Automatic Layer-Based Generation of System-On-Chip Bus Communication Models

Andreas Gerstlauer, *Member, IEEE*, Dongwan Shin, *Member, IEEE*, Junyu Peng, *Member, IEEE*, Rainer Dömer, *Member, IEEE*, and Daniel D. Gajski, *Fellow, IEEE*

Abstract—With growing market pressures and rising system complexities, automated system-level communication design with efficient design space exploration capabilities is becoming increasingly important. At the same time, customized network-oriented communication architectures become necessary in enabling a high-performance communication among the system components. To this end, corresponding communication design flows that are supported by efficient design automation techniques need to be developed. In this paper, we present a system-level design environment for the generation of bus-based system-on-chip architectures. Our approach supports a two-stage design flow using automated model refinement toward custom heterogeneous communication networks. Starting from an abstract specification of the desired communication channels, our environment automatically generates tailored network models at various levels of abstraction. At its core, an automatic layer-based refinement approach is utilized. We have applied our approach to a set of industrial-strength examples with a wide range of target architectures. Our experimental results show significant productivity gains over a traditional communication design, allowing early and rapid design space exploration.

Index Terms—Communication synthesis, embedded systems, heterogeneous multiprocessor system-on-chip (MPSoC), system-level design, transaction-level modeling (TLM).

I. INTRODUCTION

AS SYSTEM-ON-CHIP (SoC) designs grow in size and complexity, on-chip communication is becoming an increasingly important factor. New classes of optimization problems arise as communication delays and latencies across the chip start dominating computation delays. As a result, simple communication architectures based on a single shared bus are not sufficient anymore. Therefore, new heterogeneous multibus communication architectures and design flows are needed.

Communication design for SoC poses unique challenges for the system designer, including a wide range of architectures and many opportunities for optimizations due to the application-specific nature of system design. A corresponding communication design flow is needed that enables automation and rapid design space exploration. This design flow must be well defined, with unambiguous abstraction levels, models, and transformations.

Typically, design models are manually written, which is a tedious, error-prone, and time-consuming process. Also, to achieve the required accuracy, models are often written at a low abstraction level, resulting in a slow simulation. Together, this severely limits the design space that can be explored in a reasonable time.

A. Scope of Work

We propose a communication design environment that generates SoC models and implementations through refinement from an abstract specification of the system. Automatic refinement tools produce communication models at various abstraction levels to trade off simulation speed versus accuracy. Our design flow supports complex nontraditional communication architectures using a heterogeneous network of buses.

Our design environment is interactive. Since we want to benefit from designer experience and keep the designer in the loop, all decisions are made by the designer. As such, our environment does not currently provide an automated decision making or a fully automatic synthesis. Instead, it automates the generation of models based on designer decisions. Thus, instead of dealing with model writing, the designer can focus on the decision making and exploration process.

The quickly generated models can then be evaluated through simulation for a rapid feedback about the design quality. The generated models are also used for further system implementation, e.g., through back-end synthesis.

B. Target Architecture

We target architectures that are networks of shared buses over which processing elements (PEs) communicate. Multiple buses can be connected via communication elements (CEs). All buses are assumed to be reliable, i.e., error-free and lossless. The network of buses forms a forest-of-trees topology, i.e., there are no cycles, and there exists only one path between any two PEs. PEs are connected to buses through logical ports. Each PE can have multiple ports, where each port connects to any single bus (two ports can be connected to the same bus, e.g., one master and one slave port). PEs perform computation and are communication endpoints, producing and consuming data. PEs do not perform any routing or bus translation. CEs have exactly two ports, where each port connects to a different bus. CEs are void of computation functionality and only provide bus connectivity and protocol translation.

The rest of this paper is organized as follows. After an overview of related work, our proposed design flow and the supported communication semantics are outlined in Sections II and III. In Section IV, the different communication layers, where our design flow is structured, are introduced. Sections V and VI describe the automatic network and link refinement, respectively. Finally, we present the experimental results in Section VII and conclude this paper with a summary and an outlook on future work in Section VIII.

C. Related Work

Many system-level design languages (SLDLs) are available in modeling systems at different levels of abstraction [1], [2]. However, SLDLs themselves do not define any actual design flows. Recent SLDLs have been used as vehicles for the so-called transaction-level modeling (TLM) [1], [3], [4]. However, the level of abstraction in TLM is not well defined, and TLM approaches, so far, focus only on simulation, which lacks a path to vertical integration for synthesis and implementation.

Several approaches deal with automatic generation and synthesis of communication [5], [6]. None of these, however, provides intermediate models that break the design task into smaller steps required for early exploration. Historically, a lot of work have focused on automating the decision making for communication design [7]–[11] without, however, providing corresponding design models or a path for implementation. More recently, work has been done to target automatic generation [12], [13], refinement [14], [15], or estimation [16]–[18] of communication; but in all cases, the approaches are limited to specific target architecture templates or narrow input model semantics. To our knowledge, no other approach that systematically deals with an automated implementation of customized communication over heterogeneous bus networks exists.

References [19] and [20] show SystemC/C++ extensions for modeling of systems and communication at different abstractions, with automatic or manual refinement between levels. However, none of the approaches includes a path to HW/SW implementation across multiple processor with support for advanced architectural features like arbitration and interrupt handling.

In [21]–[23], the network-on-chip (NoC) approach is proposed. Following the ISO-OSI reference layers [24], these approaches partition communication into layers to maximize reuse and provide a programmer with an abstraction of the underlying communication framework. We believe that the layered approach, if properly adapted, is well suited in addressing the problems in SoC communication. Thus, our work is layer-based in its models and transformations.

In the NoC area, a lot of recent researches deal with application mapping [25], [26] or architecture specialization [27], [28]. In contrast to NoCs which are based on regular homogeneous topologies that are scalable to varying requirements, our approach is targeted toward custom heterogeneous bus networks for multiprocessor SoC (MPSoC) designs.

In our previous works [29]–[31], we have introduced layer-based modeling and corresponding point tools for a stepwise network and communication refinement, respectively. Based on
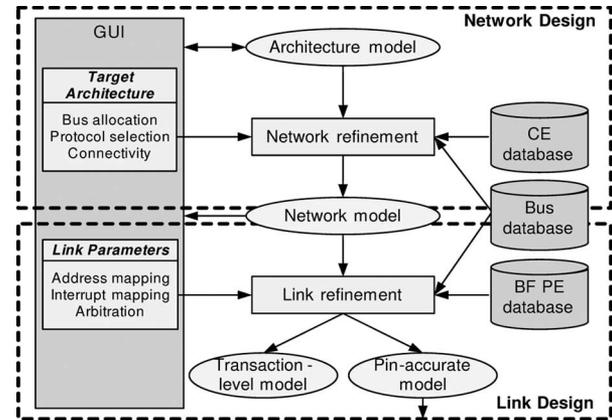

Fig. 1. Two-stage communication design flow.

these, this paper presents our overall communication design environment and clarifies and details our input semantics, database format, database contents, supported target architectures, and features implemented in each layer. Furthermore, we have extended our refinement tools to support a wider variety of target architectures and implementation options. Specifically, we support new synchronization mechanisms using shared interrupt or polling and the generation of arbiters and bus bridges. Finally, we have applied our flow to a larger set of benchmarks, including experiments with state-of-the-art MPSoC examples and industry-standard ARM/AMBA platforms.

II. COMMUNICATION DESIGN FLOW

Fig. 1 shows our proposed communication design flow which contains two successive stages: network design and link design. In each stage, design decisions are made and entered by the user through a graphical user interface. These decisions specify the desired target architecture and communication parameters. By applying the decisions to the input model, refinement tools automatically implement the communication and generate the resulting output model, relying on databases that provide models of CEs, buses, and bus-functional (BF) PEs.

A. Architecture Model

Our communication design flow starts with the architecture model which represents a virtual architecture of the system [32]. Each component in the model is a PE that executes its application in parallel with other PEs. Communication inside a PE is not a concern for our system communication synthesis. Inter-PE communication, however, takes place through abstract channels providing untimed message passing or shared memory semantics, as explained later in Section III.

Fig. 2 shows an example of an architecture model that consists of a processor CPU, a custom hardware unit HW1, a peripheral HW2, and a shared memory MEM that is running application behaviors B1 to B4. Inside the CPU, tasks are dynamically scheduled under the control of an operating system model [33]. In addition to communication via message-passing channels c1, c2, and c3 (see Section III), PEs exchange data through variables v1 and v2 that are stored in the MEM. The
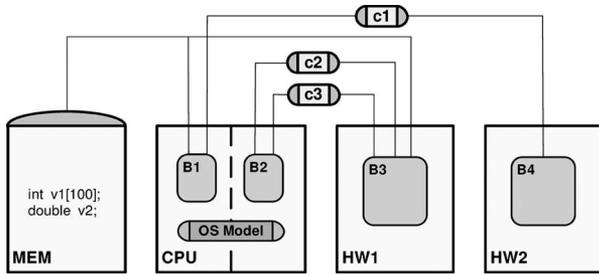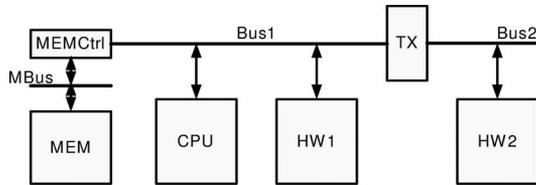
Fig. 2. Architecture model example.

Fig. 3. Target communication architecture for the example.

memory provides interface methods used to read and write each variable it contains.

B. Network Design

The overall network topology of PEs and CEs, together referred to as stations from here on, is defined in network design. CEs are introduced into the system to connect network segments and route communication between them. End-to-end messages between PEs are then implemented as packet transfers over point-to-point links between the stations.

1) Designer Decisions: Network design decisions include allocation of system buses, protocol selection, allocation and selection of CEs such as bridges and transducers, and definition of connectivity between stations and buses.

Fig. 3 shows the target architecture for the example shown in Fig. 2. Three buses have been allocated: a CPU bus (Bus1) as the main system bus, a memory bus (MBus), and a peripheral bus (Bus2). The CPU and HW1 are directly connected to Bus1, whereas a memory controller (MEMCtrl) bridges the system and memory bus protocols. Finally, a transducer CE (TX) is inserted to connect and translate between Bus1 and Bus2.

2) CE Database: CEs are taken from the CE database [34] and range from simple bridges, which transparently connect buses at the protocol level, up to complex transducers that contain buffers and translate between protocols. In our database, CEs carry attributes like name, type, and associated bus protocols. The database models, however, are empty shells that are void of any functionality. Their actual behavior will be synthesized and inserted by the refinement tools later.

C. Network Model

The network model [32] is an intermediate model in our design flow. It accurately reflects the network topology of the architecture. PEs and CEs communicate via logical links that carry streams of packets between directly connected
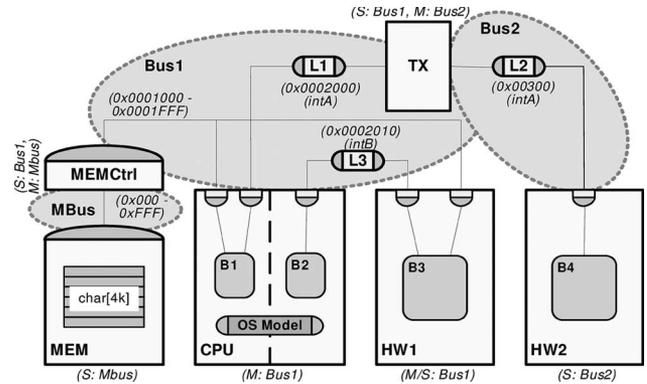

Fig. 4. Network model example with address and interrupt mapping.

components.[1] The network model can be simulated for fast validation of the overall network structure [29].

Fig. 4 shows the network model generated from the initial example (Fig. 2) using the target architecture in Fig. 3. Here, the CPU and HW1 communicate via a designated logical link L3. The communication between the CPU and HW2, on the other hand, is routed through transducer TX via logical links L1 and L2. Shared memory accesses to MEM are transparently forwarded by the MEMCtrl bridge. Inside the MEM, the variables are refined into an array of bytes reflecting the memory layout. Inside the PEs, the upper layers of the protocol stacks are generated in order to implement the application channels and variable accesses over the logical links.

D. Link Design

The logical links between adjacent stations in each network segment are implemented over the underlying physical bus. Packet transfers are implemented over the available bus transactions for the given protocol. In this process, timing-accurate protocol descriptions and BF models (BFMs) of PEs are taken from the bus and PE databases, respectively.

1) Designer Decisions: For each link in the system, designer decisions include the definition of masters and slaves, the selection of arbitration schemes and access priorities, and the assignment of bus addresses and interrupts. Depending on the channel requirements and bus capabilities, up to two addresses (for data transfers and polling) and one interrupt (for slave to master synchronization) are used per link and bus.

Fig. 4 shows the link design decisions for our example. The CPU is a master on Bus1, driving the MEM and HW1 as slaves. MEMCtrl is a slave on Bus1 and a master on MBus. HW1 is a bus-mastering direct-memory-access component on Bus1, acting as a slave which receives commands and as a master in accessing the memory. Finally, The HW2 is a slave on Bus2, where the transducer TX translates between Bus1 and Bus2. In this example, each logical link channel is assigned one interrupt and one address on its bus, e.g., L1 is assigned intA and address 0×00020000. The MEM is mapped into the

---

[1]Note that the number and the types of links in the network model depend on the decisions made during network design. This influences the link design choices about address and interrupt assignments.
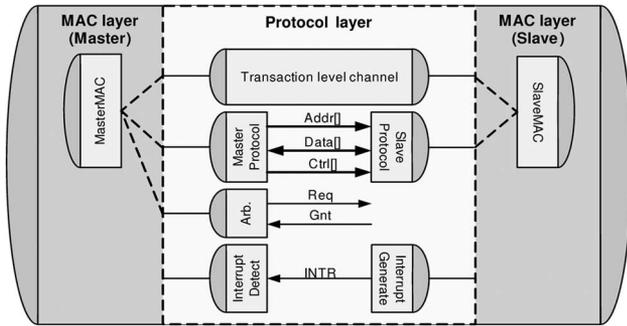
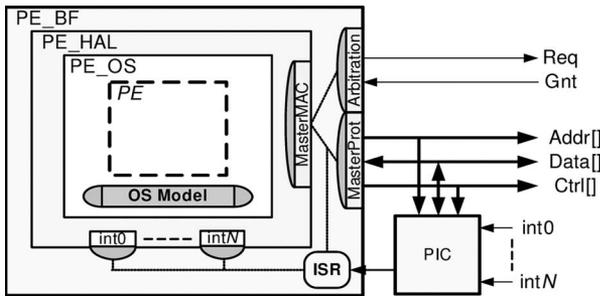Fig. 5.   Generic bus component model in the bus database.

Fig. 6.   Programmable component in the PE database.

address space of Bus1 at 0×0001000, where it occupies a range matching its size.

2) Bus Database: Our bus database contains models of buses and their protocols.[2] Our bus models consist of a stack of two layers, a protocol, and a media access (MAC) layer (Fig. 5) [34]. The protocol layer is connected to the signals representing the actual bus wires and implements the transactions defined by the bus protocol for data transfers, synchronization, and arbitration. In addition to this pin-accurate implementation, each bus model also includes a transaction-level version of the protocol layer. The MAC layer provides an abstraction of external communication into packet transfers and memory accesses. It uses the underlying bus protocol transactions to arbitrate media accesses and slice data packets into bus words. In our database, we also distinguish two sides for each layer in order to allow different implementations for masters and slaves.

3) BF PE Database: BFMs are needed for intellectual property (IP) components with fixed functionality and for programmable components with fixed communication interfaces. For the IPs, the model accurately describes the component interface at the pin level and provides functionality for simulation. For programmable components (Fig. 6), the model provides at least two layers: a BF layer describing the external pin interface and an internal hardware abstraction layer (HAL) describing the bus interface for access from the software inside. The HAL shell also provides templates of interrupt handlers for each external interrupt line of the processor. Together with models of the interrupt controller (PIC) and the interrupt service routine ISR,
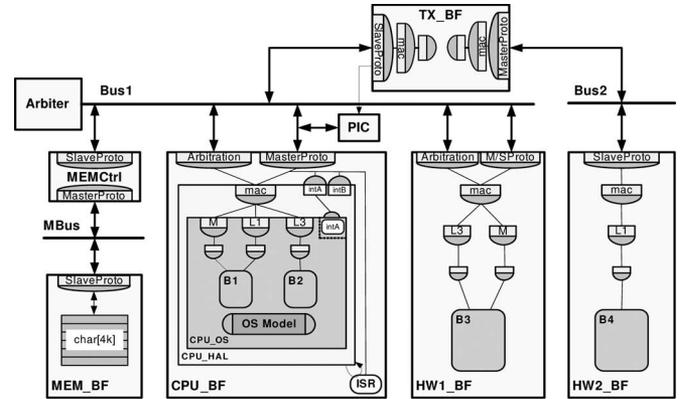

Fig. 7.   PAM example.

this accurately describes the interrupt handling of the processor. The HAL, in turn, can contain an OS shell (PE_OS) which includes an abstract model of the PEs operating system [33]. The OS shell and interrupt handlers will be filled with actual application tasks by the refinement tools later.

E. Transaction-Level and Pin-Accurate Models (PAM)

The transaction-level model and PAM [32] are the results of link design. These accurately reflect the system communication architecture in terms of both components and connections. At the top level, the PAM consists of a set of components connected by wires of the system buses. The components internally are refined down to timing-accurate BFMs communicating via their pins. The TLM, on the other hand, abstracts away pin-related protocol details [3]. Here, PEs and CEs communicate via transaction-level bus channels from the bus database. Inside the components, stacks of bus protocol layers implement each logical link down to the level of individual bus transactions or, in case of the PAM, down to the sampling and driving of individual wires. The TLM allows for a fast validation through simulation, whereas the PAM can be handed off to backend tools for RTL generation [35] and software synthesis [36].

Fig. 7 shows the PAM generated for Fig. 4. By reflecting the target architecture shown in Fig. 3, the four components communicate via buses Bus1, Bus2, and MBus. The MAC and protocol layer adapters from the bus database, together with automatically generated implementations of higher layers, are inserted into each component. For the programmable CPU, its BFM has been inserted from the PE database and connected at the MAC layer. The TLM generated for the example is equivalent to the PAM, except that the protocol layers are removed and the components communicate via corresponding TLM channels for each bus.

III. COMMUNICATION CHANNELS

We will now shift our attention to the communication semantics supported by our design environment. In general, we decided to support a limited but sufficiently general set of typical communication mechanisms. Specifically, our design flow supports synchronous and asynchronous message passing,
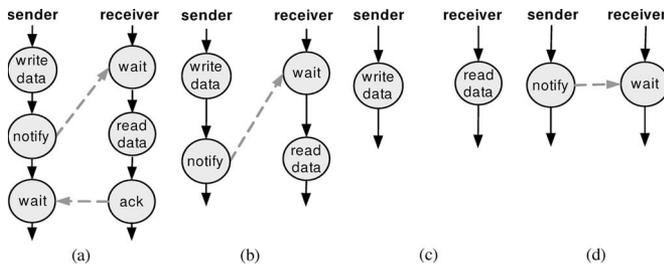
Fig. 8. Synchronization semantics of abstract communication channels. (a) Synchronous MP. (b) Asynchronous MP. (c) Memory. (d) Event.

memory access, and events. Fig. 8 shows the semantics of these four channels as state diagrams.

In a synchronous message passing [Fig. 8(a)], the sender and the receiver meet in a rendezvous fashion to safely exchange data. The sender stores the data in the channel, notifies the receiver, and then waits for acknowledgment. The receiver, on the other hand, waits for the data, copies them, and acknowledges the reception. In short, synchronous message passing utilizes a two-way handshake mechanism to ensure a reliable data transport. Both the receiver and the sender may be blocked in their execution.

In an asynchronous message passing [Fig. 8(b)], only the receiver may be blocked if data are not yet available. To avoid loss, sent data are stored in a queue channel[3] until they are received. Thus, asynchronous message passing is also reliable, but the sender does not know when the receiver actually gets the data.

A shared memory access [Fig. 8(c)] exhibits nonblocking communication for both the sender and the receiver. The sender simply writes data into the memory where they can be read by the receiver at any time. Since there is no built-in synchronization between the communicating parties, this type of data transfer is unreliable.

Finally, an event channel [Fig. 8(d)] exhibits pure synchronization semantics without any data transfer. The receiver simply waits for an event from the sender before proceeding in its execution. Note that this event channel can easily be combined with a memory channel to achieve reliable communication in a shared memory fashion.

Fig. 9 shows the four channels in our design flow. In the architecture model [Fig. 9(a)], all four channel types are supported. In the network model [Fig. 9(b)], both message-passing channels are implemented as synchronous channels via the introduced CE1. Buffering in the queue is now implemented explicitly as a buffer[4] in CE1. Note that, while communication over each link (PE1 to CE1 and CE1 to PE2) is now synchronous, the overall transfer from PE1 to PE2 may be asynchronous due to buffering in CE1.[5] In the TLM [Fig. 9(c)],

---

[3]To support asynchronous message passing, queues of infinite capacity are assumed in the input architecture model. The actual queue depths are implementation dependent.

[4]In the network model, buffers are of fixed sizes, which are determined by the user as part of the network refinement task. In case of zero buffering (no CEs), communication will end up being synchronous.

[5]For a synchronous message passing as required by the architecture model, protocols that restore synchronicity via an exchange of additional messages will be inserted into the PEs during refinement.

each bus in the network is now represented with one TLM and a set of interrupt channels (e.g., Bus1 and Int1). In the PAM [Fig. 9(d)], on the other hand, no channels are present anymore. Instead, bus wires are explicitly represented by signals of bitvector type.

IV. COMMUNICATION LAYERS

Our design approach is systematically structured along a layering of communication functionality [29]. We divide the layers based on separation of concerns, grouping of common functionality, and dependences across layers. Table I summarizes our layers for SoC communication which are based on the ISO-OSI reference model [24]. To minimize the impact of dependences, the ordering of steps, as identified by the ISO-OSI model, is the basis for our flow. However, due to the unique characteristics of SoC communication, we have tailored our layers specifically to SoC requirements. During synthesis, tools can optimize and customize layers further, depending on the specific requirements of the application and the target architecture, e.g., to eliminate unnecessary functionality or adjust hardware resource parameters. For example, transport layers are optimized away if there are no transducers between two PEs. Details of the optimizations performed for each layer are included in Sections V and VI.

Communication functionality in a SoC can often be implemented in different ways, depending on the application and the chosen target architecture. Table I summarizes the implementation options that are specifically supported by our environment [37]. Details of layer implementation and corresponding refinement are described in detail in the remainder of this paper.

V. NETWORK REFINEMENT

Network design refines the architecture into a network model that reflects the network topology of the system. Our refinement process can be divided into four main steps corresponding to the previously introduced layers: presentation, session, transport, and network layer.

A. Presentation Layer: Data Formatting

The presentation layer is responsible for data formatting in the PEs and on the network. When transferring data, we use a common format between the communicating partners. Similarly, we use a common data layout in shared memories accessed by different PEs. For each PE and shared memory, data layout parameters (size, alignment, and char width) are stored in the PE database. In addition, the database defines a canonical network endianess and includes corresponding byte-swapping functions for each PE.

For each message exchanged between PEs, its data type is converted into an untyped block of bytes passed to the next lower layer. In order to reduce overall network traffic, messages between each pair of PEs are formatted based on a fixed alignment of one (i.e., avoiding any byte padding) and by using the size parameters of the PE with smaller data types.[6]

---

[6]Note that, since value ranges are limited by the capabilities of the smaller PE, no precision is lost.
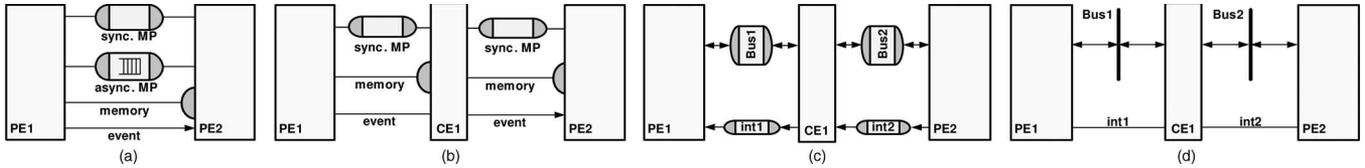
Fig. 9. Communication channels in different models of the design flow. (a) Architecture model. (b) Network model. (c) Transaction-level model. (d) PAM.

TABLE I
COMMUNICATION LAYERS

Task	OSI	Layer	Interface provided	Functionality	Implementation options
	7	Application	N/A	Computation	Produce/consume data, algorithms
Network Design	6	Presentation	End-to-end typed, named messages	Data formatting	Pack abstract data types into blocks of bytes • Fixed byte layout on the network • PE-specific size, alignment and endianess
	5	Session	End-to-end untyped, named messages	Synchronization	Realize synchronous message passing • Acknowledge messages
				Multiplexing	Merge application channels into session channels • Static session channels • Merging based on sequential order
	4	Transport	End-to-end streams of untyped messages	Packeting	Split messages into packets • Fixed size packets
				Flow control	N/A
				Error correction	N/A
	3	Network	End-to-end streams of packets	Routing	Store and forward packets via transducers • Dedicated, static routes
Link Design	2b	Link	Point-to-point logical links	Station typing	Master/slave assignments • Static, fixed master and fixed slave per link
				Synchronization	Pairwise synchronization of adjacent stations • Dedicated or shared interrupts • Polling
		Stream	Point-to-point control and data streams	Multiplexing	Merge packets into bus transfers • Merging based on sequential order
				Addressing	Set physical bus address • Dedicated bus addresses
	2a	Media Access	Shared medium byte streams	Data slicing	Slice block transfers into bus transactions • Available bus transaction types (e.g. burst)
				Arbitration	Regulate among concurrent bus master accesses • One central arbiter per bus
		Protocol	Unregulated word/frame transactions	Protocol timing	Execute bus cycles for transactions • Bus interface state machines
	1	Physical	Pins, wires	Driving, sampling	Signal levels (active high/low, tri-state)

The presentation layers in the components accessing a shared memory are responsible for converting variables in the application into size and offset values for shared memory accesses. For a memory, all variables stored inside are grouped into a single array of bytes following the memory's data layout.

B. Session Layer: Synchronization and Merging

The session layer is responsible for synchronization and multiplexing of different application channels into a number of end-to-end message streams. To maintain end-to-end synchronicity, we automatically insert acknowledge messages if the application requires a synchronous data transfer and if there are buffers in the path between endpoints. Since the channel configuration is statically known, dynamic session creation and tear-down are not needed. Instead, channel merging is implemented through static connectivity. Channels are merged based on their ordering. During refinement, sequential transactions are automatically combined using a channel-merging algorithm [30] in order to reduce the number of logical link channels in the system.

C. Transport Layer: Packeting

The transport layer is responsible for packeting, error correction, and flow control. In our case, underlying buses are
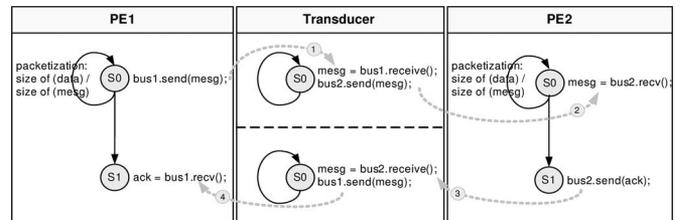
Fig. 10. Synchronous message passing over a transducer.

reliable, and error correction is not necessary. Furthermore, since lower layers can be dimensioned to avoid unnecessary network congestion,[7] flow control is not required either.

Packeting splits messages into smaller packets to reduce the required buffer sizes. Since packeting is only necessary in cases where a message is traversing a path that includes intermediate transducer stations, transport layers are empty for channels where communication endpoints are on the same (or bridged) bus. We will discuss an example of transducer communication, including implementations of transport and session layers, in Fig. 10 and Section V-D2.

[7]By implementing separate nonshared buffers for each transport path, we avoid blocking of streams due to buffer saturation by another stream.

Packet sizes are fixed and can be set by the designer for each channel. Since buffered transducers are typically used for translation of relatively slow low-volume peripheral I/O protocols (e.g., a UART that is used as a transducer between RS232 and CPU buses), packet size defaults to one. In all other cases, transport layer implementations, which pad the packets with dummy data if necessary, will be generated during refinement.

D. Network Layer: CE Synthesis and Insertion

The network layer inserts CEs from the database and routes end-to-end communication over the network of CEs and point-to-point logical links. In our target architectures, there exists only one path between any two PEs, and we assume reliable stations and links. Hence, routing is simplified to be deterministic and static over dedicated links between stations. In our environment, we distinguish between two types of CEs, bridges, and transducers. Both are empty shells in the database. Their functionality is synthesized as part of network refinement.

1) Bridges: Bridges transparently connect one bus to another at the protocol level. A bridge in the database has exactly two bus interfaces/ports. It is always a slave on one bus and a master on the other and transparently implements every matching transaction on its slave side by a corresponding transaction on its master side. The bridge maps the address space of the master side bus into the address space of its slave side bus where the bridge model 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 synchronization semantics inherent to 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. A good example of a bridge is a memory controller (see Section II) which bridges processor and memory buses.

2) Transducers: In cases where simple bus bridges are not sufficient (incompatible bus protocols), transducer CEs are used. Transducers operate on packets using a store-and-forward principle, routing packets between their incoming and outgoing links. Transducers can connect any two bus protocols and can be a master or a slave on either side. In contrast to a bridge, transducers internally buffer each individual bus transaction on one side before performing the equivalent transaction on the other side.

Transducers take part in high-level point-to-point communication protocols. As such, a transducer does not preserve synchronicity but rather decouples each end-to-end channel into point-to-point channel. Since memory transfers cannot be decoupled, memory interface transactions cannot be mapped and implemented over a transducer. In case of a synchronous message passing over a transducer, network refinement will automatically insert necessary protocol implementations into the PE endpoints in order to restore synchronicity lost over the transducer.

As shown in Fig. 10, a generated transducer contains corresponding state machines for each direction of each channel crossing the transducer. Each state machine contains its own local buffer, avoiding potential deadlocks. In the example, PE1 first packets the data and sends it to the transducer (1) which in turn forwards it to PE2 (2). In order to preserve synchronous semantics, PE2 then sends an acknowledge packet through the transducer (3) back to PE1 (4).

VI. LINK REFINEMENT

Link design refines the network model into a PAM or TLM of the system. Again, we divide our refinement into four steps, matching the link, stream, MAC, and protocol layers.

A. Link Layer: Synchronization

The link layer is responsible for implementing synchronization through interrupts, interrupt sharing, or polling. Link layers have different implementations depending on the type of station (master/slave). Methods on the master side wait for synchronization from slaves before invoking MAC layer methods to perform the actual data transfer. On the other hand, a slave will notify the master before continuing to listen for incoming data transfer requests.[8] No synchronization is necessary for slaves that are always ready (memory or memory-mapped I/O register accesses) or in case of buses with inherent two-way synchronization (e.g., RS232).

1) Interrupts: Our link refinement uses interrupts for synchronization if they are available in the master PE. BFMs for programmable components in the PE database include their interrupt capabilities. The BF PE layer defines the interrupt pins available at the physical component interface. The PEs HAL provides corresponding interrupt handler shells that define an insertion point for custom interrupt code inside a properly set up context. During link refinement, interrupt lines from slaves are connected to the interrupt pins of programmable components, and interrupt handlers and interrupt tasks are generated in the HAL and OS shell, respectively. In this process, the existing interrupt handler templates in the HAL are filled with codes to spawn an interrupt task that, in turn, notifies the link layer through a semaphore.

Fig. 11 shows the state machines synthesized for master and slave components that are synchronized by interrupts. When a slave process reaches the communication point, it notifies the master that it is ready to start data transfer by sending an interrupt (1). Upon receiving the interrupt event, the interrupt handler in the master sets a SlaveReady flag. The master side process waits until the flag is set to initiate the bus transfer (3). Finally, the slave component waits for the master to initiate the bus transfer by checking the address bus (4). This mechanism retains the two-way blocking property of any original synchronous message-passing communication. Once the data transfer is complete, the master component resets the SlaveReady flag to prepare for the next request.

---

[8]Note that, on a bus, master/slave assignments are independent from and orthogonal to sender/receiver status. Moreover, the same station can have different designations for different incoming/outgoing links.
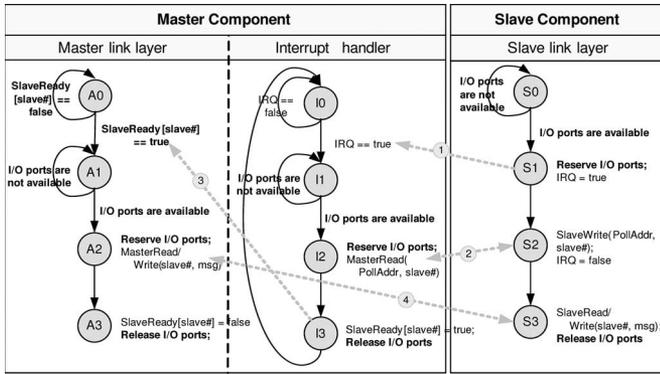
Fig. 11.   Slave to master synchronization.

Fig. 12.   Round-robin arbiter.

2) Interrupt Sharing and Polling: In case of an insufficient number of interrupts available in the master, we implement interrupt sharing. For each incoming request, interrupt handling is extended to determine the source of the request via polling of all interrupt sources. As shown in Fig. 11, link refinement optionally inserts additional code to poll slaves as part of interrupt-based synchronization (2): After receiving an interrupt, the master queries the state of all associated slaves using a unique address (PollAddr). When polling, only the slave that actually triggered the interrupt will answer the request by writing its ID to the bus. Note that, in case of master PEs without interrupt capabilities (e.g., nonprogrammable PEs), a similar polling of slaves is implemented. In this case, code is inserted to poll all connected slaves in regular intervals (busy-waiting) directly in the master's link layer.

B. Stream Layer: Addressing

The stream layer multiplexes multiple logical links over a single bus by separating them through addressing. During link design, each logical link is assigned a unique bus address by the designer.[9] Stream layer instances are generated inside the PEs and CEs to add the associated link address around each transaction before passing it to the shared MAC layer underneath. Note that, in case of point-to-point buses (e.g., serial RS232 protocols), addressing is not supported, and only one logical link per bus can be implemented.

C. MAC Layer: Data Slicing and Arbitration

The MAC layer is responsible for slicing of data packets into bus transactions and arbitration of conflicting transactions. For data slicing, MAC layer instances are taken from the bus database and inserted into the components during refinement. MAC layers split data packets into individual bus transactions supported by the underlying protocol. By using the available transaction types (including burst modes), slicing is optimized to perform at its best.

[9]Since in our bus-based systems we typically have a large number of bus addresses available, in this way, we can avoid sharing of bus addresses and associated overhead for additional IDs in the packet headers. Note that transfers of the same link carry the same address since they can implicitly be separated based on their order and the fixed packet size.
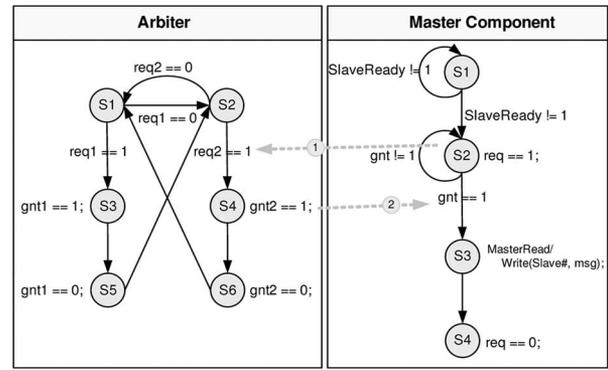
Arbitration becomes necessary in resolving conflicting accesses if multiple masters are connected to a bus. Inside the masters, arbitration protocols from the bus database are inserted as part of the bus master protocol. In case of a distributed arbitration scheme, arbitration protocols regulate bus accesses among themselves. In case of a centralized arbitration scheme, an arbiter is added to the system architecture and connected to arbitration protocol instances in all masters.

A generic arbiter supports bus request and bus grant ports (req and gnt, respectively), as shown in Fig. 12. Based on designer decisions, we generate a priority-based or round-robin arbiter component. Fig. 12 shows a round-robin arbiter with support for two masters. Upon receiving a request (1), the arbiter chooses a master to serve and sends the grant to the selected master (2).

D. Protocol Layer: Bus Interface Synthesis

The protocol layer is responsible for implementing the state machines that drive and sample the actual bus wires according to the timing diagrams and constraints defined by the protocol. For any synthesizable component, bus protocol state machines from the database are inserted into the component during refinement. On the other hand, special handling is required for BF PEs with fixed protocols and for bridges.

BFMs for PEs with fixed predefined bus interfaces (IPs or programmable processors) already include a timing-accurate description of the PEs interface at the pin level. For those PEs, BFMs are inserted from the database into the design.

Bridges transparently translate between two bus protocols directly at the protocol level. A bridge state machine is generated as the product of the two bus protocol state machines. In the process, the two protocols are properly interleaved such that data dependences and timing constraints are observed. Between listening for and serving transactions on the slave side, it interleaves corresponding mirror transactions on its master side (blocking the slave side if necessary).

Fig. 13 shows a bridge which handles data transfers between PE1 on its slave side (bus1) and PE2 on its master side (bus2). When idle, the bridge listens for requests on its slave side. Upon receiving a write request (1), it reads the data from the slave bus by serving the slave write request (2) before performing a
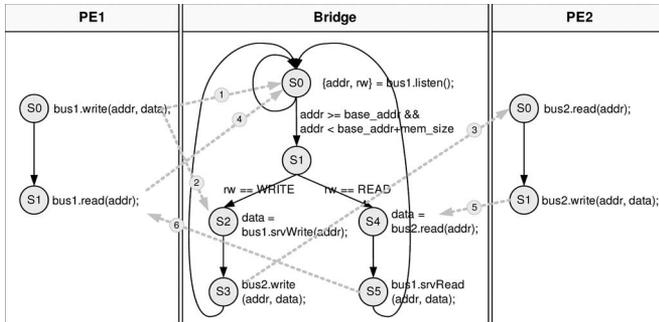
Fig. 13.   Bridge model in protocol layer.

TABLE II
DESIGN EXAMPLES AND TARGET ARCHITECTURES

Examples		Buses (Masters → Slaves)	Chnls (no.)	Traffic (bytes)
JPEG	A1	CF → IP	6	2244
	A2	CF → IP,HW	13	3420
Vocoder	A1	DSP → HW	12	46944
	A2	DSP → HW1,HW2	36	140832
	A3	DSP → HW1,HW2,HW3	42	154524
	A4	DSP1 → HW1 DSP2 → HW2	29	57160
MP3float	A1	CF → HW1	6	89862
	A2	CF → HW1,HW2,HW3 HW1 ↔ HW3 HW2 ↔ HW3	50	62406
	A3	CF → HW1,HW2,HW3,HW4 HW1 ↔ HW3 ↔ HW5 HW2 ↔ HW4 ↔ HW5	66	169747
MP3fix	A1	ARM → 2 I/O	4	5722
	A2	ARM → 2 I/O, LDCT, RDCT	22	80076
	A3	ARM → 2 I/O, LDCT, RDCT LDCT ↔ I/O RDCT ↔ I/O	23	22723
Baseband	A1	DSP → HW, 4 I/O, T CF,DMA → Mem,BR,T,DMA BR → DCT_IP	11	178500
Cellphone	A1	ARM → 4 I/O, 2 DCT, T LDCT,RDCT → I/O DSP → HW, 4 I/O, T	37	24484

VII. EXPERIMENTS AND RESULTS

We have implemented network and link refinement tools that automatically generate design models corresponding to our communication layers [30], [31]. Given designer decisions, our tools automatically refine a virtual architecture model of the system down to TLM and PAM implementations. Both refinement tools are integrated into our overall SoC design environment [38]. For our implementation, we used the SpecC SLDL [2] in describing the designs and, hence, also as input and output of the tools.

1) Setup: To demonstrate the feasibility and benefits of our approach, we have applied our tools to several industrial-strength examples: a JPEG encoder (JPEG) [39], a voice codec for the GSM standard for mobile telephony (Vocoder) [40], floating- and fixed-point versions of an MP3 decoder (MP3float and MP3fix), a mobile phone baseband example (Baseband) [29], and a platform (Cellphone) similar to the one used in the RAZR cellphone [41].

Table II summarizes the target architectures and parameters of the examples. Each architecture is specified as lists of masters and slaves for each bus. The bus type is implicitly determined to be the protocol of the primary master on the bus. JPEG and MPfloat examples use Motorola ColdFire processors (CF) assisted by various configurations of custom hardware (HW) and DCT (IP) blocks. Vocoder uses a Motorola DSP56600 processor (DSP) and custom hardware coprocessors (HW). MP3fix is based on an ARM platform with additional IO and DCT units. In all cases, we have examined various communication architectures using DSP, CF, ARM (AMBA AHB), and simple handshake buses.

Baseband and Cellphone examples are true system designs. Baseband combines JPEG on CF with the Vocoder on DSP. Cellphone consists of an ARM subsystem running MP3fix and JPEG tasks in connection with a DSP subsystem running the Vocoder (Fig. 14). In both cases, subsystems communicate via transducers (T) that connect CF/AMBA and DSP buses.

Table II also shows the number of inter-PE channels and the total inter-PE traffic for each example. We have successfully simulated all generated models for functional correctness.

2) Results: Table III shows the results of design space exploration for the different examples. Overall model complexities are given in terms of code size using lines of code (LOC) as a metric. The results show significant growth in complexity from the input to the generated output models due to the extra implementation detail at lower abstraction levels. To quantify the actual refinement effort, the number of modified lines is calculated as the sum of lines inserted and lines deleted. Codes coming from database models are listed separately (excluded from the modified LOC count). Even when using optimistic assumptions (e.g., that a designer can correctly modify 10 LOC/h, including testing and debugging), manual refinement would require days for our reasonably complex designs. Our automatic refinement, on the other hand, completes this task in a few seconds.[10] Therefore, our approach yields significant productivity gains.

3) Analysis: Our approach benefits from automating the straightforward yet tedious and error-prone model rewriting process. Our experiments show that automatically generated models provide a functionally correct and structurally and timing-accurate representation of the target design in a manner very similar to models typically implemented by designers. In addition, due to complexity and time pressure, optimizations that can be performed by designers are limited, whereas automated tools can easily exploit optimization opportunities. For example, the channel merging implemented by our network refinement tool (Section V-B) significantly reduces the design complexity (as indicated by the code size of the network models) in the JPEG and Vocoder examples.

---

[10]In this comparison, we assumed that all designer decisions are given in both cases, i.e., the time required for decision making is the same.

write transaction on its master side (3). On the other hand, when receiving a read request (4), it first performs a corresponding read transaction on its master side (5) before serving the read request on the slave side by writing the data received on the master side to the slave bus (6).
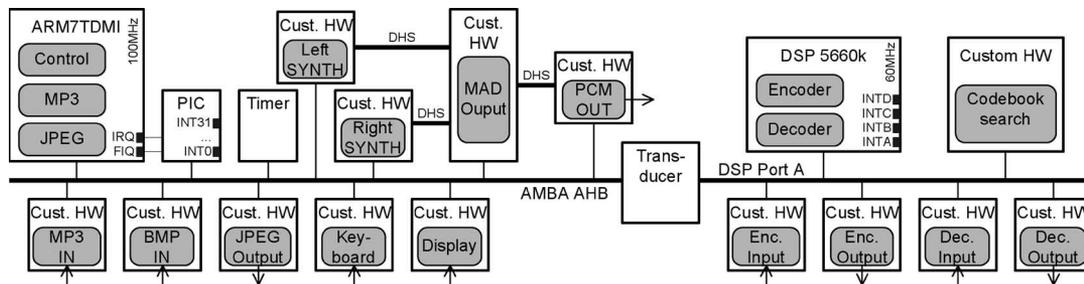
Fig. 14.   Cellphone example platform.

TABLE III
RESULTS FOR EXPLORATION EXPERIMENTS

Examples		Model Size (LOC)			DB Size (LOC)	Refinement Effort (Modified LOC)			Refinement Time Tool		
		Arch	Net	PAM		NR	LR	Total	NR	LR	Total
JPEG	A1	3451	3464	5250	1618	137	351	488	0.08 s	0.10 s	0.18 s
	A2	3712	3755	5655	1768	257	303	560	0.07 s	0.10 s	0.17 s
Vocoder	A1	10972	10980	11740	487	170	341	511	0.27 s	0.31 s	0.58 s
	A2	11386	11415	12205	487	223	405	628	0.34 s	0.33 s	0.67 s
	A3	11263	12276	13096	487	559	489	1048	0.43 s	0.39 s	0.82 s
	A4	13986	14033	15220	674	369	757	1126	0.45 s	0.84 s	1.29 s
MP3float	A1	30008	30049	32008	1584	212	904	1116	0.47 s	0.58 s	1.05 s
	A2	30923	30866	32974	1818	781	1160	1941	0.68 s	0.83 s	1.51 s
	A3	33794	33905	36361	1818	1181	1198	2379	0.92 s	1.06 s	1.98 s
MP3fix	A1	17279	17485	23541	4865	288	1283	1571	0.69 s	0.83 s	1.52 s
	A2	18977	20237	26291	4525	1738	1663	3401	2.45 s	1.09 s	3.54 s
	A3	18852	19890	26845	4909	1480	2272	3752	1.97 s	1.15 s	3.12 s
Baseband	A1	19754	20227	23027	2212	1195	1288	2483	0.75 s	1.02 s	1.77 s
Cellphone	A1	22246	23034	33106	5894	2059	3949	6008	3.05 s	1.82 s	4.87 s

As a result, using our approach, we are able to efficiently and rapidly explore the communication design space. For instance, for the Vocoder example, we were able to explore the design space and arrive at an optimal solution in less than 1 h, including the time needed for model simulations [30].

VIII. SUMMARY AND CONCLUSION

In this paper, we have presented a communication design flow with well-defined design steps and design models. By starting from a virtual architecture model with different types of abstract message-passing communication, a design model is automatically refined to a transaction-level and pin-accurate implementation through network and link design stages. We have demonstrated the feasibility and benefits of our approach using industrial-strength examples.

Our design flow supports a wide range of target communication architectures with different media and protocols. It is systematically structured along a layering of communication functionality where communication layers have been identified, defined, and tailored based on the specific requirements of SoC design. Our flow includes customization and optimization of communication layers using an automatic generation of application- and platform-specific architectures.

Out of many possible models, we have defined intermediate models based on accuracy versus simulation speed tradeoffs, allowing early validation of critical design decisions. In between design stages, the network model defines the implementation of the end-to-end network on top of point-to-point logical links. Furthermore, the transaction-level model provides an accurate simulation at a significantly improved speed compared to the PAM.

All our models and layers have been systematically defined such that they can be automatically generated. Automating the tedious and error-prone process of refining high-level communication abstractions to actual implementation results in significant gains in productivity, thus enabling rapid early exploration of the communication design space.

In the future, we plan to extend our databases with additional components, including advanced bus structures like crossbars or bus matrices (e.g., AMBA AXI). Further future work includes adding algorithms for decision making to provide a completely automated synthesis process. Finally, we plan to extend design tasks and refinement tools to implement error-correction, flow control, and dynamic routing for off-chip, long-latency, and error-prone communication media.

REFERENCES

[1] T. Grötker, S. Liao, G. Martin, and S. Swan, *System Design With SystemC*. Norwell, MA: Kluwer, Mar. 2002.
[2] D. D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, and S. Zhao, *SpecC: Specification Language and Design Methodology*. Norwell, MA: Kluwer, 2000.
[3] G. Schirner and R. Dömer, "Quantitative analysis of transaction level models for the AMBA bus," in *Proc. DATE*, Mar. 2006, pp. 1–6.
[4] M. Coppola, S. Curaba, M. Grammatikakis, and G. Maruccia, "IPSIM: SystemC 3.0 enhancements for communication refinement," in *Proc. DATE*, Mar. 2003, pp. 106–111.
[5] W. O. Cesário, D. Lyonnard, G. Nicolescu, Y. Paviot, S. Yoo, and A. A. Jerraya, "Multiprocessor SoC platforms: A component-based design approach," *IEEE Des. Test Comput.*, vol. 19, no. 6, pp. 52–63, Nov./Dec. 2002.
[6] K. van Rompaey, D. Verkest, I. Bolsens, and H. D. Man, "CoWare: A design environment for heterogeneous hardware/software systems," in *Proc. Eur. Des. Autom. Conf.*, 1996, pp. 252–257.
[7] T.-Y. Yen and W. Wolf, "Communication synthesis for distributed embedded systems," in *Proc. ICCAD*, Nov. 1995, pp. 288–294.

[8] M. Gasteier, M. Münch, and M. Glesner, "Generation of interconnect topologies for communication synthesis," in *Proc. DATE*, Mar. 1998, pp. 36–42.

[9] M. Gasteier and M. Glesner, "Bus-based communication synthesis on system-level," in *Proc. ISSS*, Nov. 1996, pp. 65–70.

[10] R. B. Ortega and G. Borriello, "Communication synthesis for distributed embedded systems," in *Proc. ICCAD*, Nov. 1998, pp. 437–444.

[11] K. Lahiri, A. Raghunathan, and S. Dey, "Efficient exploration of the SoC communication architecture design space," in *Proc. ICCAD*, Nov. 2000, pp. 424–430.

[12] W. Klingauf, H. Gädke, and R. Günzel, "TRAIN: A virtual transaction layer architecture for TLM-based HW/SW codesign of synthesizable MPSoC," in *Proc. DATE*, Mar. 2006, pp. 1–6.

[13] I. Bolsens, H. D. Man, B. Lin, K. V. Rompay, S. Vercauteren, and D. Verkest, "Hardware/software co-design of the digital telecommunication systems," *Proc. IEEE*, vol. 85, no. 3, pp. 391–418, Mar. 1997.

[14] D. Lyonnard, S. Yoo, A. Baghdadi, and A. A. Jerraya, "Automatic generation of application-specific architectures for heterogeneous multiprocessor system-on-chip," in *Proc. DAC*, Jun. 2001, pp. 518–523.

[15] W. O. Cesario, A. Baghdadi, L. Gauthier, D. Lyonnard, G. Nicolescu, Y. Paviot, S. Yoo, A. A. Jerraya, and M. Diaz-Nava, "Component-based design approach for multicore SoCs," in *Proc. DAC*, Jun. 2002, pp. 789–794.

[16] K. Lahiri, A. Raghunathan, and S. Dey, "System-level performance analysis for designing on-chip communication architectures," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 20, no. 6, pp. 768–783, Jun. 2001.

[17] X. Zhu and S. Malik, "A hierarchical modeling framework for on-chip communication architectures," in *Proc. ICCAD*, Nov. 2002, pp. 663–670.

[18] M. Lajolo, C. Passerone, and L. Lavagno, "Scalable techniques for system-level co-simulation and co-estimation," *Proc. Inst. Electr. Eng.— Computers Digital Techniques*, vol. 150, no. 4, pp. 227–238, Jul. 2003.

[19] R. Siegmund and D. Müller, "SystemC^{SV}: An extension of SystemC for mixed multi-level communication modeling and interface-based system design," in *Proc. DATE*, Mar. 2001, pp. 26–32.

[20] R. Pasko, S. Vernalde, and P. Schaumont, "Techniques to evolve a C++ based system design language," in *Proc. DATE*, Mar. 2002, pp. 302–309.

[21] M. Sgroi, M. Sheets, M. Mihal, K. Keutzer, S. Malik, J. Rabaey, and A. Sangiovanni-Vincentelli, "Addressing the system-on-a-chip interconnect woes through communication-based design," in *Proc. DAC*, Jun. 2001, pp. 667–672.

[22] L. Benini and G. D. Micheli, "Networks on chips: A new SoC paradigm," *Computer*, vol. 35, no. 1, pp. 70–78, Jan. 2002.

[23] A. Jantsch and H. Tenhunen, Eds., Networks on Chip. Norwell, MA: Kluwer, 2003.

[24] *Reference Model of Open System Interconnection*, 2nd ed., International Organization for Standardization, 1994, ISO/IEC 7498 Standard.

[25] J. Hu and R. Marculescu, "Energy- and performance-aware mapping for regular NoC architectures," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 24, no. 4, pp. 551–562, Apr. 2005.

[26] S. Murali and G. D. Micheli, "SUNMAP: A tool for automatic topology selection and generation for NoCs," in *Proc. DAC*, Jun. 2004, pp. 914–919.

[27] U. Y. Ogras and R. Marculescu, "Energy- and performance-driven customized architecture synthesis using a decomposition approach," in *Proc. DATE*, Mar. 2005, pp. 352–357.

[28] K. Srinivasan, K. S. Chatha, and G. Konjevod, "Linear-programming-based techniques for synthesis of network-on-chip architectures," *IEEE Trans. VLSI Syst.*, vol. 14, no. 4, pp. 407–420, Apr. 2006.

[29] A. Gerstlauer, D. Shin, R. Dömer, and D. D. Gajski, "System-level communication modeling for network-on-chip synthesis," in *Proc. ASPDAC*, Jan. 2005, pp. 45–48.

[30] D. Shin, A. Gerstlauer, R. Dömer, and D. D. Gajski, "Automatic network generation for system-on-chip communication design," in *Proc. CODES + ISSS*, Sep. 2005, pp. 255–260.

[31] D. Shin, A. Gerstlauer, R. Doemer, and D. D. Gajski, "Automatic generation of communication architectures," in *From Specification to Embedded Systems Application*, A. Rettberg, M. C. Zanella, and F. J. Rammig, Eds. New York: Springer-Verlag, Aug. 2005.

[32] D. Shin *et al.*, "System-on-chip modeling style guides," UC Irvine, Irvine, CA, Tech. Rep. CECS-TR-04-22 through CECS-TR-04-24, Jul. 2004.

[33] A. Gerstlauer, H. Yu, and D. D. Gajski, "RTOS modeling for system level design," in *Proc. DATE*, Munich, Germany, Mar. 2003, pp. 130–135.

[34] A. Gerstlauer *et al.*, "System-on-chip component models," UC Irvine, Irvine, CA, Tech. Rep. CECS-TR-06-10, May 2006.

[35] D. Shin *et al.*, "C-based interactive RTL design methodology," UC Irvine, Irvine, CA, Tech. Rep. CECS-TR-03-42, Dec. 2003.

[36] H. Yu, R. Dömer, and D. D. Gajski, "Embedded software generation from system level design languages," in *Proc. ASPDAC*, Jan. 2004, pp. 463–468.

[37] A. Gerstlauer *et al.*, "Necessary and sufficient functionality and parameters for SoC communication," UC Irvine, Irvine, CA, Tech. Rep. CECS-TR-06-01, May 2006.

[38] S. Abdi *et al.*, "System-on-chip environment (SCE Version 2.2.0 beta): Tutorial," UC Irvine, Irvine, CA, Tech. Rep. CECS-TR-03-41, Jul. 2003.

[39] *Digital Compression and Coding of Continous-Tone Still Images*, International Telecommunication Union (ITU), Sep. 1992, ITU Recommendation T.81.

[40] *Digital cellular telecommunications system; Enhanced Full Rate (EFR) speech transcoding*, Final draft ed., European Telecommunication Standards Institute (ETSI), 1996, GSM 06.60.

[41] C. Giridhar, *Trendy Phones Incorporate Sophisticated Engineering, EDN Asia*. [Online]. Available: http://www.edn.com/index.asp?layout=article&articleid=CA6290467

Andreas Gerstlauer (S'97–M'04) received the Dipl.Ing. degree in electrical engineering from the University of Stuttgart, Stuttgart, Germany, in 1997 and the M.S. and Ph.D. degrees in information and computer science from the University of California, Irvine (UCI), in 1998 and 2004, respectively.

He is currently an Assistant Researcher with the Center for Embedded Computer Systems, UCI. His research interests include electronic system-level (ESL) design languages, methodologies and tools, system modeling, and embedded hardware and software synthesis.

Dongwan Shin (S'00–M'04) received the M.S. degree in electronics engineering from Seoul National University, Seoul, Korea, in 1997 and the Ph.D. degree in information and computer science from the University of California, Irvine (UCI), in 2004.

From 1997 to 1999, he was with LG Semincon, Company, Ltd., Seoul, Korea. Since 2004, he has been an Assistant Project Scientist with the Center for Embedded Computer Systems, UCI. His research interests include system-level design automation, high-level synthesis, and low-power system design.
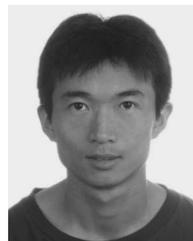
Junyu Peng (S'00–M'04) received the B.S. degree in physics from Tsinghua University, Beijing, China, in 1994 and the Ph.D. degree in information and computer science from the University of California, Irvine (UCI), in 2004.

He is currently an Assistant Project Scientist with the Center for Embedded Computer Systems, UCI. His research interests include system-level modeling and automatic synthesis of embedded systems and embedded software.

Rainer Dömer (S'95–M'00) received the Ph.D. degree in information and computer science from the University of Dortmund, Dortmund, Germany, in 2000.

He is currently an Assistant Professor in electrical engineering and computer science with the Department of Electrical Engineering and Computer Science, University of California, Irvine (UCI). He is also a member of the Center for Embedded Computer Systems, UCI. His research interests include system-level design and methodologies, embedded computer systems, specification and modeling languages, system-on-chip design, and embedded software.

Daniel D. Gajski (M'77–SM'83–F'94) received the Dipl.Ing. and M.S. degrees in electrical engineering from the University of Zagreb, Zagreb, Croatia, and the Ph.D. degree in computer and information sciences from the University of Pennsylvania, Philadelphia.

After 10 years as a Professor with the University of Illinois, he has joined University of California, Irvine (UCI), where he presently holds The Henry Samueli Endowed Chair in Computer System Design. He directs the UCI Center for Embedded Computer Systems, with a research mission that aims to incorporate embedded systems into automotive, communications, and medical applications. Being a leader in the areas of embedded systems, design methodologies, and languages, he headed the research teams that created new design methodologies, tools, and languages. He was instrumental in developing formalisms and algorithms for high-level synthesis, the definition of the finite-state-machine with data, system level languages such as SpecCharts and SpecC, and design tools such as SpecSyn and system-on-chip environment. He has authored over 300 papers and numerous textbooks, including *Principles of Digital Design* (Englewood Cliffs, NJ: Prentice Hall, 1997) that has been translated into several languages.