Interfacing Incompatible Protocols using Interface Process Generation †

Sanjiv Narayan
Viewlogic Systems Inc.
Marlboro, MA 01752

Daniel D. Gajski
Dept. of Computer Science
Univ. of California, Irvine, CA 92717

Abstract

During system design, one or more portions of the system may be implemented with standard components that have a fixed pin structure and communication protocol. This paper described a new technique, interface process generation, for interfacing standard components that have incompatible protocols. Given an HDL description of the two protocols, we present a method to generate an interface process that allows the two protocols to communicate with each other.

1 Introduction

System design maps the functionality of the system to a set of system components (chips, logic blocks on a chip, memories, processors etc.). As the complexity of systems being designed today increases, time-to-market pressures often result in a significant reuse of standard components, i.e., existing designs, off-the-shelf components etc. During system design, one or more portions of the system being designed may bound to (i.e., implemented by) these standard components, while other portions of the system may be custom designed. The pin structure and the communication protocols of these standard components are fixed and cannot be changed. Consequently, communication among system components with different communication protocols is possible only if proper interfaces are introduced.

Consider two standard components A and B that communicate data between them using two fixed protocols P_a and P_b respectively, as shown in Figure 1. If the two protocols are compatible with each other, we simply need to connect the appropriate ports on both the standard components to ensure that they are able to communicate with each other. However, if the protocols P_a and P_b are incompatible, an *interface process* needs to be inserted between the two standard components. The interface process, shown with dashed lines in the figure, is a process which facilitates data transfer between two standard components by interfacing their two incompatible protocols.

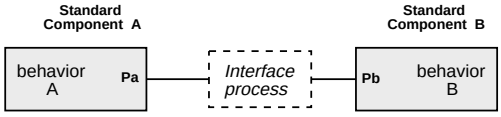

Figure 1: Interfacing standard components.

Another circumstance where interface processes may be required is when tight pin constraints on two system components during system design may cause the number of data pins in each of the system components to be different. An unequal number of data pins will lead to incompatible communication protocols for which an interface process will be required for proper communication.

2 Previous Work

While several research efforts have looked into the interaction between interface timing constraints and synthesis, only three approaches have examined protocol compatibility between standard components in detail. We will describe these approaches briefly.

Synthesis of interface *transducers* between custom chips and system buses was presented in [1, 2]. A transducer is defined as the glue logic that connects two circuit blocks. Timing diagrams of the two incompatible interfaces were specified as inputs. The output was a logic specification of the transducer circuit. First, separate event graphs are generated from the timing diagrams of the two interfaces. Next, the two event graphs are combined into a single graph by either explicitly specifying merge labels that connect nodes in the two event graphs or by examining the data dependencies between the two interfaces. A skeletal circuit is first generated using a template matching strategy for each output in the transducer circuit, and then optimized. Any timing constraint violations and race conditions in the circuit are corrected by adding appropriate logic circuitry. The main advantage of the transducer synthesis method is that it incorporates detailed timing constraints between events in the two interfaces. Second, the output of

†This work at U.C. Irvine was supported by the Semiconductor Research Corporation (grant #93-DJ-146).

32nd ACM/IEEE Design Automation Conference ®

transducer synthesis is a logic circuit – no hardware needs to be synthesized.

An approach to the synthesis of *protocol converters* was presented in [3, 4]. A protocol converter is a logic circuit that matches the control signals on each of the two protocols to enable data transfers between them. Protocol conversion assumes that the datapath of the converter is given. The protocols being interfaced are specified using Verilog-based finite-state machines. A cross product of the two state machines is obtained and optimized to obtain the state-machine description of the converter. This approach can potentially lead to a very large number of states in the protocol converter.

A method for the design of *system interface modules* was proposed for the SIERA design environment in [5]. SIERA [6] seeks to minimize the system design effort by providing a library of modules containing detailed I/O structure and protocols (specified as event graphs). Inter-module communication is abstracted to a level where the designer need only instantiate the appropriate system modules from the library and specify their interactions in terms of the interconnection of the source and destination ports using high-level primitives in a special-purpose language, IDL. First, a control flow graph is constructed from the user specification of the interconnection of the modules. Scheduling and allocation are applied to the control flow graph to generate the interface controller, and a datapath to implement the data transfers between the data lines of the two protocols. Next, the event graphs for the two protocols are obtained from the module library and interconnected based on data dependencies between the operations in the two protocols. From this event graph, a protocol controller is synthesized to respond to the control signals of the two module protocols. The main advantage of this approach is that it frees the designer from the burden of considering any low-level details (such as I/O control signals and timing constraints), since such information is stored in the module library.

3 Problem definition

Interface process generation defines an interface process between two communicating processes with fixed but incompatible protocols. An interface process responds appropriately to the control signals of both protocols and sequences the data transfers between them. In other words, the interface process translates one protocol into another.

Interface Process Generation addresses several issues in solving protocol incompatibility. Firstly, other than specifying the two incompatible protocols, the designer should not have to specify extra information that is required by the tool that attempts to resolve the incompatibility. In the Transducer Synthesis approach, the designer may be required to specify explicit merge labels in order to combine their corresponding event graphs. In addition, the names of the data lines in the two protocols must be identical. The Protocol Converter approach assumes that the datapath for the protocol converter is already designed and requires the designer to specify a third state machine called a "C-machine" which describes the set of legal sequence of operations between the two FSMs representing the protocols. In the System Interface Module approach, the designer is required to specify details of the interface module such as the sequence of data transfers and interconnection of data ports.

Secondly, the interface process generated should be simulatable with the behavioral descriptions of the components it connects. This will provide the designer with the capability to simulate the system specification and verify the functional correctness of the system once the incompatible interfaces have been resolved. In the Transducer Synthesis approach described in Section 2 the generated logic is not simulatable with the timing diagrams of the two interfaces that it connects. Similarly, in the case of the Protocol Converters and System Interface Modules, the capability to simulate the design after protocol incompatibility has been resolved is lacking.

Finally, it will often be the case that the two protocols being interfaced have different data widths (i.e., the number of data lines or pins) are different in two protocols. Any method that interfaces protocols should be able to handle data-width mismatches. The Transducer Synthesis and Protocol Converter approaches cannot handle data width mismatches. While the System Interface Module approach allows data width mismatches, it requires the designer to explicitly specify the interconnection of source and destination ports and the transfer of data between them.

4 Interface process generation

We now describe a technique for generating interface processes that meets the above requirements. The inputs for interface process generation are HDL descriptions of the two fixed protocols detailing the number of control and data lines and the sequence of data transfers over those lines. The output is an HDL description of the interface process and information related to the connection of ports of the two protocols.

We will illustrate interface process generation with the example of two processes, A and B, in Figure 2(a) that have been mapped to standard components. The two processes have fixed protocols, P_a and P_b, the HDL descriptions of which are given in Figure 2(b). Behavior A reads a 64K$\times$16 memory, modeled by variable $MemVar$ in process B. Protocol P_a has 8 address, 16 data and 4 control lines, while protocol P_b has 16 address, 16 data and 1 control line. The variables $AddrVar$ and $DataVar$ used in protocol P_a and $MemVar$ in protocol P_b are local to the corresponding processes and provide (receive) the relevant data values assigned to (read from) the data lines. For ease of identification, all port names have a "p" suffix.

A protocol typically consists of a set of *atomic* operations. There are five types of atomic operations:

1. waiting for an event on an input control line,
2. assigning a value to an output control line,
3. reading a value from input data lines,
4. assigning a value to an output data line, and
5. waiting for a fixed time interval.

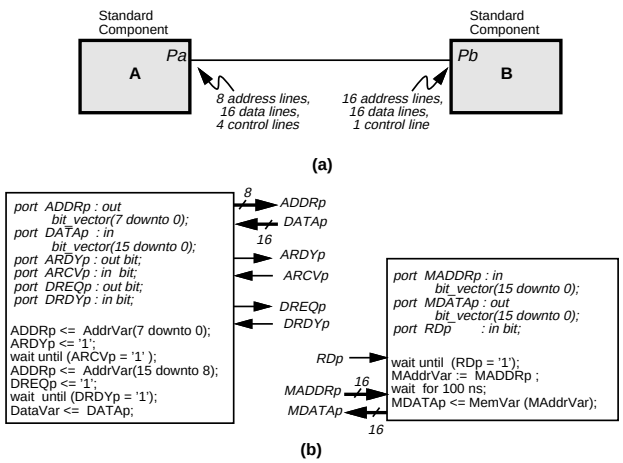

Figure 2: Representing protocols: (a) Two components A and B with fixed protocols, (b) HDL representation.

4.1 Representing protocols as ordered relations

The first step in interface process generation is representing each of the two protocols as an ordered set of relations. A *relation* defines a set of assignments to output control and data lines and the reading of values from input data lines, upon the occurrence of a certain condition. The *condition* could be an event on an input control line or a fixed delay with respect to some previous event.

Figure 3(a) shows how the set of relations can be derived from the HDL description of protocol P_a. The first two assignment statements are not preceded by any condition. Hence, the first relation, labeled A_1, consists of a default "true" condition and followed by assignments to ports $ADDRp$ and $ARDYp$. The next statement `wait until (ARCVp = '1')` represents a condition which must evaluate to true before the protocol can perform any other operation. The second relation, labeled A_2, thus consists of the condition ($ARCVp$ = '1') and the subsequent assignments to the data port $ADDRp$ and control port $DREQp$. Finally, the third relation, labeled A_3, consists of the condition ($DRDYp$ = '1') and the assignment of the value read from the data port $DATAp$ to the variable $DataVar$. In a similar manner, two relations are constructed for protocol P_b as shown in Figure 3(b).

4.2 Partitioning relations into blocks

Having derived the set of relations for the two protocols, we now need to group the relations in the two protocols into a set of relation groups. A *relation group* is an ordered subset of the set of relations that represents a unit of data transfer between the two processes. The relation groups are created in such a manner that the size of the data "generated" by the relations in the group from one protocol is identical to that expected by the relations in the group from the other protocol.

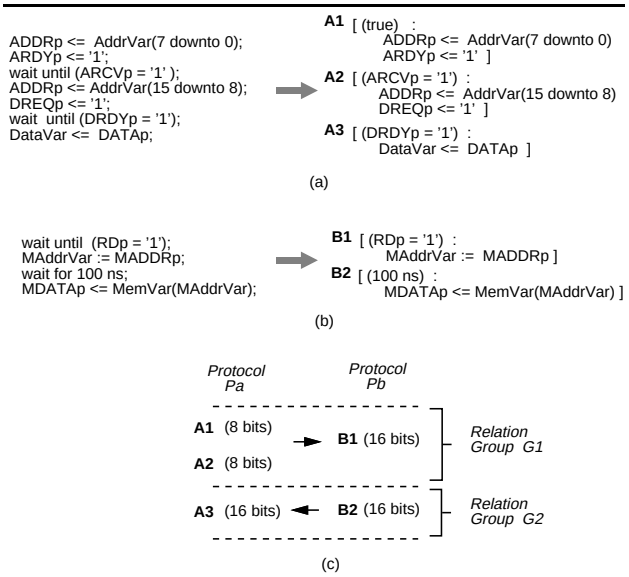

Figure 3: Deriving and partitioning of relations: (a) deriving relations for protocol P_a, (b) deriving relations for protocol P_b, (c) partitioning relations of two protocols into relation groups G_1, G_2.

Figure 3(c) shows how the relations of the two protocols are partitioned into relation groups. The relations of both protocols are listed with the number of bits transferred by the operations in each relation enclosed within parentheses. Relation B_1 of protocol P_b reads 16 bits of data from $MADDRp$. Scanning the list of relations for P_a, we see that both relations A_1 and A_2 together output 16 bits of data. Thus, the first relation group, G_1, consists of the relations A_1, A_2 and B_1. The ordering of relations belonging to two protocols within a relation group is determined by the data dependencies between the relations. B_1 can read the 16-bit address only when A_1 and A_2 have generated it. Therefore, A_1 and A_2 precede B_1 in relation group G_1:

$G_1 = (A_1\ A_2\ B_1)$

Continuing in a similar manner, we create another relation group, G_2, by merging relations A_3 and B_2. Since the 16-bit data is generated by the operations in relation B_2 and read by the operations in A_3, B_2 precedes A_3 in relation group G_2, i.e.:

$G_2 = (B_2\ A_3)$

4.3 Generating the interface process

Having combined the relations into a set of relation groups, $G = \{G_1, G_2\}$, we now generate the interface process to make the two protocols compatible. The set of operations in the relation groups taken in order represents the sequence of atomic operations across the two protocols. The interface process can be obtained by simply inverting each operation in the relation group. "Inverting" an atomic operation means replacing it with its exact *dual* or complementary operation.

Atomic operation	*HDL equivalent*	*Dual operation in Interface Process*
waiting for event	wait until (Cp = '1')	Cp <= '1'
assign control line	Cp <= '1'	wait until (Cp = '1')
read data line	var <= Dp	Dp <= TempVar
assign data line	Dp <= var	TempVar := Dp
fixed delay	wait for 100 ns	wait for 100 ns

Figure 4: Duals of atomic protocol operations.

Figure 4 shows the corresponding dual operation for each of the five atomic protocol operations. For example, waiting for an event on an input control port, Cp is represented in the interface process by its dual, i.e., an assignment to the control signal Cp. The atomic operation which assigns a value to a control line Cp has, as its dual in the interface process, an operation that waits for the same control line to attain that value. Assignments to a data port by a protocol are represented as reading the value from the data port into a local variable, *internal* to the interface process. Reading the value from a data port by a protocol is represented in the interface process by an assignment to the data port from an internal variable.

The delay operation is its own dual. To see when a delay operation is included in the interface process, consider an atomic operation o_1, which represents a delay in one protocol and operation o_2, which waits for an event on an input control line in the other protocol. If o_1 is followed by o_2 in the same relation group, then the dual of the delay operation o_1 is included in the interface process. This ensures that operation o_2 in the other protocol does not execute prematurely. For example, in Figure 3(c), relation group G_2 consists of relations B_2 and A_3. According to the definition of the relations in Figure 3, we can observe that the condition ($100\ ns$) in relation B_2 is followed by a wait for condition ($DRDY\_P$ = '1') in relation A_3. To make sure that protocol P_a does not read the data lines $DATAp$ before P_b can output the data on the lines, the delay operation must be included in the interface process.

The interface process, IP, is obtained by inverting the operations in the relation groups determined in the previous step:

$$\begin{aligned} IP &= (G_1')\ (G_2') \\ &= (A_1'\ A_2'\ B_1')\ (B_2'\ A_3') \end{aligned}$$

Replacing each operation in the above relations by the corresponding duals, we obtain the interface process of Figure 5(a). For example, consider relation A_1, which consists of the operations representing the first two statements of protocol P_a in Figure 3(a):

```
ADDRp <= AddrVar(7 downto 0);
ARDYp <= '1';
```

A_1' is the dual of these two operations, resulting in the following statements:

```
TempVar1(7 downto 0) := ADDRp;
wait until (ARDYp = '1');
```

where, $TempVar1$ is a variable internal to the interface process. These statements are the first two of the interface process, shown in Figure 5(a). Note that in the interface process, the *wait* statement above is swapped with the assignment to $TempVar1$ to ensure that $TempVar1$ is not loaded with a new value before the control signal $ARDYp$ is set to '1'. When the dual operations are generated for any relation, any resulting wait statement precedes all other operations in the interface process.

Any internal variables required by the interface process are declared within the process. Port declarations for each control and data line of both protocols are added to the interface process with the direction reversed (i.e., an "in" port of a protocol is declared as an "out" port in the interface process, and vice versa). Finally, the control and data ports on each of the two processes are connected with the corresponding ports on the interface process.

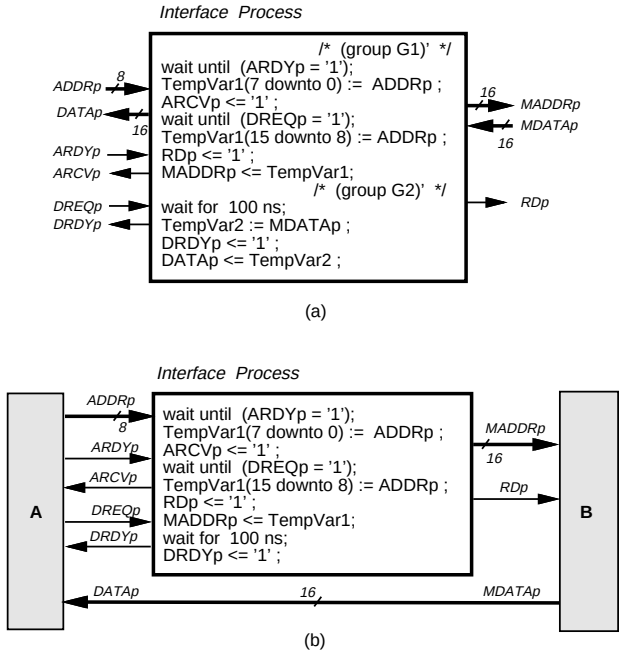

Figure 5: (a) Interface process with dual of operations in blocks G_1 and G_2, (b) interface process after interconnect optimization.

4.4 Interconnect optimization

The interface process generated in the previous step requires that all data and control lines of both protocols be connected to it, as shown in Figure 5(a). In some cases, it may be possible to connect some of the control and data ports of the two communicating processes directly, effectively bypassing the interface

process entirely. This has two advantages. First, it simplifies the interconnect in the system by reducing the number of nets in the system. Second, operations related to these ports in the interface process can be deleted altogether. This will result in a more efficient interface process, both in terms of size and performance, when hardware is synthesized.

The first type of interconnect optimization attempts to reduce the data lines connected to the interface process. Assume that two data ports, D_1 and D_2, belonging to the two protocols have the same size. If the interface process writes to port D_1 every time it reads a value from port D_2, and there is no delay (i.e., "wait" statement) between the operations, then the data ports on the two protocols can be connected directly. All writes to port D_1 and reads from D_2 can be eliminated from the interface process. Consequently, the variable generated for temporarily holding the value transferred between the ports can be eliminated. In Figure 5(a), the ports $MDATAp$ and $DATAp$ have an identical size of 16 bits, with no delay between reading of a value from $MDATAp$ and writing it to $DATAp$; consequently, these ports can be connected directly. In addition, the temporary variable $TempVar2$ and statements in the interface process for reading and writing to these ports can be eliminated.

The second type of optimization examines the control ports of the two protocols. Consider two control ports, C_1 and C_2, from each of the two protocols. If, every time the interface process waits for a particular value on C_1 it updates C_2 with the same value, and there exist no data read/write operations after the wait statement for C_1, then the ports C_1 and C_2 can be connected directly.

The optimized interface process generated for protocols P_a and P_b is shown in Figure 5(b). The data ports $MDATAp$ and $DATAp$, are connected directly. No control optimization was applicable in this example.

Algorithm 4.1 : Generate Interface Process

```
/* generate relations for each protocol */
R_a = CreateRelations(P_a)
R_b = CreateRelations(P_b)

/* partition relations into relation groups */
G = GroupRelations(R_a, R_b)

/* add dual of each operation in G */
for each relation group G_i ∈ G loop
   for each relation R_j ∈ G_i loop
      for each atomic operation o_k ∈ R_j loop
         AddDualStatement(IP, o_k)
      end loop
   end loop
end loop

CreateAndOptimizePorts(IP, P_a, P_b)
```

4.5 Summarizing Interface Process Generation

Algorithm 4.1 summarizes the steps involved in interface process generation. Given the HDL description of a protocol, $CreateRelations$ generates the set of relations that represent the protocol. The procedure $GroupRelations$ partitions the set of relations R_a and R_b into a set of relation groups represented by G. For each atomic operation of a relation in a relation group taken in order, the procedure $AddDualStatement$ adds the corresponding dual statement, as determined from Figure 4, to the interface process, IP. Once the statements for the interface process have been generated, $CreateAndOptimizePorts$ generates the set of ports between the two protocols and the interface process, and optimizes them if possible.

5 Experiments

The Interface Process Generation approach has been implemented (in approximately 9,000 lines of C) and integrated into the SpecSyn system design framework [7]. We tested our interface process generation technique by interfacing several pairs of incompatible protocols. For each pair, the two protocols were specified with VHDL sequential statements. For each experiment, Figure 6 shows the number of data, control and address ports on each of the two protocols (P_a and P_b) and the generated interface process, the number of bits required for storing temporary variables in the interface process and the number of ports in the two protocols that were directly connected after interconnect optimization.

	2−phase to 4−phase	16−bit to 8−bit Handshake	RP Program Bus to Memory Expansion Bus
Protocol Pa ports	32 data 2 control	8 data 2 control	22 address 32 data 2 control
Protocol Pb ports	32 data 2 control	16 data 2 control	22 address 16 data 1 control
Interface Process ports	4 control	24 data 4 control	48 data 3 control
Interface Process storage	−	16 bits	32 bits
Directly Connected Ports After Interconnet Optimization	32 data	−	22 addrs

Figure 6: Experiments with Interface Process Generation

The 2-phase and 4-phase protocols, shown in Figure 7(a) were adapted from [2]. The two protocols are used for handshaking between two processes that need to exchange data, which in this case is sent from a process with the 4-phase protocol to the process with the 2-phase protocol. The 2-phase protocol initiates an operation whenever a rising transition on $REQ2p$ is detected. The data on the 32-bit $DATA2p$ port is read into a local variable, $DataVar$, following which the acknowledge signal $ACK2p$ is asserted. The 4-phase protocol is identical to the 2-phase protocol except that it requires that the two control signals, $REQ4p$ and $ACK4p$ to return to logic '0' before another operation can be initiated. The interface process generated

is shown in Figure 7(b). The two data ports were directly connected after interconnect optimization, and thus, no internal variables were required by the interface process.

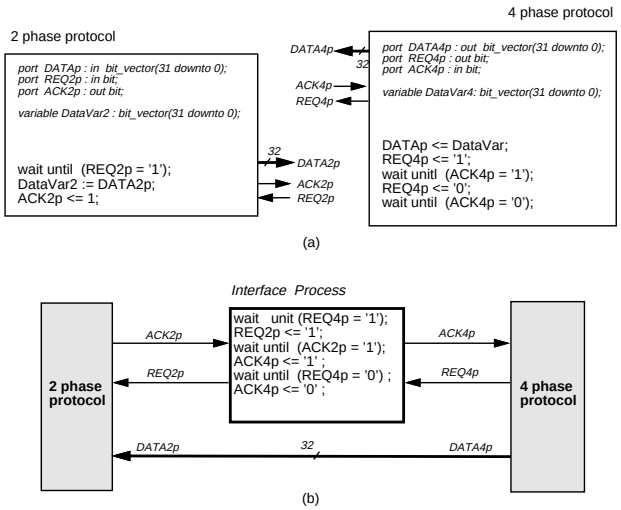

Figure 7: Interfacing 2-phase and 4-phase protocols: (a) protocol descriptions, and (b) interface process.

In the second experiment, we interfaced two handshake protocols with mismatched data widths of 16 and 8 bits respectively. A 16-bit temporary variable is synthesized for the interface process to store the data read from the 16-bit protocol, and send it to the 8-bit protocol in 2 transfers of 8-bits each.

Finally, we interfaced the 32-bit program bus (2 control lines) of the RP RISC controller and digital signal processor [8] with the 16-bit wide memory expansion bus (1 control line). Thus, a single instruction fetch over the program bus results in 2 accesses to the program memory over the memory expansion bus. A 32-bit temporary variable was required in the interface process to assemble the data fetched from the memory before transmitting it over the program bus. After interconnect optimization, the two 22-bit address ports were connected through directly.

6 Conclusion

In this paper we examined the effects on communication of binding portions of system specifications to off-the-shelf components. We described a new technique for interfacing two fixed, incompatible protocols by generating an interface process between them. Once the interface process has been generated, hardware for it can be obtained using HDL based synthesis tools.

We believe that the Interface Process Generation technique is significant and unique for several reasons. Firstly, it is the first approach to solving protocol incompatibility in the behavioral domain. This allows simulation of the interface process with the HDL descriptions of the two incompatible protocols (or even with HDLs models of the components being interfaced) to check system functionality. Second, other than the HDL description of the two protocols, the designer is *not* required to specify any additional information that are required by previous approaches (such as merge labels, extra state machines, interconnection of ports, sequences of data transfers etc.). Finally, the Interface Process Generation approach can interface protocols with different data widths, a capability not addressed by previous approaches.

We plan to extend the Interface Process Generation technique in several directions. One of the limitations of the Interface Process Generation method is that the timing information supported takes the form of non-overlapping delays between protocol operations. Since only an HDL description of the interface is generated, minimum and maximum timing constraints between events are not supported. Currently, these constraints can be passed on to the synthesis tool that will synthesize the hardware for the interface process. We are researching ways of incorporating detailed designer-specified timing constraints directly in the generation of an interface process. Second, optimizations that can be applied to interface processes generated to make two protocols compatible need to be studied. An example of such an optimization might be the minimization of the number and size of variables used by the interface process to reduce the size of the hardware that will implement the interface process. Finally, we are examining ways by which more complex protocols that have multiple modes of operations (master, slave, etc.) can be interfaced using the interface process generation technique.

References

[1] G. Borriello and R. Katz, "Synthesis and optimization of interface transducer logic," in *Proc. of the ICCAD*, 1987.

[2] G. Borriello, *A New Interface Specification Methodology and its Applications to Transducer Synthesis*. PhD thesis, Univ. of California, Berkeley, 1988.

[3] J. Akella and K. McMillan, "Synthesizing converters between finite state protocols," in *Proc. of the ICCD*, 1991.

[4] J. Akella, *I/O Performance Modeling and Interface Synthesis in Concurrently Communicating Systems*. PhD thesis, Carnegie Mellon Univ., 1991.

[5] J. Sun and R. Brodersen, "Design of system interface modules," in *Proc. of the ICCAD*, 1992.

[6] J. Sun, M. Srivastava, and R. Brodersen, "SIERA: A CAD environment for real-time systems," in *3rd Physical Design Workshop*, 1991.

[7] D. Gajski, F. Vahid, and S. Narayan, "A system-design methodology: Executable-specification refinement," in *Proc. of European Design & Test Conf.* 1994.

[8] "RSP Engineering Report." Rockwell Intl., 1991.