Automatic Generation of Bus Functional Models from Transaction Level Models

Dongwan Shin and Samar Abdi and Daniel Gajski

Technical Report CECS-03-33
November 18, 2003

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425, USA
(949) 824-8059

{dongwans,sabdi,gajski}@cecs.uci.edu

Automatic Generation of Bus Functional Models from Transaction Level Models

Dongwan Shin and Samar Abdi and Daniel Gajski

Technical Report CECS-03-33
November 18, 2003

Center for Embedded Computer Systems
University of California, Irvine
Irvine, CA 92697-3425, USA
(949) 824-8059

{dongwans,sabdi,gajski}@cecs.uci.edu

Abstract

This report presents methodology and algorithms for generating bus functional models from transaction level models in system level design. Transaction level models are often used by designers for prototyping the bus functional architecture of the system. Being at a higher level of abstraction gives transaction level models the unique advantage of high simulation speed. This means that the designer can explore several bus functional architectures before choosing the optimal one. However, the process of converting a transaction level model to a bus functional model is not trivial. A manual conversion would not only be time consuming but also error prone. A bus functional model should also accurately represent the corresponding transaction level model. We present algorithms for automating this refinement process. Experimantal results presented using a tool based on these algorithms show their usefulness and feasibility.

Contents

List of Figures

Automatic Generation of Bus Functional Models from Transaction Level Models

Dongwan Shin and Samar Abdi and Daniel Gajski
Center for Embedded Computer Systems
University of California, Irvine

Abstract

This report presents methodology and algorithms for generating bus functional models from transaction level models in system level design. Transaction level models are often used by designers for prototyping the bus functional architecture of the system. Being at a higher level of abstraction gives transaction level models the unique advantage of high simulation speed. This means that the designer can explore several bus functional architectures before choosing the optimal one. However, the process of converting a transaction level model to a bus functional model is not trivial. A manual conversion would not only be time consuming but also error prone. A bus functional model should also accurately represent the corresponding transaction level model. We present algorithms for automating this refinement process. Experimantal results presented using a tool based on these algorithms show their usefulness and feasibility.

1. Introduction

With the increasing complexity of System on a chip (SoC) designs and the pressure of the time-to-market, we are continuously faced with the challenge of implementing the design specification while meeting the strict constraints it imposes. In order to tackle these problems, Raising the level of abstraction to the system level has been touted as a main solution. However, a well-defined system level design methodology and clear and unambiguous models of the different levels of abstraction in system design are necessary.

The transaction level modeling (TLM) [GLMS02] is high-level approach to modeling the systems where details of communcations among system components are separated from the detail of the implementation of system components and communication architecture. TLM aims at communication modeling so as to optimize simulation speed. TLM has been considered to address needs for early architecture exploration and embedded software development. However, much work still needs to be done to formalize the SoC design methodology and to adopt TLM in SoC design flow.

In SoC design, Communication synthesis requires extensive design space exploration for communication architecture. 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. Since the most systems contain at least one processor, software is an essential part of the system. The software can have concurrent execution of behaviors, which should be scheduled with the help of an operating system. The currency of the software makes software impossible to debug. This makes the evaluation of our communication architecture questionable.

Finally, clear separation of decision-making step and refinement step enables designers to implement systems with their insight on the system and algorithm developers to implement many efficient algorithms independently for design exploration and synthesis.

In this report we look at how we speed up the communication synthesis process by enabling automatic model refinement. The rest of the report is organized as follows. Section 2 is a brief review of the related work in this area. Section 3 talks about our communication refinement flow. In Section 4, communication architecture will be discussed. Section 5 looks at tasks of communication refinement for the system with dynamical scheduled behaviors. Finally, we present experimental results in section 6 and concludes this report with future works.

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. Yen and Wolf [YW95] mapped a multiprocess description to processing elements (PEs) and developed heuristics to determine the communication resources on the target architecture. In [GABP98], Gogniat et al. proposed the communication interface generation method from partitioned and scheduled system model for HW/SW interfaces for co-design of embedded systems. Ortega and Borriello looked at a retargetable modeling scheme for maximum utilization of bus bandwidth in [OB98]. However, they focused mostly on reactive real time systems.

CoWare [CoW] can support shared memory among heterogeneous processors but focuses on rendezvous communication protocol based on message passing. Jerraya et al. [LYBJ01] [CBG+02] presented interesting schemes for putting together heterogeneous components on a bus using wrappers for design of application-specific multi-processor SoCs. Grötker et al. talked about transaction level modeling in [GLMS02] that aims at communication modeling so as to optimize simulation speed. However, they do not address automatic refinement of a transaction level model to produce a timing-accurate and pin-accurate bus functional model.

3. Communication Refinement Flow

Figure 1 shows how communication synthesis is performed in our SoC design methodology. We begin with a transaction model of a system. It reflects 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.

The second input is a protocol library that 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 bus functional model.

The finnal input is a set of synthesis decisions by user. The user provides a set of synthesis decisions like bus allocation, bus mapping, connectivity, bus access priorities etc. 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.

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). Our refinement implelements two-way blocking

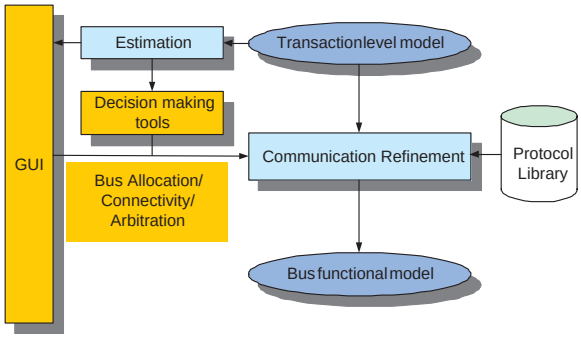

Figure 1. Communication refinement engine

message passing. In the case of two way blocking communication, 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.

4. Communication architecture

In this section, we look at communication architecture which is output of the communcation refinement. Our communication architecture contains processing elements (PEs)

which communicate through the buses. Communication between processing elements is based on either message passing and global shared memory. In general, each PE has local memories as part of the its microarchitecture. If the local memory of a PE can be accessed from other PEs it becomes global system memory. The union of all member variables in PE will be stored in its local memory.

The PEs are decomposed of master and slave. The master components can initiate the transaction and read (write) data from (to) slaves. The slaves have memory-mapped registers which can be read and written by masters. In case of design with more than one slaves on a bus, the system needs interrupt controllers. Each master component has its own interrupt controller to resolve the multiple interrupt requests from slaves. For design with more than one masters on a bus, we need an arbiter to resolve multiple requests to bus from masters. For communication between masters, we use the queue to buffer the transactions, because the masters cannot communicate each other directly. In the case of multiple bus designs, system might need bus bridge which connects buses. Figure shows the typical communication architecture. Inside the PEs, behavior models of bus drivers

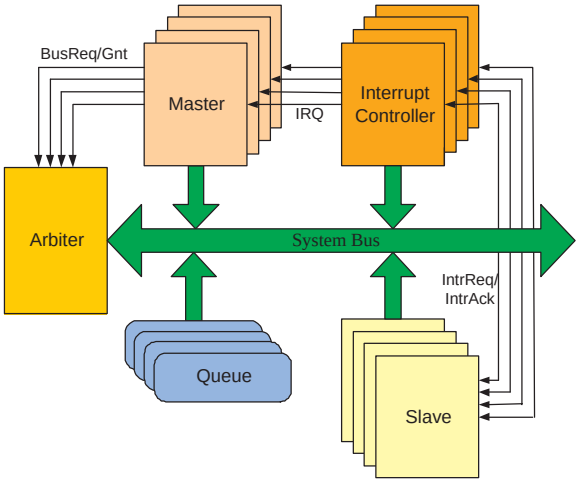

Figure 2. A typical communication bus architecture

and bus interfaces describe the PE's communication functionality, i.e. the implementation of the message-passing communication over the bus protocols. Those bus adapters specify how the PE implements the semantics of the abstract channels by driving and sampling the wires of the system bus. Behavioral blocks inside the PEs, in turn, connect to the equivalent message-passing channel interface provides by bus adapters.

The bus adapter channels are hierarchically composed of two layers: a high-level *application layer* and a low-level *protocol layer*. The protocol layer performs actual bus transactions by driving and sampling bus wires. The protocol layer is instantiation of the bus protocol from the protocol library. At its interfaces to application layer, sits on top of the protocol layer and provides the adapter's outer interface to the external world. Using the protocol layer primitives, it performs the necessary synchronization, data slicing, addressing to implement the communication over the bus protocol. The application layer will be explained in the next section in detail.

5. Communication refinement

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.

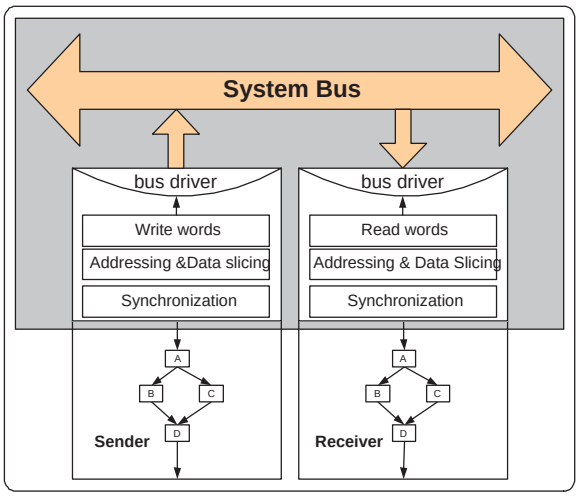

Figure 3. Application layer for simple architecture

5.1. Application layer

Application layer wraps up the protocol layer and provides the adapter's outer interface to the external world. Using the protocol layer primitives, it performs the necessary addressing, data slicing and synchronization to implement the communication over the bus protocol.

5.1.1 Addressing

Virtual addresses on the application side have to be turned into a bus adressing scheme. In general, bus address are a combination of source PE, destination PE, and ID of the message to be transfered. Depending on the application, the bus addressing scheme can be simplified. For example, if there is a predefined order of messages between two PEs (which means PEs are statically scheduled), the message ID can be removed from the address.

5.1.2 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. On the receiver side, this bit stream needs to be identified and reassembled to the original variable. Data slicing algorithm is explained in [ASG03] in detail.

5.1.3 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.

If there are a set of concurrent behaviors inside components, the behaviors can be scheduled statically or dynamically. 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. This is well explained in this paper [ASG03]. In this report, we will focus on communication between components with dynamically scheduled behaviors.

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 (message) should be assigned a different address. Moreover, the behavior of the interrupt handler on the master side would be shown in Figure 4. On the master side, we will also need separate *message id* (msg#_ih) register for each message. The master is waiting for message id (msg#_waiting). The interrupt handler on receiving an interrupt event reads the message id (msg#_ih) from the registers inside the slave with slave address. After getting an acknowledge from master, the slave will put the variable address on the bus.

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 the semaphore for the IO protection of the software components, and for harad-ware component, hardware protection mechanism like test-and-set. Note in Figure 4 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. subsectionMultiple Slaves This is a typical SoC de-

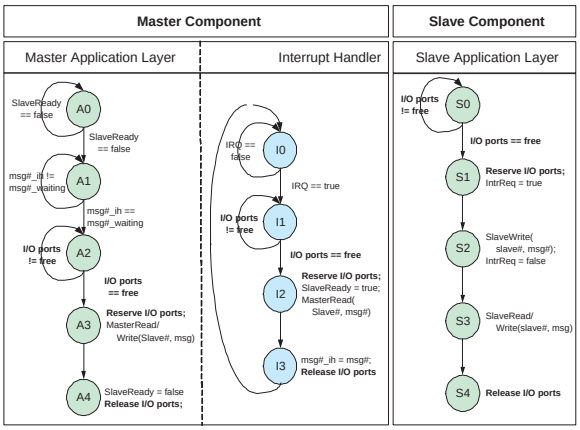

Figure 4. Master and slave communication mechanism for simple architecture

sign 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 previ-

ous 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.

5.1.4 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 Acknowledge* ports labeled *IntrReq* and *IntrAck* respectively, as shown in Figure 5. Depending on the synthesis decision, we generate a priority based or round-robin interrupt controller. In Figure 5, **Select** function implements these arbitration scheme in interrupt controller. Upon choosing a slave request, the controller sends an interrupt event to the master component and an acknowledge signal to the chosen slave. For the master component, there is no change

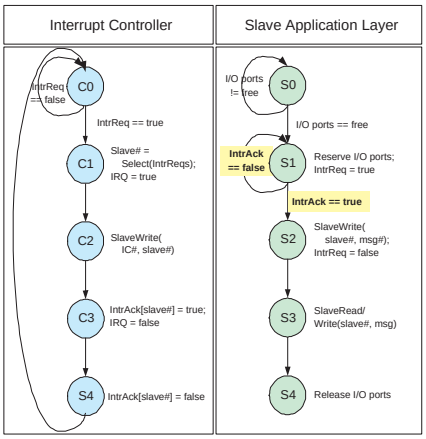

Figure 5. Interrupt controller and Slave communication mechanism

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 5, the slave sends an interrupt request to the interrupt controller and waits for the acknowledge. If the controller gives acknowledge to another slave, the request signal must be kept high to compete for the next acknowledge cycle. The interrupt handler inside the interrupted master will get the memory mapped register address of the slave from the interrupt controller, and read the address of the variable on the bus from the slave. After getting an acknowledge signal, the slave will put the variable address into its memory-mapped register and continue as before.

5.2. Multiple Masters

5.2.1 Queue

The master components cannot communicate directly without the help of the components which buffer the message among master components. Because the master components have the out-going address buses and can't be slave on the bus. In our implementation, we choose the FIFO queues to perform data transactions among master components. The FIFO queue serves slave on the bus. Each master component has its own queue to get the message from other master components. Master components will write message ID and message to the queue of other master components which they want to transfer messages. Then, if there are messages in the FIFO queue, the FIFO queue will interrupt the the master component. The master component will read message ID and message data from the queue.

5.2.2 Arbiter

For buses that support arbitration, the designer may designate more than one master as shown in Figure 6. 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. The arbitration is implemented inside protocol layer, because generally, it is the part of bus protocol.

6. Experiment 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 4 different architectures of the GSM Vocoder. The total traffic per

Table 1. Experimental results for various vocoder architectures

Number of Components	Number of System Buses	Traffic/ sample (bytes)	Schedule Method	TLM Size	BFM Size	Modified (LOC)	Automatic refinement (seconds)	Manual refinement (person-hr)
1 DSP 56600 + 1 standard HW	1 (Motorola DSP bus)	36512	Static	10270	12554	2284	0.701	230
			Dynamic	10307	13304	2997	0.714	300
1 DSP 56600 + 2 standard HW	1 (Motorola DSP bus)	46944	Static	9968	11279	1311	0.439	130
			Dynamic	10010	11611	1601	0.464	160
2 DSP 56600 + 2 standard HW	1 (Motorola DSP bus)	57276	Static	11049	15373	4324	1.597	430
			Dynamic	11103	16739	5636	1.687	560
2 DSP 56600 + 3 standard HW	2 (Motorola DSP bus)	121924	Static	35309	44771	9462	3.047	950
			Dynamic	35496	45557	10661	3.156	1070

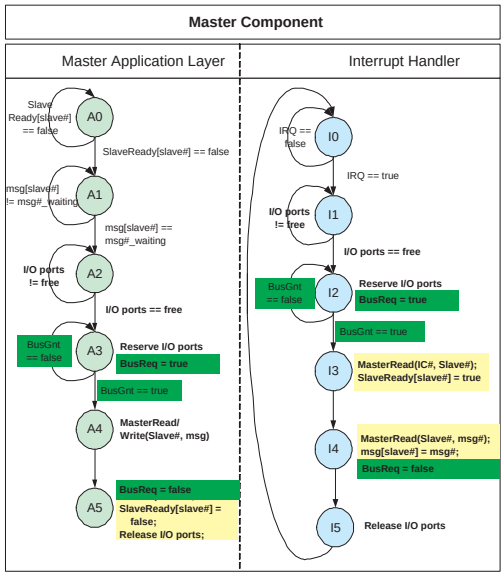

Figure 6. Interrupt handler and Master communication mechanism

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.

Snapshots from the GUI of our SoC design environment [APY+03] are shown in Figure 7 and Figure 8. The design has 4 components, DSP0, DSP1, HW0 and HW1 communicating with abstract channels as seen in Figure 7. One Motorola DSP56600 bus is used. The generated bus functional model's snapshot can be seen in Figure 8. Our refinement tool inserts two interrupt controllers for two DSP processors and a queue and an arbiter. Note that the top level consists of the components connected with wires of the system buses.

7. Conclusion and Future Works

In this report, we suggested a methodology algorithms to automatically generate bus functional models from transaction level model with abstract message passing semantics. A tool has been developed and experiments were performed to validate this concept. Simulations were done on input transaction level models and output bus functional models to ensure their semantic equivalence. Our main contribution in this report 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 automatic generation of transducer to make different bus protocols compatible.

References

[APY+03] Samar Abdi, Junyu Peng, Haobo Yu, Dongwan Shin, Andreas Gerstlauer, Rainer Dömer, and Daniel D. Gajski. System-on-chip Environment (SCE Version 2.2.0 beta): Tutorial. Tech-

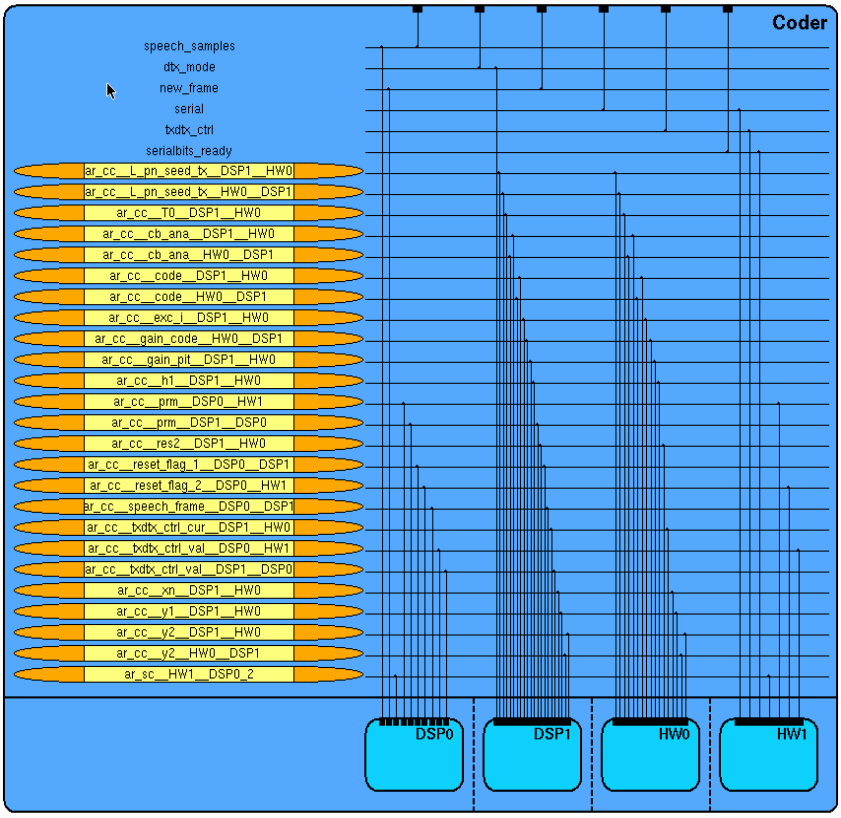

Figure 7. Top level of the transaction level model

nical Report CECS-TR-03-18, Center for Embedded Computer Systems, University of California, Irvine, July 2003.

[ASG03] Samar Abdi, Dongwan Shin, and Daniel D. Gajski. Automatic communication refinement in system-level design. In *Proceedings of the Design Automation Conference*, pages 300–305, June 2003.

[CBG+02] W. O. Cesario, A. Baghdadi, L. Gauthier, D. Lyonnard, G. Nicolescu, Y. Paviot, S. Yoo, A. A. Jerraya, and M. Diaz-Nava. Component-baed design approach for multicore SoCs. In *Proceedings of the Design Automation Conference*, pages 789–794, June 2002.

[CoW] CoWare N2C. Available at `http://www.coware.com/cowareN2C.html`.

[GABP98] Guy Gogniat, Michel Auguin, Luc Bianco, and Alain Pegatoquet. Communication synthesis and HW/SW integration for embedded system design. In *Proceedings of the International Workshop on Hardware-Software Codesign*, pages 49–53, March 1998.

[GLMS02] Thorsten Grötker, Stan Liao, Grant Martin, and Stuart Swan. *System Design with SystemC*. Kluwer Academic Publishers, March 2002.

[GVNG94] Daniel D. Gajski, Frank Vahid, Sajiv Narayan, and Jie Gong. *Specification and Design of Embedded Systems*. Prentice-Hall, 1994.

[LYBJ01] Damien Lyonnard, Sunjoo Yoo, Amer Baghdadi, and Ahmed 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.

[OB98] Ross B. Ortega and Gaetanno Borriello. Communication synthesis for distributed embedded systems. In *Proceedings of the International Conference on Computer-Aided Design*, pages 437–444, November 1998.

[RSV97] James A. Rowson and Alberto Sangiovanni-Vincentelli. Interface based design. In *Proceedings of the Design Automation Conference*, pages 178–183, June 1997.

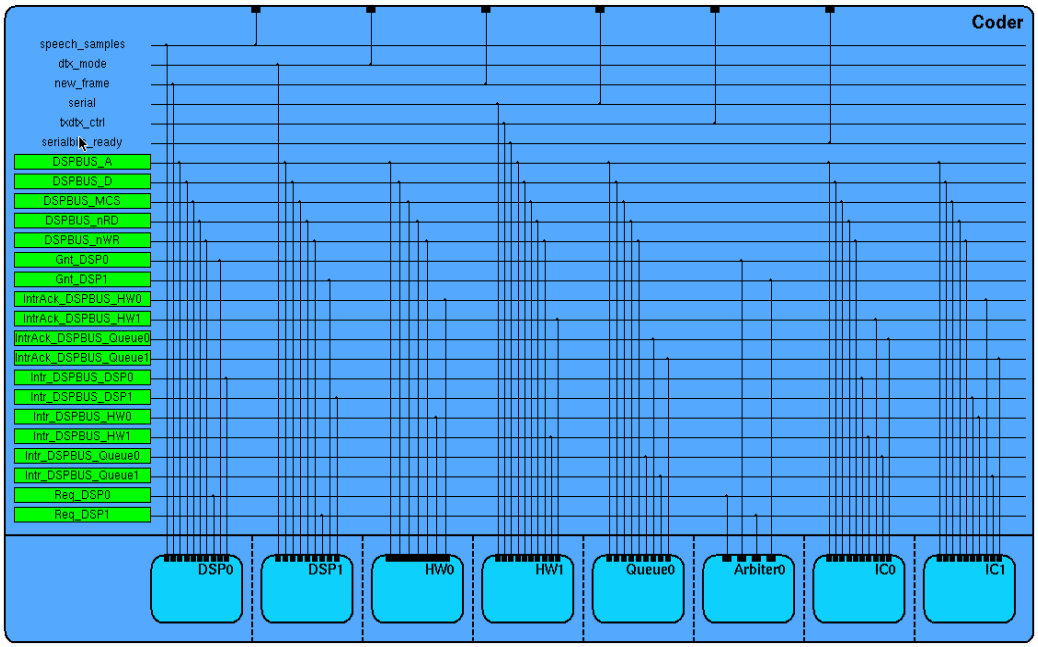

Figure 8. Top level of the generated bus functional model

[VT97] Frank Vahid and Linus 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.

[YW95] Ti-Yen Yen and Wayne Wolf. Communication synthesis for distributed embedded systems. In *Proceedings of the International Conference on Computer-Aided Design*, pages 288–294, November 1995.