

Software Methodologies for VHDL Code Static Analysis based on Flow Graphs

Luciano Baresi, Cristiana Bolchini, and Donatella Sciuto
Dipartimento di Elettronica e Informazione
Politecnico di Milano
P.zza L. Da Vinci, 32 - 20133 Milano, Italy
[baresi|bolchini|sciuto]@elet.polimi.it

Abstract

At a high level of abstraction, the VHDL specification of the functionalities that a circuit shall perform is given by defining the behavioral model. The similarity with procedural programming languages suggested to tailor some software analysis techniques to VHDL behavioral description analysis. The paper presents several analyses of the code, based on data flows, aimed at identifying significant properties of the final circuit from the synthesis and testability points of view.

1. Introduction

VHDL is now the most widespread language for building software models of hardware systems. However, although the complexity of the described models is rapidly increasing, no support is given to the designer to evaluate the quality of its code. Such a problem has been already analyzed by the software engineering research area and different techniques for evaluating concurrent and sequential software specifications have been proposed.

The aim of this paper is to consider static analysis techniques to identify significant properties of the implementation, from the behavioral VHDL description. The information collected from the code analysis is very important for (a) discovering and correcting possible errors, inconsistencies, and incompletenesses, without affecting the following design phases; (b) deriving test patterns, that will be used to validate the circuit during simulation; (c) identifying code fragments, that are known to be critical with respect to synthesis; and (d) predicting the outputs of a synthesis tool, i.e., how a synthesized circuit should look like.

Approaches to VHDL analysis for quality evaluation have been already published in literature [1, 5, 11, 8]. New commercial tools mainly dealing with code coverage are already available [15, 17]. More sophisticated analyses for synthesizability, simulation, complexity evaluation and testability are under study in the EEC supported OMI Project REQUEST.

This paper presents the first results of this project by proposing a data flow based methodology to statically extract information from the VHDL specification, which is interesting both for testability analysis and for synthesis purposes.

The syntactic similarity between behavioral VHDL and procedural programming languages suggested to try to apply software analysis techniques to investigate VHDL code. The work describes an approach for identifying deadlock conditions within VHDL specifications. Hints and ideas have been taken from reachability analysis [13] and symbolic execution [7]. While proposing this technique, we adapted the concept of deadlock to the hardware domain. Many proposals to uncover deadlock conditions in concurrent software systems use the dining philosophers problem to validate their approach. This is why a VHDL model of the problem has been the starting point of the work. Besides this, several techniques based on data flow analysis are presented to highlight useful properties of VHDL behavioral descriptions.

The rest of this paper is organized as follows. Section 2 proposes a VHDL model of the dining philosophers problem, pointing out its usefulness as far as VHDL and hardware systems are concerned. Moreover it informally figures out an analysis technique to discover potential deadlocks. Static analysis techniques, all based on some annotations of the flow graph of the VHDL code under analysis, are described in Section 3. Finally, Section 4 draws some conclusions.

2. A VHDL Model of the Dining Philosophers

Dijkstra's *Dining Philosophers problem* [6] was chosen to study deadlocks in VHDL code and their impact on the resulting circuits. The problem can be formulated as follows:

Five philosophers are seated around a table. Between each philosopher there is a single fork. The life of a philosopher consists of periods of eating and thinking. When a philosopher gets hungry, he tries to get his left and right fork, one at a time, in either order. If successful in acquiring the two forks, he eats for a while, then puts down the forks and thinks.

A problem can arise if each philosopher simultaneously grabs his left (right) fork and then waits for his right (left) fork. Since right (left) forks are not available, all philosophers starve and a deadlock occurs.

This problem can be modeled in VHDL by exploiting the features of the language. *Processes* may be used to model the philosophers whereas *signals* are suited for modeling the forks. Main role for the characterization of the problem is played by the VHDL synchronization mechanism.

Communications in VHDL are asynchronous, based on events on signals. As to model execution, during the start-up phase, all processes are executed either till the end or till they are suspended waiting for some events to occur. After that, processes are executed each time there is a change on the signals they are sensible to. The outputs always remain available, and there is no run-time supervisor: all the processes are executed in parallel.

It is meaningful to study whether a modeled system can enter deadlock situations. Quoting [16], deadlock can be defined formally as follows:

A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause.

Because all processes are waiting, none of them will ever cause any event, and all the processes remain blocked. This definition applies directly to hardware systems.

The differences, with respect to software systems, are:

- a closed system, such as our model of the dining philosophers problem, is either always or never deadlocked. There is no run-time supervisor that defines the actual scheduling among processes. If a system can enter a deadlock condition, this is not a possibility, but a certainty.
- The dynamics of an open system is not hard-coded within the specification of the system itself. It depends on the arrival of external signals. This indeterminism turns the certainty of deadlocks into the possibility of having them.
- An event is a change on a signal. A process can either be suspended waiting for an event or need an event to change its outputs. In both cases, events are necessary to allow a circuit to evolve.
- A blocked circuit does not stop working, but it evolves through dummy executions. This condition is easily detectable by an external observer who sees always the same output values.

In case of closed systems, then, the first and the fourth points imply that a VHDL behavioral description, with deadlocks, should be synthesized by a set of constant values, instead of a real logic circuit. In the case of open systems, the absence of a reset signal may become a critical point if the code is characterized by the possible presence of deadlocks. In fact, the reset signal would provide the means for exiting the deadlock situation, placing the circuit in its initial state.

Now, the problem is how deadlocks can be uncovered in hardware circuits and, even better, in VHDL behavioral specifications. The aforementioned condition is easily detectable by simulation, but, to the best of our knowledge, not by the analysis of designed models.

This is the reason why we defined a VHDL model of the dining philosophers problem. Each resource is modeled by a separate process. Thus, both philosophers and forks are coded as processes. So doing, we can say that we do not actually model a fork, but a sort of dispatcher that oversees resource allocation.

Characteristic of the modeled dining philosophers problem is the fact that all philosophers behave in the same way and the non-determinism has been eliminated from the model.

Looking at the code [2], not included here due to space considerations, we noticed that:

- two distinct signals (FORKNL and FORKNR) are used to represent the request for acquiring a fork: one for each adjacent philosopher. This solution has been adopted not to have multiple-driven signals, constituting not synthesizable code.
- The actual availability of a fork is represented by a boolean signal (FORKN).
- All the philosophers are controlled by the same external clock, i.e., they all obey the same timing. The adoption of more than one clock would have led to not synthesizable models for commercial logic synthesis tools [15, 12].
- It is supposed that all the philosophers try to start eating at the same time. This is the easiest choice to have a deadlock. Moreover it does not require the use of external signals to control the philosophers, i.e., we have a closed system.

Note that, the approach here adopted is not restricted to closed systems only, but it is suited for open systems too.

The simulation of the designed model highlighted what expected: outputs are actually constants in the time domain, after a first initialization phase. On the contrary, the synthesis, using commercial logic synthesis tools [15, 12] did not produce simple constants (including set logic), but an unexpected logical circuit. This is due to the fact that having defined different processes, different finite state machines are synthesized, sharing the logic but not the states. Hence the tool is not able to identify the possible occurrence of a deadlock and the correspondence to a constant value.

Therefore, deadlock analysis can be effective to evaluate the quality of the design with respect to the specific problem.

2.1. Deadlock

A first step to detect deadlocks is the analysis of the information that flows among the processes involved in the communication. To do this, we define the *information flow graph* of a VHDL model. An *information flow graph* is a directed graph, where nodes represent processes, and edges information flow. There is an edge, labeled with v , between a process S and a process T , if T reads the variable v updated by S .

A trivial condition that could lead to deadlocks is then the absence of loops in the flow graph. A process (node) cannot recompute its outputs as a consequence of external events. On the contrary, when information flows around a loop, processes can be fed with new inputs and thus produce new outputs. It is true, however, that clocked processes could modify their inputs due to internal events only. For instance, an output signal could be incremented at each clock sample, even if the inputs do not change.

Information flow graphs do not often provide definitive solutions, but they are easy to build and can be regarded as the first step in analyzing a VHDL model. According to this, we defined the *information flow graph* (Figure 1) for the philosophers problem. For the sake of simplicity, we consider two philosophers only. In this

graph, processes controlled by a sensitivity list are highlighted by a circle around the node. Moreover signals are divided into signals that appear into sensitivity lists, depicted by dashed lines, and the others, represented by solid lines.

It can be immediately noticed that the condition on information flow does not hold: each process can be fed with new inputs. It is true, however, that the two philosophers wait for the same two signals.

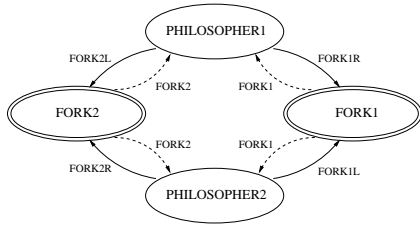


Figure 1. Philosophers: information flow graph

This fact suggests to extend the analysis by considering the dependencies among the events necessary to each process. In other words, we define a *dependencies graph*. Nodes are the signals that appear in an *information flow graph*, and edges state eventual dependencies among signals. An edge between a node s and a node t means that there is a change on s only if there is a change on t . The *dependencies graph* for the philosophers problem is shown in Figure 2. It can be noticed that, after the start-up phase, there is an event on FORK1R, if there is an event on FORK1. Unfortunately, this relation holds also in the other way around: an event on FORK1 requires an event on either FORK1R or FORK1L. The same circularities exist also among signals FORK2L, FORK2R, and FORK2. This means that the whole system has entered a circular wait condition, i.e., a deadlock condition.

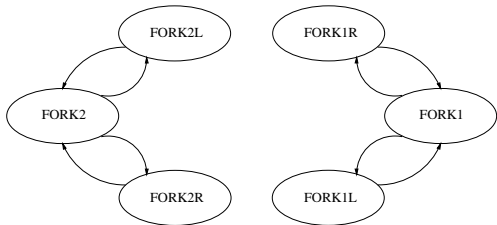


Figure 2. Philosophers: dependencies graph

The previous methods can also suggest partial simplifications in synthesizing a model. The conditions could hold for a subset of the involved processes. Simplifications would apply to those processes only, instead of to the whole model.

The use of boolean signals for regulating both the request and the acquisition of a fork, i.e., the presence of semaphores both in philosopher processes and in fork processes, imposes a rendezvous-like communication model. This means we can define how the model evolves in its execution space, borrowing some ideas from reachability analysis [13].

Looking at how signals are used and at the *information flow graph* of Figure 1, we can define four high-level functions:

set request: a process P probes the variable v and sends a request to set v .

reset request: a process P sends a request to reset v .

set: a process P sets the variable v .

reset: a process P resets the variable v .

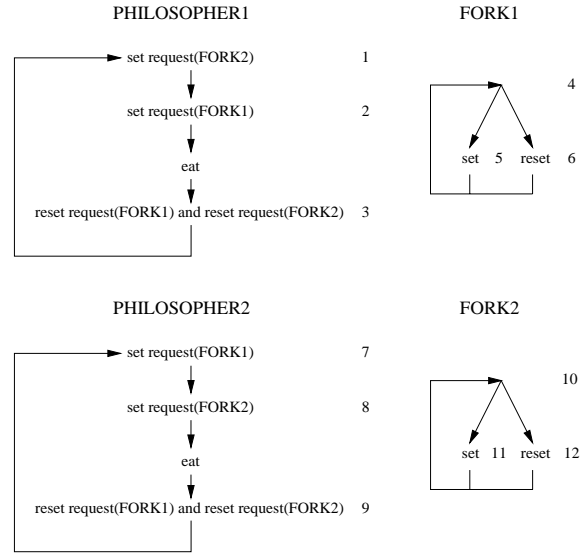


Figure 3. Philosophers: graph representations.

The identification of these macro-instructions allows us to extract from the processes, two philosophers and two forks, a graph representation, shown in Figure 3, that resembles *flowgraphs* (the numbers by the graph represent the labels of the states used in the execution space). According to the VHDL model we defined [2], the graph for a philosopher process translates the finite state machine coded by the case statement. In the same way, the graph of a fork represents the two alternatives offered by an `if` statement. In both cases the loop is required by the dynamic semantics of VHDL.

These four graphs help us in building the execution space, Figure 4, of the model. VHDL processes execute and, if possible, change their state all in parallel. As expected, it is not possible to leave the state where each philosopher tries to get the second fork. The dashed line indicates that the whole system is not blocked, but it goes on executing in the same state.



Figure 4. Philosophers: execution space

When an open system is taken into consideration, Figure 2 is modified by adding new ingoing edges in the nodes (signals) sensible to external events. In this way we model the sensitivity of the processes to unpredictable signals. These signals do not modify the basic interdependencies between the philosophers and the forks, but they introduce undeterminism in model's dynamics. This unpredictability makes the proposed analysis highlight the possibility, instead of the certainty, of deadlocks.

3. Flow Graph based Static Analyses

After having examined the communication among processes focusing the attention on the problem of possible deadlocks, let us now take into account the sequential programming language aspects of VHDL. A variety of different methods have been successfully proposed and applied for analyzing sequential software programs [9]. This section aims at concentrating on static analysis techniques, that, even if not as powerful as dynamic analysis, offer valuable results and are easier to apply.

All the techniques presented in the next sections are based on appropriate annotations on the nodes of a graph. As already done in Section 2, VHDL code is translated into a graph-like representation, called *flow graph*. A *flow graph* [9], basically a directed graph, is composed of nodes, representing program statements¹, and edges, defining execution flow. Notice that, for readers familiar with the LEDA tool [10], the representation proposed here is very similar to the graphs built by the tool.

The obtained graph is then decorated with the information needed for the particular analysis. Section 3.1 takes into account delays associated with variables². The operations performed on each variable are used in Section 3.2 to uncover inferred latches. Finally “transparent” variables, are addressed in Section 3.3.

3.1. Timings

A first analysis that can be done on the *flow graph* of a VHDL process concerns the delays associated with variables, i.e., which is the actual effect of `after` statements on variables propagation.

The analysis can help in studying problems related to controllability and observability for testability verification [4]. We can statically decide whether all the combinations of the possible values are observable on process inputs. Different delays could actually forbid some combinations.

Let us define a *path* as a directed path from the entry node to the terminal node of the *flow graph*. The presence of loops within a *flow graph* leads to, at least from a theoretical point of view, an infinite set of *paths*. Fortunately, the problem can be overcome by exploiting both the peculiarities of VHDL and the specific needs of the current analysis. To have a specification that could be synthesized, loops must be upperbounded. The aforementioned assumption would definitively solve the problem. Moreover, to have a synthesizable specification, delays cannot be put within loops. This means that, as far as current analysis is concerned, loops can be discarded and the graph becomes a directed acyclic graph (DAG).

Graph nodes, in which variables are assigned, are annotated with a pair $\langle N, LD \rangle$. N is the variable name and LD is the local delay. LD is 0 if the variable is not delayed, i.e., if the statement `after` does not follow the assignment. After that, all the *paths*, within the modified *flow graph*, are searched. For each *path*, delays associated with each variable are summed, defining new pairs $\langle N, PD \rangle$. N is still the variable name and PD is the delay on the *path*. At the end, for each variable, the minimum

¹A “statement” can be both a single actual statement and a whole process invocation, considered as a macro-statement.

²A variable indicates both an actual variable and a signal.

```
MUX: PROCESS(I0, I1, I2, I3, A, B)
  VARIABLE muxval: INTEGER;

BEGIN
  muxval := 0;
  IF (A = '1') THEN
    muxval := muxval + 1;
  END IF;

  IF (B = '1') THEN
    muxval := muxval + 2;
  END IF;

  CASE muxval IS
    WHEN 0 => Q <= I0 AFTER 10 ns;
    WHEN 1 => Q <= I1 AFTER 10 ns;
    WHEN 2 => Q <= I2 AFTER 10 ns;
    WHEN 3 => Q <= I3 AFTER 10 ns;
    WHEN OTHERS => NULL;
  END CASE;
END PROCESS MUX;
```

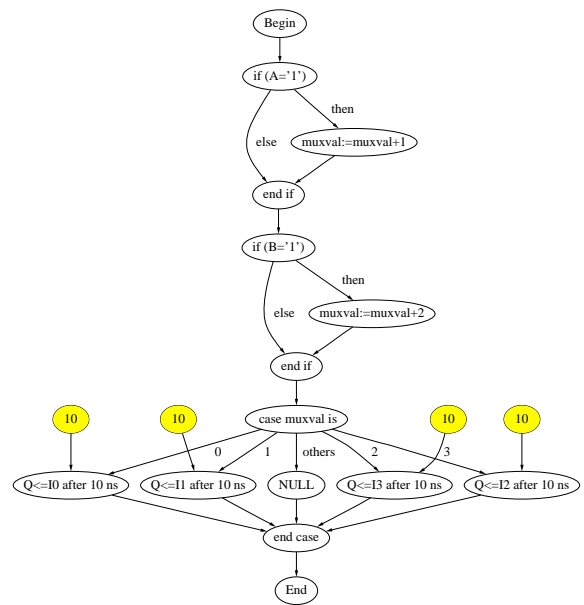


Figure 5. An example of Timings analysis

and maximum values of PD define its delay domain. If the two values are 0, the variable is actually not delayed.

The analysis of all leaf processes is the precondition to be able to reason on a set of related processes. In this case, the *flow graph* contains nodes that are whole processes. These nodes are not associated with a single pair, but with a set of pairs: the results of the analysis on the single process. The set comprises a tuple $\langle N, LDmin, LDmax \rangle$ for each significant variable. Again, N is the variable name, $LDmin$ and $LDmax$ are the minimum and maximum delay evaluated for the variable. After that, the analysis goes on in almost the same way.

Figure 5 presents the VHDL code of a multiplexer. In this toy example, it can be noticed at a first glance that output Q is always deferred of 10 nsecs. The proposed analysis provides the same information. If we consider annotated nodes only, i.e., the ones in which variable Q is assigned, we identify four paths. In all cases

```

ENTITY state_machine IS
  PORT (clk: in std_logic;
        x: in std_logic;
        z: out std_logic);
END state_machine;

ARCHITECTURE arch OF state_machine IS
  TYPE states IS (S0, S1);
  SIGNAL state: states;
BEGIN
  PROCESS (clk)
  BEGIN
    IF (clk='1' and clk'event) THEN
      CASE state IS
        WHEN S0 => IF (x='1') THEN
                     state<= S1;
                     z<='1';
                   ELSE
                     z<='0';
                   END IF;
        WHEN S1 => IF (x='1') THEN
                     state<= S0;
                     z<='0';
                   ELSE
                     z<='1';
                   END IF;
      END CASE;
    END IF;
  END PROCESS;
end arch;

```

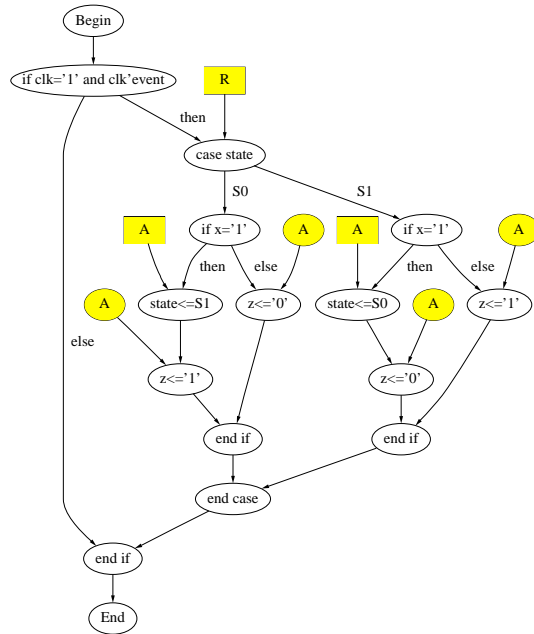


Figure 6. Implicit Memory Elements

the corresponding pair is $\langle Q, 10 \rangle$. Hence Q is always deferred of 10 nsecs.

3.2. Implicit Memory Elements

This section sketches a way to find out implicit memory elements based on pattern searching in regular expressions. The starting point is again the *flow graph* of a VHDL process, as defined in Section 3. We are not interested in searching all the *paths*. For each variable, we define a regular expression summing up the significant actions performed on the variable itself. Notice that, in this case, loops in *flow graphs* are not a problem. Remembering regular expression theory, they are simply translated by means of Kleene stars (*).

According to [3], given a VHDL process, a variable requires an implicit memory element each time (a) it is read before being assigned; (b) it is assigned before a `wait` statement; (c) it is assigned in a clocked process; and (d) it is not assigned in all conditional branches.

This means that, as to this analysis, we need to take into account read, write (assignment), and wait operations. These actions correspond to a *r*, an *a*, and a *w*, respectively, within regular expressions. Thus *ar* means that the variable has been assigned a value and the value has been subsequently read. *w(r|a)* states that, after the `wait` statement the variable is either read or assigned.

It should be clear now, that looking for implicit memory elements corresponds to looking for strings within regular expressions. The first two conditions above are easily translated by strings *ra* and *aw*. The third case means looking for *a*, but it applies to clocked processes only. Finally, the last rule states that, looking at possible alternatives, if one is an assignment, then all the alternatives must be assignments. Hence, the alternatives can

be actually reduced to a single possibility. When the condition does not hold, an implicit memory element is needed.

Consider, as an example, the VHDL code of Figure 6. It models a Moore machine by defining a process with a sensitivity list. No reset signals are used. Designer's experience can infer that a flip-flop for the variable *state* and one for the output *z* are needed. These memory elements can be uncovered using the method described so far.

Looking at the corresponding *flow graph*, shown in Figure 6, the regular expression for the variable *state* is $r(a|a) = ra$. It straightly matches the first condition. Moreover, the expression for the output *z* is $(a|a) | (a|a) = a$. Since the example refers to a clocked process, an implicit memory element is inferred by the third rule.

Due to the nature of the analysis, it must be applied on single processes only. It would be meaningless trying to extend the analysis in a hierarchical fashion, as proposed for timings in Section 3.1.

3.3. Transparency

Strictly speaking, a variable remains transparent to a process if it is not used within the process itself. This definition would imply that transparent variables would uncover only warnings or inconsistencies in a given specification. A more useful definition states that a transparent variable is a variable that is only read and re-assigned to another variable, without being logically or arithmetically manipulated.

If a variable is transparent for a process, it can be used to propagate test patterns through the process itself. Being able to statically identify which variables can be propagated through

```

MUX_PR: PROCESS(A, B, sel)
BEGIN
  IF (sel='0') THEN
    Z<=A;
  ELSE
    Z<=B;
  END IF;
END PROCESS MUX_PR;

MUX_PR: PROCESS(A, B, sel)
BEGIN
  IF (sel='0') THEN
    Z<=A+B;
  ELSE
    Z<=A-B;
  END IF;
END PROCESS MUX_PR;

```

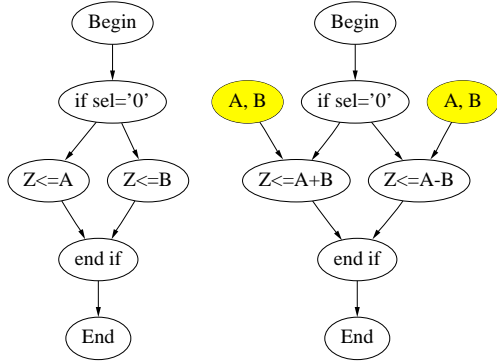


Figure 7. Transparent variables

which processes provides more control on test patterns and thus is useful in testing hardware circuits.

Once more, let us consider the *flow graph*. Each node is associated with a set V of variables for which the given definition of transparency does not hold anymore due to the statement in the node. Moreover, we define another set TV (Transparent Variables): it will contain the variables still transparent after each step.

At the beginning, TV contains all the variables of the process. After that, visiting each *flow graph* node, variables in V are subtracted from TV . At the end, i.e. after having visited all the nodes exactly once, TV contains the variables that are actually transparent.

In this case too, we can extend the *flow graph* to cope with a set of related processes. As in Section 3.1, a node corresponds to a whole process. Its set V is simply the difference between all the variables and the set TV of the invoked process.

As an example, consider the two fragments of VHDL code of Figure 7 and their associated *flow graphs*. In the first case, the set V of each node is empty. The trivial application of the proposed analysis highlights that both variable A and variable B are transparent. On the contrary, in the second case, it can be proved that the two variables are not transparent. They both are used in arithmetic computations.

4. Conclusions and Future Work

The paper presents a first overview for examining interrelations between the hardware and software domains, and provided a first basis for further proposals.

The techniques in this paper gave encouraging results as to the case studies used to validate them. In particular, in Section 2 we adapted to VHDL domain the concept of deadlock and presented a way to discover deadlock conditions within VHDL models. Future work will try to generalize and formalize the proposals. Moreover,

other, and maybe more significant, case studies will be taken into account to improve our confidence and to better tailor the proposed methods.

The work revealed also the two different finalities and interpretations of testing a software program and a VHDL behavioral specification. In the former case, testing is used to validate produced code, i.e., to try to correct possible errors in the implementation. In the latter case, testing is not only employed to uncover problems, lacks or inconsistencies, but also to generate test patterns which will be used during circuit validation [14].

References

- [1] A. Balboni, M. Mastretti, and M. Stefanoni. Static Analysis of VHDL Model Evaluation. In *Proceedings of Euro-VHDL*, pages 586–591, 1994.
- [2] L. Baresi. Software Methodologies in VHDL Code Analysis. Technical Report 96.010, Politecnico di Milano - Dipartimento di Elettronica e Informazione, February 1996.
- [3] J. Bergé, A. Fonkoua, S. Maginot, and J. Rouillard. *VHDL Designer's Reference*. Kluwer Academic Publishers, 1992.
- [4] M. Bombana, G. Buonanno, P. Cavalloro, F. Ferrandi, D. Sciuto, and G. Zaza. ALADIN: A Multi-Level Testability Analyzer for VLSI System Design. *IEEE Transactions on VLSI Systems*, 2(2):157–171, 1994.
- [5] S. Carlson and E. Girczyc. Increasing Design Quality and Engineering Productivity through Design Reuse. In *Proceedings of 30th Design Automation Conference*, 1993.
- [6] E. Dijkstra. *Co-operating Sequential Processes*. Academic Press, 1965.
- [7] C. Ghezzi, D. Mandrioli, S. Morasca, and M. Pezzè. Symbolic Execution of Concurrent Systems using Petri Nets. *Computer Languages*, 14(4):263–281, 1989.
- [8] X. Gu, K. Kuchcinski, and Z. Peng. Testability Analysis and Improvement from VHDL Behavioral Specifications. In *Proceedings of EURO-VHDL '94*, pages 644–649, 1994.
- [9] J. Kam and J. Ullman. Global Data Flow Analysis and Iterative Algorithms. *Journal of the ACM*, 23:158–171, 1976.
- [10] LEDA S.A. *LEDA VHDL System - User's Manual*, 1993. version 3.2.0.
- [11] O. Levia. Writing High Performance VHDL Models. In *Proceedings of Euro-VHDL*, 1991.
- [12] Mentor Graphics. *Autologic VHDL Reference Manual*, 1993. version 8.2.
- [13] M. Pezzè, R. Taylor, and M. Young. Graph Models for Reachability Analysis of Concurrent Programs. *ACM Transactions on Software Engineering and Methodology*, 4(2):171–213, 1995.
- [14] J. Santucci, A. Courbis, and N. Giambiasi. Behavioral Testing of Digital Circuits. *Journal of Microelectronic Systems Integration*, 1(1):55–77, 1993.
- [15] Synopsys. *Synopsys User's Manual*, 1994.
- [16] A. Tanenbaum. *Modern Operating Systems*. Prentice-Hall International Editions, 1992.
- [17] Veda. *VHDL Cover User's Manual*.