Automatic Communication Refinement for System Level Design

Samar Abdi
Center for Embedded Computer Systems
University of California Irvine
CA 92697 USA
sabdi@cecs.uci.edu

Dongwan Shin
Center for Embedded Computer Systems
University of California Irvine
CA 92697 USA
dongwans@cecs.uci.edu

Daniel Gajski
Center for Embedded Computer Systems
University of California Irvine
CA 92697 USA
gajski@cecs.uci.edu

ABSTRACT

This paper presents a methodology and algorithms for automatic communication refinement. The communication refinement task in system-level synthesis transforms abstract data-transfer between components to its actual bus level implementation. The input model of the communication refinement is a set of concurrently executing components, communicating with each other through abstract communication channels. The refined model reflects the actual communication architecture. Choosing a good communication architecture in system level designs requires sufficient exploration through evaluation of various architectures. However, this would not be possible with manually refining the system model for each communication architecture. For one, manual refinement is tedious and error-prone. Secondly, it wastes substantial amount of precious designer time. We solve this problem with automatic model refinement. We also present a set of experimental results to demonstrate how the proposed approach works on a typical system level design.

Keywords

Model Refinement, System modeling, System level design, Communication, System bus

1. INTRODUCTION

With the increase in complexity of system level designs, we are continuously faced with the challenge of implementing the design specification while meeting the strict constraints it imposes. Communication synthesis requires extensive design space exploration. With a greater number and variety of components being put together on a chip, the task of communication synthesis becomes more complicated. In order to choose the right communication architecture for our designs, we need to generate models that reflect the communication architecture. These models are then evaluated through simulation to test their "goodness".

Typically, these models are handwritten, which poses a number of problems. First of all, a lot of time is spent in writing these models which is a serious handicap to the exploration process. The fewer architectures we test, the lower is the probability of choosing the optimal one. Secondly, model rewriting is an error prone process. It is possible to introduce several errors while manually rewriting the model. This makes the evaluation of our communication architecture questionable.

In this paper we look at how we speed up the communication synthesis process by enabling automatic model refinement. Figure 1 shows how communication synthesis is performed in our system design methodology. We begin with a model of our partitioned system design, which represents various system components executing concurrently. These components communicate with each other via abstract channels as shown. Each channel comprises of the data itself and *Send/Receive* methods that enable the data transaction. The user provides a set of synthesis decisions like bus allocation, connectivity, bus access priorities etc. System buses may be inserted in the design by instantiations from a bus protocol library. With these inputs, the communication refinement tool produces an output model that reflects the bus architecture of the system. In the output model, the top level of the design consists of system components and wires of the system bus(es). The components themselves are refined to their bus functional models that communicate using the system bus(es). The rest of the paper is organized as follows. Section 2 is a brief review of the related work in this area. Section 3 talks about the characteristics of abstract communication in the input model. Section 4 looks at the basic tasks of communication refinement for a simplistic example with two components and one bus. Section 5 builds up on the simple example by adding multiple components and multiple buses. Finally, we present experimental results in section 7 and wind up with a summary and conclusion.

2. RELATED WORK

In recent years a lot of attention has been given to modeling and synthesis of bus architectures. Most of the work has been done in optimizing communication architectures for specific designs. In [7], Gogniat et al. describe mechanism for interfacing HW/SW interfaces for co-design of embedded systems. Ortega et. al look at a retargettable modeling scheme for maximum utilization of bus bandwidth in [8]. However, they focus mostly on reactive real time systems. Vahid and Tauro [10] propose using a parameterized communication library. Rowson et. al propose the Interface based design methodology [9] that also aims at successive refinement of communication from abstract tokens to implementation.

SystemC methodology talks about transaction level modeling in [2] that aims at communication modeling so as to optimize simulation speed. However, it does not address automatic generation of such models. CoWare [1] supports heterogeneous proces-

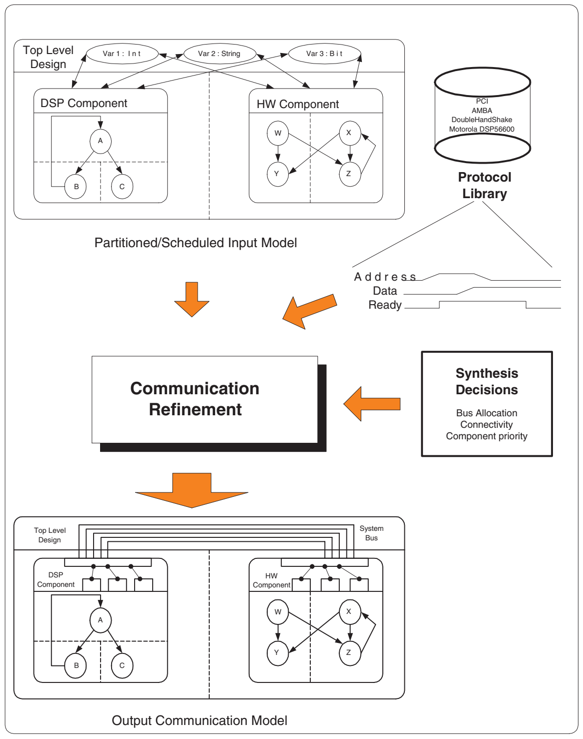

Figure 1: Communication refinement engine

sors but focuses on shared memory communication. Jerraya et al. [4] [6] present interesting schemes for putting together heterogeneous components on a bus using wrappers. SpecC methodology [5] suggests four system level models and proposes refinement-based synthesis approach.

3. INPUTS TO COMMUNICATION REFINEMENT

As discussed earlier and show in Figure 1, we have basically three inputs to the communication refinement engine. The first input is the input model with abstract communication. The second input is a protocol library that supports a variety of protocols including generic and processor specific protocols. The final input is a set of synthesis decisions that guide the communication refinement engine on the required transformations that need to be applied to the input model.

3.1 Input model

The input model to communication refinement should follow certain pre-specified semantics. It should reflect the intended architecture of the system with respect to the components that are present in the design. Each component executes a specific behavior in parallel with other components. Communication inside a component takes place through local memory of that component, and is thus not a concern for communication refinement. Inter-component communication is point-to-point and takes place through abstract channels that support Send and Receive methods.

Communication between components can be modeled via three schemes as shown in Figure 2. In the case of two way blocking communication as shown in Figure 2 (a) , both the sender and receiver must be blocked until the transaction has completed. This mechanism is modeled using events and blocking wait statements.

As we can see, the sender writes the data on a shared variable in the channel and follows up by notifying the receiver. The receiver cannot read the data until it gets the sender's notification. This guarantees the safety of the transaction. The ack event guarantees that the sender cannot rewrite on the channel until the previous transaction has completed. Such a mechanism is deterministic. Non-deterministic communication mechanisms can be seen in Figure 2(b) and Figure 2(c). In Figure 2(b), the one way blocking mechanism is used that ensures that the receiver cannot read the data until it is written by the sender. However, there is no way to stop the sender from re-writing that data in a subsequent iteration. The mechanism shown in figure Figure 2 (c) is completely non-deterministic. These mechanisms are typically used in real time systems where a time out strategy is employed. In the course of this paper, we will look only at refinement of two way blocking communication. The other two mechanisms can be implemented easily once we have support for two way blocking communication.

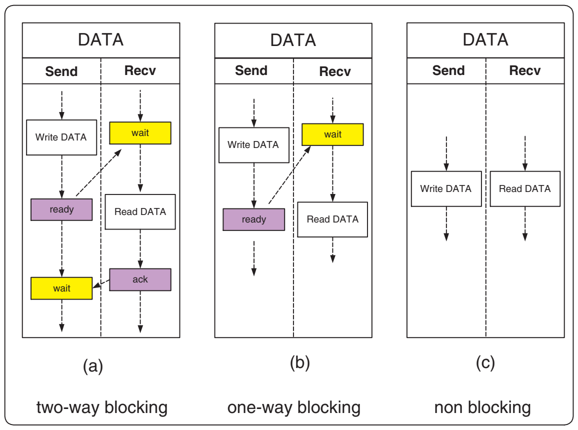

Figure 2: Communication mechanisms in abstract channels

3.2 Protocol Library

The protocol library is a set of channels that model the protocols of system buses. These channels provide for the standard read/write methods for the bus protocol. Additional methods may be required for more complex designs that support arbitration, multiple interrupt signals etc. Each bus transaction also requires definition of a master and slave. Therefore, the protocol library must provide for unique channels for both master and slave sides. The ports of the bus protocol channel represent the actual bus wires which are later exposed in the communication model.

3.3 Synthesis decisions

The refinement engine works on directions given to it by the communication synthesis decisions. The synthesis process can either be automated or interactive as per the designer's methodology. However, the decisions must input to the refinement engine using a specific format. Some typical features of the communication architecture include the choice of system buses, the mapping of abstract communication to these buses, the connectivity between components and buses etc. Based on these decisions, the refinement engine imports the required protocols from the bus protocol library and generates interfaces and drivers for components so that they may talk over the system buses. For the purpose of our implementation, we annotated the input model with the set of synthesis decisions. The refinement tool then detects and parses these annotations to perform the requisite model transformations.

4. REFINEMENT OF A SIMPLE DESIGN

In this section we look at communication refinement of a simplistic model. We will look at the basic tasks involved in the refinement process before moving on to more complex architectures. The design consists of two components (a processor and a HW unit) communicating with two-way blocking channels. All this communication needs to be mapped to a single system bus in order to get a simple bus architecture as shown in Figure 3. Four communication points are shown in the master and slave component. Each communication point is labeled such that node A of master talks to node A of slave, node B of master talks to node B of slave and on. Implementation of data transactions on the system bus is done by the *Application Layer* for that variable. Each component in the design has a unique *Application Layer* for every variable that it sends or receives. The *Application Layer* essentially substitutes the original abstract communication channel by implementing the data transfer on the system bus. Additions made to the model as a result of communication refinement are highlighted in Figure 3.

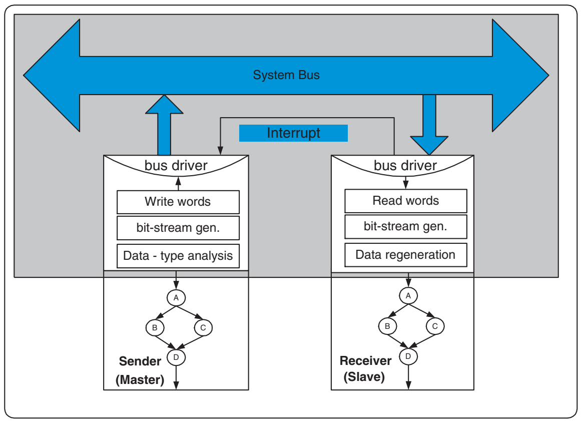

Figure 3: Simple architecture

4.1 Data slicing

The abstract communication channels of the input model perform transaction of complex variables with the help of events. These complex variables could be structures, multi-dimension arrays or integers. Eventually, they need to be translated to a bit-stream to be sent over the system bus. Figure 3 shows how abstract data is processed into data bus words and sent to the receiver. On the receiver side, this bit stream needs to be identified and translated back to the original variable. The entire data slicing forms part of the component's application layer for that variable.

4.1.1 Type analysis

As shown in the Figure 3, data type analysis is the first step on the sender's side. For simplicity, we consider only integral data. The approach can be extended to floating point data or user defined data as well. We begin by doing a *depth first search* on the complex data structure. The key idea is to reduce the data structure to a string of items of primitive types. The primitive integral type data can then be directly converted to bit vectors to be sent on the system bus. Figure 4 shows a recursive algorithm to generate code for the sender side for doing this data processing. D, the input to the algorithm, is a definition of the complex variable. The type of D is checked to see if it is a complex type. For complex types, the algorithm is called recursively with the subtype element of D. For conversion of primitive integral types to bit streams, the modeling language must have constructs or supporting libraries. This is true for most system design languages and hence our algorithm can be used to generate code in those system modeling languages. Note that in the algorithm, the use of *codegen* refers to generating statements in the model description.

```
protocol   Bus
procedure GenerateSendCode ( definition D )
   if  TypeOf (D) = STRUCT
   do
         foreach elem in D
         do
                  call GenerateSendCode( elem )
         end for
   end if
   if TypeOf(D) = ARRAY
   do
         codegen : loop i = 0 to D.length -1
         call GenerateSendCode ( D[ i ] )
         codegen : end loop
   end if
   if TypeOf(D) = INTEGRAL_TYPE
   do
         codegen : bit[D.size-1 : 0] temp = D
         call GenerateSendCode ( temp )
   end if
   if TypeOf(D) = BIT_VECTOR
   do
         codegen :  slices =  ceil (D.size / Bus.datawidth)
         codegen :  loop   j  =  0   to slices - 1
         if (D.direction = DOWNTO)
         do
                  codegen : bus.Write ( SlaveAddr,
                           D [D.LeftBound : D.LeftBound - Bus.datawidth + 1] )
                  codegen :  ShiftLeft (D,  Bus.datawidth)
         else
                  codegen : bus.Write ( SlaveAddr,
                           D [D.LeftBound : D.LeftBound + Bus.datawidth - 1] )
                  codegen :  ShiftLeft (D,  Bus.datawidth)
         end if
         codegen : end loop
   end if
end procedure
```

Figure 4: Algorithm for generating code for data slicing

4.1.2 Creating Bus Transactions

Once the complex variable has been converted to a series of bit vectors, the methods of the protocol channel can be used for completing the data transfers. As shown in the second half of the *GenerateSendCode* algorithm in Figure 4, we determine the number of bus transactions required to send the entire variable. To perform these transactions, we need slices of the variable of the size of data bus width. The slices are made from left to right in the bit vector. Each of these slices are then sent using the bus protocol's write method. *Receive* procedure on the other side is an exact dual of the *Send* method shown here. The incoming bit stream is read using the protocol channel's *Read* method. Since we already know the variable type, it is thus easy to regenerate it at the *Receiver* end. Thereby we retain the functionality of the original *Receive* and *Send* methods in the input model.

4.2 Refining Synchronization

Besides converting abstract data to bus words, we also need to preserve the communication semantics of the input model. In the case of abstract channels, each data transaction is independent and does not interfere with other transactions. However, once all those independent data transactions are mapped on the same bus, they have to share the same communication medium and synchronization events. Therefore, it is necessary to generate additional synchronization code so as to avoid conflicts on the bus. This synchronization is inserted around the data splitting and transfer code in the application layer.

4.2.1 Synchronization for Statically Scheduled Components

This is the simplest case for handling synchronization. If the two communicating components have statically scheduled behaviors, there would be no possibility of temporal overlap of communication. In the two component design scenario, this amounts to communication between two concurrent processes. The communication takes place as follows.

When the slave process reaches the communication point, it notifies the master that it is ready to start the data transfer. This notification is typically done by means of an interrupt. Upon receiving the interrupt event, the slave suspends its execution and sets the *SlaveReady* flag as shown in Figure 5. The interrupt mechanism requires the simulator support and also programming language construct to model it. The master side process waits till the flag is set to initiate the bus transfer. The slave component waits for the master to initiate the bus transfer by checking the address bus. This mechanism retains the two-way blocking property of the original abstract communication. Once the data transfer is complete, the application layer resets the *SlaveReady* flag to prepare for the next slave request. In Figure 5 the actions performed in each state are shown in **bold** as compared to the state transition inputs. The slave component waits for the master to initiate the bus transfer by checking the address bus.

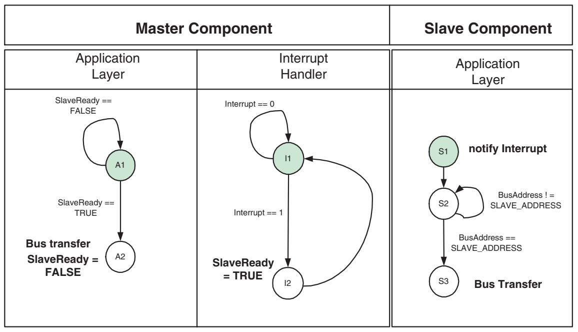

Figure 5: Master and Slave communication mechanisms for statically scheduled behaviors

4.2.2 Synchronization for Dynamically Scheduled Components

With dynamically scheduled components, we are faced with a scenario where we might have temporal overlap of communication. For instance, in Figure 3, transactions **B** and **C** might overlap in time. In such a case, we have two issues to look into.

Firstly, we have to determine the source of the data transfer request. If the master gets an interrupt from the slave, there is no way to tell if the slave is ready for transaction **B** or **C**. In a normal addressing scheme, a query by the master will only result in the slave component's address. To distinguish between the two transaction requests, each variable should be assigned a different address. Moreover, the behavior of the interrupt handler on the master side would also change as shown in Figure 6. On the master side, we will also need separate *SlaveReady* flags for each transaction address. The interrupt handler on receiving an interrupt event reads the variable's address from the bus. A special address labeled as *GLOBAL_ADDRESS* is used by the interrupt handler to notify the slave application layer to write its variable's address on the data bus. Depending on the slave address it sets the corresponding *SlaveReady* flag.

Secondly, with temporal overlap of communication, we need to control access to the IO port of the component. Therefore each data transfer has to be treated like a critical section. To ensure this we can use semaphores or hardware flags in components. Note in Figure 6 that each access to the IO port is protected. The code generated in application layer for each component must ensure that the IO port is reserved before it is used.

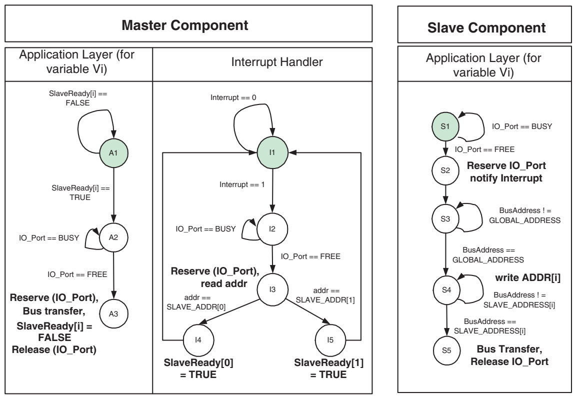
Master Component
Application Layer (for variable Vi)
Interrupt Handler
Slave Component
Application Layer (for variable Vi)
SlaveReady[i] == FALSE
A1
SlaveReady[i] == TRUE
IO_Port == BUSY
A2
IO_Port == FREE
Reserve (IO_Port), Bus transfer, SlaveReady[i] = FALSE Release (IO_Port)
A3
Interrupt == 0
I1
Interrupt == 1
IO_Port == BUSY
I2
IO_Port == FREE
Reserve (IO_Port), read addr
I3
addr == SLAVE_ADDR[0]
addr == SLAVE_ADDR[1]
I4
I5
SlaveReady[0] = TRUE
SlaveReady[1] = TRUE
S1
IO_Port == BUSY
IO_Port == FREE
S2
Reserve IO_Port notify Interrupt
S3
BusAddress ! = GLOBAL_ADDRESS
BusAddress == GLOBAL_ADDRESS
S4
write ADDR[i]
BusAddress ! = SLAVE_ADDRESS[i]
BusAddress == SLAVE_ADDRESS[i]
S5
Bus Transfer, Release IO_Port

Figure 6: Master and Slave communication mechanisms for dynamically scheduled behaviors

5. REFINEMENT OF COMPLEX DESIGNS

For more complex designs the refinement engine needs to do more work to ensure that synchronization is maintained. The basic tasks of data transfer remain essentially the same. For designs with more than one slave, we need to generate an interrupt controller. For multiple masters on a bus, we have to generate bus arbitration mechanism to make sure that the communication model is correct. In the case of multiple bus designs, components might interface to more than one bus, which requires generation of multiple bus drivers.

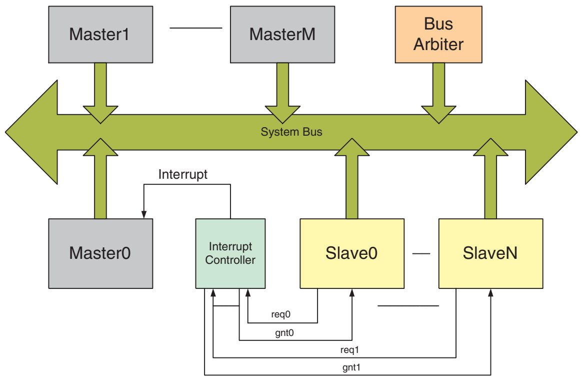

Figure 7: A complex bus architecture

5.1 Multiple Slaves

This is a typical system design where several slave components talk to a single master. In some ways, this case is similar to multi-threaded dynamically scheduled components that we discussed in the previous section. However, there are several independent interrupt lines and the master, which is typically a processor, has only one incoming interrupt line in its bus functional model. Some processors may have more than one interrupt, with an interrupt controller built in. The way we handle this is by parameterizing the processor components.

5.1.1 *Interrupt Controller*

If the number of slaves is more than the number of interrupt ports on the processor's interface, we generate an interrupt controller. A generic interrupt controller for a master component consists of *Interrupt Request* ports, *Interrupt Grant* ports labeled *req* and *gnt* respectively, as shown in Figure 7. Depending on the synthesis decision, we generate a priority based or round-robin interrupt controller.Figure 8 shows how a round-robin interrupt controller works. Upon choosing a slave request, the controller sends an interrupt event to the master component and a grant signal to the chosen slave.

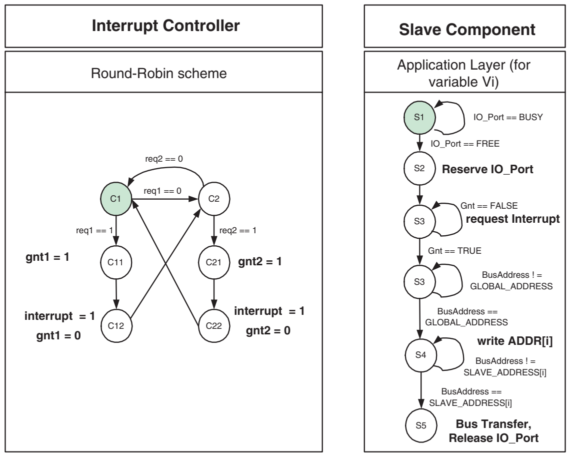

Figure 8: Round-robin interrupt controller and Slave communication mechanism

5.1.2 *Application layers*

For the master component, there is no change in the application layer. Since each variable carries its own address, the master does not make any distinction based on the slave component. However, the operation of the slave component has to be changed in the presence of other competing slaves. As shown in Figure 8, the slave sends an interrupt request to the interrupt controller and waits for the grant. If the controller gives grant to another slave, the request signal must be kept high to compete for the next grant cycle. On getting a grant signal, the slave monitors the address bus for the *GLOBAL_ADDRESS* to put its variable address on the data bus and continue as before.

5.2 Multiple Masters

For buses that support arbitration, the designer may designate more than one master as shown in Figure 7. The arbitration mechanism could either be distributed or centralized. For distributed arbitration, we rely on the protocol channel to provide for an appropriate method to request bus arbitration. Essentially, the master side protocol should have a special method which is annotated to be identified as the bus arbitration method. If such a channel method is not found, we have to generate a centralized bus arbiter as per the requirements. Based on synthesis decisions, we generate a priority-based or round-robin arbitration unit. The arbiter behavior is exactly like that of an interrupt controller, except that it resolves conflicts between masters.

6. EXPERIMENTAL RESULTS

Based on the described methodology and algorithms, we developed a communication refinement tool in C++. The example was chosen as the GSM Vocoder which is employed worldwide for cellular phone networks. The model was based on the bit-exact reference implementation of the ETSI standard in ANSI C. It encodes 5 ms of speech data consisting of 163 frames. Different architectures using the Motorola DSP56600 processor and custom hardware units were generated and various bus architectures were tested. Table 1 shows the data from tests conducted on 5 different architectures of the GSM Vocoder. The total traffic per speech sample refers to the amount of data exchanged between components during course of one simulation with a sample speech of 163 frames. Note that this data increases with greater partition, which increases communication time. To compare against the manual effort of model refinement, we used the Lines of Code (LOC) metric. Even with a very optimistic estimate of 10 LOC per person hour, manual communication refinement takes several hundred hours for reasonably complex designs. Automatic refinement on the other hand completes in the order of a few seconds. The productivity gain is enormous as a result of automatic refinement.

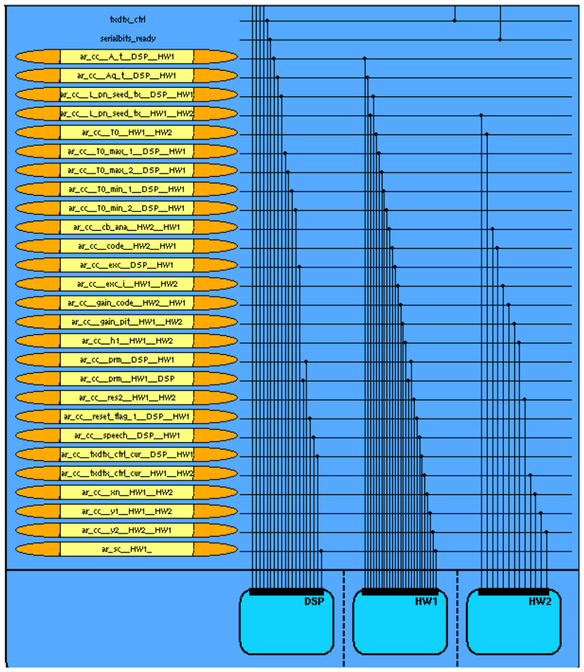

Figure 9: Top level of the input model

Snapshots from the GUI of the SCE design environment [3] are shown in Figure 9 and Figure 10. The design has three components DSP, HW1 and HW2 communicating with abstract channels as seen in Figure 9. Two buses viz. the Motorola DSP56600 bus and a generic 32-bit double handshake bus are used. The generated communication model's snapshot can be seen in Figure 10. Note that the top level consists of the components connected with wires of the system buses.

7. CONCLUSION AND FUTURE WORK

In this paper, we suggested a methodology and algorithms to automatically generate communication models. A tool was developed

Table 1: Experimental results for various vocoder architectures

Number of Components	Number of System Buses	Total Traffic/ speech sample (bytes)	Input Size	Output Size	Modified (LOC)	Automatic refinement (seconds)	Manual refinement (estimated person-hr)
1 DSP 56600 + 1 standard HW	1(Motorola DSP bus)	60962	7992	8448	1299	0.291	130
1 DSP 56600 + 2 standard HW	1(Motorola DSP bus)	66830	11292	12581	2392	0.480	240
1 DSP 56600 + 2 standard HW	1(Motorola DSP bus) + 1(HandShake bus)	70092	21248	22418	13020	0.644	1300
1 DSP 56600 + 4 standard HW	1(Motorola DSP bus) + 2(HandShake bus)	131378	25470	28950	19927	1.923	2000
2 DSP 56600 + 7 standard HW	2(Motorola DSP bus) + 2(HandShake bus)	60692	31481	37629	21375	3.761	2140

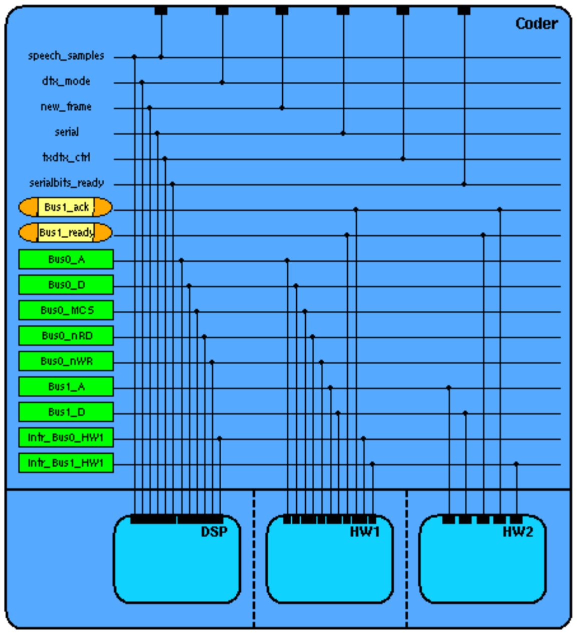

Figure 10: Top level of the generated communication model

and experiments were performed to validate this concept. Simulations were done on input models and output communication models to ensure their semantic equivalence. Our main contribution in this paper is the automation of a time-consuming and error prone process to achieve better designer productivity. It also enables designers to evaluate several design points during exploration. Future work in this direction would involve parameterizing of protocols and developing libraries for interrupt controllers and arbiters. We are also looking at automatic mapping of abstract communication on buses to minimize communication delay.

8. ACKNOWLEDGMENTS

The authors are grateful to reviewers for their insightful comments. Thanks to Dr. Rainer Doemer for his help with simulation tools and to Andreas Gerstlauer for his help with SCE System design environment. This work is funded in part by Semiconductor Research Corporation.

9. REFERENCES

[1] CoWare N2C. Available: http://www.coware.com/cowareN2C.html.

[2] SystemC, OSCI[online]. Available: http://www.systemc.org/.

[3] S. Abdi, J. Peng, R. Doemer, D. Shin, and et. al. SCE Environment - Tutorial. Technical Report ICS-TR-02-28, University of California, Irvine, September 2002.

[4] L. Gauthier, S. Yoo, and A. Jerraya. Automatic generation of application specific architectures for heterogeneous multiprocessor system-on-chip. In *Proceedings of the Design Automation Conference*, pages 518–523, June 2001.

[5] A. Gerstlauer, R. Domer, J. Peng, and D. Gajski. *System Design: A Practical Guide with SpecC*. Kluwer Academic Publishers, May 2001.

[6] F. Gharsalli, D. Lyonnard, S. Meftali, F. Rousseau, and A. A. Jerraya. Unifying memory and processor wrapper architecture in multiprocessor soc design. In *Proceedings of the International Symposium on System Synthesis*, Oct 2002.

[7] G. Gogniat, M. Auguin, L. Bianco, and A. Pegatoquet. Communication synthesis and hw/sw integration for embedded system design. In *Proceedings of the International Workshop on Hardware-Software Codesign*, pages 35–98, March 1998.

[8] Passerone, J. A. Rowson, and A. Sangiovanni-Vincentelli. Automatic synthesis of interfaces between incompatible protocols. In *Proceedings of the International Conference on Computer-Aided Design*, pages 437–444, November 1998.

[9] J. A. Rowson and A. Sangiovanni-Vincentelli. Interface based design. In *Proceedings of the Design Automation Conference*, pages 178–183, June 1997.

[10] F. Vahid and L. Tauro. An object-oriented communication library for hardware/software codesign. In *Proceedings of the International Workshop on Hardware-Software Codesign*, pages 81–86, March 1997.