Communication Synthesis for Reuse

Jon D. Kleinsmith
Daniel D. Gajski

Technical Report ICS 98-06
February 1998

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92692-3425, USA
(714) 824-8059

jon@ics.uci.edu
gajski@ics.uci.edu

Abstract - In this report we discuss a set of techniques needed to generate and synthesize communication interfaces in a System Design context. Given a behavioral specification, we present the transformations necessary for generating a communication model containing channels and protocol. This work is being conducted in conjunction with codesign tools being developed in the CADLAB at the University of California, Irvine.

I. INTRODUCTION

The increasing complexity of mixed hardware/software systems and short time-to-market demands requires that tools and methodologies be developed to support this co-design at higher levels of abstraction in order to allow the designer to rapidly specify, explore and refine a given product. In a system design methodology, a given design can be described as a set of interacting behaviors each communicating with the others [1]. The communication between these behaviors can take on many forms, from global memory accesses to complex protocol over common buses. At the system level, communication can be seen as auxiliary behavior which may be synthesized into hardware or translated into software within the system.

The designer may spend a great deal of time defining and specifying protocols and communication interfaces between system components. It is our intent to shorten this task and relieve the designer of some of the tedium of communication synthesis including protocol generation, communication channel insertion and interface specification. Ideally, a designer should be able to specify the mode and method of communication as part of the system being designed. However, this assumes that the final partitioning of the system has been completed in the designer's mind and will not change during synthesis.

More realistically, though, a system partitioning will evolve through allocation, scheduling and estimation, until a viable partition is reached. At this point, shared variables across partitions can then be examined. These variables may then be mapped to the local memories of individual processing units (PEs) and, once an architecture has been established, a message-passing scheme may be imposed allowing variable sharing through allocated buses. References to shared variables must be replaced by protocol-specific functionality that will allow transfer of these variables across a communication channel between the components.

In order to achieve a high level of reuse, change to the system must be compartmentalized or limited in scope such that change remains local. The model and methodology presented in this paper ensure that system functionality is separated from communication in such a way that system components may be easily replaced. Communication between components is then generated in accordance with their individual protocols.

In this paper, we introduce a communication synthesis methodology and describe the steps involved to take a system-level description and refine it into a model that contains the behavior necessary to describe detailed communication between components including complex protocol generation. An example of the channel insertion and refinement process will be presented in order to illustrate the details of the methodology.

In the first section, we present the communication synthesis methodology and give a brief overview of each step in the process. Next, we discuss communication refinement, wherein shared variables are replaced by abstract channels and remote procedure calls enact communication between components. The *Channel Insertion* algorithm is developed and discussed. We then describe the two remaining steps in communication synthesis, inlining and interface generation. Algorithms and examples of both are introduced and detailed.

II. COMMUNICATION SYNTHESIS

Communication synthesis is an integral step in a system design methodology. As proposed in [1] and [3], communication synthesis is performed after the greater portion of the system has been specified and behavior mapped to components in a selected architecture. In this methodolgy, the designer specifies the behavior or functionality of the system using a specification language, in particular SpecC. Following specification, a target architecture is generated through the *allocation* of processing elements from a library of such components. The specification must then be mapped or *partitioned* onto these processing elements. Allocation and partitioning may be repeated until a feasible architecture is achieved. At this point, one or more distinct behavioral tasks may have been mapped to a single processing element. If this is the case, an ordering or *scheduling* of these tasks must be performed while maintaining correct functionality and satisfaction of system constraints.

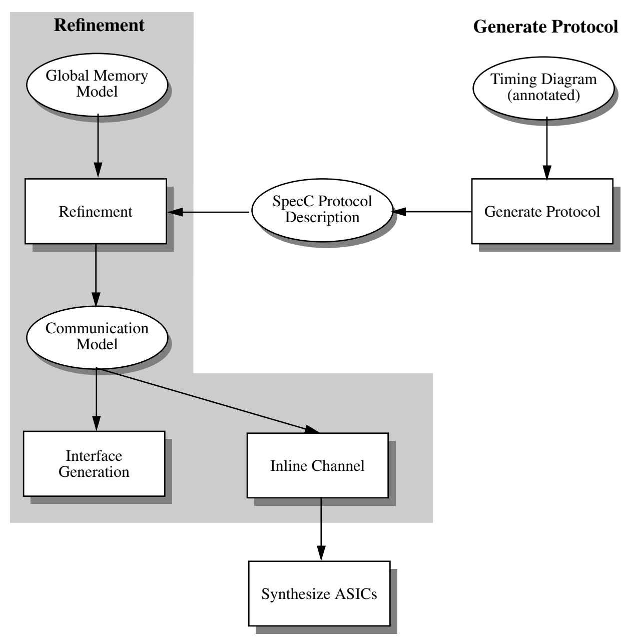

Fig. 1. Communication Synthesis Flow

Communication synthesis then begins with a partitioned specification, wherein various components or behaviors entities interact and communicate via shared variables left over from the original, unpartitioned specification. It is assumed that scheduling has already been performed on the partitioned model. This specification is referred to as the **Global Memory Model** in *Figure 1* and can be seen graphically in *Figure 3*.

The goal of communication synthesis should be to produce a complete specification, describing the structure and functionality of the system. Hardware objects in the specification should be synthesizable and ready for hand-off to available synthesis tools. Software elements should be compiler ready.

Communication synthesis consists of five main tasks:

A. Generate protocol.

The first task in communication synthesis is the generation of protocols, that is, communication protocols must be described in a form usable by high-level synthesis tools and yet suitable for simulation. This process starts with an annotated timing diagram describing the given protocol. This diagram would not only include the temporal state of signals but the causal relationships between them with associated delay as seen in *Figure 2*. Using the causal relationships, the various events (signal changes) can be grouped and scheduled according to direction, concurrency and dependencies. This schedule can then be translated into a behavioral specification, using an FSM model to delineate the various states in the schedule in order to ensure synthesizability.

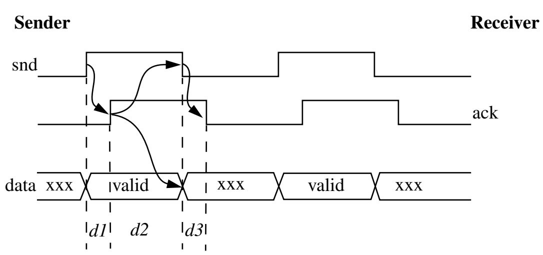

Fig. 2. Annotated Timing Diagram

B. Channel Refinement.

Starting from a partitioned behavioral specification with components communicating through shared global variables, the specification will undergo a series of transformations that result in a model in which each shared variable is encapsulated by an abstract communication channel. The channel controls access to the variables and captures the functionality of a specified protocol. In addition, the component is modified to reflect this design decision. We use a remote-procedure-call (RPC) scheme to facilitate the passage of data from one component to another. The channel object maintains the definition of the remote functions which describe the channel protocol. The process of incorporating channel objects into the specification is discussed in the next section. This process is referred to as channel refinement.

The designer must select one or more channel objects from a library of channels. These channels can either be generic, containing the description for a simple bus and handshake protocol, or they may describe complex communication via PCI, VME or some such standard protocol. Additionally, the designer may have a custom scheme in mind which too, may be part of the library. In any case, channel(s) must be allocated for the system. Next, shared variables between components must be partitioned to one or more instances of an allocated channel, meaning one or more channels will exist between communicating components. Once variables have been encapsulated by channels, channels may be examined for compatibility, throughput and performance. The designer may wish to group or *merge* channels based on width, variable lifetime, access frequency or some other metric. Channels merged in this fashion become *superchannels* and may require resolution or arbitration between multiple components. Details of channel merging and related design issues are not discussed in this paper and are the subject of future work.

C. Interface Generation

In the case of previously designed components, those having their own established communication protocols, interfaces to the remaining components in the system must be generated and can take the form of transducers [2] between components. The transducer object may be viewed abstractly as a set of dual, ordered relations between opposing signal groups. That is, a particular "view" of the protocol, either on the sender side or receiver side, is captured and reversed. As such, the transducer generates the signals necessary to satisfy the protocols of either component it is linking. This object can then be realized as a FSM and may be synthesized as another hardware component, merged with another synthesizable component or be translated to software running on an associated processor [4].

D. Inline Channel.

Behavior that has been partitioned and allocated to an undefined component may have its communication functionality inlined. Namely, the behavior or logic necessary to initiate a communication transaction, formerly located in the channel object is placed inside the partition that utilizes that functionality. This communication behavior can the be handed-off to be synthesized with the rest of the component's functional behavior.

E. Synthesize ASICs.

Finally, after protocols have been inlined and transducers have been generated, synthesis of the undefined behavioral components, including interface components, may be completed using current high-level synthesis techniques.

In the next section, the refinement process is more fully explained and an example is given, illustrating the steps involved.

III. COMMUNICATION REFINEMENT

Communication refinement begins with a partitioned specification which is assumed to share data through common, global memory objects as shown in *Figure 3*, and a behavioral description of the protocols to be used. Transformations are then performed on the specification in order to remove global variable references and replace them with the appropriate behavior to realize the given protocol.

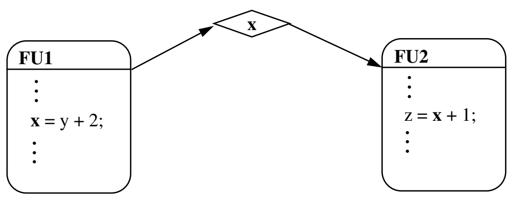

Fig. 3. Global Memory Model

We begin with the partitioned behavioral specification of a system containing two components, *ASIC1* and *ASIC2*. These components are communicating through the global variable *Gdata* which is an array of bytes and can been specified as follows:

```
Entity System is concurrent subbehaviors
      ASIC1;
      ASIC2;
end System;

type byte is bit_vector(7 downto 0);
variable Gdata is array(1 to n) of byte;

ASIC1 : process
      variable local_byte byte;
begin
      for i in 1 to n loop
         -- generates byte of data;
         Gdata[i] <= local_byte;
      end loop;
      for i in 1 to n loop
         Gdata[i] <= Gdata[i] XOR MASK1;
      end loop;
end ASIC1;

ASIC2 : process
      variable x byte;
begin
      while (!done) loop
         x := Gdata[i] AND MASK2;
         i := i+1;
         if (condition1) then
            done := 1;
         end if;
      end loop;
end ASIC2;
```

The protocol specification would have been generated from a timing diagram that has been annotated to indicate signal direction as well as causal relationships between signals. The designer may generate a protocol specification by dividing the timing digram into a set of events in which independent signal assignments may occur. Causal dependencies between signals result in wait statements and act as boundaries between events. For example, a protocol similar to the one presented in *Figure 2* may be decomposed and would result in the following procedures to send and receive *data* across the signal *port*. This protocol is a simple asynchronous scheme where the sending process raises a signal *snd* and applies the *data* to the data *port*. These signals will be maintained until the data is received and an acknowledgment signal *ack* is raised indicating that data is no longer needed. At this point *snd* and *data* are lowered or are no longer valid. Conversely, the receiving process waits for a valid data which is recognized through a raised *snd* signal. After latching the data, the receiver will raise an acknowledgment signal *ack* to complete the handshake.

```
send_byte(byte data, byte port) is
begin
      snd <= '1';
      port <= data;
      wait on ack;
      snd <= '0';
end send_byte;

receive_byte(byte data, byte port) is
begin
      ack <= '0'
      wait on snd;
      data <= port;
      ack <= '1';
      wait on !snd;
      ack <= '0';
end receive_byte;
```

A. Channel Insertion

The operations necessary to transform the global memory model begin with the replacement of each global variable reference within each component of the system. A component containing such a reference must have a local variable instantiated representing the global variable. Structurally, this has the effect of placing the variable in a memory local to the functional unit. Next, each occurrence of the global variable as referenced in the functional description is replaced with a reference to the local variable.

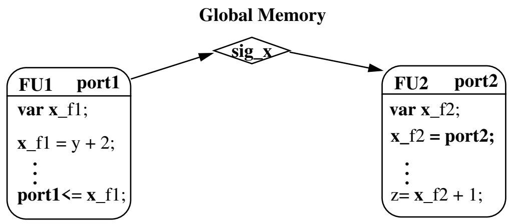

Fig. 4. Local Memory Model

Next, each component is examined for instances of global variable references. If found, a local variable is inserted and each global reference is replaced by a reference to the local variable. This is illustrated in *Figure 4*, and can be seen textually in the code segment below.

```
signal byte_port is byte;
--variable Gdata is array(1 to n) of byte;
ASIC1 : process
  variable local_byte byte;
  variable ASIC1_Gdata is array(1 to n) of byte;
begin
    for i in 1 to n loop
      -- generates byte of data;
      -- Gdata[i] <= local_byte;
      ASIC1_Gdata[i] <= local_byte;
    end loop;
    for i in 1 to n loop
      -- Gdata[i] <= Gdata[i] XOR MASK1;
      ASIC1_Gdata[i] <= ASIC1_Gdata[i] XOR MASK1;
    end loop;
    for i in 1 to n loop
      send_byte(ASIC1_Gdata[i], port);
    end loop;
end ASIC1;
```

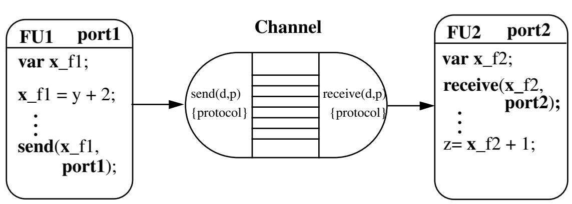

Fig. 5. Channel Placement

Next, *send* and *receive* calls must be inserted in the partitions for each global variable. Send and receive represent remote procedure calls to the behavioral specification of the protocol. In partitions acting as producers, a send call in appended, whereas, consumers require a prepended receive call. This can be seen in *Figure 5*.

```
ASIC2 : process
    variable ASIC2_Gdata is array(1 to n) of byte;
    variable x byte;
begin
    for i in 1 to n loop
      receive_byte(ASIC2_Gdata[i], port);
    end loop;
    while (!done) loop
      -- x := Gdata[i] AND MASK2;
      x := ASIC2_Gdata[i] AND MASK2;
      i := i+1;
      if (condition1) then
        done := 1;
      end if
    end loop;
end ASIC2;
```

Finally, ports are added as described in the protocol specification and width differences between the transferred objects and the ports are resolved, inserting loops or padding of the data as necessary.

B. Channel Insertion Algorithm

Based upon the previous description, the channel insertion process may be encapsulated in an algorithm, which will be the basis for a tool to automatically perform the necessary transformations on a given specification. The channel insertion algorithm takes as input a protocol specification containing send and receive procedures and a partitioned system description communicating through global variables and returns a communication model wherein components communicate across channels via send and receive procedures.

```
Algorithm : Channel Insertion
Build(Global_Var_List);
foreach Component in Specification loop
  foreach Global_Var in Global_Var_List loop
    if Find_Reference(Global_Var, Component) then
      Declare_Local_Var(Global_Var, Component);
      Replace_Global_References(Global_Var, Component);
      if Is_Producer(Global_Var, Component) then
        Append(Protocol.send, Component);
      else
        Prepend(Protocol.receive, Component);
      end if;
      Resolve_Variable_Width(Global_Var, Port);
    end if;
  end loop;
end loop;
Insert(Channel_Specification, Specification);
return Specification;
```

IV. CHANNEL INLINING

Following channel replacement the model, as shown in Fig. 5, contains remote procedure calls that facilitate communication across channels between components. Before we can pass this model on for further synthesis, the channel must be resolved to a bus architecture and the remote procedure calls must either be inlined in the calling component or encapsulated in a transducer object that will in turn become the basis for a component's interface.

In the current model, components can be divided into two classes, fixed or synthesizable. Fixed components are those that will later be realized by *Intellectual Properties (IPs),* physical objects that have previously been synthesized and have an internal protocol that must be followed. This includes processor elements wherein the behavior will be mapped to software. In this case, the channel protocol will need to be resolved against the component's protocol and a transducer or interface will be generated.

On the other hand, synthesizable components are those objects in the partition that have not already been realized in silicon and are still malleable in that we may define the component's protocol in terms of the channel's protocol. Thus, we will *inline* the channel protocol functionality into the behavior of the component.

Initially, the each behavior in the specification must be identified as either synthesizable or fixed. In the case of the later, interfaces will need to be developed to match the protocol as discussed in the next section. If a behavior is synthesizable, then the interface functional definition, either *send* or *receive* is placed inside of the component description. Next, ports are generated to accommodate the signals within the communication function. In the previous example, should we inline the *send* function, ports would need to be declared for *snd, ack,* and *data*. Connectivity to the rest of the system would then be resolved through these ports. Finally, the original function call within the behavior must be updated to match the local function name. This procedure can be expressed in an algorithm.

Algorithm : Channel Inline
foreach Component **in** Specification **loop**
  if Is_Synthesizable(Component) **then**
    if Is_Producer(Component) **then**
      Inline(Protocol.send, Component);
      Add_ports(Protocol.send, Component);
    else
      Inline(Protocol.receive, Component);
      Add_ports(Protocol.receive, Component);
    end if;
    Resolve_function_calls(Component);
    Update_connectivity;
  end if;
end loop;
return Specification;

The resulting specification would appear graphically as shown in *Figure 6*.

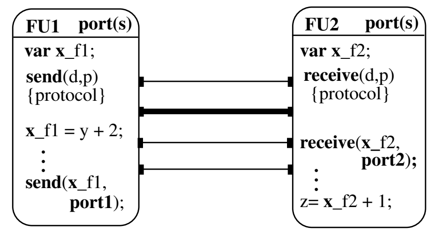

Fig. 6. Inlined Communication Behavior

V. INTERFACE GENERATION

The interface generation process is intended to develop interface transducers between fixed components with differing protocols. We will illustrate the process of transduction through an example and later develop an algorithm that encapsulates this process.

The process begins with the existence of two differing protocols between fixed components. Since we will not be able to incorporate the protocol of one component into the other as was the case between sythesizable objects, a transducer will be generated tying the two protocol together.

In order to perform transduction, the protocols in question must be annotated or otherwise specified to indicate causal relationships between signals. From such an annotated diagram as seen in *Figure 7*, a protocol specification may be derived as timed or scheduled signal assignments. An example of such a protocol specification can be seen below.

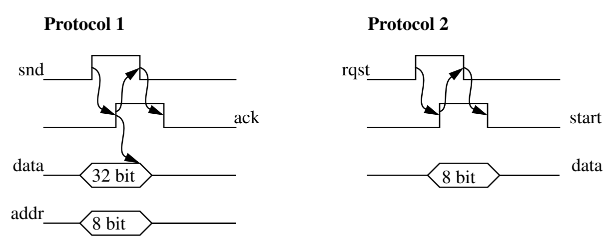

Fig. 7. Disjoint Protocols

Protocol 1
ports(RDYp **out** bit;
    DATAp **out** bit_vector(31 downto 0);
    ADDRp **out** bit_vector(7 downto 0);
    ACKp **in** bit);
begin
  RDYp <= '1';
  DATA1p <= Data_var1;
  ADDRp <= Addr_var1;
  wait until (ACKp = '1');
  RDYp <= '0';
  wait until (ACKp = '0');
end;

```
Protocol 2
ports(RQSTp out bit;
        ACKp in bit;
        DATA2p in bit_vector(7 downto 0));
begin
  RQSTp <= '1';
  wait until (STARTp = '1');
  Data_var2 <= DATA2p;
  RQSTp <= '0';
end;
```

From this specification, we must derive an interface process that acts as the dual of each protocol thus acting as an intermediary between the two communicating processes. To accomplish this, ordered relationships between signals must be obtained. An ordered relation in the specification can be defined as one or more statements having a common causal action. In other words, we will impose an ordering on the protocol specification that groups statements according to a common event.

In our example, timing or wait statements act as natural borders between relation blocks. Thus we could have a grouping as seen in *Figure 8*.

```
Protocol 1

  begin
    RDYp <= '1';
    DATA1p <= Data_var1;          Block 1
    ADDRp <= Addr_var1;
    ---------------------------------------
    wait until (ACKp = '1');      Block 2
    RDYp <= '0';
    ---------------------------------------
    wait until (ACKp = '0');      Block 3
  end;

Protocol 2

  begin
    RQSTp <= '1';                 Block 1
    ---------------------------------------
    wait until (STARTp = '1');
    Data_var2 <= DATA2p;          Block 2
    RQSTp <= '0';
  end;
```

Fig. 8. Relationship Groupings

In Block 1, from the figure, we see that each assignment executes upon entry to this block and are dependent only upon entry to that block. The statements in Block 2, following the *wait* statement are dependent upon the signal on *ACKp*.

To form the interface to these blocks, we take the dual operation of each statement and place it in the new interface behavior. The dual of a given statement can be viewed as a complementary operation that take one of several forms. Specifically, operations can be divided into three main categories: control assignment, data assignment and fixed delay. A control assignment is identified as assignment of a constant value to a port or signal. Data assignment occurs when a local variable is assigned to or written from a port. Finally, fixed delay takes the form of any wait statement that contains unconditional delay, in that the wait is independent of signal events. The dual of any of these operations can be found by obtaining the inverse relation between assignment and waiting for events. Namely, for a control assignment, the dual is obtained by waiting for the assigned event to occur on that signal. For instance, in our example, the dual of the first statement in Protocol 1, "*RQSTp <= '1';*", would be constructed as: "**wait until** (RQSTp = '1');". Additionally, the reverse also holds true, and conditional wait statements can be dualed to obtain control assignments. Variable assignment from a port translates as a dual statement where the variable in question is written to the port. Again, the reverse holds true as well. In such a case, each variable in the original spec must be replaced by a temporary variable in the interface.

The interface is formed when dual statements are found for each statement in the original protocol specifications and are used to form the new interface behavior, first for the sending protocol then for the receiving protocol. In this way we allow the capture of data from one component, temporary storage of transmitted data and then the transmission to the other component.

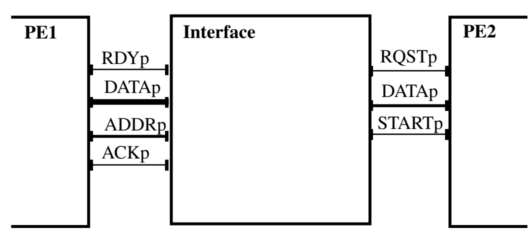

Fig. 9. Interface Connectivity

Next, we place each port from the protocol specifications into the interface. Since we have taken the dual of each of their assignments, we must inverse their in/out status. Finally, we must resolve data width conflicts between protocols. In the example, the transmitted data is 32-bit while the receiver expects 8-bits of data. In order to send the data, the receiver must be sent portions of the data until it is all received. To accomplish this, a loop will enclose the receiver's dual statements with loop control and data widths resolved through modulo arithmetic. This process was performed for our example and the resulting interface process can be seen in the following code segment and graphically in the *Figure 10*.

```
Interface Process
ports(RDYp in bit;
        DATAp in bit_vector(31 downto 0);
        ADDRp in bit_vector(7 downto 0);
        ACKp out bit;
        RQSTp in bit;
        ACKp out bit;
        DATA2p out bit_vector(7 downto 0));

variable Temp1 bit_vector(31 downto 0);
variable Temp2 bit_vector(7 downto 0);
variable n integer;

begin
  wait until (RDYp = '1');
  Temp1 <= DATA1p;
  Temp2 <= ADDRp;
  ACKp <= '1';
  wait until (RDYp = '0');
  STARTp <= '1';
  DATA2p <= Temp2;
  wait until (RQSTp = '0');
  STARTp <= '0';
  while n <= 24 loop
    STARTp <= '1';
    DATA2p <= Temp1(7+n downto 0+n)
    n := n+8;
    wait until (RQSTp = '0');
    STARTp <= '0';
  end loop;
end;
```

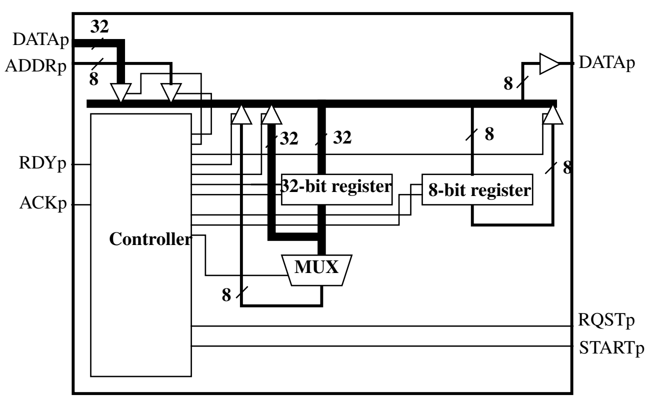

Fig. 10. Interface Diagram

The example shows the result of the interface generation process. We can express this process in terms of the *Interface Generation Algorithm* presented below. It should be noted. however, the some amount of user interaction is currently required to make this generation possible. It needs to be determined which signals in the protocol specification are actual data objects being passed. At this time, we do not have a method that automates this recognition, however, if naming conventions are adhered to, it may be possible to enable this recognition by an automated tool.

```
Algorithm : Interface Generation
begin
  Gsend = ID_Relation_Groups(Protocol.sender);
  Grec = ID_Relation_Groups(Protocol.receiver);
  Gsend' = Dual(Gsend);
  Grec' = Dual(Grec);
  if (Protocol.sender.data > Protocol.recever.data) then
     Add_loop(Grec');
  else
     Add_loop(Gsend');
  end if;
  Interface = Gsend' + Grec';
  Add_ports(Interface, Protocol.sender, Protocol.receiver);
end;
```

VI. CONCLUSION

In this paper we have presented a communication synthesis methodology that guides the refinement of a system specification, transforming it into a communication model containing the behavior necessary to capture complex communication protocols between the various system components. In this way, we can allow for more accurate simulation using the channel mechanism presented in this paper. Specifically, protocols must be captured into the specification language, the system specification must be refined to include the protocol and finally, transducers and refined components must be generated for later synthesis and production. Future work will include a more detailed analysis and experimentation with protocol and transducer generation, namely optimization techniques to allow for higher performance during communication, leading towards final system synthesis. Finally, this work is intended to be an integral part of a larger, system design methodology under development at the CADLAB of the University of California, Irvine.

REFERENCES

[1] D.D. Gajski, J. Zhu, R Doemer. "Essential Issues in Codesign." *Hardware/Software Codesign: Principles and Practices.* Kluwer: Boston, MA. 1997.

[2] S. Narayan, D.D. Gajski. "Interfacing System Components by Generation of Interface Processes." *Proceedings of the 32nd Design Automation Conference.* June 1995.

[3] D.D. Gajski, F. Vahid, S. Narayan, J. Gong. *Specification and Design of Embedded Systems.* Prentice Hall: Princeton, NJ. 1994.

[4] G. Borriello, R.H. Katz. "Synthesis and Optimization of Interface Transducer Logic." *Proceedings of the International Conference on Computer-Aided Design.* 1987.