Synthesis of VHDL Concurrent Processes

Petru Eles* Krzysztof Kuchcinski† Zebo Peng† Marius Minea* ‡

* Computer Science and Engineering Department
Technical University of Timisoara
Romania

† Dept. of Computer and Information Science
Linköping University
Sweden

Abstract

This paper presents two methods for synthesis of VHDL specifications containing concurrent processes. Our main objective is to preserve simulation/synthesis correspondence during high-level synthesis and to produce hardware that operates with a high degree of parallelism. The first method supports an unrestricted use of signals and wait statements and synthesizes synchronous hardware with global control of process synchronization for signal update. The second method allows hardware synthesis without the strict synchronization imposed by the VHDL simulation cycle. Experimental results have shown that the proposed methods are efficient for a wide spectrum of digital systems.

1. Introduction

This paper addresses the problem of high-level synthesis from a behavioral VHDL description that contains interacting concurrent processes. Our goal is to conform to the VHDL standard semantics during synthesis and to produce hardware that operates with a high degree of parallelism. One of the most difficult issues in this context originates from the VHDL semantics of signal assignments and wait statements which is specified in terms of simulation. As stated in the language definition [8], unlike variables which are updated as soon as they are assigned a value, signals are only updated at the end of a simulation cycle. This means that the update of signal values must be synchronized with the execution of a wait statement by *every* process in the system and has to be performed simultaneously for all signals that change their values in that simulation cycle.

The synthesis strategies we present in this paper preserve a partial ordering relation of operations on signals and ports from the simulation model to the synthesized hardware structure. Thus, we are achieving *simulation/synthesis correspondence* which means that both the simulation model and the synthesized hardware react with the same values (sequences of values) of the signals and ports to identical sequences of stimuli applied at the inputs.

Most of the high-level synthesis systems which accept VHDL specifications restrict themselves to a virtually sequential subset of the language with a very limited use of signals due to some of the problems discussed above [3]. According to SynVHDL [13] and VSYNTH [7] an architecture body may only contain a single process. Silicon 1076 [10] restricts the use of signal assignments to output ports and requires the design to contain only one process described at the architectural level. In CALLAS [2] the designer is required to use an explicit global clock signal. The entire behavior has to be described only in terms of variables, and the use of signals is virtually limited to input and output ports. In the context of restrictions like these, hardware implementation of standard VHDL semantics for process interaction through signals can be avoided.

On the other hand papers dedicated to questions concerning the synthesis of signals (such as [14] and [9]), do not address the implications for synthesis of signal assignment semantics as it is defined, in terms of the simulation cycle, by the VHDL standard. DSS [15] supports the synthesis of interacting VHDL processes to a synchronous hardware of strongly coupled FSMs with lockstep execution of processes. The synthesis system HIS [1], designed at IBM, imposes several restrictions on the VHDL specification style. It restricts the use of signals to the explicit clock signal in the so-called sequential synchronous model. The system accepts only very strictly defined description styles: sequential synchronous model, explicit state machine model, and a dataflow description using only concurrent signal assignments.

The main objective of our approach is to preserve the computational effects of the simulation cycle with minimal additional costs and minimal impact on the performance of the synthesized hardware. To achieve this goal we have developed a method for compiling VHDL into an internal design representation which explicitly captures its essential semantics with respect to process synchronization. The developed compiler automatically generates synthesis

‡ at present: School of Computer Science, Carnegie Mellon University

This work has been partially sponsored by the Swedish National Board for Industrial and Technical Development (NUTEK)

© 1994 ACM 0-89791-685-9/94/0011 $1.50

structures which are later transformed by high-level synthesis algorithms. Our approach supports two different solutions, one with unrestricted use of signals and wait statements, and the other with reduced synchronization between processes. Which solution will be applied for a certain synthesis task depends on the description style adopted by the user. Selection of the description style is decided according to the features of the designed hardware. The proposed solutions have been implemented, and tested with the CAMAD high-level synthesis system [12].

This paper is divided into 6 sections. Section 2 gives a short introduction to the basic design environment for high-level synthesis and our internal design representation. Sections 3 and 4 describe the unrestricted and reduced-synchronization specification styles respectively and present the corresponding features of the synthesized hardware. In section 5 we discuss some synthesis results obtained with CAMAD. Section 6 presents our conclusions.

2. The Design Environment

The VHDL related design environment comprises currently the S'VHDL compiler and several high-level synthesis algorithms. S'VHDL consists of a large subset of standard VHDL [4], and accepts specifications consisting of interacting concurrent processes. The compiler translates S'VHDL descriptions into an internal design representation which can be later synthesized. It provides several options to allow generation of different representation formats as well as acceptance of different description styles. This is used to make the synthesis process more efficient and give the designer a possibility to choose suitable synthesis styles. In our approach, the CAMAD system is currently used to carry out the synthesis task.

The internal design representation into which S'VHDL programs are translated is called ETPN (extended timed Petri net) [12]. ETPN consists of two separate but related parts: control part and data path. The *data path* is represented as a directed graph with nodes and arcs. The nodes are used to capture data manipulation and storage units. The arcs represent the connections of the nodes. The *control part*, on the other hand, is represented as a timed Petri net with restricted transition firing rules.

The S'VHDL compiler translates an S'VHDL description into an ETPN representation by generating the corresponding data path and the control Petri net. Each S'VHDL statement is mapped into a separate control place in the Petri net. An arc representing data (or control) dependence between two nodes is modelled as a transition. If the dependence is conditional, then the corresponding transition is guarded by a condition generated by the data path. Construction of the ETPN data path is carried out by the convention that each scalar variable will be mapped into a node, i.e., each scalar variable is assumed at this point to be implemented by one register. Each operation *instance* will be mapped into an individual node. The arcs between the data path nodes are then introduced to model the data communication between the nodes. These arcs will be guarded by corresponding control places. Note that each statement will usually be mapped into a control place. However, complex statements will later be decomposed in the synthesis process.

In the examples throughout the paper data path nodes will be represented as rectangles with labels indicating the functions of the nodes or their names. Transfer of data from one node to another via an arc is controlled by control signals coming from the control part. This control relation is indicated by using control place labels to guard arcs. When a control place in the Petri net holds a token (a control signal is sent), its guarded arcs in the data path are open for data to flow. A transition may be guarded by one or more conditions produced from the data path. It may be fired when it is enabled (all its input places have a token) and the guarding condition is true.

After an S'VHDL program is translated into an ETPN representation, parallelism extraction is carried out. As a result, a new ETPN with maximal parallelism is obtained. Scheduling/allocation is then performed using an iterative transformation approach implemented in CAMAD [12]. This means that the models described in sections 3 and 4 are inputs to high-level synthesis and subject to design transformations and optimizations. When the final ETPN is generated it will be converted into a netlist and one or several FSMs which represent the final design.

3. The Unrestricted Model

Our approach supports two basic models for specifying VHDL concurrent processes, the unrestricted model and the reduced-synchronization model. The unrestricted model offers the freedom to express process interaction using signals conforming to the full S'VHDL synthesis subset. From the point of view of synthesis the model implies practically the hardware implementation of the simulation cycle. This means that processes have to wait for each other until all of them are executing a wait statement in order to update the signal values.

A "wait on signal" statement is represented in ETPN by associating a condition to the transition in the control part corresponding to the waiting process [4]. The condition will be produced as the result of an assignment to the corresponding signal. Figure 1 shows the ETPN representation of signals for this model and the control part corresponding to a wait statement. Signals are modeled by two register nodes (s and s') in the data path. The value referred to by the processes accessing the signal is stored in node s while the node s' stores the last assigned value. Condition C_s indicates an event on the signal s. Updating the signal, by passing the value from node s' to node s, is controlled by place Q (shown in Figure 2) that will hold a token only when all pro-

cesses are executing a wait statement. This structure can also be extended to produce the condition corresponding to a transaction on the signal [4]. For reasons of simplicity we use a compressed data path representation for signals depicting only the two register nodes (as, for example, in Figure 2).

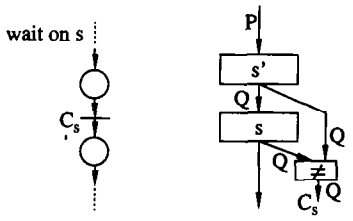

Fig. 1. Design representation of signals and wait statements in the unrestricted model

The hardware synthesized for this model is controlled, optionally, either by a single FSM or by several FSMs working synchronously together.

3. 1. Synthesis of a Collection of State Machines

Synthesis of several FSMs, one for each process, is performed on a design representation containing several independent control Petri nets synchronizing through shared data path conditions. The representation in Figure 2 corresponds to an architecture that includes k processes and n signals. For simplicity reasons only one wait statement has been depicted in each of the processes **P1, P2, ..., Pk.** A

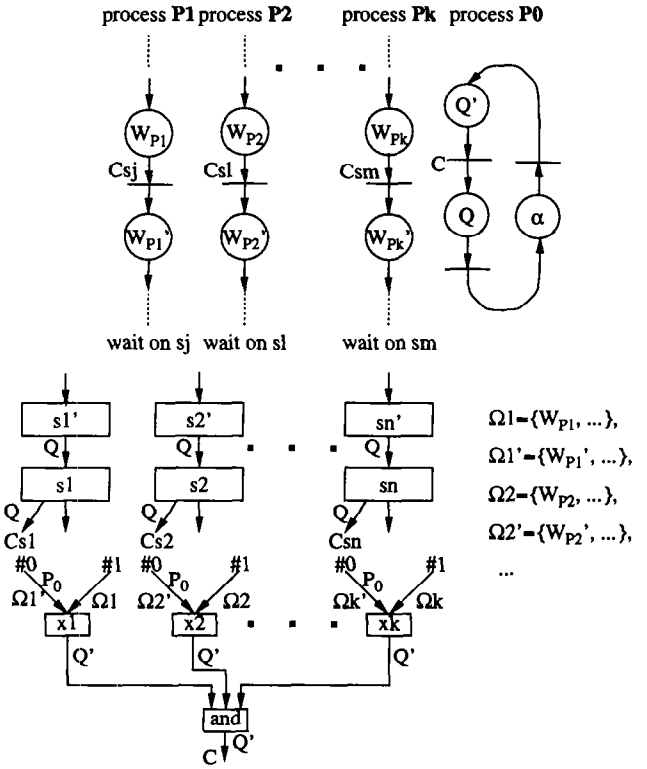

Fig. 2. Design representation for generation of a collection of FSMs

supervisor process **P0** is automatically generated during compilation of the S'VHDL description, and is responsible for the synchronized updating of signals (under the control of place Q). The required synchronization is achieved by using the one-bit register nodes $x_1, x_2, ..., x_k$, one for each process in the design. Node x_i is initially reset by the place P_0 that holds the initial token in the control Petri net. Setting of x_i is controlled by any of the places in the set Ω_i, where Ω_i consists of all control places corresponding to the wait statements that belong to process **Pi.** Node x_i will be reset under the control of any place in the set Ω_i', where Ω_i' consists of the places that are the direct successors of those in Ω_i.

Place α in process **P0** does not control any arc. This dummy place indicates that a certain delay has to be introduced to allow resetting of the one-bit nodes (controlled by places in the sets Ω_i') before the same nodes are referred to under the control of place Q' (the direct successor of α) in process **P0.**

Since the control Petri nets corresponding to the processes are disjoint, synthesis of a separate state machine for each process is possible. A reachable marking generation algorithm with as-soon-as-possible transition firing rule, which has been implemented in CAMAD, is used to transform each Petri net into an FSM [12]. The set of FSMs is synchronized by the FSM corresponding to process **P0.** With the help of the nodes $x_1, x_2, ..., x_k$ in the data path, **P0** coordinates the global synchronization so that signals are updated only when all processes are in a wait state.

3. 2. Synthesis of a Single State Machine

If the complexity and/or the number of processes are not very large, the control structure can be synthesized to a single FSM. This state machine is generated using a design representation that differs from that in Figure 2. If the user asks for synthesis of the k processes to a single FSM the S'VHDL compiler generates a control Petri net like the one in Figure 3. Synchronization between waiting processes in order to update signals is moved entirely into the control part where it becomes explicit. The control places $Q1, Q2, ..., Qk$, one for each process, hold a token only when the corresponding process is executing a wait statement. When *all* processes are waiting, the transition T (in the middle of Figure 3) can be fired; thus the places $Q1', Q2', ..., Qk'$ will get tokens and the signals will be updated. If condition C_{si} associated to a signal si, on which process **Pj** is waiting, becomes true, process **Pj** will continue (the token is passed from Qj' to the output place of the transition on which the process was waiting). If C_{si} is false (the expected event did not happen) **Pj** enters again its waiting state (the token is passed back from Qj' to Qj).

Moving synchronization entirely into the control part increases the complexity of the control Petri net. But this complexity does not entail the generation of a higher number

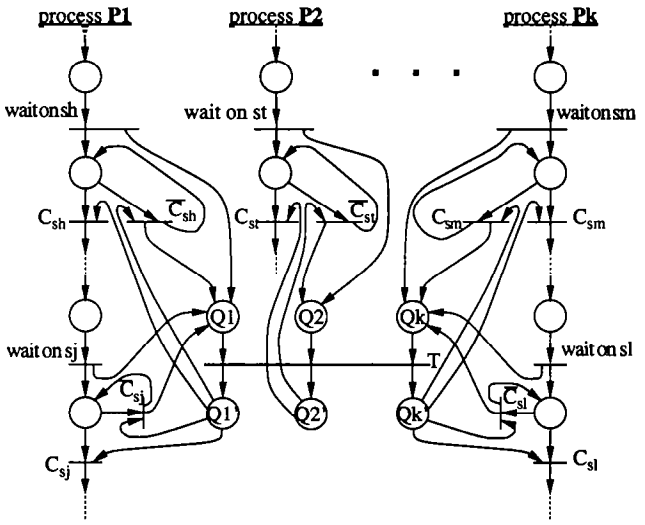

Fig. 3. Design representation for generation of one FSM

of FSM states. The additional constraints introduced into the control part are used by CAMAD to eliminate unreachable states at FSM generation and result actually in the reduction of states.

For this solution no "supervisor" process **P0** has to be generated and there is no need for the register nodes x_1, x_2, ..., x_k in the data path. The control parts corresponding to the processes are tightly interconnected and thus it is appropriate to generate just one single FSM. Handling the whole representation globally during synthesis for the design of a single FSM allows more control on the allocation of data path elements and offers the possibility of sharing hardware between different processes during the synthesis process. During the high-level synthesis process, hardware modules can be shared across the process boundaries if the related operations are scheduled into different time steps, which is illustrated by the synthesis results given in section 5.

4. The Reduced-Synchronization Model

The unrestricted model entails the implementation of the VHDL simulation cycle in hardware. Thus it results in a strong synchronization of the processes, which very often exceeds the level needed for the correct functionality of the circuit. This *oversynchronization* is the price payed for an unrestricted use of signals while preserving at the same time simulation semantics for synthesis. Without giving up simulation/synthesis correspondence, the oversynchronization can be relaxed when the correct behavior of the described hardware does not rely on the implicit synchronization enforced by the simulation cycle. We call such a description *well synchronized.* In a well synchronized VHDL description all the assumptions that provide the proper synchronization and communication between processes are *explicitly* stated by operations on signals.

We will now present a synthesis strategy that does not reproduce the simulation cycle in hardware while maintaining simulation/synthesis correspondence. It accepts designs specified according to a certain description style and produces independent FSMs which work in parallel. The S'VHDL descriptions conforming to this style are implicitly well synchronized.

4. 1. The Designer's View

With the reduced-synchronization model, the designer describes hardware as a set of S'VHDL processes communicating through signals. Any number of processes can communicate through a given signal (we say that these processes are connected to the signal) but only one of these processes is allowed to assign values to it. Assignment of a value to a signal is done by a *send* command. Processes that refer to the signal will wait until a value is assigned to it, by calling a *receive* command. Both *send* and *receive* have the syntax of ordinary procedure calls.

A *send* command, denoted as send(X, e), where X is a signal, e is an assignment expression, and e and X are type compatible, is executed, by a process **P**, in two steps:

1) process **P** waits until all other processes connected to signal X are executing a *receive* on this signal (if all these processes are already waiting on a *receive* for X, then process **P** enters directly step 2);
2) expression e is evaluated and its value is assigned to signal X. This value becomes the new value of X. After that process **P** continues its execution.

A *receive* command, denoted as receive(X), where X is a signal, causes the executing process to wait until a send on signal X is executed. Communication with *send* and *receive* can also be achieved through several signals [5].

The definition of the *send/receive* commands ensures that between the execution by a process of two consecutive receives on a given signal X, the value of this signal remains unchanged. This is due to the fact that in this interval no *send* on that signal can become active. This property is very important from a synthesis point of view (see section 4.2). Communication with *send* and *receive* requires synchronization between processes. However, it is important to note that this synchronization does not affect *all* processes, but only those involved in the specific communication (the processes connected to a given signal).

To avoid undesired blocking of a process on a *receive* command, the boolean function *test* is provided. Test(X), where X is a signal, returns *true* if there is a process waiting to execute *send* on X; otherwise the function returns *false*.

An S'VHDL description corresponding to this model can be transformed by a preprocessor into an equivalent standard VHDL model for simulation [5]. Starting from the same description, the S'VHDL compiler generates the ETPN internal representation that will be synthesized by the CAMAD system. Simulation/synthesis correspondence will

be preserved during the synthesis process.

4. 2. Reduced-Synchronization Model Synthesis

For the reduced-synchronization model a signal will be represented as a simple data path node. After an assignment (as the result of a *send* executed on the signal) the value of the node is directly updated. Synchronization between the process assigning to a signal and all those accessing it, imposed by the *send/receive* mechanism, makes an assignment in two steps unnecessary.

The synchronization protocol between the process executing a *send* and those executing *receive* on a given signal is implemented using one-bit register nodes (e.g., A_X in Figure 4). In Figure 4 we show the communication protocol for sending/receiving on signal X. The condition C_X is used to synchronize the process executing *send* on X while the complementary condition $\overline{C}_X$ controls the continuation of the *receive* command. Petri net places P_1, R_1, P_2 and R_2 are used to implement the proper synchronization for the handshaking protocol while the place P_e evaluates the expression of the *send* command. An extension to the described formulation has also allowed the use of multiple *send* and *receive* commands [5].

This model leads to hardware structures that work at a higher degree of parallelism than those synthesized for the previous one. It doesn't require a global synchronization of *all* processes. Signals need not be implemented by double registers with additional functional elements in the data path. They are represented and updated exactly like ordinary variables. As a consequence of the fact that process interaction is based on a handshaking protocol and does not need any global control, the Petri net controllers of the processes can be implemented as independent FSMs working in parallel.

To benefit from the advantages of this synthesis approach the designer has to adapt his description to the style required by the reduced-synchronization model. Using *send* and *receive* instead of wait statements and signal assignments is more natural and simpler for higher level descriptions. A similar approach is proposed for hardware/software co-specifications and successfully used in a co-design project [6]. Our results support automated synthesis and more efficient application of this model for process communication. This modelling style facilitates also the organization of high-level design into loosely coupled processes with a well structured interface. If such an organization is possible, the reduced-synchronization model is a natural approach for synthesis and results in efficient and highly parallel hardware implementations.

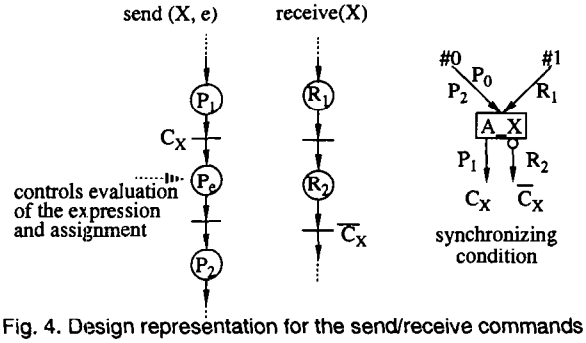

Fig. 4. Design representation for the send/receive commands

5. Experimental Results

The first example is an architecture known as the "move machine" [15] which is capable of moving instructions and data between memory and processor. ALU operations are considered to be associated to addresses in memory; arithmetic and logic operations are side effects of moving data to and from these locations. The original VHDL description of the move machine given in [15] contains three processes: one for loading the next instruction, the second for computing operand address, and the third for executing the instruction. The three processes are activated sequentially in a loop, one after the other, which corresponds to the classical execution chain of an instruction.

The original description of the move machine architecture has been changed according to our reduced-synchronization model. To increase parallelism of the design we organized the architecture as a control unit (one process) connected to two memory modules (two processes), the first one for instructions and the second one for data. The process representing control unit loads an instruction, computes the address, and executes the instruction. During the execution of an instruction the next one is loaded to achieve more parallelism. Communication between the control unit and the memories is done through the bit-string signals representing the command (read or write), the memory address, and the transferred data. The S'VHDL code of the move machine can be found in [5].

The synthesis results of CAMAD are presented in Table 1. For the move machine we used first the description given in [15] and synthesized it to a single FSM, according to our unrestricted model. The modified version of the move machine in reduced-synchronization style has been synthesized to three FSMs, each corresponding to one process. Results reported in [15], for the synthesis of the move machine to a synchronous hardware, indicate 118 states and a CPU time needed for the controller generation of 2.27 seconds. This time was obtained by using a multiprocessor machine. Our CPU times given in the table correspond to the whole synthesis process and are obtained on a SUN Sparc ELC station. Comparing the synthesis of a single FSM with that of more FSMs we observe that the second approach results in the use of two additional comparators. This is the consequence of the fact that generating separate FSMs makes it impossible to share hardware (see section 3.2), but on the other hand results in a lower complexity of the control structure and in a higher degree of parallelism.

For the elliptic filter in Table 1 we used the VHDL benchmark given in [16]. Its results are included here to

Example	Model		States	Function units	CPU time (s)
Move machine	Unrestricted		61	1 ALU, 1 decoder	20.65
	Reduced-Synchronization	Process_1	5	1 comparator	0.70
		Process_2	5	1 comparator	0.70
		Process_3	25	1 ALU, 1 decoder	14.27
Am 2901	Unrestricted		22	1 ALU, 1 adder, 1 decoder,1 inverter	112.62
Elliptic filter	Unrestricted		19	1 adder, 1 multiplier	10.83

Table 1: Summary of synthesis results of CAMAD

show that our system performs well also with arithmetic dominated hardware, although the elliptic filter example consists of only one process. Results reported in [11] indicate 18 states by using three adders and one multiplier with a CPU time of 360 seconds. In [15] the elliptic filter is synthesized to 19 states, with three adders and one multiplier, in 107 seconds.

Finally, by synthesizing the Am 2901 four-bit microprocessor slice listed in Table 1, we demonstrate that CAMAD is able to synthesize VHDL specifications of standard commercial microprocessor structures with results that are similar to the original manual design.

6. Conclusions

This paper addresses one of the most difficult aspects in the hardware synthesis of behavioral VHDL specifications, namely synthesis of concurrent processes while preserving standard VHDL simulation semantics.

We first developed a model that allows a practically unrestricted use of signals and wait statements by producing a synchronous hardware with a global control of process synchronization for signal update. The hardware can be controlled either by a single state machine or by a collection of FSMs working synchronously together.

With our second model we have shown that it is possible to relax the strong synchronization imposed by the VHDL simulation cycle without affecting the semantic correctness of the synthesized circuit. S'VHDL descriptions written according to this style are synthesized to hardware with a higher degree of parallelism and asynchrony, without any need for additional global synchronization.

The results we report in the paper show that the CAMAD high-level synthesis system can efficiently handle ETPN design representations produced by the S'VHDL compiler including designs described as interacting concurrent processes according to the proposed models. More research is needed, however, in the area of high-level specific transformations applicable to concurrent processes and communication protocols.

References

[1] Bergamaschi, R. A., Kuehlmann, A., *A System for Production Use of High-Level Synthesis*, IEEE Transactions on Very Large Scale Integration (VLSI), vol. 1, no. 3, Sept. 1993, pp. 233-243.

[2] Biesenack, J., et. al., *The Siemens High-Level Synthesis System CALLAS*, IEEE Transactions on Very Large Scale Integration (VLSI), vol. 1, no. 3, Sept. 1993, pp. 244-253.

[3] Camposano, R., Saunders, L. F. and Tabet, R. M., *VHDL as Input for High-Level Synthesis*, IEEE Design and Test of Computers, March 1991, pp. 43-49.

[4] Eles, P., Kuchcinski, K., Peng, Z., Minea, M., *Compiling VHDL into a High-Level Synthesis Design Representation*, Proc. EURO-DAC/EURO-VHDL'92, 1992, pp. 604-609.

[5] Eles, P., Kuchcinski, K., Peng, Z., Minea, M., *Two Methods for Synthesizing VHDL Concurrent Processes*, Research Report, LiTH-IDA-R-93-22.

[6] Ecker, W., *Using VHDL for HW/SW Co-Specification*, Proc. EURO-DAC/EURO-VHDL'93, 1993, pp. 500-505.

[7] Harper, P., Krolikoski, S., Levia, O., *Using VHDL as a Synthesis Language in the Honeywell VSYNTH System*, in J.A. Darringer, F.J. Rammig (Editors), *Computer Hardware Description Languages and their Applications*, North Holland, 1990, pp. 315-330.

[8] *IEEE Standard VHDL Language Reference Manual*, IEEE Std. 1076-1987, IEEE Computer Soc. Press, 1987.

[9] Müller, J., Krämer, H., Analysis of Multi-Process VHDL Specifications with a Petri Net Model, in: *Proc. EURO-DAC/ EURO-VHDL'93* (1993), pp. 474-479.

[10] Nagasamy, V., Berry, N., Dangelo, C., *Specification, Planning, and Synthesis in a VHDL Design Environment*, IEEE Design & Test of Computers, June 1992, pp. 58-68.

[11] Paulin, P.G., Knight, J.P., *Force-Directed Scheduling for the Behavioral Synthesis of ASIC's*, IEEE Transactions on Computer-Aided Design, vol. 8, no. 6, June 1989, pp. 661-679.

[12] Peng, Z., Kuchcinski K., *Automated Transformation of Algorithms into Register-Transfer Level Implementation*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 13, no. 2, Feb. 1994, pp. 150-166.

[13] Postula, A., *VHDL Specific Issues in High Level Synthesis*, Proc. Euro-VHDL'91, 1991, pp. 70-77.

[14] Ramachandran, L., Vahid, F., Narayan, S., Gajski, D., *Semantics and Synthesis of Signals in Behavioral VHDL*, Proc. EURO-DAC/EURO-VHDL'92, 1992, pp. 616-621.

[15] Roy, J., Kumar, N., Dutta, R., Vemuri, R., *DSS: A Distributed High-Level Synthesis System*, IEEE Design & Test of Computers, June 1992, pp. 18-32.

[16] Vemuri, R., Roy, J., Mamtora, P., Kumar, N., *Benchmarks for High Level Synthesis*, Technical Memo-ECE-DDE-91-11, University of Cincinnati, 1991.