VHDLDIAG+: Value-level Diagnosis of VHDL Programs

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

Abstract

We describe the application of model-based diagnosis to the debugging of VHDL programs. In our previous work in VHDL-based software diagnosis, we have relied upon a very abstract representation to make it possible to diagnose full-sized applications (up to 1MLOC) at the cost of reduced discrimination between diagnoses. This paper describes a more detailed representation for VHDL programs that explicitly reasons about the computation of variable and signal values, making it possible to examine the internal workings of VHDL processes to find a more exact cause of the errors in a program. The higher computational effort involved requires that the new representation be used as a refinement in tandem with the older, more abstract representation. Models for the full VHDL language extent are currently being developed. The implementation is being applied to test problems.

Introduction

Over the last years, the application of model-based diagnosis to software debugging has been examined by a number of researchers. The use of model-based diagnosis in fixing faults for logic programs was described in (Console, Friedrich, & Dupré 1993), with extensions proposed in (Bond 1994; 1996). A new application area for model-based diagnosis was opened with the VHDL-DIAG system (Friedrich, Stumptner, & Wotawa 1996) which searches for errors in VHDL programs. While software diagnosis presents a number of challenges for diagnosis approaches, VHDL possesses the expressiveness of a general purpose programming language, posing new problems in the search for diagnoses beyond those of systems which restrict themselves to declarative languages such as logical or functional programs.

The VHDLDIAG system described in (Friedrich, Stumptner, & Wotawa 1996) uses an abstract system description automatically converted from the VHDL source code, and can be used effectively to diagnose very large programs. However, it is limited in discriminating diagnosis candidates. In (Stumptner & Wotawa 1996b) and (Stumptner & Wotawa 1996a), proposals avoiding this drawback are introduced. They use a concise model which follows directly from the programming language semantics. A mode that allows replacing program parts by others is used, which means faults can not only be found but also corrected. The main assumption needed to make this approach feasible is to assume that small variants of a faulty one are more likely to be correct than completely different ones.

VHDL is an Ada-like programming language developed for describing hardware components and systems. Programs written in VHDL consist of processes which are executed in parallel. Processes themselves contain sequential statements. See (Navabi 1993) for an introduction. VHDL has the expressiveness of a general-purpose programming language and allows using recursive functions and loops. In practice, however, the final stage of the development of VHDL programs is the *register transfer level* (RTL), from which the gate level description of the circuit is synthesized (i.e., converted automatically). The gate level description then serves as direct input to the IC manufacturing process (which includes task such as layouting, routing etc.). RTL programs are not allowed to contain recursive functions and loops (except for loops the iterate over array elements, and these can be expressed by other means).

The VHDLDIAG system can be used in a standard VHDL development environment consisting of a compiler, a simulator, and other tools used to convert VHDL programs into hardware. Since hardware development projects are usually very large, several programmers are engaged producing small parts which are put together, tested, and debugged. Such tests are not always be done by the programmers themselves. In this situation, VHDLDIAG can save time in focusing on faulty program parts. During the hardware design process, changes to the specification are usually made where VHDLDIAG can help to localize the program parts which must be changed to fit the new specification. Apart from interactive use, scripts allow to automate the debugging process. In other words, searching for bugs overnight will still save overall project costs in an environment where multi-day simulation runs are not rare.

Our representation of VHDL programs has two parts.

One part, derived from the semantics of VHDL, is static and does independent of individual programs. The other part is directly derived from the individual program's code. In this paper, we will introduce a concise model for the sequential part of VHDL programs and give a more complete and general view of program debugging than in previous work.

We use the standard consistency-based view of model-based diagnosis (Reiter 1987; de Kleer & Williams 1987). A system description SD and a set of observations OBS are given. For every component there is a set of possible mode assignments (de Kleer, Mackworth, & Reiter 1990), each associated with a corresponding behavior. A set Δ of mode assignments to components from the system is a *diagnosis* if it has the property that SD and OBS are consistent with the mode assignments from Δ.

The paper is organized as follows: In the next section we give an example VHDL program, discuss its (faulty) execution and the signal semantics, which are the basis for gaining observations about the program. We then explain the basis for the value-level system description, the outcome of the diagnosis process using this description, and discuss the special heuristics needed for software debugging.

A VHDL Example

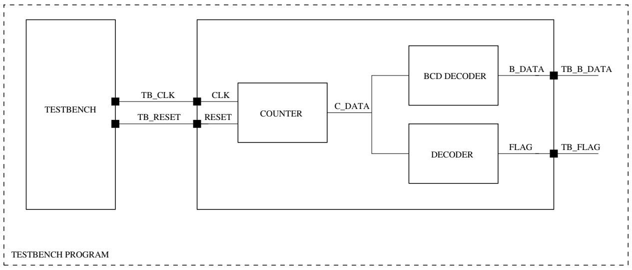
VHDL programs are divided into *entities* which represent physical parts or subdivisions of the circuit. The internal structure of an entity is described by its *architecture*. In common programming language terminology, an entity is an abstract data type and the architecture describes its internal structure.

The behavior of an entity is described by its body, which usually consist of the description of a *process*. The parallel execution of the processes of different entities, resulting in the simulated behavior of the hardware unit to be designed, is simulated by executing the VHDL program.

A process statement has two parts. One part contains a list of signals and variable declarations. The second consists of statements – in particular, variable and signal assignments, and branching (if) statements. Whenever at least one signal from the input signal list of a process p changes its value, the statement part of p is executed, with resulting changes to output signals being propagated to other processes.

To illustrate our approach we use a small VHDL program consisting of one counter and two decoder processes. The verbal specification of this program is as follows:

The counter can be reset to 0 by setting the reset input to 1. Counting happens incrementally by adding 1 on the rising edge of each clock pulse. If 10 is reached, the counter starts again at 0. One decoder converts the output of the counter to a number which can be displayed on a decimal LED display. The other delivers 1 if the counter value is



(a) Schematic of the Counter-Decoder-Example

```
entity E_COUNTER is
  port(
    CLK, RESET: in bit;
    FLAG: out bit;
    B_DATA: out bit_vector(1 TO 7));
end;

architecture BEHAVIOR of E_COUNTER is
  signal C_DATA: integer;
begin
  COUNTER: process(CLK)
    variable INTDATA: integer;
  begin
    if RESET = '1' then
      INTDATA := 0;
    else
      if CLK = '1' then
        if INTDATA > 9 then
          INTDATA := 0;
        else
          INTDATA := INTDATA - 1;
        end if;
      end if;
    end if;
    C_DATA <= INTDATA;
  end process;

  DEC: process(C_DATA)
  begin
    if C_DATA < 5 then
      FLAG <= '1';
    else
      FLAG <= '0';
    end if;
  end process;

  BCD: process(C_DATA)
  begin
    if C_DATA = 0 then B_DATA <= "1111110" endif;
    if C_DATA = 1 then B_DATA <= "0000110" endif;
    if C_DATA = 2 then B_DATA <= "1011011" endif;
    if C_DATA = 3 then B_DATA <= "1001111" endif;
    if C_DATA = 4 then B_DATA <= "0100111" endif;
    if C_DATA = 5 then B_DATA <= "1101101" endif;
    if C_DATA = 6 then B_DATA <= "1111101" endif;
    if C_DATA = 7 then B_DATA <= "1000110" endif;
    if C_DATA = 8 then B_DATA <= "1111111" endif;
    if C_DATA = 9 then B_DATA <= "1101111" endif;
    if (C_DATA < 0) or (C_DATA > 9) then
      B_DATA <= "1111001";
    end if;
  end process;
end;
```

(b) The (faulty) Counter-Decoder Implementation

Figure 1: A VHDL Example

less than 5 and 0 otherwise.

The schematic of the underlying circuit together with a testbench is given in figure 1(a). A program implementing the intended behavior is given in figure 1(b). It consists of the processes *COUNTER*, *DEC*, and *BCD*. To test this implementation, we use a test case consisting of a reset followed by incrementing the counter. The waveform trace from figure 2 represents the signal values over time produced by the test case. It it is easy to see that the implementation behaves faulty. The first discrepancy occurs at time 20 ms, with *TB_B_DATA* being set to hex *79* instead of hex *06*.

The Diagnosis Approach

In (Friedrich, Stumptner, & Wotawa 1996), we presented a system description based on *functional dependencies* (FDs) for diagnosing VHDL programs. Basically, it works by analyzing the dependencies between processes in the program and abstracting away from individual values. There is no examination of the exact computations that take place within a process. To use this representation for locating the bug in the example, we first compute the FDs for every process.

Process	FDs
COUNTER	C_DATA ← { CLK, RESET, C_DATA }
DEC	FLAG ← { C_DATA }
BCD	B_DATA ← { C_DATA }

The comparison of the traces results in the set of observations
$O = \{ok(CLK), ok(RESET), ok(FLAG), \neg ok(B\_DATA)\}$.
Diagnosis based on the FD system description together with O produces two single diagnoses, $\{$*COUNTER*$\}$ and $\{$*BCD*$\}$. Note that the output of *COUNTER* is used by *BCD* and *DEC*. Since *DEC*s output is correct, there is a strong indication that only *COUNTER* is buggy. By adding the discrepancy observed at time 100 ms to the observations, we can confirm this assumption.

Now we are interested in focusing on the bug within the *COUNTER* process. In this case, the trace supplies us with the values of all variables and signals at every point in time. This information can be collected from standard VHDL simulation environments. If we examine the situation at the first point in time where a signal shows a discrepancy, the values are:

Time	Values
20ms	Env = {CLK = 1, RESET = 0, C_DATA = 0, INTDATA = 0}
	Env' = {CLK = 1, RESET = 0, C_DATA = 0, INTDATA = -1}
	Hist = { C_DATA(0) = -1 }

Env denotes the environment (i.e., the set of signal and variable values) before the *COUNTER* process is run for this point in (simulated) time, and *Env'* the environment immediately after leaving the process. Note that the signal values in the environment are not changed directly within a process, but instead are entered into the signal's *history*. Signal assignments can specify a time delay for the assignment to take effect by using the keyword **after**. If a signal assignment has no **after** part, a delay of 0 is assumed. Formally, an environment is a function $Env : VARIABLE \cup SIGNAL \mapsto DOMAIN$ associating a value with a variable or signal within a process. A history is a function storing the value which is supposed to supersede the current value at a specific time, i.e., $Hist : SIGNAL \times TIME \mapsto DOMAIN$. Since multiple processes can attempt to assign conflicting values to a particular signal, the values proposed by the different processes have to be distinguished. Each process p places his assignments on a private version of the history that is called a *driver*. A particular value-time pair that is established in p by an assignment to a signal s (and therefore entered into the driver of p for the history of s) is called a *transaction* (*TA*). The update of the driver of s is specified by:

New TA will occur BEFORE existing TA	Overwrite existing transaction
New TA will occur AFTER existing TA	Overwrite existing transaction if different values otherwise keep both.

If multiple drivers contain transactions for the same time with different values, the new signal value is chosen by a so-called *resolution function* that is itself written in VHDL. The semantics of signal assignments also include a variant assignment that includes so-called *transport delay* with slightly different semantics.

The above definitions are used throughout this paper where a model for the process statement part is given.

A Value Level VHDL Model

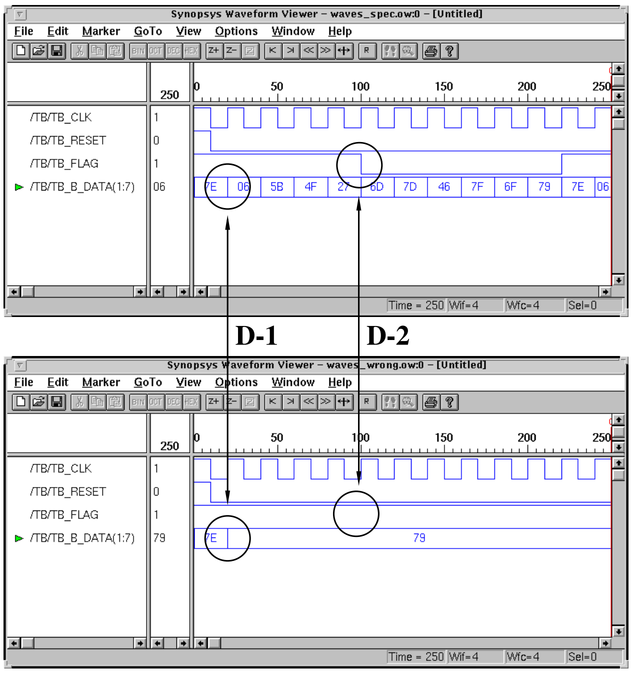
To diagnose a VHDL program, we first have to construct a system description of it. To be able to create the system description automatically, it must be possible to describe the system's behavior in terms of the behaviors of its components, just as in hardware diagnosis. That means the system description must be specified by describing the behavior of the components in the program.

Component Structure

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

- A *Variable assignment* has the form *V* **:=** *Expr*. The value of the variable *V* after execution is equal to the value of the expression *Expr*.

SPECIFICATION WAVEFORM TRACE



OBSERVED WAVEFORM

Figure 2: Comparing Specification with Implementation

- A *signal assignment* of the form *S <= Expr* **after** *TExpr* attempts to change the history of signal *S* using the value of expression *Expr* at the time *TExpr* according to the VHDL semantics, i.e., the transaction $(Expr, Texpr)$ is placed on the driver of S.

- An *if-statement* is given by **if** *Cond* **then** *St1* **else** *St2* **end if** where the **else** part is optional. If the Expression *Cond* evaluates to *true*, then *St1* is executed. Otherwise, the sequence of statements *St2* is executed.

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

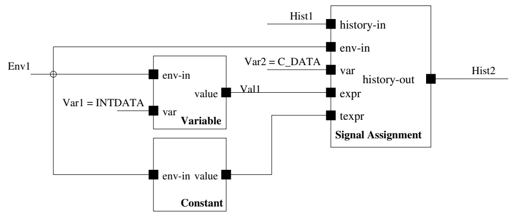
Just as with statements, the expressions contained in these statements can be mapped to a component oriented view. However, in this paper, we do not consider expressions as individual components for diagnosis for efficiency reasons.

- Signals and variables expression are mapped a components so that the current environment and the name of the signal (variable) are inputs and the value of the signal (variable) is the output.

- Operators have the values of the operands as input and the result as output

- Constants only have an output for propagating their value

See (Stumptner & Wotawa 1996a) for details of a similar representation and its use in diagnosing expressions. For example, the assignment statement *C_DATA <= INTDATA* is mapped to:



Component behavior

After describing the conversion of processes, we now give a model of the behavior of each diagnosis component. In VHDLDIAG+, consistency based diagnosis is used. We therefore introduce a model of the correct behavior. The function *value* is defined to return the value of a port *p*, e.g., *value(env_in)* returns the value of port *env_in*. We explain the model as a constraint propagation system. Values are propagated from component ports to other component ports. A contradiction arises if contradicting values are propagated.

- *Variable assignment* S: *"Var := Expr"* If statement S behaves correctly (written $nab(S)$), the environment is changed, i.e., the value of the variable given by the *var* port is set to the value obtained by the *expr* port. *value(expr)* (now equal to *value(var)*) is propagated to the *env_out* port. All other value assignments from the input environment are propagated to the output.

  In addition, propagation also occurs from *env_out* to *env_in*. All environment entries to variables other than

Var are propagated to *env_in* unchanged. If *Var*= V is specified for *env_out*, then the value V is propagated to the *value* port.

If an assignment statement S is assumed to be incorrect (written $ab(S)$), we have no further knowledge about the value that should be assigned to its variable *Var*. Therefore, no value for *Var* is propagated from *value* to *env_out*.

- *Signal assignment* S: *"Sig <= Expr* **after** *TExpr"*

  The behavior of signal assignments is similar to variable assignment statements, except that the signal history is changed instead of an environment. Note that this does not mean the signal will actually change to that value, since the assignment can later be overwritten by another assignment according to the signal update semantics.

- *If-Statement* IS: *"***if** *Cond* **then** *st1* **else** *st2* **end if***"*

  If $nab(IS)$, the propagation depends on the value of the *cond* port. If *value(cond)* is true, every element of *st1* (*env_out*) is propagated to *env_out* (*st2*). Otherwise, the same is done for *st2* instead for *st1*. Similar, history values are propagated.

  If IS is has a wrong condition, (written $wCond(IS)$), then propagation depends on the value of the *cond* port. Instead of propagating elements from *st1* to *env_out* in the case where *value(cond)* is true, elements from *st2* are used and vice versa. The same is done for the history values.

  An incorrect if-statement ($ab(IS)$) only propagates environments for signals from *env_in* to *env_out* and vice versa. No history is updated in this case.

Debugging a Program

We now show how to use the previously presented system description for VHDL process statements for debugging VHDL programs, illustrated on the *COUNTER* example. First, the *COUNTER* process has to be converted into a diagnosis component oriented representation. Figure 3 shows the resulting diagnosis problem. The system consists of seven diagnosis components which correspond to the seven sequential statements of the process.

Selection Criteria

Consider the semantics of the *ab* mode for conditional statements. There is a separate mode (*wCond*) that expresses the case where the condition is incorrect and should have been inverted (i.e., the other branch should have been followed). The general mode, *ab*, stands for the case where the whole conditional (together with its statement parts) must be revised to correct a bug. This is a more severe alteration than

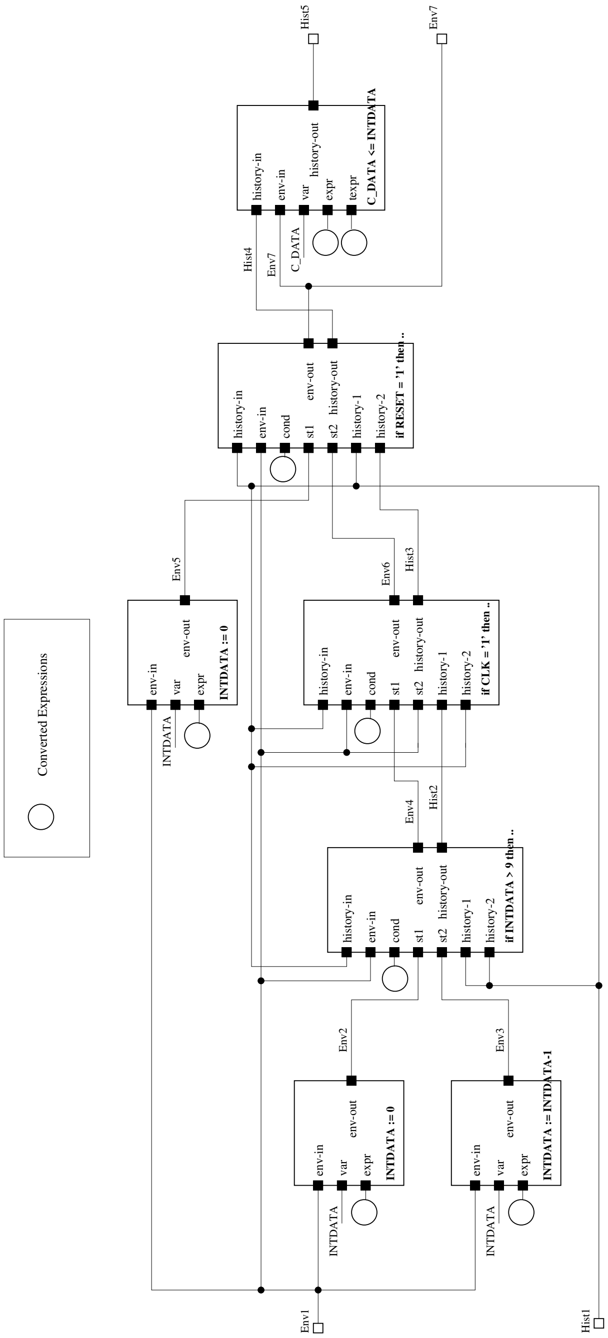


Figure 3: The *COUNTER* process converted into a diagnosis system

changing parts of the statement, e.g., adding a negation operator to the test expression. Given that we have a statement $s1$ that is contained within another statement $s2$, e.g., $s1$ is in the **then** branch of the if statement $s2$, it holds that $ab(s1)$ is more specific than $ab(s2)$ in the sense that for any diagnosis Δ_1 containing $ab(s1)$ there is always going to be an otherwise equal diagnosis Δ_2 containing $ab(s2)$ (but not vice versa). Therefore, we prefer Δ_1 to Δ_2.

We can also introduce a ranking guideline for diagnoses based on the underlying assumption that only minor changes are necessary to correct a program (Stumptner & Wotawa 1996a).

First, we have defined different fault modes for a statement. As pointed out in (Stumptner & Wotawa 1996a), the different nature of fault modes in software diagnosis means that the number of potential fault modes is effectively infinite, since a particular fault mode means a specific way in which a statement or expression fails to be correct, and therefore implies a specific bug fix (what would be a repair in hardware diagnosis). To cover all such circumstances, we require the existence of the universal fault mode ab for each type of statement, i.e., a mode which catches all kinds of errors. Of course, the generic nature of the mode implies that it is impossible to deterministically specify the nature of the fix that would correct the statement. Conversely, a more specific mode means that we have a clear idea of how the error influences the outcome produced by the program. We can express this in terms of the generality of fault modes.

Definition 1 *Let SD be a system description and OBS a set of observations. A fault mode m_1 of a statement S is more specific than a mode m_2, written $m_1 < m_2$, if, for every diagnosis Δ for (SD,OBS) with $m_1(S) \in \Delta$, it holds that $\Delta \setminus \{m_1(S)\} \cup \{m_2(S)\}$ is also a diagnosis for (SD,OBS).*

As an example, for a conditional statement S, a diagnosis containing $wCond(S)$ will be preferred over one containing $ab(S)$, all other modes therein being equal.

Definition 2 *A Diagnosis* Δ subsumes Δ' *if*

1. *there is at least one $m(S) \in \Delta$ and $m'(S) \in \Delta'$, such that $m < m'$, and $\Delta \setminus \{m(S)\}$ is preferred to $\Delta' \setminus \{m'(S)\}$ (i.e., in particular, there is no $m'(S') \in \Delta$ and $m''(S') \in \Delta'$ such that $m' > m''$ for any S').*

2. *there is at least one $ab(S) \in \Delta$ and $ab(S') \in \Delta'$, such that S is a substatement of S', and $\Delta \setminus \{ab(S)\}$ is preferred to $\Delta' \setminus \{ab(S')\}$ (i.e., in particular, there is no $m'(S') \in \Delta$ and $m''(S') \in \Delta'$ such that $m' > m''$ for any S').*

Another, heuristic ranking that can be considered is the complexity of the program part involved. All other things being equal, a more complex expression or statement will have a higher probability of a programmer error (typo or conceptual error) occurring within it. From this view, for example, *INTDATA := INTDATA - 1* is more complex than *C_DATA <= INTDATA*. As an approximation for complexity of an expression, we use the term depth of the statement (this is 2 for the former and 1 for the latter statement above).

Thus, the diagnosis process consists of three steps: First, diagnoses are computed using a standard algorithm, then subsumed diagnoses are removed, and finally, the remaining candidates are ranked according to heuristic criteria such as the one above.

Diagnosis

To debug the process, the intended behavior of the program part must be known. The behavior is assumed to be specified by an oracle. This can either be a user or a formal specification, or an executable higher level VHDL specification. It is the latter which would produce a waveform trace for comparison as depicted in Figure 2. At least the input environment and the expected process history must be given. If we make the assumption that the error was caused during the last process call, the information we want is the input environment and current signal history after the call.

The input environment (including signal values) can be automatically obtained from the simulation environment. However, it requires a greater level of detail than is provided by the standard waveform traces, which list changes at time points, but not the causal relationships between the actual signal events.[1]. However, commercial VHDL simulators also provide a monitoring trace of signal changes over time from which the values can be derived. This trace is rarely used in practice since it contains too much information to be routinely digestible by a user, and is not needed for displaying the waveform traces. Figure 4 shows the monitoring result for our example, allows to obtain the relevant signal and variable values within the *COUNTER* process. Only the events marked with arrows on the left are relevant for the example. The signal *B_DATA* exhibits a discrepancy (1). *B_DATA* is functionally dependent on *C_DATA* and *C_DATA* is dependent on the set { *INTDATA*, *CLK*, *RESET* } (2). Their values can be directly obtained from the monitor output.

In addition, we need an oracle (the user, or a separate specification trace containing the same relevant set of signals) .

In our case, the expected specification data are

	Specified Values
Input Env.	Env1 = {CLK = 1, RESET = 0, C_DATA = 0, INTDATA = 0}
Final History	Hist5 = { C_DATA(0) = 1 }

[1]Functional dependencies used for debugging do not handle delay time and other temporal aspects.

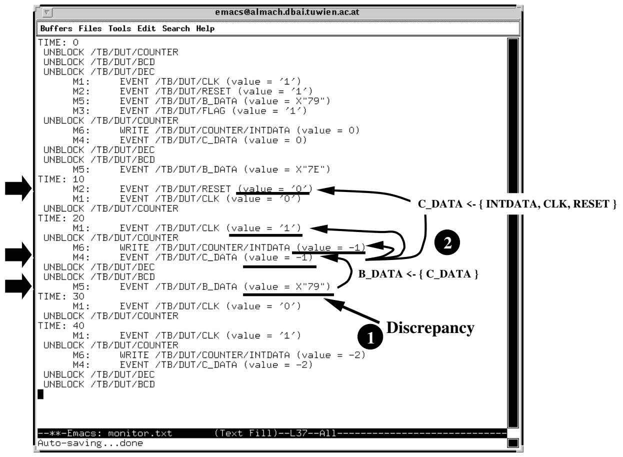


Figure 4: Monitor output for the VHDL example program

whereas the actual observed environment and history (at time 20ms in the example) were given above in Section .

For illustration purposes, let us assume diagnosis starts by assuming the correctness of all components. Because of the specified input values (*Env1*) *INTDATA = -1* is element of environment *Env3*. Since *INTDATA* is less than 9 all elements of *Env3* are propagated to *Env4*. Propagation continues finally leading to *INTDATA = -1* as an element of *Env7*. Because of the behavior of the signal assignment *C_DATA <= INTDATA* the value *C_DATA(0) = -1* is added to the history port *Hist5*, leading to a contradiction and a non-minimal conflict. From here on the diagnosis process continues, for example using the standard construction of the HS-DAG (Reiter 1987). Ultimately, we finally receive the set of most preferred diagnoses:

$\{\{$ *ab(INTDATA:=INTDATA-1* $\}$*, ab(C_DATA<=INTDATA)*$\}\}$

with the following diagnoses being removed because of subsumption:

$\{\{$*ab(if INTDATA>9 then. . .)*$\}$,$\{$*ab(if CLK='1' then. . .)*$\}$,
$\{$*ab(if RESET='1' then . . .)*$\}\}$

Examining the two final diagnoses, we find that both fault modes are equally weak, but the first diagnosis, $\{\{$ ab(INTDATA := INTDATA - 1 $\}\}$, is the most specific one. Since both are abnormal, then the assignment (which is the actual source of the error in the example since '-' should be '+') is preferred.

Discussion

If we compare the representation used in this paper to other work such as the dependency-based representation of (Friedrich, Stumptner, & Wotawa 1996), we find the following differences. First, the value-level description allows the analysis of VHDL programs inside processes, statements, and resolution functions. In addition, the value-level representation makes it possible to define more specific fault modes such as the $wCond$ mode, and allows for the incorporation of a replacement-based approach as in (Stumptner & Wotawa 1996a) (although we do not discuss it in this paper). The same principle can be applied to the diagnosis of individual expressions occurring in assignments or other statements, by assigning fault modes to expressions. However, this will certainly require a hierarchical approach due to the increased problem size if individual expressions are included. The size of individual processes typically does not exceed several hundred expressions and statements. This is small enough to make diagnosis easy to compute if the number of candidate processes can be restricted using an abstract model (see (Friedrich, Stumptner, & Wotawa 1996)). In comparison to (Stumptner & Wotawa 1996b), we cover a larger part of the VHDL semantics (sufficient to diagnose RT level programs).

The diagnoses computed by the system will be complete (i.e., the set of diagnoses returned contains the actual incorrect statements) with two important exceptions, both based on the notion of structural faults. First, if a directly observed signal (or variable) is incorrect because it is not altered by any statement, no diagnosis will be computed (because there is no component in the program that is directly responsible for the incorrect value). If, on the other hand, other assignments to the signal (or variable) exist, or the signal (variable) is not directly observed, the set of diagnoses will contain other diagnoses instead of the intended one, similar to bridge faults in hardware diagnosis. To solve this kind of faults, a separate model for locating possible "bridges" must be present (Böttcher 1995), such as provided for example by the replacement approach (Stumptner & Wotawa 1996a). This is beyond the scope of this paper.

The model-based approach to software debugging, compared with more traditional ones, such as a search for errors using program traces exclusively, shows improved results in many cases. In this context, a *trace diagnosis* is a program statement executed during program evaluation. For our example. a trace diagnosis approach would provide 5 single diagnoses $\{$ ab(INTDATA := INTDATA - 1 $\}$, $\{$ ab(C_DATA <= INTDATA) $\}$, $\{$ ab(if INTDATA > 9 then . . .) $\}$, $\{$ ab(if CLK = '1' then . . .) $\}$, $\{$ ab(if RESET = '1' then . . .) $\}$. For another example, consider the following program:

S1 := **not**(I1);
S2 := **not**(I2);
OS <= **and**(S1,S2);

The input environment $\{$ I1 = 1, I2 = 0 $\}$ and the expected output history $\{$ OS(0) = 1 $\}$ lead to a contradiction. Using value-level diagnosis (VLD) we obtain two single diagnoses $\{$OS <= **and**(S1,S2);$\}$ and $\{$ S1 := **not**(I1); $\}$. A program trace diagnosis would return all three statements as single fault diagnoses.

Lemma 1 *Let* Π *be a VHDL process, and* tc *a test case. The set of diagnoses using a program trace* $diags_{TRACE}(\Pi, tc)$ *is a superset of the set of single* *fault diagnoses* $diags_{VLD}(\Pi, tc)$ *obtained by using model-based diagnosis techniques described in this paper.*

$$diags_{TRACE}(\Pi, tc) \supseteq \{x | x \in diags_{VLD}(\Pi, tc) \wedge |x| = 1\}$$

Status and Outlook

The improved VHDLDIAG implementation based on (Reiter 1987) including the value-level system description is currently being tested and extended since not all VHDL operators and statement types are yet supported. The diagnosis run for the example lasts about 1 second with the current, unoptimized implementation. The full implementation (VHDLDIAG+) includes models for all VHDL operators used in RTL programs.

While the system provides more detailed analysis than the original VHDLDIAG system (Friedrich, Stumptner, & Wotawa 1996), there are still some restrictions. While the example above shows how it is possible to define iterative effects via the interaction of parallel processes (in fact, that is how such effects are ultimately expressed in the hardware resulting from the VHDL design) , explicit sequential loop statements, i.e., iterations *inside* processes, are not handled. Second, more rigorous analysis of the temporal aspects of VHDL programs. Third, an exact abstraction mapping between the new representation and the FD-based representation of VHDLDIAG has to be defined, to enable us to use the abstract representation in an initial diagnosis stage, and then use the results from that stage for focusing on the detailed workings of individual candidate processes. Finally, the incorporation of repair actions will have to be studied as it may provide a more effective ranking of diagnoses. As we have pointed out in (Stumptner & Wotawa 1996a), repair actions, expressed as replacement fault modes, are both effective in controlling search and offer the only possibility to provide good discrimination in cases involving what corresponds, in software debugging, to "bridge faults" in hardware diagnosis, e.g., misnamed variables or functions.

In summary, this paper continues our work on the automated diagnosis of VHDL programs. Compared to previous work that concentrated on diagnosing logic programs, this area provides additional challenges as there is no immediate mapping from the program semantics to a logic-based system description. Compared to the system description of (Friedrich, Stumptner, & Wotawa 1996), the representation shown in this paper analyses the sequential statements in the bodies of VHDL processes down to the level of individual terms. Therefore, a much more detailed analysis of the potential error sources found by a diagnostic tool is made possible. We have presented the mapping from unannotated VHDL source code to the system description, and discusses the heuristics needed for ranking multiple diagnosis candidates. It should be noted again that the representation of (Friedrich, Stumptner, & Wotawa 1996) could not have discriminated between any of the diagnosis candidates in the example, as it considered exclusively parallel signal assignments without analyzing the inner, sequential semantics of VHDL processes. Together with the original representation, the goal is to let this view provide the additional expressiveness that is needed for better discrimination between diagnoses, while using the abstract representation for initial filtering that provides the required efficiency for large, real-world systems.

Acknowledgement

This work was partially supported by Siemens Austria under project grant DDV GR 21/96106/4. The authors would like to thank Thomas Havelka for his work on the current implementation of VHDLDIAG.

References

Bond, G. W. 1994. *Logic Programs for Consistency-Based Diagnosis*. Ph.D. Dissertation, Carleton University, Faculty of Engineering, Ottawa, Canada.

Bond, G. W. 1996. Top-down consistency based diagnosis. In *Proceedings on the Seventh International Workshop on Principles of Diagnosis*.

Böttcher, C. 1995. No faults in structure? How to diagnose hidden interaction. In *Proc. IJCAI*.

Console, L.; Friedrich, G.; and Dupré, D. T. 1993. Model-based diagnosis meets error diagnosis in logic programs. In *Proc. IJCAI*, 1494–1499. Chambery: Morgan Kaufmann Publishers, Inc.

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

de Kleer, J.; Mackworth, A. K.; and Reiter, R. 1990. Characterizing diagnoses. In *Proceedings AAAI*, 324–330. Boston: Morgan Kaufmann Publishers, Inc.

Friedrich, G.; Stumptner, M.; and Wotawa, F. 1996. Model-based diagnosis of hardware designs. In *Proc. ECAI*.

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.

Stumptner, M., and Wotawa, F. 1996a. A model-based approach to software debugging. In *Proceedings on the Seventh International Workshop on Principles of Diagnosis*.

Stumptner, M., and Wotawa, F. 1996b. Model-based diagnosis of hardware description languages. In *Proc. Computational Engineering in Systems Applications CESA'96 IMACS Multiconference, Invited Session on Model-Based Systems*.