Interface Refinement for System Design

ZOLTAN BARUCH

Computer Science Department,
Technical University of Cluj-Napoca
e-mail: Zoltan.Baruch@cs.utcluj.ro

ABSTRACT: *Interface refinement* is the task of generating buses and their protocols for the abstract channels of a digital system specification. The *bus generation* determines the width of the bus that will implement a group of channels. The *protocol generation* selects and generates the communication protocol that will implement the data transfer over the bus. In this paper we describe methods for interface refinement. We describe techniques for determining the buswidth for implementing a group of channels, and we present a method for protocol generation.

KEY WORDS: digital systems, high-level synthesis.

1. Introduction

A system specification consists of *functional objects* such as processes, variables and communication channels. During system design, these functional objects in the specification are grouped into a set of *system components* such as processors, ASICs, memories and buses. While functional objects are devoid of any structure, system components have a well-defined structure, such as the number of pins on a chip, the number and size of words in a memory, or the number of wires in a bus. Updating the specification to reflect the transformation of the functional objects into system components is called *specification refinement*.

Concurrent processes in a specification communicate with one another by sending messages over abstract communication channels. To minimize interconnect cost, the channels in the system are grouped in such a way that each group of channels is implemented by a common physical medium called a bus. A *bus* consists of a set of wires over which the actual data transfer takes place under a bus protocol. The task of generating buses and their protocols for each group of channels is called *interface refinement*. In this paper we describe methods for interface refinement. We shall begin by presenting the parameters that characterize the channels and buses.

For any channel, only one *master* process initiates and controls the data transfer and one or more *slave* processes respond to communication initiated by the master process. If the master process sends (or receives) data over the channel, then the *direction* associated with the channel is write (or read). Channels are usually unidirectional, which implies that if a process both reads and writes to a variable in another process, distinct channels for each direction of data transfer are needed.

Channels are characterized by four parameters. Channel **data size**, *bits(C)*, represents the number of bits in a single message transferred over channel *C*. The data size includes any address bits that may be required to access array variables over the channel. The **number of accesses**, *access(P, C)*, represents the number of times that process *P* transfers data over channel *C* in its lifetime. The **channel average rate**, *avg_rate(C)*, is the rate at which data is sent over channel *C* over the lifetime of the processes communicating over the channel. The

channel peak rate, *peak_rate(C)*, is the rate at which a single message is transferred over the channel *C*.

Any bus implementation of a channel or a group of channels can be characterized by four parameters. The **buswidth**, *buswidth(B)*, is the number of data lines in bus *B* over which the message can be transferred between the processes. Associated with each bus is a protocol that defines the exact sequence of operations that implement the message transfer over the set of data lines. The **protocol delay**, *prot_delay(B)*, is the total delay of the protocol employed for a single transfer of data over the bus. The **average bus rate**, *avg_rate(B)*, is the rate at which data can be transferred across the bus. The **peak bus rate**, *peak_rate(B)*, is the maximum rate at which data can be transferred across the bus. The peak bus rate and the buswidth have the following relation with one another:

$$peak\_rate(B) = \frac{buswidth(B)}{prot\_delay(B)} \quad (1)$$

Given a group of abstract communication channels, interface refinement determines the buswidth and the protocol for the bus that will implement the channels. Interface refinement is driven by two conflicting goals. First, it attempts to minimize the interconnect cost between the system components that use a bus by reducing the buswidth, *buswidth(B)*. Second, it attempts to maximize the communication performance over the bus by increasing the peak rate of the bus, *peak_rate(B)*, and consequently increasing the *buswidth(B)*.

Interface refinement consists of two tasks: bus generation and protocol generation. Given a set of constraints, *bus generation* determines the width of the bus that will implement the group of channels. After the desired buswidth has been selected, *protocol generation* selects and generates the communication protocol that will actually implement the data transfer over the bus.

2. Bus Generation

In this section, we describe techniques for determining the buswidth for implementing a group of channels.

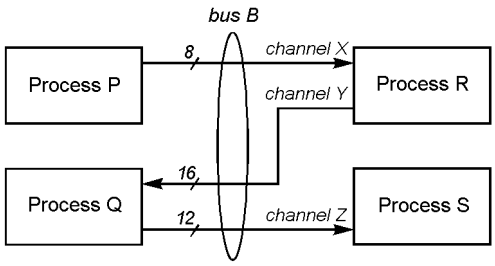

Figure 1. A typical bus formed by merging channels transferring data of different sizes between several processes.

A simple case of bus generation in which all the channels in a group have an identical message size is presented in [1]. In this case, channels are merged in such a way that all channels in any group are used exclusively over time to communicate between the same two processes. Consequently, each channel group is implemented with a buswidth identical to the size of any channel.

In a more general case, processes communicating over channels that have been grouped together transfer data over the shared physical medium simultaneously. In addition, different channels may be transferring messages of different sizes between the processes. Such a case is shown in **Figure 1**. Processes *P*, *Q*, *R* and *S* communicate over channels *X*, *Y* and *Z*, which have been grouped by system partitioning to be implemented as a single bus, *B*. The three channels transfer data of different sizes – 8, 16 and 12 bits respectively – over the bus. In addition, process *P* may need to transfer data to process *R* at the same time that process *Q* needs to send data to process *S*. We describe the bus generation for this case.

2.1. Determining the bus rate

Consider two channels, *X* and *Y*, which transfer 8-bit and 16-bit messages respectively, as shown in **Figure 2**. The number of bits associated with each message transfer is indicated above the message. We assume that the four second time interval shown in the figure represents the data transfer over the lifetimes of the processes that communicate over channels *X* and *Y*. Channels *X* and *Y* have average rates of 4 and 12 bits/second, respectively. If channels *X* and *Y* are merged into a single bus *B*, then the bus needs to send data at a rate of at least 16 bits/second to be able to satisfy the data transfer requirements of the two original channels.

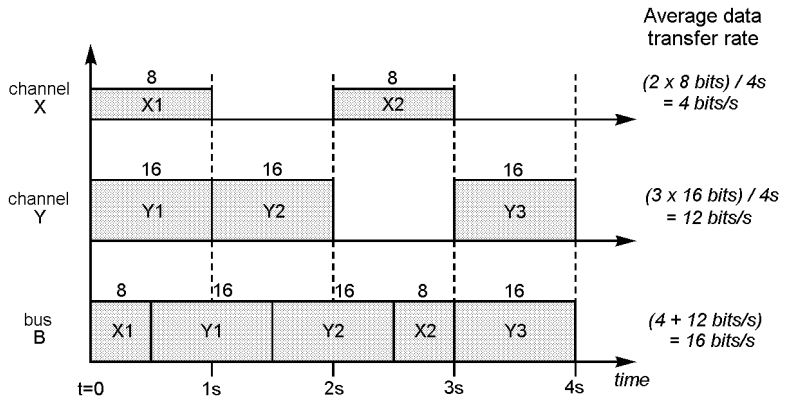

Figure 2. Merging channels *X* and *Y* into bus *B*.

The individual messages transferred over the channels have been labeled in the figure to make it easier to associate them with the data transferred over the shared bus. Consider the message labeled *Y2* transferred at the $t = 1$ second in the original channel *Y*, which is now transferred on bus *B* at $t = 1.5$ seconds. While individual message transfers may be delayed due to the bus access conflicts, the total number of bits transferred over the individual channels before channel merging are still sent over the shared bus in the same amount of time.

For the synthesis of bus *B* in **Figure 2**, we take into account that the individual channels will not always be transferring data. One channel's idle time slots are utilized for data transfers of other channels by synthesizing a bus over which data is always being transferred at a constant rate.

Before being merged into a bus, if a channel is transferring data at a certain average rate, it should be able to transfer the data over the bus at the same average rate. This can be achieved if the average rate, *avg_rate(B)*, of bus *B* is greater than the sum of the individual channel average rates. Thus,

$$avg\_rate(B) \geq \sum_{C \in B} avg\_rate(C) \quad (2)$$

The goal of bus generation should be to synthesize a bus with a minimum number of wires and an average rate given by Equation (2). The most efficient bus implementation will be achieved if the bus is never idle, and if it is constantly transferring data at a fixed rate. Under such ideal conditions, the bus peak and average rates will be identical:

$$peak\_rate(B) = avg\_rate(B) \quad (3)$$

2.2. Constraints for bus generation

For a given set of channels that have been grouped together to be implemented as a single bus, constraints and relative weights can be specified for several bus and channel parameters.

A minimum/maximum *buswidth* constraint may be derived from the overall pin constraints specified for the modules or chips to which the processes communicating over the bus have been mapped.

Channel average rate may be constrained to ensure that the processes are not slowed down due to communication delays over the bus. Given constraints on the execution time of a process that communicates over several channels, the designer may allocate the amount of time spent for communication over the various channels, from which a minimum channel average rate constraint can be derived. A maximum channel average rate may be specified in cases when one of the processes communicating over the channel represents a slow device that is incapable of sending or receiving data faster than a certain rate.

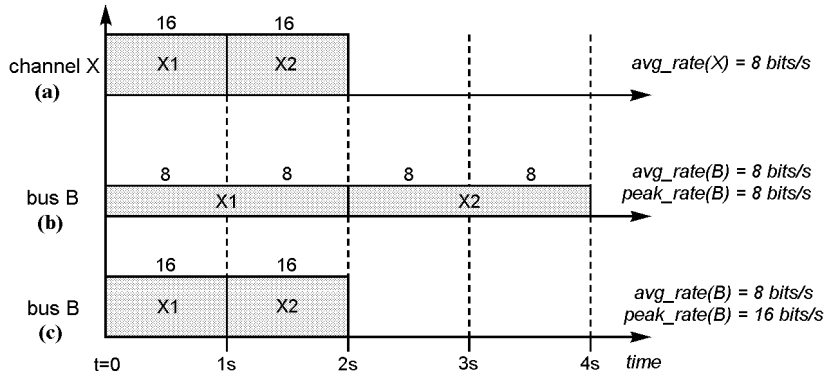

Figure 3. Channel peak rate constraints.

In certain cases, a minimum *channel peak rate* may be specified to ensure that a single message transfer over the channel does not take an excessive amount of time. For example, consider channel *X* in **Figure 3(a)**, which transfers a 16-bit message in each of the first two time slots, $t = 0$ and $t = 1$. The other two time slots are used to perform internal computations by the process that communicates over channel *X*. If we were to implement the channel as a bus by itself, from Equation (2), the bus average rate would be 8 bits/second, resulting in the execution trace of **Figure 3(b)**. However, the process now requires four time slots just for communicating over the bus. This is unacceptable, since two additional time slots will still be

required to perform the internal computations of the process (originally performed in time slots $t = 2$ and $t = 3$ in **Figure 3(a)**).

If a minimum peak rate of 16 bits/second is specified for channel *X*, we will get the desired bus implementation of **Figure 3(c)**, one that does not require any additional time slots. Thus, for all channels *C* that have a minimum peak rate constraint associated with them,

$$peak\_rate(B) > peak\_rate(C) \quad (4)$$

In case a minimum channel peak rate constraint is specified, the resulting bus may be idle at times.

2.3. Algorithm for determining buswidth

We present an algorithm for determining the width of a bus implementation [3]. The algorithm assumes that the data and control lines are disjoint. At any given instant, only one channel can transfer data over the bus. If the width of the bus is greater than the address and data bitwidths, then the address and data bits are sent simultaneously over the bus, otherwise, they are sent separately in two distinct transfers. In the latter case, the address bits have to be latched in the receiving process. If the buswidth is smaller than the size of the message being sent over it, the message is sent in multiple transfers.

All the processes which communicate over one of the channels in the bus are assumed to having a synchronous implementation. Thus, transferring a message over the bus requires a whole number of clock cycles. A variable accessed over the bus is modeled by a separate process that sends and receives its value over the bus in response to requests from other processes. Thus, while computing the execution time of a master process that accesses a variable over the bus, the slave process which model that variable is assumed to be always ready for data transfer, i.e., no synchronization delays occur for variable accesses over the bus.

The input to the buswidth-generation algorithm consists of a set of channels to be implemented on a single bus and constraints on the channel rates and buswidth. The output of the algorithm is the width of the bus that will implement that channel group.

The algorithm examines a range of potential buswidths. For each buswidth, the bus peak rate and the individual channel average rates are computed. For the synthesis of a bus that constantly transfers data, Equation (2) and (3) require that the bus peak rate should be greater than the sum of the channel average rates. Each buswidth for which the above condition holds represents a feasible bus implementation. From the set of feasible bus implementations, each corresponding to a different buswidth, we select the one which has the least cost. In case no constraints are specified, a unit buswidth corresponding to a serial data transfer is selected.

The buswidth-generation algorithm is presented in **Figure 4**. First, the range of buswidths examined by the algorithm is determined. The largest buswidth, *max_width*, is the size of the largest message transferred by any channel. The lowest buswidth, *min_width*, is 1.

The variable *curr_width* represents the current buswidth being evaluated by the algorithm. For each value of *curr_width* in the range (*min_width*, *max_width*), the bus peak rate is computed using Equation (1).

We denote by *access(P, C)* the average number of accesses performed by the process *P* to the channel *C*. The number of bits sent or received over the channel C in a single message is denoted by *bits(C)*. If a process accesses an array variable over the bus, the number of address bits are also included in *bits(C)*. For instance, if a process accesses a scalar variable *x*

of 16 bits and an array variable *y* of 32 words × 16 bits over the channels *Cx* and *Cy*, respectively, then *bits(Cx)* is 16, and *bits(Cy)* is 21 (5 address bits and 16 data bits).

```
if no constraints specified then
    return (1)
end if
/* compute range of buswidths */
min_width = 1
max_width = Max (bits(C))
mincost = ∞
mincost_width = ∞
for curr_width in min_width to max_width loop
    /* compute bus peak rate */
    peak_rate (B) = curr_width / prot_delay (B)
    /* compute sum of channel average rates for curr_width */
    avg_rate_sum = 0;
    for all channels C ∈ B loop
        avg_rate(C) = access(P,C) × bits(C) / (comp_time(P) + comm_time(P))
        avg_rate_sum = avg_rate_sum + avg_rate (C)
    end loop
    if (peak_rate (B) > avg_rate_sum) then
        /* feasible solution, determine minimal cost */
        curr_cost = ComputeCost (curr_width)
        if (curr_cost < mincost) then
            mincost = curr_cost
            mincost_width = curr_width
        end if
    end if
end loop
if (mincost = ∞)
    then return (failure)
    else return (mincost_width)
end if
```

Figure 4. Buswidth generation algorithm.

For simplicity, assume that a process *P* has exactly one channel *C* over which messages are transferred. In case than current buswidth is less than the number of bits in the message, several transfers ($\lceil bits(C) / curr\_width \rceil$) may be required to implement a single message transfer. The communication time for the process *P* is calculated as follows:

$$comm\_time(P) = access(P,C) \times \left(\left\lceil \frac{bits(C)}{curr\_width} \right\rceil \times prot\_delay(B) \right) \qquad (5)$$

Using the value of *comm_time(P)* computed above, the average rate for each channel mapped to the bus can be estimated. The sum of the channel average rates for a specific value of *curr_width* is stored in *avg_rate_sum*.

If the bus peak rate is lower than the sum of the average rates of all the channels, then *curr_width* represents an infeasible implementation for the bus. We repeat the computation of the bus peak rate and sum of the channel rates with the next higher buswidth in the range (*min_width*, *max_width*).

If the bus peak rate is greater than the sum of the individual channel rates, then *curr_width* represents a feasible implementation of the bus. For any of the constraints on buswidth, channel average and peak rates specified for the bus, the procedure *ComputeCost* calculates the cost of a feasible bus implementation as the sum of the squares of any constraint violations weighted by the relative weights specified for that constraint. For example, assume that the only constraint specified was a maximum buswidth constraint represented by *max_wires*. Let *k* represent the relative weight specified for this constraint. For any value of the buswidth, *curr_width*, the cost function for the bus would be defined as follows:

$$cost = \begin{cases} (k(curr\_width - max\_wires))^2 & \text{if } curr\_width > max\_wires \\ 0 & \text{otherwise} \end{cases} \quad (6)$$

If more than one feasible solution exists for the group of channels, the buswidth with the lowest cost is selected for implementing the bus. Variable *mincost* represents the minimum cost computed for all feasible implementations, while variable *mincost_width* represents buswidth, which corresponds to the minimum cost.

If there were no feasible solutions for all the buswidths examined, then an implementation for the group of channels would not be possible. Any implementation for such a group of channels would progressively delay the processes communicating over the bus. Such a situation can arise when several channels with very high average rates are grouped together to be implemented as a single bus. One solution would be to split the group of channels to be implemented further by more than one bus. Alternatively, the lowest cost infeasible buswidth may be selected.

3. Protocol generation

After the selection of the appropriate buswidth, protocol generation defines the exact mechanism of data transfer over the bus. A bus consists of three sets of wires.

Data lines are used to send data over the bus. The number of data lines can be determined by the buswidth-generation algorithm, or it can be specified by the system designer.

Control lines are required to synchronize the processes that communicate over the bus. The number of control lines required depends on the type of protocol selected to implement the data transfers. The set of control lines are shared by all the channels mapped to the same bus.

Identification or mode lines are required to identify the particular channel that is transferring data over the bus at any moment. Since the bus control signals are shared by all channels, such identification (ID) lines are essential to enable processes to recognize when the control signals over the bus are meant for them. Each channel in the bus is assigned a unique ID, which serves as its address. Every time a master process initiates transfer of data over the bus, it places the corresponding ID of the relevant channel on the bus ID lines so that only the corresponding slave process will respond to the control signals. The ID lines can also be directly encoded into the addresses of data accessed over the bus. In such cases, the slave processes must have an address detection mechanism which examines each address placed on the

bus, to determine whether they should respond to the control signals set by the master process.

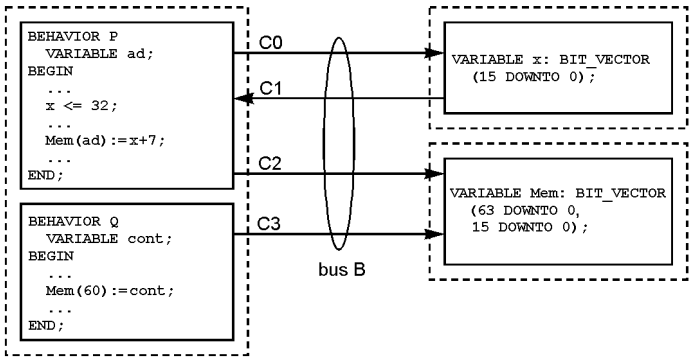

Figure 5. Processes accessing variables over channels grouped into a bus.

We shall examine protocol generation through a simple example, shown in **Figure 5** [2]. Variables *x* and *Mem* are accessed by processes *P* and *Q*. The dashed lines indicate the assignment of the processes and variables to system components. Channels *C0*, *C1*, *C2* and *C3* are grouped into a single bus *B*, whose width has been determined to be 8 bits. Protocol generation consists of several steps:

- **Protocol selection**. Various communication protocols may be selected for a bus implementation, such as a full-handshake, half-handshake, or hardwired ports. Each protocol requires a different number of control lines. For bus *B* in **Figure 5**, a full-handshake protocol is selected.
- **ID assignment**. If *N* channels are implemented on the same bus, $log_2(N)$ lines will be required to encode the channel ID. The four channels in **Figure 5** require two ID lines. Channel *C0* is assigned the ID “00”, *C1* is assigned “01” and so on.
- **Bus structure and procedure definition**. The structure defined for the bus (i.e. the data, control and ID lines) is defined in the specification. For each channel mapped to the bus, appropriate send and receive procedures are generated. **Figure 6** shows the declaration of an 8 bit bus, with two control lines and two ID lines. The bus *B* is declared to be a global variable (a signal in the *VHDL* language) so that all processes can access it. Process *P* writes to the 16-bit variable *x* over channel *C0*. Since the buswidth is only 8 bits, procedures *SendC0* and *ReceiveC0* in **Figure 6(b)** transfer the 16-bit message associated with channel *C0* over the bus, in two transfers of 8 bits each.
- **Update variable-references**. References to a variable that has been assigned to another system component by system partitioning must be updated in processes that were originally referencing it directly. Accesses to variables are replaced by the send and receive procedure calls corresponding to the channel over which the variable is accessed. For example, in **Figure 5**, process *P* writes the value "32" directly to variable *x*. Channel *C0* represents the write to variable *x*. The statement "*x <= 32*" is replaced by the send procedure call "*sendC0 (32)*" as shown in **Figure 7**. The statement "*Mem(60) := cont*" in process *Q* is updated to "*sendC3 (60, cont)*, indicating that the value in *cont* is to be written to address 60 of array *Mem*.

```
TYPE HandShakeBus IS RECORD
  start, done: BIT;
  ID: BIT_VECTOR (1 DOWNTO 0);
  DATA: BIT_VECTOR (7 DOWNTO 0);
END RECORD;

SIGNAL B: HandShakeBus;

PROCEDURE ReceiveC0 (rxdata: OUT BIT_VECTOR) IS
BEGIN
  FOR j IN 1 TO 2 LOOP
    WAIT UNTIL (B.start = '1') AND (B.ID = "00");
    rxdata (8*j - 1 DOWNTO 8*(j-1)) <= B.DATA;
    B.done <= '1';
    WAIT UNTIL (B.start = '0');
    B.done <= '0';
  END LOOP;
END ReceiveC0;

PROCEDURE SendC0 (txdata: IN BIT_VECTOR) IS
BEGIN
  B.ID <= "00";
  FOR j IN 1 TO 2 LOOP
    B.DATA <= txdata (8*j - 1 DOWNTO 8*(j-1));
    B.start <= '1'
    WAIT UNTIL (B.done = '1');
    B.start <= '0'
    WAIT UNTIL (B.done = '0');
  END LOOP;
END SendC0;
```

Figure 6. Defining bus *B* and the send and receive protocols for channel *C0*.

- **Generate processes for variables**. In order to obtain a simulatable system specification, a separate process is created for each group of variables accessed over a channel. Appropriate send and receive procedure calls are included in the process to respond to access requests to the variable over the bus. In **Figure 5**, the variables *x* and *Mem* were assigned to different system components, as shown by the dashed lines. In **Figure 7**, processes *xProc* and *MemProc* have been created for these two variables.

The protocol generation presented has several advantages. First, the refined specification is simulatable, and the design functionality, after insertion of buses and communication protocols, can be verified. Second, by encapsulating data transfer over the bus in terms of send and receive procedures, the description of the process remains much ordered than it would be if we were to insert the assignments for the control and data lines at each communication point in the procedure. Finally, if at a later stage another communication protocol were selected for communication over the bus, only the bus declaration and send and receive procedures need be changed. The system's process descriptions, including the send and receive procedure calls, remain unchanged.

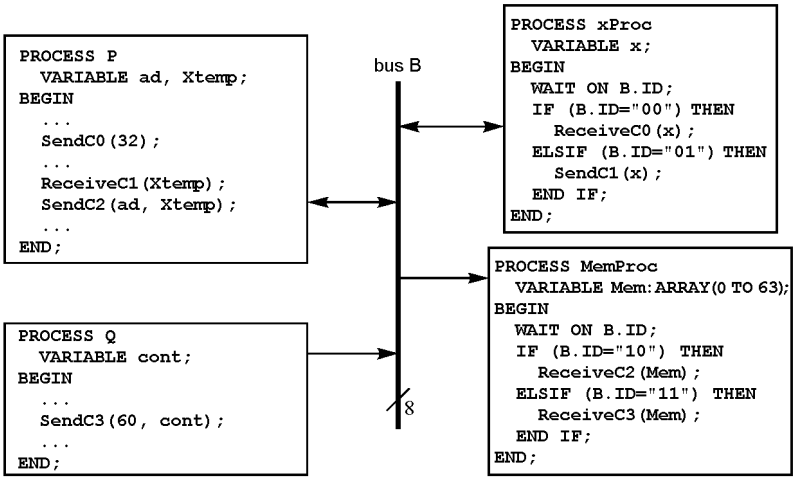

Figure 7. Refined specification after protocol generation.

4. Conclusions

In this paper we presented a set of tasks that have to be performed to refine the interface specifications. A method for bus generation was introduced, and the tradeoffs between buswidths and system performance were evaluated. For a selected buswidth, we showed how communication protocols can be generated. We presented the advantages of this protocol generation method.

The work presented can be extended in several directions. Methods for resolving access conflicts must be developed, for different arbitration schemes. Techniques are needed for interfacing fixed incompatible protocols. During protocol generation, we need to develop metrics to evaluate several candidate protocols that may be selected for implementing transfers over the bus. We need to incorporate the effect of arbitration delays on the process execution times and channel average rates. The effect of different bus arbitration schemes on the performance of the processes communicating over a bus needs to be investigated.

References

[1] Filo, D., Ku, D., Coelho, C. N., De Micheli, G., "Interface optimization for concurrent systems under timing constraints", in *IEEE Transactions on Very Large Scale Integration Systems*, September 1993, pp. 268-281.

[2] Gajski, D. D., Vahid, F., Narayan, S., Gong, J., *Specification and Design of Embedded Systems*, P T R Prentice Hall, 1994.

[3] Narayan, S., Gajski, D. D., "Features supporting system specification in HDLs", in *Proceedings of the European Design Automation Conference*, 1994.