# Automatic Network Generation for System-on-Chip Communication Design

Dongwan Shin, Andreas Gerstlauer, Rainer Dömer and Daniel D. Gajski
Center for Embedded Computer Systems
University of California Irvine
CA 92697 USA
{dongwans, gerstl, doemer, gajski}@cecs.uci.edu

## ABSTRACT

With growing system complexities, system-level communication design is becoming increasingly important and advanced, network-oriented communication architectures become necessary. In this paper, we extend previous work on automatic communication refinement to support non-traditional, network-oriented architectures beyond a single bus. From an abstract description of the desired communication channels, the refinement tools automatically generate executable models and implementations of the system communication at various levels of abstraction. Experimental results show that significant productivity gains can be achieved, demonstrating the effectiveness of the approach for rapid, early communication design space exploration.

## Categories and Subject Descriptors

J.6 [**Computer-Aided Engineering**]: CAD

## General Terms

Design, Alogithms

## Keywords

System level design, communication synthesis

## 1. INTRODUCTION

With the ever increasing complexities and sizes of system level designs, system-level communication is becoming an increasingly dominant factor, e.g. in terms of overall latencies. As the number and types of components on a chip increases, traditional, single-bus structures are not sufficient anymore and new, network-based communication architectures are needed. In order to explore the communication design space, designers use models which are evaluated through simulation. Typically, these models are handwritten, which is a tedious and error-prone process. Furthermore, to achieve

required accuracies, models are written at low levels of abstraction with resulting slow simulation performance. Together, this severely limits the amount of design space that can be explored in a reasonable time-to-market.

In this paper, we propose a communication design process that can automatically generate models and implementations of system communication through refinement from an abstract description of the partitioned system processing architecture. The automatic refinement tools can produce communication models at various levels of abstraction in order to trade off simulation accuracy and speed. In previous work [2], we presented automatic communication refinement for simple, single-bus based architectures. In this paper, we extend this approach to support complex, non-traditional architectures with communication over a heterogeneous network of busses or other communication media. We introduce an additional network refinement tool that automatically generates the necessary implementation of upper network protocol layers for bridging and routing of end-to-end communication over a network of point-to-point links.

The rest of the paper is organized as follows: in the remainder of this section, we introduce the overall design flow followed by a brief overview of related work. Section 2 then shows the inputs and outputs of the network design task and Section 3 will present the details of the network refinement process. Finally, experimental results are shown in Section 4 and the paper concludes with a summary in Section 5.

### 1.1 Communication Design Flow

Figure 1 shows the refinement-based communication design flow [8] which is divided into two tasks: *network design* and *link design*. During the network design, the topology of communication architecture is defined and abstract message passing channels between system components are mapped into communication between adjacent communication stations (e.g. processing elements and communication elements) of the system architecture. The network topology of communication stations connected by logical link channels is defined, bridges and other communication elements are allocated as necessary and abstract message passing channels are routed over sets of logical link channels. The result of the network design step is a refined link model of the system. The link model represents the topology of communication architecture where components and additional communication stations communicate via logical link channels.

Network design is followed by *link design* [15] where logical link channels between adjacent stations are then grouped
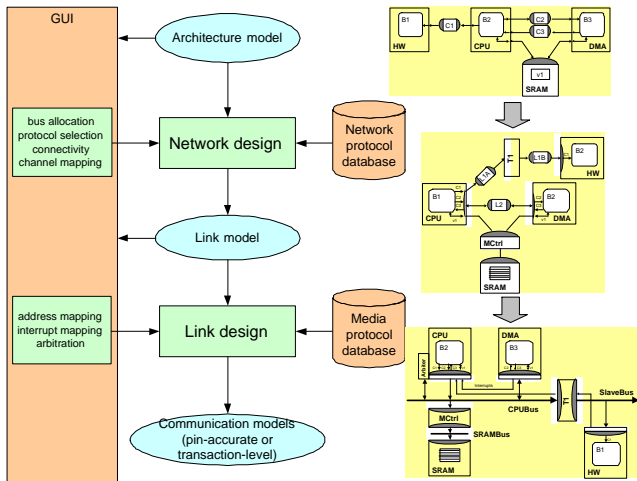
**Figure 1: Communication design flow.**

and implemented over an actual communication medium (e.g. system busses) where each group of links can be implemented separately. Logical links are grouped, a communication medium with associated protocol is selected for each group, types of stations connected to each medium are defined and media parameters like addresses and interrupts are assigned to each logical link.

As a result of the communication design process, communication models such as pin-accurate models and transaction-level models of a system are generated. The final pin-accurate model is a fully structural model where components are connected via pins and wires and communicate in a cycle-accurate manner based on media protocol timing specifications.

## 1.2 Related Work

Recently, transaction-level modeling (TLM) [6, 10] has been proposed for modeling and exploration of SoC communication. TLM proposals, however, focus on modeling and simulation only. By themselves, they do not provide solutions for generating such models. Historically, a lot of work has focussed on automating the decision making process [16, 7, 13, 11] for communication design without, however, providing corresponding design models or a path to implementation. There are several approaches dealing with automatic generation [4] and refinement of communication [12, 5] but these approaches are usually limited to specific target architecture templates or narrow input model semantics. Benini et al. [3] proposed the Network on Chip (NoC) approach which partitions the communication along OSI layers. While the OSI layering also builds the basis for our approach, they do not provide an actual automatic generation of such layers.

## 2. NETWORK DESIGN

The network design task consists of implementations for upper layers of the protocol stack, namely presentation, session, transport, and network layers [8]. The presentation layer is responsible for data formatting. It converts abstract data types in the application to blocks of ordered bytes as defined by the canonical byte layout requirements of the lower layers.

The session layer implements end-to-end synchronization to provide synchronous communication as required between system components in the application. Furthermore, it is responsible for multiplexing messages of different channels into a number of end-to-end sequential message streams.

The transport layer splits messages into smaller packets (e.g. to reduce required intermediate buffer sizes) and implements end-to-end flow control and error correction to guarantee reliable transmission.

Finally, the network layer is responsible for routing and multiplexing of end-to-end paths over individual point-to-point logical links. As part of the network layer, additional communication stations are introduced as necessary, e.g. to bridge two different bus systems.

Network refinement takes three inputs: an architecture model, design decisions and a network protocol database. With these inputs, the network refinement tool produces an output link model that reflects the topology of communication architecture of the system.

## 2.1 Input Architecture Model

The architecture model is the starting point for communication design. In the architecture model, system components communicate via message-passing channels. The communication design process gradually implements these channels and generates a new model for each layer of communication functionality inserted.

The architecture model follows certain pre-specified semantics. 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 design. Inter-component communication is end-to-end and takes place through abstract channels that support *send* and *receive* methods.

For data object communicated between system components, the model contains corresponding typed message-passing channels. Communication between components can be modeled via three schemes: two-way blocking, one-way blocking and non-blocking communication. In the paper, we will look at refinement of two-way blocking communication because it is used for unbuffered data transfers and can be implemented directly over standard bus-based communication protocols. The other two mechanisms can be implemented easily once we have support for two-way blocking communication.

Figure 2 shows an example of the architecture model. During partitioning, the application has been mapped onto a typical system architecture consisting of a processor (*PE2*), a custom hardware (*PE1*), an IP (*IP1*) and a system memory (*M1*). Inside *PE2*, tasks are dynamically scheduled under the control of an operating system model [9] that sits in an additional operating system shell of the processor (*PE2_OS*).

In the architecture model, the shared memory component is modeled as a special behavior with interfaces. The memory behavior encapsulates all variables mapped into the shared memory component. At its interface, the memory behavior provides two methods for each variable to read and write the value of the variable from/to memory.
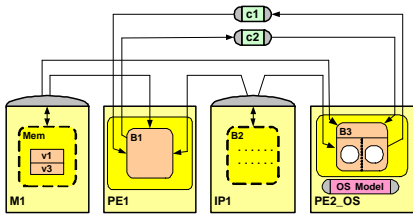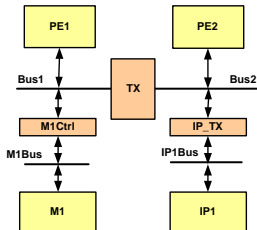
**Figure 2: Input architecture model.**



**Figure 3: Connectivity definition for Figure 2.**

## 2.2 Design Decisions

During network design, design decisions include *allocation of system busses*, *protocol selection*, *selection of communication elements*, *connectivity definition between components and busses*, *mapping of abstract communication to busses*, and *byte layout of system memories*. Based on these decisions, the network refinement maps the application channels onto logical links, synthesizes the implementations of the communication elements and finally generates the resulting link model.

For example, for the implementation of the previously introduced architecture Figure 2 we made the following design decisions: three busses (*Bus1* for *PE1*, *Bus2* for *PE2*, *IP1Bus* for *IP1* and *M1Bus* for *M1*) are allocated and the connectivity is defined as shown in Figure 3.

## 2.3 Network Protocol Database

The network protocol database contains a set of communication elements (CEs) such as transducers and bridges. The database contains models of bridges and transducers that include attributes like name, type and associated bus protocols. Models of the CEs in the database are empty templates that are void of any functionality and will be synthesized by the refinement tools.

## 2.4 Output Link Model

The link model is an intermediate model of the design process between network and link design. The link model reflects the network topology of the communication architecture where components communicate via logical link channels which implement message-passing semantics.

For each application channel between components, upper layers of the protocol stack such as presentation, session, transport and network layers are inlined into the corresponding components. In the link model, end-to-end channels have been replaced with point-to-point logic link channels between components that will later be physically implemented via directly connected bus wires. In the link model, communication elements are inserted from the network protocol database and synthesized for bridging different busses.
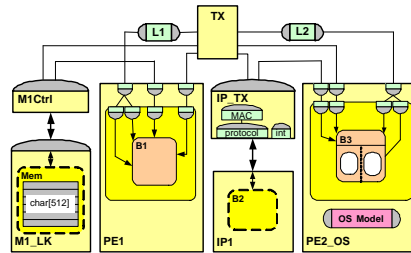


**Figure 4: Output link model.**

Figure 4 shows an example of a link model generated from the initial architecture model example (Figure 2) using the network design decisions in (Figure 3). Inside each component, implementations of the presentation layer are inserted. For instance, the corresponding implementations of application channels (*c1*, *c2*, *IP1* and *M1*) are inlined into the behavior of the corresponding system components. A transducer (*TX*) is introduced to perform protocol translation between *Bus1* and *Bus2*. The communication between *PE1* and *PE2* is routed over logical link channels, *L1* and *L2* via *TX*.

Also, a system memory (*M1*), which has its own interface protocol (*M1Bus*), has been connected to *Bus1* and therefore, a bridge (*M1Ctrl*) for protocol translation between them is necessary. The memory component model has been refined down to an accurate representation of the byte layout. All variables stored inside the memory are replaced with and grouped into a single array of bytes.

For an IP component (*IP1*), the transducer for the IP (*IP_TX*) is introduced, and the the IP transducer contains protocol stacks with implementations of media access and protocol layers for communication with the IP.

## 3. NETWORK REFINEMENT

Network refinement refines the input architecture model into a link model that reflects the network topology of a system. Network refinement implements the functionality of presentation, session, transport and network layers.
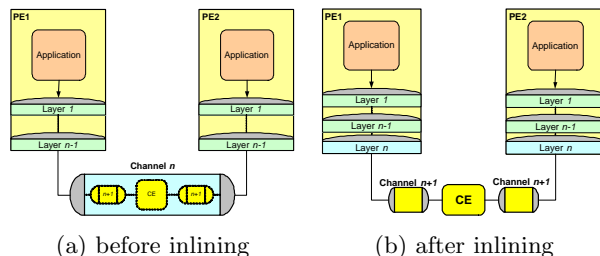


(a) before inlining      (b) after inlining

**Figure 5: Inlining of protocol stack.**

With each step in the design flow, an additional layer of communication functionality is inlined into the PEs of the design model. By replacing the communication between PEs with channels and communication elements that model the transaction semantics at the interface of the next lower layer, a new system model at the next lower level of abstraction is generated. Then, model refinement is performed through channel inlining of protocol stacks as shown in Figure 5 Note that as part of network refinement, layers may be merged for cross-optimizations. Furthermore, during synthesis, tools
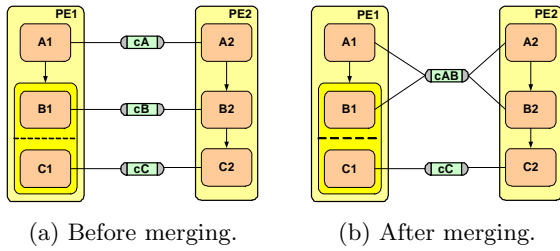
(a) Before merging.　　(b) After merging.

**Figure 6: An example for channel merging.**

will optimize and customize layers depending on the specific requirements of the application and the target architecture, e.g. to eliminate unnecessary functionality or to adjust hardware resource parameters.

The refinement process can be divided into four main steps corresponding to the previously introduced layering: data formatting (presentation layer), channel merging (session layer), flow control and error correction (transport layer), and communication element synthesis (network layer).

## 3.1  Presentation Layer: Data Formatting

The presentation layer converts abstract data types in the application to blocks of ordered types as defined by the canonical byte layout requirements of the lower layers. For example, the presentation layer takes care of component-specific data type conversions and endianess (byte order) issues.

In the link model, models of components contain presentation layer implementations in the form of adapter channels that provide the services (interface) of the presentation layer to the application on one side, while connecting and calling network layer methods on the other side. The presentation layer performs data formatting for every message data type found in the application (primitive/complex data type to void pointer type conversion).

The presentation layers inside components accessing a global, shared memory are responsible for converting variables in the application into size and offset for shared memory accesses. For a memory component, all variables stored inside the memory are replaced with and grouped into a single array of bytes. The memory component is modeled as a channel which provides methods (*read/write*) to access each variable by providing the offset of the variable in the memory.

## 3.2  Session Layer: Channel Merging

The session layer is responsible for multiplexing of different channels into a number of end-to-end sequential message streams. If communication channels are guaranteed to be accessed sequentially over time and transaction are guaranteed to never overlap, they can be merged during network refinement. In other words, sequential transactions are merged over single stream as much as possible in order to reduce the number of logical link channels in the system. Channel merging is implemented through static connectivity, i.e. if channels are merged, they are multiplexed by connecting their presentation layers to the same lower transport layer instance.

In the example of Figure 6(a), three double handshake channels are used for message passing between three components. They are all mapped onto one bus. On *PE1*, *A1* is followed by a parallel composition of *B1* and *C1* while

on *PE2*, *A2* is followed by *B2* and *C2*. Channel *cB* and *cC* can not be shared, because the execution order between *B1* and *C1* is not known in advance. If we shared *cB* and *cC*, then *B1* would potentially receive data from *C1* which was intended for *C1*. However, we can safely share *cA* and *cB* because we know that *A1* is always executed first without causing any deadlock and violation of the data transfer sequence. By doing this, we can eliminate one channel.

Then, our problem is to determine how to group channels. Based on the observation from the example, we conclude that two channels can be merged if three conditions are met: (a) two channels must be assigned to same bus, (b) two channels send data from the same source component to the same destination component, (c) both sending and receiving behaviors of both channels execute sequentially. If two channels satisfy the aforementioned conditions, then we say that they are *compatible*, otherwise they conflict with each other.

Our channel merging algorithm is based on the conflict analysis as follows: The first step is to build the conflict graph for all channels by checking the conditions. In the conflict graph, vertices represent the channels and edges represent the conflict between them. The second step is then to color the conflict graph with the minimum number of colors under the requirement that two vertices on the same edge can not have the same color. Graph coloring is known to be NP-complete and heuristics can be used when graphs are large. For our implementation, we used greedy graph coloring algorithm. After coloring, the channels with the same color are merged into one channel.

Note that merging of channels in the link model implies sharing of bus addresses and CPU interrupts. Therefore, if the application framework defined through the architecture model describes a sequential relationship of communication channels, the application programmer generally can not convert the sequential code into parallel tasks later. The opposite case, however, holds true, i.e. parallel specification tasks can be serialized at any time without affecting correctness. The initial specification should therefore in general already expose all the required application task parallelism.

## 3.3  Transport Layer: Flow Control

The transport layer is responsible for paketing of messages. It splits messages into smaller packets to reduced required buffer sizes. Depending on the links and stations in lower layers, the transport layer implements end-to-end flow control and error correction to guarantee reliable transmission. However, in cases where underlying medium is guaranteed to be error-free (e.g. standard bus-based communication), error correction and flow control are not required. Therefore, in these cases transport layers are simplified.

## 3.4  Network Layer: CE Synthesis

The network layer is responsible for routing and multiplexing of end-to-end paths over individual point-to-point logical links. As part of the network layer, additional communication stations such as bridges and transducers are introduced as necessary, e.g. to bridge two different bus systems. The communication stations split the system of connected system components in the architecture model into several bus sub-systems. If a communication element is allocated from the communication element database, its functionality is not implemented yet. Model refinement will gen-
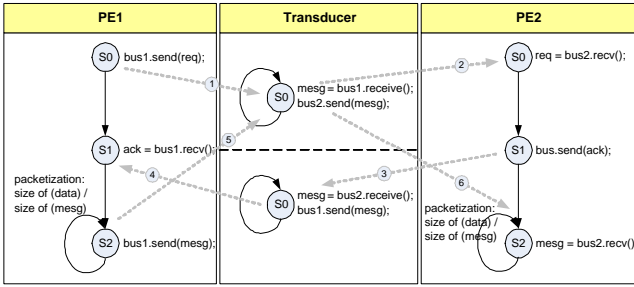
**Figure 7: State machine of transducer (network).**

erate functionality of the communication element, based on its type and types of channels communicated over it.

Assuming reliable stations and logical links, routing SoCs is usually done statically, i.e. all packets of a channel take the same fixed, pre-determined path through the system. In standard bus-based communication, dynamic routing is not required, therefore, network layers are simplified.

### 3.4.1 Bridges

Bridges in general are CEs that transparently connect one bus to another. Bridge implementations are taken out of the network protocol database where the database has to contain specific bridge component for every two busses that the user should be able to bridge during network design.

A bridge in the database has exactly two bus interfaces/ports. A bridge is always slave on one bus and master on the other. It transparently transforms and implements every matching transaction on its slave side by performing a corresponding transaction on its master side. Basically, the bridge maps the address and interrupt space of the master side bus into the slave side bus where the bridge in the database specifies the range of addresses mapped.

A bridge does not buffer a complete transaction but rather blocks the transaction on its slave side until the shadow transaction on its master side is complete. Therefore, a bridge preserves semantics (e.g. synchronization) inherent in the bus protocols. As such, a bridge is a CE that only covers conversions at the protocol level and that is transparent to higher communication layers.

One type of bridge is a memory controller which bridges processor and memory busses. The memory controller model exports an interface that matches the memory interface which provides *read* and *write* methods as shown in Figure 4. The interface of the memory controller is connected to the PE on one side and to the memory on the other side. Inside its interface methods, the memory controller invokes the interface methods of the memory to read/write data. The memory controller model serves as a basis for link design and it implements the identity function at the high level of abstraction.

### 3.4.2 Transducers

In cases where simple bus bridges are not sufficient, transducer CEs are allocated out of the network protocol database during network design in order to connect incompatible bus protocols. In general, transducers can connect any two bus protocols and they can be master or slave on either side. In contrast to a bridge, transducers internally buffer each individual bus transactions/transfer received on one side be-

fore performing the equivalent transaction on the other side. Note that transducers contain separate buffers for each direction of each channel, i.e. buffers are not shared, avoiding potential deadlocks.

Transducers take part in high-level point-to-point communication protocols. As such, a transducer does not preserve synchronicity but rather decouples end-to-end channels into two point-to-point channels each. Since memory transfers can not be decoupled, memory interface transactions can not be mapped and implemented over a transducer. In case of two-way blocking double handshake communication over a transducer, network refinement will automatically insert necessary protocol implementations into the PE endpoints in order to restore synchronicity lost over the transducer by exchanging additional ready and acknowledge packets (*req/ack* in Figure 7) such that end-to-end double handshake semantics are preserved as shown in Figure 7. In order to resolve deadlock in case the buffers in the transducer are full which causes a cycle wait, we use separate buffers for each channel, which means buffers are not shared in the transactions.

## 4. EXPERIMENTAL RESULTS

Based on the described methodology and algorithms, we developed a network refinement tool, which is integrated in our SoC design environment [1]. We applied the tool to four examples: a JPEG decoder (*JPEG*), a GSM Vocoder (*Vocoder*), a mobile phone baseband platform (*Baseband*) and a MP3 Decoder (*MP3*). The Baseband example is composed of JPEG decoder and GSM Vocoder running in parallel.

Different architectures using Motorola DSP56600 processors (*DSP*), Motorola ColdFire processors (*CF*) and custom hardwares blocks (*HW*) were generated and various bus architectures (*DSP* Bus, *CF* Bus and simple handshake bus) were tested. Table 1 shows the design characteristics (number of message passing channels and the total traffic) and the design decisions made during network design. In this table, channel mapping from application channels to link channels is done by automatic the network refinement tool which implements the channel grouping as shown in Section 3.2. In the Baseband example, the *Bridge* is used to connect two busses, DSP bus and CF bus. In the MP3 decoder example, the ColdFire processor communicates with four dedicated hardware units over its bus whereas the hardware units communicate with each other through four separate handshake busses. Table 1 also shows the results of network refinement. Model complexities are given in terms of code size (using Lines of Code (LOC) as a metric) and number of channels in the design. The number of channels are reduced during refinement based on channel grouping (as shown in the fifth and sixth columns) which turns out to reduce the number of lines of code significantly in the JPEG and Vocoder examples. Results show significant differences in complexity between input and generated output models due to extra implementation detail added between abstraction levels. To quantify the actual refinement effort, the number of modified lines is calculated as the sum of lines inserted and lines deleted whereas code coming from database models is excluded. We assume that a person can modify 10 LOC/hour. Thus, manual refinement would require several weeks for reasonably complex designs. Automatic refinement, on the other hand, completes in the order of seconds. Results show that a productivity gain of around

**Table 1: Experimental results for network refinement.**

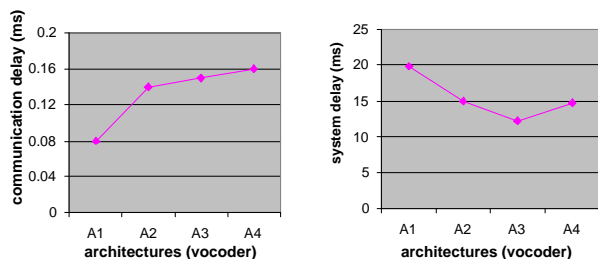| Examples | | Traffic (bytes) | Medium (masters/slaves) | Channels | | Lines of Code | | | Tool (sec) | Human (hr) | Gain |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | Arch | Link | Arch | Link | Mod. | | | |
| JPEG | A1 | 7560 | DSP Bus (DSP/HW) | 7 | 1 | 2940 | 2969 | 133 | 0.08 | 13.3 | 798 |
| Vocoder | A1 | 46944 | DSP Bus (DSP/HW) | 12 | 1 | 10972 | 10980 | 170 | 0.27 | 17.0 | 1020 |
| | A2 | 56724 | DSP Bus (DSP/(2 HWs)) | 22 | 2 | 11386 | 11415 | 223 | 0.34 | 22.3 | 1338 |
| | A3 | 76284 | DSP Bus (DSP/(3 HWs)) | 42 | 3 | 11263 | 12276 | 559 | 0.43 | 55.9 | 3354 |
| | A4 | 57160 | DSP1 Bus (DSP1, HW1), DSP2 Bus (DSP2/HW2) | 29 | 2 | 13986 | 14033 | 369 | 0.45 | 36.9 | 1107 |
| Baseband | A1 | 1113801 | DSP Bus (DSP/5 HWs), CF Bus (CF/(MEM,IP)) | 32 | 13 | 19754 | 20227 | 1195 | 0.75 | 119.5 | 3385 |
| MP3 | A1 | 103289 | CF Bus (CF/5 HWs) 4 Handshake Buses (5 HWs) | 65 | 10 | 33794 | 33905 | 1181 | 0.92 | 111.1 | 952 |



**Figure 8: Exploration results (Vocoder).**

1000 times can be expected using the presented approach and automatic model refinement.

Figure 8 shows the results of exploration of the design space for the vocoder example. We used 4 different architecture for vocoder as shown in Table 1 and measured the communication delay and whole system delay of each architecture. As shown in Figure 8, as the number of system components increases with each architecture, the overall performance of the system is improved while communication delays increase. Given the design decisions made by the user, it took less than 1 hour to obtain 4 different communication models from an executable specification model by architecture exploration [14] and communication design.

# 5. CONCLUSIONS

In this paper, we presented a methodology, algorithms and tools to automatically generate models and implementations of advanced, network-oriented SoC communication designs from a partitioned virtual architecture model of a system. On top of previous work and existing tools for automated link design, a network design process implements end-to-end communication semantics between system components which is mapped into point-to-point communication between communication stations of a network architecture. A corresponding network refinement tool has been developed and integrated into our SoC design environment. Using industrial-strength examples, the feasibility and benefits of the approach have been demonstrated. Automating the tedious and error-prone process of refining a high-level, abstract description of the design into an actual implementation results in significant gains in designer productivity, thus enabling rapid, early exploration of the communication design space. Future work includes extending network design to implement error correction, flow control and dynamic routing for unreliable and long-latency underlying media. Furthermore, we plan to add algorithms for automated design making and optimization in order to provide fully automated network and communication synthesis for extensive communication design space exploration.

# 6. REFERENCES

[1] S. Abdi, et al. System-on-Chip Environment (SCE Version 2.2.0 beta): Tutorial. Technical Report CECS-TR-03-41, University of California, Irvine, 2003.

[2] S. Abdi, et al. Automatic communication refinement in system-level design. In *Proc. of DAC*, 2003.

[3] L. Benini, et al. Networks on chips: A new SoC paradigm. *IEEE Computer*, 2002.

[4] I. Bolsens, et al. Hardware/Software co-design of the digital telecommunication systems. *Proc. of IEEE*, 1997.

[5] W. O. Cesario, et al. Component-baed design approach for multicore SoCs. In *Proc. of DAC*, 2002.

[6] M. Coppola, et al. IPSIM: SystemC 3.0 enhancements for communication refinement. In *Proc. of DATE*, 2003.

[7] M. Gasteier, et al. Generation of interconnect topologies for communication synthesis. In *Proc. of DATE*, 1998.

[8] A. Gerstlauer, et al. System-level communication modeling for Network-on-Chip synthesis. In *Proc. of ASPDAC*, 2005.

[9] A. Gerstlauer, et al. RTOS modeling for system level design. In *Proc. of DATE*, 2003.

[10] T. Grötker, et al. *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[11] K. Lahiri, et al. Efficient exploration of the SoC communication architecture design space. In *Proc. of ICCAD*, 2000.

[12] D. Lyonnard, et al. Automatic generation of application-specific architectures for heterogeneous multiprocessor System-on-Chip. In *Proc. of DAC*, 2001.

[13] R. B. Ortega, et al. Communication synthesis for distributed embedded systems. In *Proc. of ICCAD*, 1998.

[14] J. Peng, et al. Automatic model refinement for fast architecture exploration. In *Proc. of ASPDAC*, 2002.

[15] D. Shin, et al. Automatic generation of communication architectures. In A. Rettberg, et al., editors, *From Specification to Embedded Systems Application*, 2005. Springer.

[16] T.-Y. Yen, et al. Communication synthesis for distributed embedded systems. In *Proc. of ICCAD*, 1995.