, true) \\ (O1, false) \\ (O2, true) \\ (V, false) \end{array} \right]_{T_3}^2$$

From the intermediate starting environments $E_{T_1}^0$ and $E_{T_2}^0$ we only have knowledge of the value of the CLK and $RESET$ signal:

$$\left[\begin{array}{c} \dots \\ (CLK, true) \\ (RESET, false) \\ \dots \end{array} \right]_{T_1}^0, \left[\begin{array}{c} \dots \\ (CLK, false) \\ (RESET, false) \\ \dots \end{array} \right]_{T_2}^0$$

In terms of our framework, the relationship between the environments and the process instances can be represented in the following compacted form:

$$(E_{T_0}^0, \{ mem^{(1)}, comb_in^{(1)} \}, E_{T_0}^1),$$

$$(E_{T_0}^1, \{ mem^{(2)}, comb_in^{(2)} \}, E_{T_0}^2),$$

$$(E_{T_1}^0, \{ mem^{(3)}, comb_in^{(3)} \}, E_{T_1}^1),$$

$$(E_{T_1}^1, \{ mem^{(4)}, comb_in^{(4)} \}, E_{T_1}^2),$$

$$(E_{T_2}^0, \{ mem^{(5)}, comb_in^{(5)} \}, E_{T_2}^1),$$

$$(E_{T_2}^1, \{ mem^{(6)}, comb_in^{(6)} \}, E_{T_2}^2),$$

$$(E_{T_3}^0, \{ mem^{(7)}, comb_in^{(7)} \}, E_{T_3}^1),$$

$$(E_{T_3}^1, \{ mem^{(8)}, comb_in^{(8)} \}, E_{T_3}^2)$$

In our example a BSC consists of 2 processes, mem and $comb_in$. Since we assumed the process activation graph to be acyclic, for every process we have to reserve at most 2 instances. Thus, a single point in time requires 4 instances to be created. In total, since we considered 4 points in time, 16 instances have to be forseen for unfolding our model in time.

After invoking the diagnosis procedure we obtained 7 single fault diagnoses and 15 double fault diagnoses where each double fault diagnosis can be mapped back to a single bug in the source code when using the φ function. In summary the diagnosis correspond to the statements 9, 11, 15, 22, 25, 27 and include the introduced bug. The statements 9, 11, 15, and 22 correspond to a process or conditional statement. Only the diagnosis for statements 25 and 27 can be mapped back to functional faults, e.g., wrong operators, in the code. This result indicates that the new approach can localize the statement that is responsible for the misbehavior. Moreover, because of the information at which point in time which statements must be assumed to behave faulty, the user gets more information.

4. CONCLUSION

In this paper we introduced a modeling framework for VHDL programs that extents previous research and naturally represents the VHDL semantics. As an advantage some limitations of previous research regarding observations can be avoided. Moreover, we discussed a case study that proves the principle applicability of the approach. However, more sophisticated examples are required to allow for a detailed analysis of the outcome of the approach. Such an analysis should give an answer to the following questions. Which kind of faults cannot be detected (if any)? Can the approach be used for larger examples?

REFERENCES

- de Kleer, Johan and Brian C. Williams (1987). Diagnosing multiple faults. *Artificial Intelligence* **32**(1), 97–130.
- Friedrich, Gerhard, Markus Stumptner and Franz Wotawa (1999). Model-based diagnosis of hardware designs. *Artificial Intelligence* **111**(2), 3–39.
- IEEE (1988). IEEE Standard VHDL Language Reference Manual LRM Std 1076-1987. Institute of Electrical and Electronics Engineers, Inc. IEEE.
- Navabi, Zainalabedin (1993). *VHDL Analysis and Modeling of Digital Systems*. McGraw-Hill.
- Reiter, Raymond (1987). A theory of diagnosis from first principles. *Artificial Intelligence* **32**(1), 57–95.
- Weiser, Mark (1982). Programmers use slices when debugging. *Communications of the ACM* **25**(7), 446–452.
- Wotawa, Franz (2002). Debugging Hardware Designs using a Value-Based Model. *Applied Intelligence* **16**(1), 71–92.