OCCN: A NoC Modeling Framework for Design Exploration

Marcello Coppola[1], Stephane Curaba[1], Miltos D. Grammatikakis[2, 3], Riccardo Locatelli [4], Giuseppe Maruccia[1]and Francesco Papariello[1]
[1] ST Microelectronics, AST Grenoble Lab, 12 Jules Horowitz 38019 Grenoble, France
{marcello.coppola, stephane.curaba, giuseppe.maruccia, francesco.papariello}@st.com
[2] ISD S.A., K. Varnali 22, 15233 Halandri, Greece, mdgramma@isd.gr
[3] Computer Science Group, TEI-Crete, Heraklion, Crete, Greece
[4] Info Engineering, Uni-Pisa, v. Diotisalvi 2, 56122 Pisa, Italy, r.locatelli@iet.unipi.it

Abstract

The On-Chip Communication Network (OCCN) project provides an efficient framework, developed within SourceForge, for the specification, modeling, simulation, and design exploration of network on-chip (NoC) based on an object-oriented C++ library built on top of SystemC. OCCN is shaped by our experience in developing communication architectures for different System-on-Chip (SoC). OCCN increases the productivity of developing communication driver models through the definition of a universal Application Programming Interface (API). This API provides a new design pattern that enables creation and reuse of executable transaction level models (TLMs) across a variety of SystemC-based environments and simulation platforms. It also addresses model portability, simulation platform independence, interoperability, and high-level performance modeling issues.

1. Introduction

Due to steady downscaling of CMOS device dimensions, manufacturers are increasing the amount of functionality on a single chip. It is expected that by the year 2005, complex systems, called *Multiprocessor System-on-Chip* (MPSoC), will contain billions of transistors. The canonical MPSoC view consists of a number of processing elements (PEs) and storage elements (SEs) connected by a complex communication architecture. PEs implement one or more functions using programmable components, including general purpose processors and specialized cores, such as digital signal processor (DSP) and VLIW cores, as well as embedded hardware, such as FPGA or application-specific intellectual property (IP), analog front-end, peripheral devices, and breakthrough technologies, such as micro-electro-mechanical structures (MEMS) [16] and micro-electro-fluidic bio-chips (MEFS) [52] .

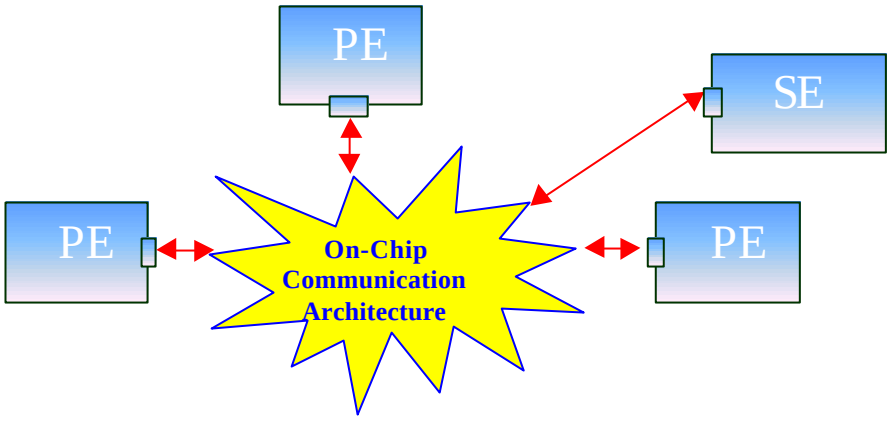

Figure 1. MPSoC configured with on-chip communication architecture, processing, and storage elements

As shown in Figure 1, a global On-Chip Communication Architecture (OCCA) interconnects these devices, using a full crossbar, a bus-based system, a multistage interconnection network, or a point-to-point static topology [35]. OCCA bandwidth and data transfer parameters, e.g. acquisition delay and access time for single transfer or burst, often limit overall SoC performance.

OCCA provides the communication mechanisms necessary for distributed computation among different processing elements. For high performance protocols, crossbars are attractive, since they avoid bottlenecks associated with shared bus lines and centralized shared memory switches. Currently there are two prominent types of OCCA.

- T*raditional and semi-traditional on-chip buses*, such as AMBA[2], STBus [44, 45], and Core Connect [31]. Bus-based networks are usually synchronous and offer several variants. Buses may be reconfigurable, hierarchical (partitionable into smaller sub-systems), might allow for exclusive or concurrent read/write, and may provide multicasting or broadcasting facilities.
- The next generation *network on-chip* is able to meet application-specific requirements through a powerful communication fabric based on repeaters, buffer pools, and a complex protocol stack [3, 23, 35]. Innovative network on-chip architectures include LIP6's SPIN [23], MIT's Raw network [39], and VTT's Eclipse [21].

The Spin NOC, proposed by the University of Pierre and Marie Curie - LIP6, uses packet switching with wormhole routing and input queuing in a fat tree topology. It is a scalable network for data transport, but uses a bus network for control. It is a best-effort network, optimized for average performance, e.g. by the use of optimistic flow control coupled with deflection routing. Commitment is given for packet delivery, but latency bounds are only given statistically. However, input queuing causes head-of-line blocking effects, thus being a limiting factor for providing a latency guaranty for the data network.

The Raw network tries to implement a simple, highly parallel VLSI architecture by fully exposing low-level details of the hardware to the compiler, so that the compiler (or the software) can determine and implement the best allocation of resources, including scheduling, communication, computation, and synchronization, for each possible application. Raw implements fine-grain communication between local, replicated processing elements and, thus, is able to exploit parallelism in data parallel applications, such as multimedia processing.

Embedded Chip-Level Integrated Parallel SupErcomputer (Eclipse) is a scalable high-performance computing architecture for NoC. The PEs are homogeneous, multithreaded, with dedicated instruction memory, and highly interleaved (cacheless) memory modules. The interconnect is a high capacity, 2-d sparse-mesh that exploits locality and avoids memory hotspots (and partly network congestion) through randomized hashing of memory words around a module's memory banks. The programming model is a simple lock-step-synchronous EREW PRAM model.

OCCA choice is critical to performance and scalability of MPSoC[1]. An OCCA design for a network processor, such as MIT's Raw network on-chip, will have different communication semantics from another OCCA design for multimedia MPSoC.

[1] SoC performance varies up to 250% depending on OCCA, and up to 600% depending on communication traffic [3].

Furthermore, for achieving cost-effectively OCCA scalability, we must consider various architectural, algorithmic, and physical constraint issues arising from Technology [33]. Thus, within OCCA modeling we must consider architecture realizability and serviceability. Although efficient programmability is also important, it relates to high-level communication and synchronization libraries, as well as system and application software issues that fall outside of the OCCA scope [24].

Realizability is associated to several network design issues that control system parallelism by limiting the concurrency level [25], such as

- network topology, size, packetization (including header parsing, packet classification, lookup, data encoding, and compression), switching technique, flow control, traffic shaping, packet admission control, congestion avoidance, routing strategy, queuing and robust buffer management, level of multicasting, cache hierarchy, multithreading and pre-fetching, and software overheads,
- memory technology, hierarchy, and consistency model for shared memory, and architecture efficiency and resource utilization metrics, e.g. power consumption, processor load, RTOS context switch delay, delays for other RTOS operations, device driver execution time, and reliability (including cell loss), bandwidth, and latency (including hit ratios) for a given application, network, or memory hierarchy,
- VLSI layout complexity, such as time-area tradeoff and clock-synchronization to avoid skewing; an open question is “for a given bisection bandwidth, pin count, and signal delay model, maximize clock speed and wire length within the chip”.

The new nanometer technologies provide very high integration capabilities, allowing the implementation of very complex systems with several billions of transistors on a single chip. However, two main challenges should be addressed.

- How to handle escalating design complexity and time-to-market pressures for complex systems, including partitioning into interconnecting blocks, hardware/software partitioning of system functionality, interconnect design with associated delays, synchronization between signals, and data routing.
- How to solve issues related to the technologies themselves, such as cross-talk between wires, increased impact of the parasitic capacitance and resistors in the global behavioral of system, voltage swing, leakage current, and power consumption.

There is no doubt that future NoC systems will generate errors, and their reliability should be considered from the system-level design phase [18]. This is due to the non-negligible probability of failure of an element in a complex NoC that causes transient, intermittent, and permanent hardware and software errors, especially in corner situations, to occur anytime. Thus, we characterize NoC serviceability with corresponding reliability, availability, and performability metrics.

- Reliability refers to the probability that the system is operational during a specific time interval. Reliability is important for mission-critical and real-time systems, since it assumes that system repair is impossible. Thus, reliability refers to the system's ability to support a certain quality of service (QoS), i.e. latency, throughput, power consumption, and packet loss requirements in a specified operational environment. Notice that QoS must often take into account future traffic requirements, e.g. arising from multimedia applications, scaling of existing applications, and network evolution, as well as cost vs. productivity gain issues.
- System dependability and maintainability models analyze transient, intermittent, and permanent hardware and software faults. While permanent faults cause an irreversible

system fault, some faults last for a short period of time, e.g. nonrecurring transient faults and recurring intermittent faults. When repairs are feasible, fault recovery is usually based on detection (through checkpoints and diagnostics), isolation, rollback, and reconfiguration. Then, we define the availability metric as the average fraction of time that the system is operational within a specific time interval.

- While reliability, availability and fault-recovery are based on two-state component characterization (faulty, or good), system performability measures degraded system operation in the presence of faults, e.g. increased congestion, packet latency, and distance to destination when there is no loss (or limited loss) of system connectivity.

The rapid evolution of *Electronic System Level* (ESL) methodology addresses MPSoC design. ESL focuses on the functionality and relationships of the primary system components, separating system design from implementation. Low-level implementation issues greatly increase the number of parameters and constraints in the design space, thus extremely complicating optimal design selection and verification efforts. Similar to near-optimal combinatorial algorithms, e.g. travelling salesman heuristics, ESL models effectively prune away poor design choices by identifying bottlenecks, and focus on closely examining feasible options. Thus, for the design of MPSoC, OCCA (or NoC) design space exploration based on analytical modeling and simulation, instead of actual system prototyping, provides rapid, high quality, cost-effective design in a time-critical fashion by evaluating a vast number of communication configurations [1, 17, 33, 34, 37, 38, 51].

The proposed On-Chip Communication Network methodology (OCCN) is largely based on the experiences gained from developing communication architectures for different SoC. OCCN-based models have already been used by Academia and Industry, such as ST Microelectronics, for developing and exploring a new design methodology for on-chip communication networks. This methodology has enabled the design of next generation networking and home gateway applications, and complex on-chip communication networks, such as the STMicroelectronics proprietary bus STBus, a real product found today in almost any digital satellite decoder [44, 45].

OCCN focuses on modeling complex on-chip communication network by providing a flexible, open-source, object-oriented C++-based library built on top of SystemC. We have also developed a methodology for testing the OCCN library and for using it in modeling various on-chip communication architectures.

Next, in Section 2, we focus on generic modeling features, such as abstraction levels, separation of function specification from architecture and communication from computation, and layering that OCCN always provides. In Section 3, we provide a detailed description of the OCCN API, focusing on the establishment of inter-module communication refinement through a layering approach based on two SystemC-based modeling objects: the Protocol Data Unit (Pdu), and the MasterPort/SlavePort interface. In Section 3, we also describe a generic, reusable, and robust OCCN statistical model library for exploring system architecture performance issues in SystemC models. In Section 4, we outline a transmitter/receiver case study on OCCN-based modeling, illustrating inter-module communication refinement and high-level system performance modeling. In Section 5, we provide conclusions and ongoing extensions to OCCN. We conclude this paper with a list of references.

2. Generic Features for NoC Modeling

OCCN extends state-of-the-art communication refinement by presenting the user with a powerful, simple, flexible and compositional approach that enables rapid IP design and system level reuse. The generic features of our NoC modeling approach, involving abstraction levels, separation of communication and computation, and communication layering are described in this Section. These issues are also described in the VSIA model taxonomy that provides a classification scheme for categorizing SoC models [49].

2.1 Abstraction Levels

A key aspect in SoC design is model creation. A model is a concrete representation of functionality for a target SoC. In contrast to component models, *virtual SoC prototype* (or virtual platform) refers to modeling the overall SoC. Thus, virtual SoC combines processor emulation by back-annotating delays for specific applications, processor simulation using an instruction set simulator and compiler, e.g. for ARM V4, PowerPC, ST20, or Stanford DLX model, RTOS modeling, e.g. using a pre-emptive, static or dynamic priority scheduler, on-chip communication simulation (including OCCA models), peripheral simulation (models of the hardware IP blocks, e.g. I/O, timers, and DMA, and environment simulation (including models of real stimuli). Virtual platform enables integration and simulation of new functionalities, evaluation of the impact that these functionalities have on different SoC architectural solutions, and exploration of hardware/software partitioning and re-use at any level of abstraction.

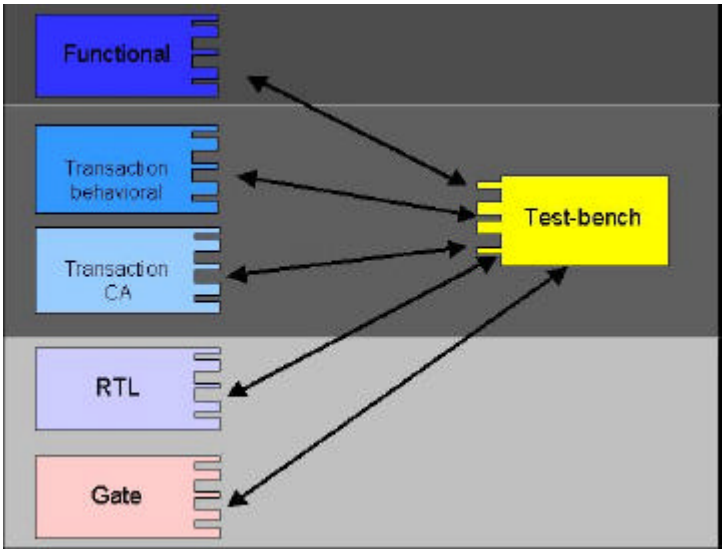

Figure 2. Modeling in various abstraction levels

Notice that virtual SoC prototype may hide, modify or omit SoC properties. As shown in Figure 2, abstraction levels span multiple levels of accuracy, ranging from functional- to transistor-level. Each level introduces new model details [27]. We now describe abstraction levels, starting with the most abstract and going to the most specific.

Functional models usually have no notion of resource sharing or time. Thus, functionality is executed instantaneously, or as an ordered sequence of events as in a functional TCP model, and the model may or may not be bit-accurate. This layer is suitable for system concept validation, functional partitioning between control and data, including abstract data type definition, hardware or software communication and synchronization mechanisms, lightwave versions of RTOS, key algorithm definition, integration to high-

level simulation via C, C++, Ada, MPI, Corba, DCOM, RMI, Matlab, ODAE Solver, OPNET, SDL, SIMSCRIPT, SLAM, or UML technology, key algorithm definition, and initial system testing. Models are usually based on core functionality written in ANSI C and a SystemC-based wrapper.

Transactional behavioral models (denoted simply as transactional) are functional models mapped to a discrete time domain. Transactions are atomic operations with their duration stochastically determined. Although general transactions on bus protocols can not be modeled, transactional models are particularly important for protocol design and analysis, communication model support, i.e. shared memory, message passing or remote procedure call, RTOS introduction, functional simulation, pipelining, hardware emulation, parameterizable hardware/software co-design, preliminary performance estimation, and test bench realization.

Except for asynchronous models, *transactional clock accurate models* (denoted transactional CA) map transactions to a clock cycle; thus, synchronous protocols, wire delays, and device access times can be accurately modeled. This layer is useful for functional and cycle-accurate performance modeling of abstract processor core wrappers (called bus functional models), bus protocols, signal interfaces, peripheral IP blocks, instruction set simulator, and test benches, in a simple, generic and efficient way using discrete-event systems. Transactional CA models are similar to corresponding RTL models, but they are not synthesizable.

Register-transfer level models (RTL) correspond to the abstraction level from which synthesis tools can generate gate-level descriptions (or netlists). RTL systems are usually visualized as having two components: data and control. The data part is composed of registers, operators, and data paths. The control part provides the time sequence of signals that evoke activities in the data part. Data types are bit-accurate, interfaces are pin-accurate, and register transfer is accurate. Propagation delay is usually back annotated from gate models.

Gate models are described in terms of primitives, such as logic with timing data and layout configuration. For simulation reasons, gate models may be internally mapped to a continuous time domain, including currents, voltages, noise, clock rise and fall times. Storage and operators are broken down into logic implementing the corresponding digital functions, while timing for individual signal paths can be obtained.

Thus, an embedded physical SRAM memory model may be defined as:

- a collection of constraints and requirements described as a functional model in a high-level general programming language, such as Ada, C, C++ or Java,
- implementation-independent RTL logic described in VHDL or Verilog languages,
- as a vendor gate library described using NAND, flip-flop schematics, or
- at the physical level, as a detailed and fully characterized mask layout, depicting rectangles on chip layers and geometrical arrangement of I/O and power locations.

2.2 Separation of Communication and Computation Components

System level design methodology is based on the concept of orthogonalization of concerns [22]. This includes separation of

- function specification from architecture, i.e. **what** are the basic system functions vs. **how** the system organizes software, firmware and hardware components in order to implement these functions, and
- communication from computation (also called behavior).

This orthogonalization implies a refinement process that eventually maps specifications for behavior and communication interfaces to the hardware or software resources of a particular architecture, e.g. as custom-hardware groupings sharing a bus interface or as software tasks. This categorization process is called system partitioning and forms a basic element of co-design [35]. Notice that function specification, i.e. behavior and communication, is generally independent of the particular implementation. Only in exceptional cases, specification may guide implementation, e.g. by providing advice to implementers, or compiler-like pragmas.

Separation between communication and computation is a crucial part in the stepwise transformation from a high-level behavioral model of an embedded system into actual implementation. This separation allows refinement of the communication channels of each system module. Thus, each IP consists of two components.

- *A behavior component* is used to describe module functionality. At functional specification level a behavior is explained in terms of its effect, while at design specification level a behavior corresponds to an active object in object-oriented programming, since it usually has an associated identity, state and an algorithm consuming or producing communication messages, synchronizing or processing data objects. Access to a behavior component is provided via a communication interface and explicit communication protocols. Notice that this interface is considered as the **only** way to interact with the behavior.
- *A communication interface* consists of a set of input/output ports transferring messages between one or more concurrent behavior components. The interface supports various communication protocols. Behaviors must be compatible, so that output signals from one interface are translated to input signals to another. When behaviors are not compatible, specialized channel adapters are needed.

Notice that by forcing IP objects to communicate solely through communication interfaces, we can fully de-couple module behavior from inter-module communication. Therefore, inter-module communication is never considered in line with behavior, but it is completely independent. Both behavior and communication components can be expressed at various levels of abstraction. Static behavior is specified using untimed algorithms, while dynamic behavior is explained using complex simulation-based architectures, e.g. hierarchical finite state machines or Threads. Similarly, communication can be either abstract, or close to implementation, e.g. STMicroelectronics' proprietary `STbus` [44, 45], OCP [27], `VCI` interfaces [49], or generic interface prototypes.

Moreover these C++-based objects support protocol refinement. Protocol refinement is the act of gradually introducing lower level detail in a model, making it closer to the real implementation, while preserving desired properties and propagating constraints to lower levels of abstraction. Thus, refinement is an additive process, with each detail adding specificity in a narrower context.

2.3 OSI-Like Layering for Inter-Module Communication Refinement

Communication protocols enable an entity in one host to interact with a corresponding entity in another remote host. One of the most fundamental principles in modeling complex communication protocols is establishing protocol refinement. Protocol refinement allows the designer to explore model behavior and communication at different level of abstractions, thus trading between model accuracy with simulation speed. Thus, a complex IP could be modeled at the behavioral level internally, and at the cycle level at its interface allowing validation of its integration with other components. Optimal design methodology is a combination of top-down and bottom-up refinement.

- In top-down refinement, emphasis is placed on specifying unambiguous semantics, capturing desired system requirements, optimal partitioning of system behavior into simpler behaviors, and refining the abstraction level down to the implementation by filling in details and constraints.
- In bottom-up integration, IP-reuse oriented implementation with optimal evaluation, composition and deployment of prefabricated architectural components, derived from existing libraries from a variety of sources, drives the process. In this case, automatic IP integration is important, e.g. automatic selection of a common or optimal high-speed communication standard.

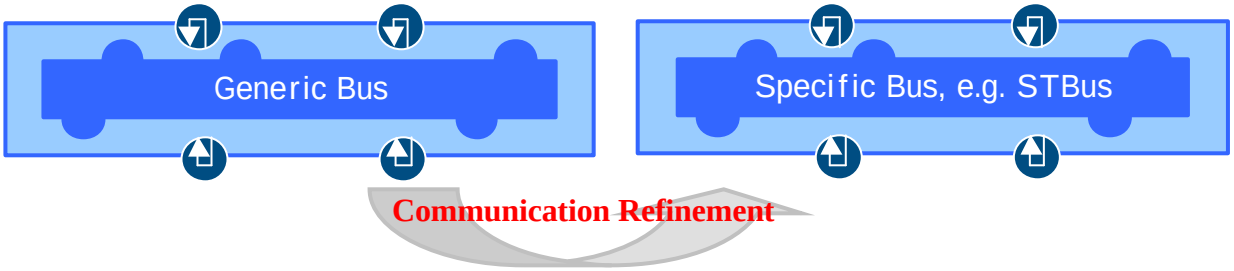

Figure 3. Enabling communication refinement

As shown in Figure 3, communication refinement refers to being able to modify or substitute a given communication layer, without changing lower communication layers, computational modules, or test benches. In communication refinement, the old protocol is either extended to a lower abstraction level, or replaced by a completely new bus protocol implemented at a similar or lower abstraction level. Inter-module communication refinement is fundamental to addressing I/O and data reconfiguration at a any level of hierarchy without re-coding, and OCCA design exploration. Communication refinement is often based on communication layering.

Layering is a common way to capture abstraction in communication systems. It is based on a strictly hierarchical relationship. Within each layer, functional entities interact directly only with the layer immediately below, and provide facilities for use by the layer above it. Thus, an upper layer always depends on the lower layer, but never the other way round. An advantage of layering is that the method of passing information between layers is well specified, and thus changes within a protocol layer are prevented from affecting lower layers. This increases productivity, and simplifies design and maintenance of communication systems.

Efficient inter-module (or inter-PE) communication refinement for OCCA models depends on establishing appropriate communication layers, similar to the OSI communication protocol stack. This idea originated with the application- and system-level transactions in Cosy [6], which was based on concepts developed within the VCC

framework [11, 19, 20, 30, 40, 41, 43]. A similar approach, with two distinct communication layers (message and packet layer) has been implemented in IPSIM, an ST Microelectronics-proprietary SystemC-based MPSoC modeling environment [12, 14, 15].

- The *message layer* provides a generic, user-defined message API that enables reuse of the packet layer by abstracting away the underlying channel architecture, i.e. point-to-point channel, or arbitrarily complex network topology, e.g. Amba, Core Connect, STBus. Notice that the firing rule, determining the best protocol for token transmission, is not specified until later in the refinement process.
- The *packet layer* provides a generic, flexible and powerful communication API based on the exchange of packets. This API abstracts away all signal detail, but enables representation of the most fundamental properties of the communication architecture, such as switching technique, queuing scheme, flow control, routing strategy, routing function implementation, unicast/multicast communication model. At this abstraction level, a bus is seen as a node interconnecting and managing communication among several modules of two kinds (masters and slaves).

2.4 SystemC Communication

The primary modeling element in SystemC is a module (`sc_module`). A module is a concurrent, active class with a well-defined behavior mapped to one or more processes (i.e. a thread or method) and a completely independent communication interface.

In SystemC inter-module communication is achieved using interfaces, ports, and channels as illustrated in Figure 4. An interface (circle with one arrow) is a pure functional object that defines, but does not implement, a set of methods that define an API for accessing the communication channel. Thus, interface does not contain implementation details. A channel implements interfaces and various communication protocols. A port, shown as a square with two arrows in Figure 4, enables a module, and hence its processes, to access a channel through a channel interface. Thus, since a port is defined in terms of an interface type, the port can be used only with channels that implement this interface type. SystemC port, interface and channel allow separating behavior from communication.

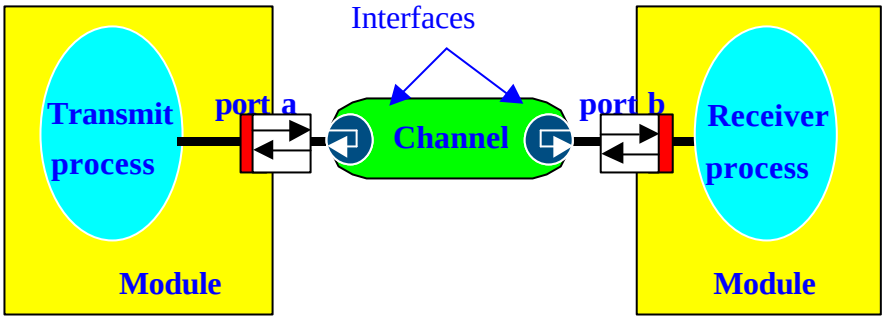

Figure 4. SystemC module components: behavior and inter-module communication

Access to a channel is provided through specialized ports (small red squares in Figure 4). For example, for the standard `sc_fifo` channel two specializations are provided: `sc_fifo_in<T>` and `sc_fifo_out<T>`. They allow FIFO ports to be read and written without accessing the interface methods. Hereafter, they are referred to as Port API.

An example is shown below.

```
class producer : public sc_module {
  public:
   sc_fifo_out<char> out; // define “out” port;
   SC_CTOR(producer) { SC_THREAD(produce); }
   void produce() {
     const char *str = “hello world!";
     while(*str){ out.write(*str++); } // call API of “out”
};
```

3. The OCCN Methodology

As all system development methodologies, any SoC object oriented modeling would consist of a modeling language, modeling heuristics and a methodology [42]. Modeling heuristics are informal guidelines specifying how the language constructs are used in the modeling process. Thus, the OCCN methodology focuses on modeling complex on-chip communication network by providing a flexible, open-source, object-oriented C++-based library built on top of SystemC. System architects may use this methodology to explore NoC performance tradeoffs for examining different OCCA implementations.

Alike OSI layering, OCCN methodology for NoC establishes a conceptual model for inter-module communication based on layering, with each layer translating transaction requests to a lower-level communication protocol. As shown in Figure 5, OCCN methodology defines three distinct OCCN layers. The lowest layer provided by OCCN, called *NoC communication layer*, implements one or more consecutive OSI layers starting by abstracting first the Physical layer. For example, the STBus NoC communication layer abstracts the physical and data link layers. On top of the OCCN protocol stack, the user-defined *application layer* maps directly to the application layer of the OSI stack. Sandwiched between the application and NoC communication layers lies the *adaptation layer* that maps to one or more middle layers of the OSI protocol stack, including software and hardware adaptation components. The aim of this layer is to provide, through efficient, inter-dependent entities called communication drivers, the necessary computation, communication, and synchronization library functions and services that allow the application to run. Although adaptation layer is usually user-defined, it utilizes functions defined within the OCCN communication API.

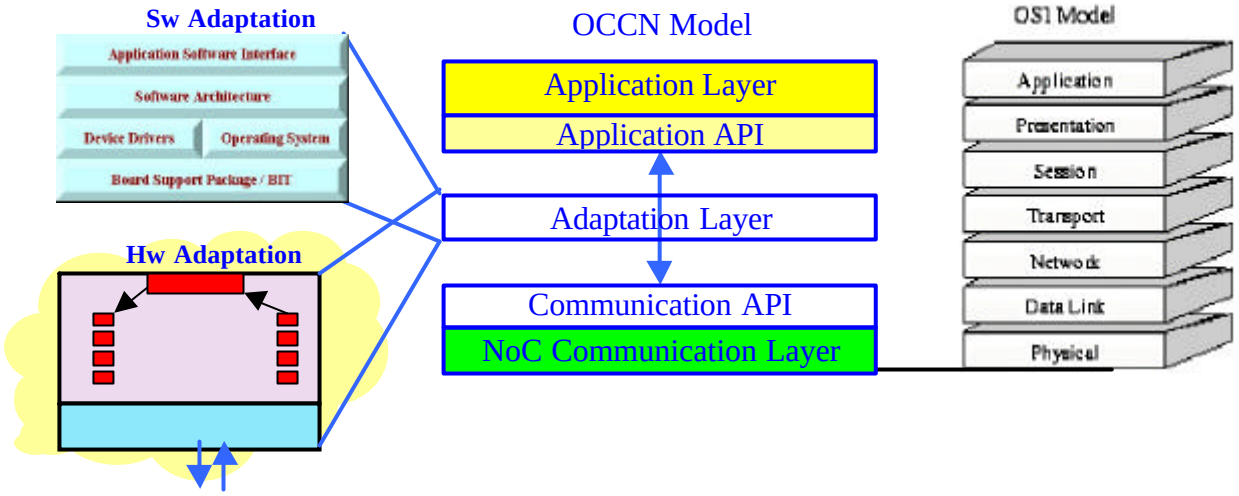

Figure 5. OSI-like OCCN layering model with APIs shown

An implementation of an adaptation layer includes software and hardware components, as shown in the left part of Figure 5. A typical software adaptation layer includes several sub-layers. The lowest sub-layer is usually represented by the board support package

(BSP) and built in tests (BIT). The BSP allows all other software, including the Operating System (OS), to be loaded into memory and start executing, while BIT detects and reports hardware errors. On top of this sub-layer we have the OS and device drivers. The OS is responsible for overall software management, involving key algorithms, such as job scheduling, multitasking, memory sharing, I/O interrupt handling, and error and status reporting. Device drivers manage communication with external devices, thus supporting the application software. Finally, the software architecture sub-layer provides execution control, data or message management, error handling, and various support services to the application software.

The OCCN conceptual model defines two APIs.

- The *OCCN communication API* provides a simple, unique, generic, ultra-efficient and compact interface that greatly simplifies the task of implementing various layers of communication drivers at different level of design abstraction. The API is based on generic modeling features, such as IP component reuse and separation between behavior and communication. It also hides architectural issues related to the particular on-chip communication protocol and interconnection topology, e.g. simple point-to-point channel vs. complex, multilevel NoC topology supporting split transactions, and QoS in higher communication layers, thus making internal model behavior module-specific. The OCCN communication API is based on a message-passing paradigm providing a small, powerful set of methods for inter-module data exchange and synchronization of module execution. This paradigm forms the basis of the OCCN methodology, enhancing portability and reusability of all models using this API.
- The *application API* forms a boundary between the application and adaptation 1ayers. This API specifies the necessary methods through which the application can request and use services of the adaptation layer, and the adaptation layer can provide these services to the application.

The OCCN implementation for inter-module communication layering uses generic SystemC methodology, e.g. a SystemC port is seen as a service access point (SAP), with the OCCN API defining its service. Applying the OCCN conceptual model to SystemC, we have the following mapping.

- The *NoC communication layer*, is implemented as a set of C++ classes derived from the SystemC `sc_channel` class. The communication channel establishes the transfer of messages among different ports according to the protocol stack supported by a specific NoC.
- The *communication API* is implemented as a specialization of the `sc_port` SystemC object. This API provides the required buffers for inter-module communication and synchronization and supports an extended message passing (or even shared memory) paradigm for mapping to any NoC.
- The *adaptation layer* translates inter-module transaction requests coming from the application API to the communication API. This layer is based on port specialization built on top of the communication API. For example, the communication driver for an application that produces messages with variable length may implement segmentation, thus adapting the output of the application to the input of the channel.

The fundamental components of the OCCN API are the Protocol Data Unit (Pdu), the MasterPort and SlavePort interface, and high-level system performance modeling. These components are described in the following sections.

3.1 The Protocol Data Unit (Pdu)

Inter-module communication is based on channels implementing well-specified protocols by defining rules (semantics) and types (syntax) for sending and receiving protocol data units (or Pdus, according to OSI terminology). In general, Pdus may represent bits, tokens, cells, frames, or messages in a computer network, signals in an on-chip network, or jobs in a queuing network. Thus, Pdus are a fundamental ingredient for implementing inter-module (or inter-PE) communication using arbitrarily complex data structures.

A Pdu is essentially the optimized, smallest part of a message that can be independently routed through the network. Messages can be variable in length, consisting of several Pdus. Each Pdu usually consists of various fields.

- The *header* field (sometimes called *protocol control information*, or *PCI*) provides the destination address(es), and sometimes includes source address. For variable size Pdus, it is convenient to represent the data length field first in the header field. In addition, routing path selection, or Pdu priority information may be included. Moreover, header provides an operation code that distinguishes: (a) request from reply Pdus, (b) read, write, or synchronization instructions, (c) blocking, or nonblocking instructions, and (d) normal execution from system setup, or system test instructions. Sometimes performance related information is included, such as a transaction identity/type, and epoch counters. Special flags are also needed for synchronizing accesses to local communication buffers (which may wait for network data), and for distinguishing buffer pools, e.g. for pipelining sequences of nonblocking operations. In addition, if Pdus do not reach their destinations in their original issue order, a sequence number may be provided for appropriate Pdu reordering. Furthermore, for efficiency reasons, we will assume that the following two fields are included with the Pdu header.
  - The *checksum* (CRC) decodes header information (and sometimes data) for error detection, or correction.
  - The *trailer* consisting of a Pdu termination flag is used as an alternative to a Pdu length sub-field for variable size Pdus.
- The *data* field (called *payload*, or *service data unit*, or *SDU*) is a sequence of bits that are usually meaningless for the channel. A notable exception is when data reduction is performed within a combining, counting, or load balancing network.

Basic Pdus in simple point-to-point channels may contain only data. For complicated network protocols, Pdus must use more fields, as explained below.

- Remote read or DMA includes header, memory address, and CRC.
- Reply to remote read or DMA includes header, data, and CRC.
- Remote write includes header, memory address, data, and CRC.
- Reply from remote write includes header and CRC.
- Synchronization (fetch&add, compare&swap, and other read-modify-write operations) includes header, address, data, and CRC.
- Reply from synchronization includes header, data, and CRC.
- Performance-related instructions, e.g. remote enqueue may include various fields to access concurrent or distributed data structures.

Furthermore, within the OCCN channel, several important routing issues involving Pdu must be explored (see Section 1). Thus, OCCN defines various functions that support simple and efficient interface modeling, such as adding/striping headers from Pdus, copying Pdus, error recovery, e.g. checkpoint and go-back-n procedures, flow control,

segmentation and re-assembly procedures for adapting to physical link bandwidth, service access point selection, and connection management. Furthermore, the Pdu specifies the format of the header and data fields, the way that bit patterns must be interpreted, and any processing to be performed (usually on stored control information) at the sink, source or intermediate network nodes.

The Pdu class provides modeling support for the header, data field and trailer as illustrated in the following C++ code block.

```
  template <class H, class BU, int size>
    class Pdu  {
      public:
        H hdr; // header (or PCI)
        BU body[size]; // data (or SDU)

        // Assignments that modify & return lvalue:
        Pdu& operator=(const BU& right);
        BU& operator[](unsigned int x); // accessing Body, if size > 1

        // Conditional operators return true/false:
        int operator==(const Pdu& right) const;
        int operator!=(const Pdu& right) const;

        // std streams display purpose
        friend ostream& operator<< <>(ostream& os, const Pdu& ia);
        friend istream& operator>> <>(istream& is, Pdu& right);

        // Pdu streams for segmentation/re-assembly
     friend Pdu<H,BU,size>& operator<< <> (Pdu& left, const Pdu& right);
     friend Pdu<H,BU,size>& operator>> <> (Pdu& left, Pdu& right);
}
```

Depending on the circumstances, OCCN Pdus are created using four different methods. Always `HeaderType` (H) is a user-defined C++ `struct`, while `BodyUnitType` (BU) is either a basic data type, e.g. `char` and `int`, or an encapsulated Pdu; the latter case is useful for defining layered communication protocols.

- Define a simple Pdu containing only a header of `HeaderType`:
  `Pdu<HeaderType> pk2`
- Define a simple Pdu containing only a body of `BodyUnitType`:
  `Pdu<BodyUnitType> pk1`
- Define a Pdu containing a header and a body of `BodyUnitType`:
  `Pdu<HeaderType, BodyUnitType> pk3`
- Define a Pdu containing a header and a body of `length` many elements of `BodyUnitType`:
  `Pdu<HeaderType, BodyUnitType, length> pk4`

Processes access Pdu data and control fields using the following functions.

- The `occn_hdr(pk, field_name)` function is used to read or write the Pdu header.
- The standard operator "=" is used to
  - read or write the Pdu body,
  - copy Pdus of the same type.
- The operator s ">>"and "<<" are used to
  - send or receive Pdu from input/output streams, and
  - segmentation and re-assembly Pdus.

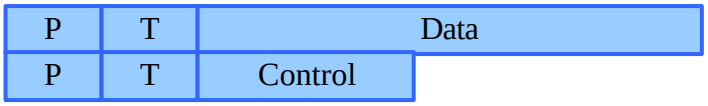

Figure 6. A simple token DS LINK (IEEE 1355)

As an example, let us consider the high-speed digital serial links known as IEEE1355. IEEE 1355 specifies the physical media and low-level protocols for a family of serial scalable interconnect systems. Pdus defined in the character layer are called tokens. They include a parity bit (P) plus a control bit (T) used to distinguish between data and control tokens. In addition, a data token contains 8 bits of data, and control tokens contain two bits to denote the token type. This is illustrated in Figure 6. The OCCN modeling structure is as follows.

```
struct DSLINK_token {
 uint P :1;
 uint T :1; }
Pdu<DSLINK_token, char> pk1, pk2;
occn_hdr(pk1,P)=1; // parity field in pk1 is set equal to 1
occn_hdr(pk1,T)=0; // we set pk1 as data token, because T=0
uint tmp = occn_hdr(pk1,P); // tmp is set equal to 1
char body =  'a';
pk1 = body; // pk1 contains 'a'
pk2 = pk1; // now pk2 and pk1 are the same, since "=" copies Pdus
char x = pk2; // x assume the value of 'a';
// since pk1 is equal to pk2 and x is equal to 'a',
// the cout statement is executed
if ((pk1 == pk2) && (x == 'a'))
   cout << "pk1 == pk2" << endl;
```

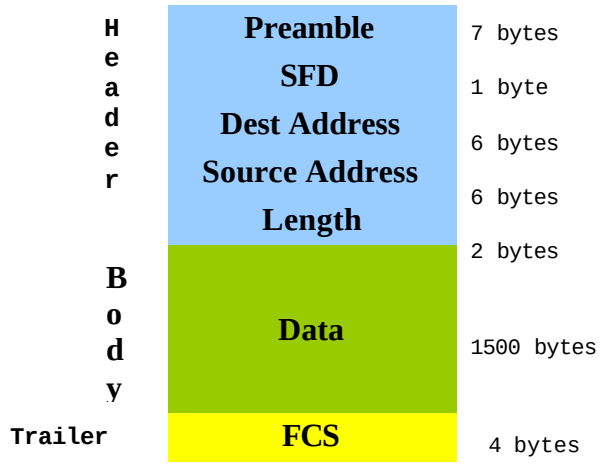

Figure 7. A complex Pdu for a CSMA/CD LAN

Alternatively, as illustrated in Figure 7, a complex Pdu containing both header and body for a local area network CSMA/CD may be defined as follows.

```
struct CSMA_CD_crtl {
 int8 preamble[7];
 int8 sfd;
 int8 dest[6];
 int8 source[6];
 int16 length;
 int32 FCS;
};
// first argument is the type, while last one is the size of the body
```

```
Pdu< CSMA_CD_ctrl, char, 1500> pk1, pk2;
occn_hdr(pk1,sfd)=3; // set sfd field in pk1 to 3
int8 tmp=occn_hdr(pk1,sfd); // set tmp to 3
char body[1500] =”The body of a Pdu”;
pk1=body; // pk1 contains ”I am the body of a Pdu” (all data copied)
pk2=”I am an OCCN Pdu”; // pk2 contains ”I am an OCCN Pdu”;
// since pk1 is not equal to pk2 the cout statement is executed
if (pk1!=pk2)
   cout << "pk1 != pk2" << endl;
```

Segmentation and re-assembly are very common operations in network protocols. Thus, OCCN supports the overloaded operators “>>”, “<<”. The following is a full segmentation/ reassembly example.

```
typedef struct { int32 seq;} Headermsg;
Pdu< Headermsg, char, 4 > pk0, pk1, pk2, pk3;
Pdu< Headermsg, char, 8> msg1, msg2;

occn_hdr(msg1, seq)=13;
msg1=”abcdefgh”; // msg1 contains ‘abcdefgh’;

msg1>>pk0; // pk0 contains 13;
msg1>>pk1; // pk1 contains ‘abcd’;
msg1>>pk2; // pk2 contains ‘efgh’ and msg1 is empty;;
msg1>>pk3  // pk3 is empty since msg1 was empty

//previous three statements are equivalent to
msg1>>pk1>>pk2>>pk3;

msg2<<pk0; // msg2 contains ‘+’
msg2<<pk1; // msg2 contains ‘abcd’;
msg2<<pk2; // msg2 contains ‘abcdefgh’

// the previous statements are equivalent to the statement:
// msg2<<pk0<<pk1<<pk2;

int tmp=occn_hdr(msg2,seq);  // tmp is equal to 13
```

3.2 The MasterPort & SlavePort API

In this Section we describe the transmission/reception interface of the OCCN API. The OCCN API provides a message-passing interface, with send and receive primitives for point-to-point and multi-point communication. If the Pdu structure is determined according to a specific OCCA model, the same base functions are required for transmitting the Pdu through almost any OCCA. A great effort is dedicated to define this interface as a reduced subset of functions providing users with a complete and powerful semantic. In this manner, we can achieve model reuse and inter-module communication protocol refinement through a generic OCCA access interface.

Message passing systems may efficiently emulate a variety of communication paradigms based on shared memory, remote procedure call (RPC), or Ada-like rendezvous. Furthermore, the precise semantic flavor of message-passing primitives defining basic data exchange between user-defined tasks can often be mixed. For example, they may reflect completion semantics, i.e. specifying when control is returned to the user process that issued the send or receive, or specifying when the buffers (or data structures) can be reused without compromising correctness.

For message passing, there are two major point-to-point send/receive primitives: *synchronous* and *asynchronous*. Synchronous primitives are based on acknowledgments, while asynchronous primitives usually deposit and remove messages to/from application and system buffers. Within the class of asynchronous point-to-point communications, there are also two other major principles: *blocking* and *non-blocking*. While non-blocking operations allow the calling process to continue execution, blocking operations suspend execution until receiving an acknowledgment or timeout. Although, we often define various buffer-based optimization-specific principles, e.g. the standard, buffered, and ready send/receive in the MPI standard, we currently focus only on the major send/receive principles.

Synchronous blocking send/receive primitives offer the simplest semantics for the programmer, since they involve a handshake (rendezvous) between sender and receiver.

- A synchronous send busy waits (or suspends temporarily) until a matching receive is posted and receive operation has started. Thus, the completion of a synchronous send guarantees (barring hardware errors) that the message has been successfully received, and that all associated application data structures and buffers can be reused. A synchronous send is usually implemented in three steps.
  - First, the sender sends a request-to-send message.
  - Then, the receiver stores this request.
  - Finally, when a matching receive is posted, the receiver sends back a permission-to-send message, so that the sender may send the packet.
- Similarly, a synchronous receive primitive busy waits (or suspends temporarily) until there is a message to read.

With *asynchronous blocking* operations we avoid polling, since we know exactly when the message is sent/received. Furthermore, in a multi-threaded environment, a blocking operation blocks only the executing thread, allowing the thread scheduler to re-schedule another thread for execution, thus resulting in performance improvement. The communication semantics for point-to-point asynchronous blocking primitives are defined as follows.

- The blocking send busy waits (or suspends temporarily) until the packet is safely stored in the receive buffer (if the matching receive has already been posted), or in a temporary system buffer (message in care of the system). Thus, the sender may overwrite the source data structure or application buffer after the blocking send operation returns. Compared to a synchronous send, this allows the sending process to resume sooner, but the return of control does not guarantee that the message will actually be delivered to the appropriate process. Obtaining such a guarantee would require additional handshaking.
- The blocking receive busy waits (or suspends temporarily) until the requested message is available in the application buffer. Only after the message is received, the next receiver instruction is executed. Unlike a synchronous receive, a blocking receive does not send an acknowledgment to the sender.

Asynchronous non-blocking operations prevent deadlocks due to lack of buffer space, since they avoid the overhead of allocating system buffers and issuing memory-to-memory message copies. Non-blocking also improves performance by allowing communication and computation overlap, e.g. via the design of intelligent controllers based on parallel programming or multithreading principles. More specifically, for point-

to-point asynchronous non-blocking send/receive primitives, we define these following semantics.

- A non-blocking send initiates the send operation, but does not complete it. The send returns control to the user process before the message is copied out of the send buffer. Thus, data transfer out of the sender memory may proceed concurrently with computations performed by the sender after the send is initiated and before it is completed. A separate *send completion* function, implemented by accessing (probing) a system communication object via a handle, is needed to complete the communication, i.e. for the user to check that the data has been copied out of the send buffer, so that the application data structures and buffers may be reused[2]. These functions either block until the desired state is observed, or return control immediately reporting the current send status.
- Similarly, a non-blocking receive initiates the receive operation, but does not complete it. The call will return before a message is stored at the receive buffer. Thus, data transfer into receiver memory may proceed concurrently with computations performed after receive is initiated and before it is completed. A separate *receive completion* function, implemented by accessing (probing) a system communication object via a handle, is needed to complete the receive operation, i.e. for the user to verify that data has been copied into the application buffer[1]. These probes either block until the desired state is observed, or return control immediately reporting the current receive status.

We can combine synchronous, asynchronous blocking, and asynchronous non-blocking send/receive pairs. However, in some cases, the semantics become very complex. In standard message passing libraries, such as MPI, there exist over 50 different (and largely independent) functions for point-to-point send/receive principles. Moreover, there are literally hundreds of ways to mix and match these function calls.

The precise type of send/receive semantics to implement depends on how the program uses its data structures, and how much we want to optimize performance over ease of programming and portability to systems with different semantics. For example, asynchronous sends alleviate the deadlock problem due to missing receives, since processes may proceed past the send to the receive process. However, for non-blocking asynchronous receive, we need to use a probe before actually using the received data.

For efficiency reasons, the OCCN MasterPort/SlavePort API is based only on synchronous blocking send/receive and asynchronous blocking send primitives. The proposed methods support synchronous and asynchronous communication, based on either a synchronous (cycle-based), or asynchronous (event-driven) OCCA model.

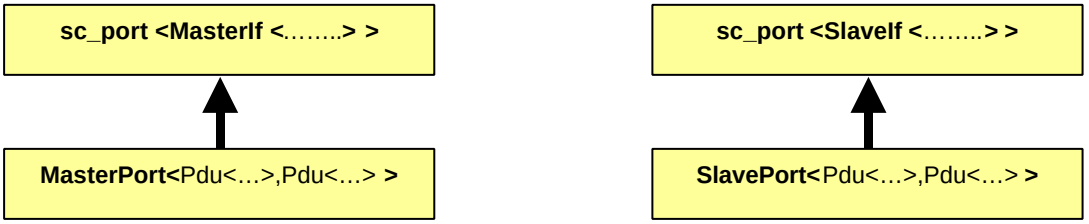

Figure 8. `MasterPort` and `SlavePort` derived from `sc_port`

[2] Blocking send (receive) is equivalent to non-blocking send (resp., receive) immediately followed by a blocking send (resp., receive) completion function call.

As shown in Figure 8, this API is implemented using two specializations of the standard SystemC `sc_port<…>`, called `MasterPort<…>` and `SlavePort<…>`. The name Master/Slave is given just for commodity reasons, there is no relationship with the Master/Slave library provided in SystemC, or RPC concepts introduced earlier by CoWare [7]; this encompasses the limitation that the Master port is always connected to a Slave one. In our case, the name Master is associated to the entity, which is responsible to start the communication. Master and Slave ports are defined as templates of the outgoing and incoming Pdu. In most cases, the outgoing (incoming) Pdu for the Master port is the same as the incoming (respectively, outgoing) Pdu for the Slave port.

Hereafter, a list of the OCCN MasterPort/SlavePort API is provided.

- `void send(Pdu<…>* p, sc_time& time_out=-1, bool& sent);` This function implements synchronous blocking send. Thus, the sender will deliver the Pdu `p`, only if the channel is free, the destination process is ready to receive, and the user-defined `timeout` value has not expired. Otherwise, the sender is blocked and the Pdu is dispatched. While the channel is busy (or the destination process is not ready to receive), and the `timeout` value has not expired, waiting sender tasks compete to acquire the channel using a FIFO priority scheme. Upon function exit, the boolean flag `sent` returns false, if and only if the `timeout` value has expired before sending the Pdu; this event signifies that the Pdu has not been sent
- `void asend(Pdu<…>* p, sc_time& time_out=-1, bool& dispatched);` This function implements asynchronous blocking send. Thus, if the channel is free and the user-defined `timeout` value has not expired, then the sender will dispatch the Pdu `p` whether or not the destination process is ready to receive it. While the channel is busy, and the user-defined `timeout` value has not expired, waiting sender tasks compete to acquire the channel using a FIFO priority scheme. In this case, the boolean flag `dispatched` returns false, if and only if the `timeout` value has expired before sending the Pdu; this event signifies Pdu loss.
- The OCCN API implements a synchronous blocking receive using a pair of functions: `receive` and `reply`.
  - `Pdu<…>* receive(sc_time& time_out=-1, bool& received);` This function implements synchronous blocking receive. Thus, the receiver is blocked until it receives a Pdu, or until a user-defined `timeout` has expired. In the latter case, the boolean flag `received` returns false, while the Pdu value is undefined.
  - `void reply(uint delay=0)` or `void reply(sc_time delay);` After a dynamically variable `delay` time, expressed as a number of bus cycles or as absolute time (sc_time), the receiver process completes the transaction. Return from `reply` ensures that the channel send/receive operation is completed and that the receiver is synchronized with the sender. The following code is used for receiving a Pdu.

```
sc_time timeout = ….;
bool received;
// Suppose that in is an OCCN SlavePort
Pdu<…> *msg = in.receive(timeout, received);
if (!received)
  // timeout expired: received Pdu not valid
else
  // received Pdu is valid; user may perform elaboration on Pdu
  reply(); // synchronizing sender & receiver after 0 bus cycles
```

Notice that when the delay of a transaction is measured in terms of bus cycles, OCCN assumes that the channel is the only one to have knowledge of the clock, allowing asynchronous processes to be connected to synchronous clocked communication media. In both cases the latency of `reply` can be fixed or dynamically calculated after the `receive`, e.g. as a function of the received Pdu.

An example of fixed receive latency delay is provided below.

```
sc_time latency(2, SC_NS);
msg=in.receive(); // in is an OCCN SlavePort
// msg now available
addr=occn_hdr(*msg, addr);
seq=occn_hdr(*msg, seq);
// managing the payload
reply(latency); // receive operation is completed,
                // receiver is synchronized with the transmitter
```

An example of dynamic delay is given below.

```
uint latency=5; // latency expressed in number of bus clock cycles
msg=in.receive();    // obtain msg with a dynamic delay
addr=occn_hdr(*msg, addr);
seq=occn_hdr(*msg, seq);
latency=delay_function(msg); // latency is dynamic (depending on msg)
reply(latency); // receive operation is completed,
          // receiver is synchronized with the transmitter
```

Furthermore, notice that a missing reply to a synchronous send could cause a deadlock, unless a sender timeout value is provided. In the latter case, we allow that the Pdu associated with the missing acknowledgment is lost. Notice that a received Pdu is also lost, if it is not accessed before the corresponding reply.

Sometimes tasks may need to check, enable, or disable Pdu transmission or reception, or extract the exact time(s) that a particular communication message arrived. These functions enable optimized modeling paradigms that are accomplished using appropriate OCCN channel setup and control functions. A detailed description of these functions falls outside the scope of this document [13].

Another OCCN feature is protocol in-lining, i.e. the low-level protocol necessary to interface a specific OCCA is automatically generated using the standard template feature available in C++ enabled by user-defined data structures. This implies that the user does not have to write low-level communication protocols already provided by OCCN, thus, making instantiation and debugging easier. Savings are significant, since in today's MPSoC there are 20 or more ports, and 60 to 100 signals per port.

3.3 High-Level Performance Modeling

High-level system performance modeling is an essential ingredient in NoC and MPSoC design exploration and co-design. Performance metrics help identify system bottlenecks by recording *instant values* commonly in time-driven simulation, and *duration values* usually in event-driven simulation. OCCN provides a statistical package based on two general monitoring classes for collecting instant and duration performance characteristics from on-chip network components with the following functionality.

- In *time-driven simulation*, monitored objects usually have instantaneous values. During simulation, these values are recorded by calling a library-provided public member function called `stat_write`.

- In *event-driven simulation*, recorded statistics for events must include arrival and departure time (or duration). Since the departure time is known at some point later in time, the interface can be based on two public member functions.
  - First a `stat_event_start` function call records the arrival time, and saves in a local variable the unique location of the event within the internal table of values.
  - Then, when the event's departure time is known, this time is recorded within the internal table of values at the correct location by calling the `stat_event_end` function with the appropriate departure time.

Since the above methods enable simple and precise measurement, they are useful for obtaining system performance metrics. The statistics collection based on `stat_write` and `stat_event_start/end` operations may either be performed by the user, or directly by a specialized on-chip model, e.g. APB, AHB, CoreConnect, and StBus, using library-internal object pointers. In the latter case, probes are inserted into *the source code of the models*, either manually by setting sensors and actuators, or more efficiently through the use of a monitoring segment which automatically compiles the necessary probes. Such probes share resources with the system model, thus offering small cost, simplicity, flexibility, portability, and precise application performance measurement in a timely, frictionless manner.

Furthermore, observe that from the above classes representing time- and event-driven statistics, we may derive specialized classes on which throughput, latency, average and instant size, packet loss, and average hit ratio statistics can be based. Furthermore, for 2D graphs, the statistical API can be based on a possibly overloaded `enable_stat()`function that specifies the absolute start and end time for statistics collection, the title and legends for the x and y axes, the time window for window statistics, i.e. the number of consecutive points averaged in order to generate a single statistical point, and the unique (over all modules) object name. Thus, for example, the event-driven statistic class allows the derivation of simple duration statistics based on an `enable_stat_delay` library function for Register, FIFO, memory, and cache objects.

In addition to the previously described classes that cover all basic cases, it is sometimes necessary to combine statistical data from different modeling objects, e.g. for comparing average read vs. write access times in a memory hierarchy, or for computing cell loss in ATM layer communications. For this reason, we need new *joint or merged statistic classes* that inherit from time- and event-driven statistics. Special parameters, e.g. boolean flags, for these joint detailed statistic classes can lead to detailed statistics.

The statistical data obtained from the above processes is analyzed online using visualization software, e.g. the open source `Grace` tool, or dumped to a file for subsequent processing, e.g. via an electronic spreadsheet or a specialized text editor.

4. Case-Study: OCCN Communication Refinement

In our case study, we examine a simple, transport layer, inter-module data transfer protocol. After illustrating the implementation with reference to an OCCN generic channel, we illustrate OCCN inter-module communication refinement using the configurable STBus NoC. We also provide several examples that highlight the use of OCCN performance modeling.

4.1 A Transport Layer Inter-Module Data Transfer Protocol

This Section provides a case study for the OCCN methodology, focusing on the user point of view. This example shows how to develop the Adaptation Layer on top of the basic OCCN Communication API consisting of the `MasterPort` and `SlavePort` classes (see Figure 8). Using layered OCCN communication architecture, with each layer performing a well-specified task within the overall protocol stack, we describe a simplified transport layer, inter-module transfer application from a transmitter to a specified receiver. The buffer that has to be sent is split into frames. Thus, the `TransportLayer` API covers the OSI stack model up to the transport layer, since it includes segmentation. This API consists of the following basic functions.

- `void TransportLayer_send(uint addr, BufferType& buffer);` Notice that destination address `addr` identifies the target receiver and the buffer `buffer` to be sent. The `BufferType` is defined in the `inout_pdu.h` code block using the OCCN Pdu object.
- `BufferType* TransportLayer_receive();` This function returns the received buffer data.

We assume that the channel is unreliable. NoC is becoming sensitive to noise due to technology scaling towards deep submicron dimensions [5]; Thus, a simple stop-and-wait data link protocol with negative acknowledgments, called Positive Acknowledgment with Retransmission (PAR) (or Automatic Repeat reQuest, ARQ), is implemented [28, 46]. Using this protocol, each frame transmitted by the sender (I-frane) is acknowledged by the receiver using a separate acknowledgment frame (ACK-frame). A timeout period determines when the sender has to retransmit a frame not yet acknowledged.

The I-frame contains a data section (called payload) and a header with various fields:

- a `sequence` number identifying an order for each frame sequence; this information is used to deal with frame duplication due to retransmission, or reordering out-of-order messages due to optimized, probabilistic or hot-potato routing; in the latter case, messages select different routes towards their destination [25];
- a destination `address` field related to routing issues at Network Layer;
- an `EDC` (Error Detection Code) enabling error-checking at the Data Link Layer for reliable transmission over an unreliable channel [50], and
- a `source_id` identifying the transmitter for routing back an acknowledgment.

The ACK-frame sent by the receiver consist of only a header, with the following fields:

- a positive or negative `ack` field that acknowledges packet reception according to the adopted Data Link protocol, and
- a `source_id` identifying the receiver where to route the acknowledgment.

A frame is retransmitted only if the receiver informs the sender that this frame is corrupted, either through a special error code or by not sending an acknowledgement.

From the transmitter (Tx) side, the PAR protocol works as follows.

- **Tx.1**: send an I-frame with a proper identifier in the sequence field.
- **Tx.2**: wait for acknowledgment from the receiver until a timeout expires;
- **Tx.3**: if the proper acknowledgment frame is received, then send the next I-frame, otherwise re-send the same I-frame.

From the receiver (Rx) point of view, the PAR protocol works as follows.

- **Rx.1**: wait for a new I-frame from the Tx.
- **Rx.2**: detect corrupted frames using the sequence number (or possibly EDC).
- **Rx.3**: send a positive or negative ACK-frame based on the outcome of step Rx.2.

For simplifying our case-study, we assume that no data corruption or packet-loss occurs during the data exchanges. EDC is provided for a future more complex implementation.

4.2 Implementation using the OCCN StdChannel

Figure 9 illustrates OCCN implementation of our file transfer protocol, using inter-module communication between two SystemC modules (Transmitter and Receiver) through a synchronous, point-to-point OCCN channel called StdChannel. This channel implements the timeout capability (see Section 4.1) and random packet loss by emulating channel noise.

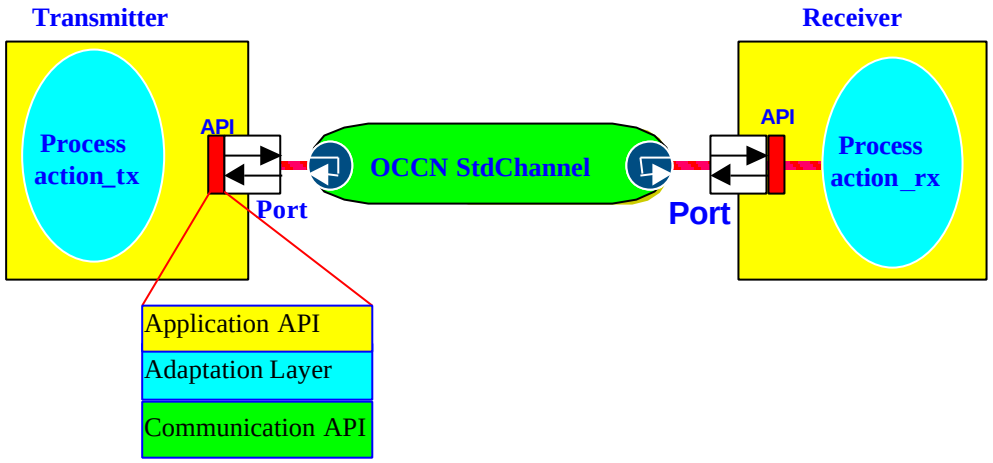

Figure 9. Transport Layer send/receive implementation with OCCN StdChannel

The StdChannel is accessed through the OCCN Communication API defined in Section 3.2, while the Transmitter and Receiver modules implement the higher-level Application API defined in Section 4.1. This API is based on Adaptation Layer classes MasterFrame, and SlaveFrame, specialized ports derived from MasterPort and SlavePort, respectively (see Figure 10). This SystemC-compliant approach allows design of the communication-oriented part of the application on top of the OCCN Communication API.

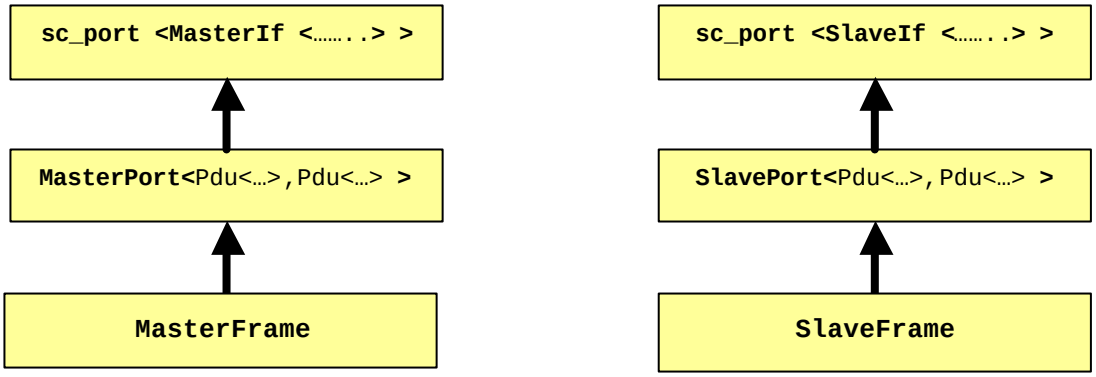

Figure 10. Inheritance of classes and specialized ports.

Due to the context of this document and space limitation only the code for transmitter is provided below. Comments are introduced within each code block to illustrate the most important design issues.

- "`inout_pdu.h`" – Pdu definitions for all ports,
- "`transmitter.h`" – interface definition of Transmitter module,
- "`transmitter.cc`" – implementation of Transmitter module,
- "`MasterFrame.h`" – interface definition of Transmitter frame adaptation layer,
- "`MasterFrame.c`" – implementation of Transmitter frame adaptation layer, and
- "`main.cc`" – description of main SystemC module.

4.2.1 The Pdu Definition

With respect to data type definition, we define the buffer and frame data structures using the OCCN Pdu object (see Section 3.1).

The buffer is an OCCN Pdu without header and a body of `BUFFER_SIZE` number of characters. Mapping of the frame to an OCCN Pdu is shown in Figure 11. Assuming in-order transmission, the sequence number can be represented by a single bit. However, we assign a progressive, arithmetic sequence to each frame, partly for clarity reasons and for the possibility to support unordered transmissions in the future. Since `StdChannel` is point-to-point, `addr` is actually useless, but could be used in the future in a more general implementation. Moreover, a reserved number (`ERROR_CODE`) in *`ack`* indicates an error code to distinguish, among all non acknowledgement conditions, the ones relating to data corruption detected by EDC.

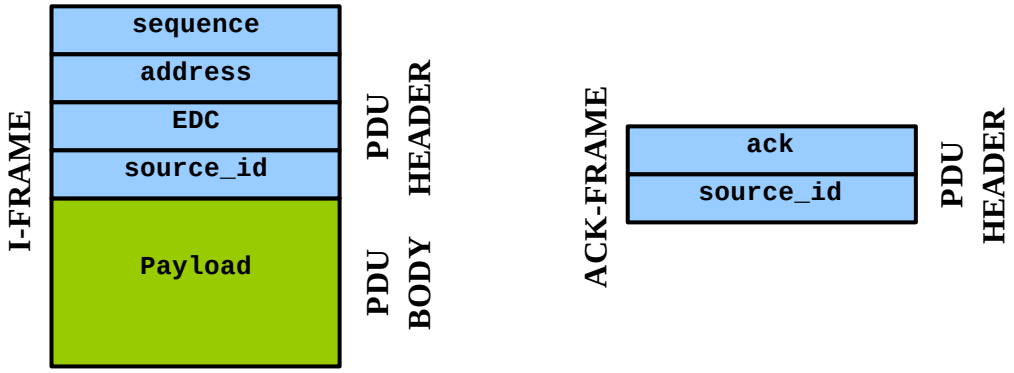

Figure 11. I- and ACK-frame data structures and corresponding OCCN Pdu objects.

To simplify the code, we assume that the data size of the StdChannel equals the frame size, i.e. Pdu sent by the OCCN Communication API is directly the I-frame in the transmitter to receiver direction and the ACK-frame in the opposite.

The "`inout_pdu.h`" code block provides the Pdu type definitions for inter-module communication. Notice the definitions of the `I_FrameType` and `ACK_FrameType` that are used throughout the example.

inout_pdu.h

```
#include <occn.h>
#define BUFFER_SIZE 256
#define ERROR_CODE  0xFFFFFFFF

typedef uint seq_num;
typedef uint EDC_type;

// I_HeaderType for the header of the Pdu I_FrameType
 struct I_frame_header {
  seq_num sequence;
  uint address;
  EDC_type EDC; };

// ACK_HeaderType for the header of the Pdu ACK_FrameType
 struct ACK_frame_header {
  seq_num ack;
  uint source_id; };

// I_FrameType PDU has a body size of 32 bits (unsigned int)
typedef Pdu<I_frame_header, N_uint32> I_FrameType;

// ACK_FrameType PDU has no body section
typedef Pdu<ACK_frame_header> ACK_FrameType;

// The PDU for the buffer parameter of TransportLayer_send
typedef Pdu<char, BUFFER_SIZE> BufferType;
```

4.2.2 The Transmitter Module

The Transmitter module implements the `SC_THREAD action_tx`. A buffer filled with random letters in the range 'A' to 'Z' is sent through the channel by calling the application API `TransportLayer_send` function through the MasterFrame `sap_tx` access port. This operation is repeated `NB_SEQUENCES` times. The Transmitter module interface described in the code block "`transmitter.h`" includes

- the `MasterFrame` definition, which is the same as in the Receiver module; thus it is obtained directly from "`inout_pdu.h`".
- the transmission layer interface definitions, defined in "`MasterFrame.h`".
- the thread name and action (`action_tx`) routine,
- the thread definition,
- the transmission layer name, as well as
- other internal objects and variables, such as `buffer`.

Transmitter.h

```
#include <systemc.h>
#include <occn.h>
#include "inout_pdu.h"
#include "MasterFrame.h"
#define NB_SEQUENCES 15

class transmitter :  public sc_module {
  public:
    transmitter(sc_module_name name, sc_time time_out);
    MasterFrame sap_tx; // MasterFrame specialized port
    SC_HAS_PROCESS(transmitter);
  private:
    void action_tx();
    BufferType buffer_tx; };
```

Transmitter.cc

```
#include <stdlib.h>
#include "transmitter.h"

// Transmitter constructor
transmitter::transmitter(sc_module_name name, sc_time time_out)
        : sc_module(name),
          sap_tx(time_out),
          buffer_tx()
{ SC_THREAD(action_tx); }

// Transmitter SC_THREAD process action_tx
void transmitter::action_tx() {
  uint addr = 0;
  Random rnd; // Random is an OCCN class for random generation
  do {
    for (int i=0; i < buffer_tx.sdu_size; i++)
       buffer_tx[i] = (char) (rnd.integer(26)+65);
    // Report
    cout << sc_time_stamp() << ": body of buffer_tx #" << addr;
    cout << "transmitted with TL_send" << endl << buffer_tx;
    sap_tx.TransportLayer_send(addr*buffer_tx.sdu_size, buffer_tx);
    addr++;
  } while(addr < NB_SEQUENCES);
  sc_stop();
}
```

The Transmitter module provides a transmission layer interface described in code blocks "`MasterFrame.h`" and "`MasterFrame.cc`". This layer defines a very simple communication API. based on the `TransportLayer_send` function; its behavior can be summarized into two main actions:

- segmentation of the buffer into I-frames, with the relevant header construction and insertion; this action exploits Pdu class operators.
- sending the I-frame according to the PAR protocol.

MasterFrame.h

```
#include <systemc.h>
#include <occn.h>
#include "inout_pdu.h"

class MasterFrame : public MasterPort<I_FrameType, ACK_FrameType> {
  public:
   MasterFrame(sc_time t);
   ~MasterFrame();
   void TransportLayer_send(uint addr, BufferType& buffer);

  private:
   uint address_function(uint addr);
   EDC_type EDC_function_tx(I_FrameType frame);
   sc_time timeout;
};
```

MasterFrame.cc

```
#include "MasterFrame.h"

MasterFrame::MasterFrame(sc_time t) : timeout(t){}
MasterFrame::~MasterFrame() {}

void MasterFrame::TransportLayer_send(uint addr,
                                      BufferType& buffer_tx) {
  I_FrameType& frame_tx = *(new I_FrameType);
  ACK_FrameType& frame_ack_rx = *(new ACK_FrameType);
  bool received_ack;
  const seq_num total_frames = buffer_tx.sdu_size/frame_tx.sdu_size;
  seq_num i = 0;
  buffer_tx >> frame_tx; // Segmentation (Transport-Layer)
  do {
    occn_hdr(frame_tx, sequence) = i; // frame header set up
    // Network-layer function determining the address of the packet
    occn_hdr(frame_tx, address) = address_function(addr);
    // Data-link Layer function which adds the Error Detection Code
    occn_hdr(frame_tx, EDC) = EDC_function_tx(frame_tx);
    occn_hdr(frame_tx, source_id) = 1;
    asend(&frame_tx); // MasterPort API: non-blocking send
    frame_ack_rx = *receive(timeout, received_ack);
    if (received_ack) {
      if (occn_hdr(frame_ack_rx, ack)==i) { // pos ack => next frame
         i++;
         buffer_tx >> frame_tx; }
      reply(0); }
 } while (i<total_frames); }

//non-implemented, dummy functions
uint MasterFrame::address_function(uint addr) {return addr; }
EDC_type MasterFrame::EDC_function_tx(I_FrameType frame) {return 1;}
```

4.2.3 The Top Level `main.cc`

main.cc

```
#include <systemc.h>
#include <occn.h>
#include "inout_pdu.h"
#include "transmitter.h"
#include "receiver.h"

int main() {
  sc_clock clock1("clk",10,SC_NS);
  sc_time timeout_tx(40, SC_NS); // time out for the PAR protocol
  transmitter my_master("Transmitter", timeout_tx);
  receiver my_slave("Receiver");

  StdChannel<I_FrameType, ACK_FrameType> channel("ch"); // Stdchannel
  channel.clk(clock1); // clock associated to StdChannel
  my_master.sap_tx(channel); // port bind
  my_slave.sap_rx(channel);  // port bind

  sc_start(-1);
  return -1; }
```

Finally, the `main.cc` references to all modules, and as shown in the code block above, it

- instantiates the simulator Clock and defines the timeout delay,
- instantiates the SystemC Receiver and Transmitter modules,
- instantiates the StdChannel,
- distributes the Clock to the Receiver and Transmitter modules,
- connects the Receiver and Transmitter modules using a point-to-point channel (and in more general situations a multipoint) channel, and
- starts the simulator using the `sc_start` command.

4.3 OCCN Communication Refinement using STBus

This Section explains communication refinement, if the proprietary STBus NoC is used instead of the generic OCCN StdChannel. The refinement is described after a brief presentation of the STBus

4.3.1 The StBus NoC

The ST Microelectronics STBus may be considered as a first generation implementation of a NoC for system-on-chip applications, such as set-top-box, digital camera, MPEG decoder, and GPS. STBus consists of packet-based general-purpose transaction protocols, interfaces, primitives and architecture specifications layered on top of a highly integrated physical communication infrastructure. STBus is a scalable interconnect that can cater for all attached modules in the same way. It also provides interoperability across different vendors. This ensures that the model designer is isolated from system issues, while allowing the system designer to control and tune architecture- and implementation-specific parameters, such as topology, and arbitration logic for optimal performance. Thus, most STBus implementations are richly connected networks that are efficient in controlling the delivery of bandwidth and latency. This sharply contrasts with the monolithic structures common in many other on-chip interconnects.

STBus is the result of evolution of the interconnect developed for micro-controllers dedicated to consumer application, such as set top boxes, ATM networks, digital still cameras and others. Interconnect design was based on the accumulation of ideas converging from different sources, such as the transputer (ST20), the Chameleon program (ST40, ST50), MPEG video processing and VCI (Virtual Component Interface) organization. Today STBus is not only a communication system characterized by protocol, interfaces, transaction set, and IPs, but also a technology allowing to design and implement communication networks for SoC. Thus, STBus supports a development environment including tools for system level design, architectural exploration, silicon design, physical implementation and verification.

There exist three types of STBus protocols, with different complexity and implementation characteristics and various performance requirements.

- Type1 is the simplest protocol. The protocol is intended for peripheral register access. Th protocol acts as an RG protocol, without any pipelining. Simple load, and store operations of several bytes are supported.
- Type 2 includes pipelining features. It is equivalent to the “basic” RGV protocol. It supports an operation code for ordered transactions. The number of request cells in a packet is the same as the number of response cells.
- Type 3 is an advanced protocol implementing split transactions for high bandwidth requirements (high performance systems). It supports out of order execution. The size

of a response packet may be different than the size of a request packet. The interface maps the STBus transaction set on a physical set of wires defined by this interface.

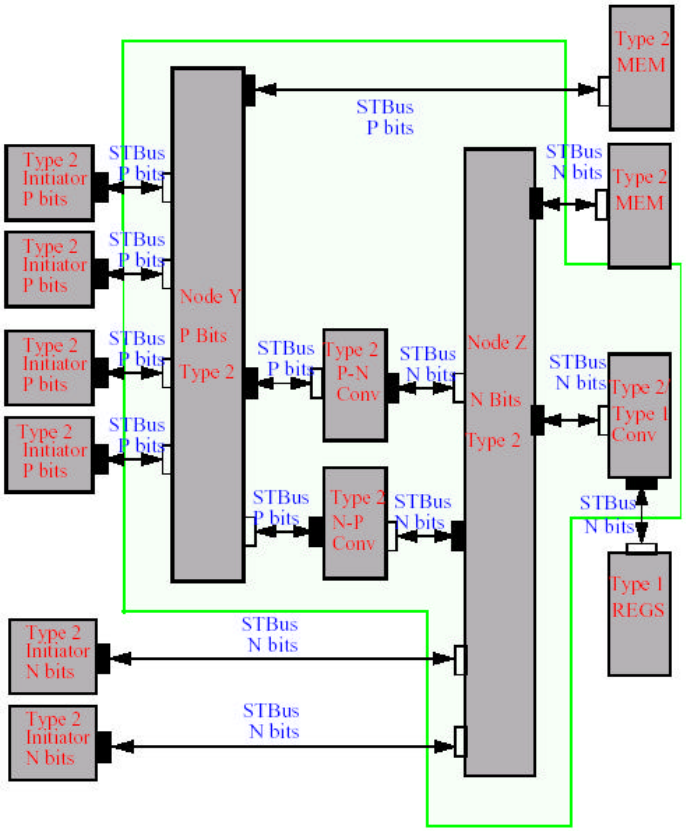

Figure 12. The STBus NoC

As shown in Figure 12, STBus NoC is built using several components, such as node, register decoder, type converter, size converter. The node is the core component, responsible for the arbitration and routing of transactions. The arbitration is performed by several components implementing various algorithms. A TLM cycle-accurate model for STBus has been developed using OCCN. The model provides all OCCN benefits, such as in simplicity, speed, and protocol in-lining. System architects are currently using this model in order to define and validate new architectures, evaluate arbitration algorithms, and discover trade-offs in power consumption, area, clock speed, bus type, request/receive packet size, pipeling (asynchronous/synchronous scheduling, number of stages), FIFO sizes, arbitration schemes (priority, least recently used), latency, and aggregated throughput.

We next illustrate important concepts in communication refinement and design exploration using the STBus model. Similar refinement or design exploration may be applied to AMBA AHB or APB bus models. These models have also been developed using OCCN. For further information regarding STBus and AMBA bus OCCN models please refer to [13, 44, 45].

4.3.2 Communication refinement with STBus

Figure 13 illustrates communication refinement, if the proprietary STBus NoC, described in Section 4.3.1, is used instead of the generic OCCN StdChannel.

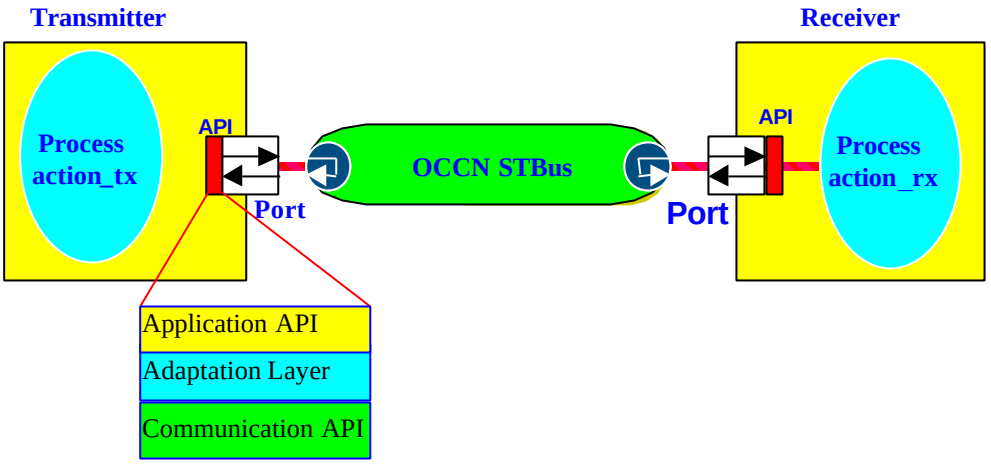

Figure 13. Transport layer protocol with STBus refinement

Refinement of the transport layer data transfer case-study is based on the simplest member of the STBus family. STBus Type 1 acts as an RG protocol, involves no pipelining, supports basic load/store operations, and is targeted at modules with low complexity, medium data rate system communication requirements.

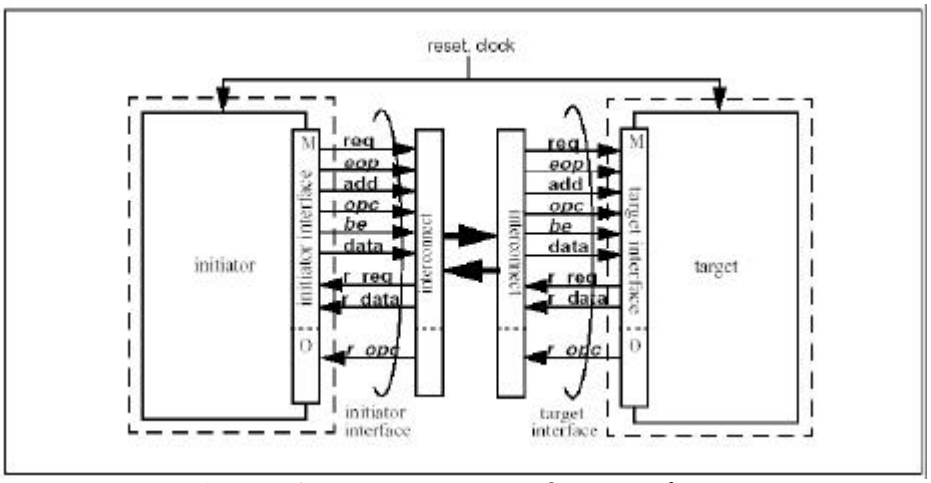

Figure 14. STBus Type 1 initiator and target interfaces

Figure 14 shows the simple handshake interface of STBus Type 1 [44, 45]. This interface supports a limited set of operations based on a packet containing one or more cells at the interface. Each request cell contains various fields: operation type (`opc`), position of last cell in the operation (`eop`), address of operation (`add`), data (`data`) for memory write, and relevant byte enable signals (`be`). The request cell is transmitted to the target, which acknowledges accepting the cell by asserting a handshake (`r_req`) and sending back a response packet. This response contains a number of cells, with each cell containing data (`r_data`) for read transactions, and optionally error information (`r_opc`) indicating that a specific operation is not supported, or that access to an address location within this device is not allowed. STBus uses `r_opc` information to diagnose various system errors.

The `TransportLayer_send` and `TransportLayer_receive` API do not change in terms of their semantics and interfaces, but the Adaptation Layer must implement the functionality required to map the Application API to the STBus intrinsic signals and

protocol. Thus, within the Adaptation Layer, we provide an STBus-dedicated communication API based on the MasterPort/SlavePort API. Due to space limitation, the code for this refinement is not provided. However, we abstract some of the key ideas used in the process.

According to Figure 14, the frame must be mapped to an `STBus_request Pdu` describing the path from initiator (transmitter) to target (receiver), and an `STBus_response` Pdu representing the opposite path. In particular,

- the payload corresponding to STBus write data lines (`data`) is mapped to the body of the STBus_request Pdu.
- buffer segmentation and reassembly are implemented as in StdChannel, by exploiting the >> and << operators of the OCCN Pdu object (the payload size is the same).
- the destination address corresponds to the STBus *addr* signal (which is a part of the STBus_request Pdu header), and
- the extra bits for the `EDC` are implemented as extra lines of the write data path.

With Type 1 protocol, completion of the initiator send means that the target has received the data. Thus, we avoid explicit implementation of the `sequence`, `source_id` (both fields) and `ack` fields in Figure 14.

Furthermore, since STBus guarantees a transmission free of packet loss, normally no timeout features are required. However, deep submicron effects may make the STBus a noisy channel. Thus, we may extend the basic idea of a PAR protocol, i.e. EDC with retransmission of erroneous data, to a real on-chip bus scheme [4, 26]. In this case, the `r_opc` signal, represented as a header field of the `STBus_response` Pdu, may an error code to the transmitter (initiator, in STBus terminology). This code may be used to determine if the same transaction must be repeated.

Since there is no direct access to the signal interface or the communication channel characteristics, we do not need to modify Transmitter or Receiver application modules, or the relevant test benches. Thus, we achieve module and tester design reuse at any level of abstraction without any rewriting and without riping up and re-routing communication blocks. This methodology facilitates OCCA design exploration through efficient hardware/software partitioning and testing the effect of various functions of bus architectures. In addition, OCCN extends state-of-the-art in communication refinement by presenting the user with a powerful, simple, flexible and compositional approach that enables rapid IP design and system level reuse

4.3 High-Level System Performance Statistical Functions

OCCN methodology for collecting statistics from system components can be applied to any modeling object. For advanced statistical data, which may include preprocessing, one may also directly use the public OCCN statistical classes. In order to generate basic statistics information appropriate `enable_stat_` function calls must be made usually from within the module constructor. For example, we show below the function call for obtaining write throughput (read throughput is similar) statistics.

```
// Enable statistics collection in [0,50000] with number of samples = 1
enable_stat_throughput_read(“statbox”, 0, 50000, 1, "Simulation Time",
                                  "Average Throughput for Write Access");
```

Considering our transport layer data transfer case-study without taking into account retransmissions, we measure the effective throughput in I-frame transfers in Mbytes/sec. Assuming packet-loss transmission, and a receiver that provides an acknowledgment after every clock cycle, i.e. a frame is transmitted every 2 cycles, the StdChannel can transmit a payload of 4 bytes during every (10ns) clock cycle. Thus, the maximum throughput is 200Mbytes/sec.

Figure 15 assumes a receiver with random response latency (not greater than 3 clock cycles), and an unreliable connection. In the left graph, an appropriate timeout is chosen according to the receiver latency, while in the right graph, the chosen timeout is too short, thus a high number of retransmissions occur. This obviously decreases the performance of the adopted ARQ protocol.

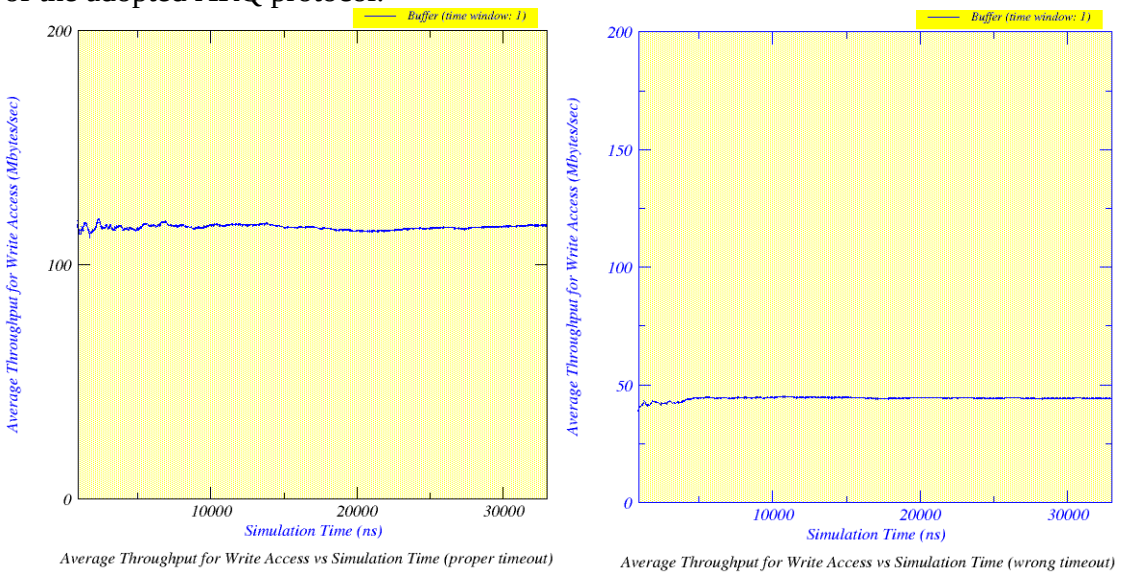

Figure 15. Performance results for transport layer protocol.

4.4 Design Exploration using OCCN

In order to evaluate the vast number of complex architectural and technological alternatives the architect is equipped with a highly-parameterized, user-friendly, and flexible OCCN methodology. This design exploration methodology is used to construct an initial architectural solution from system requirements, mapping inter-module communications through appropriate bus configuration parameters. Subsequently, this solution is refined through an iterative improvement strategy based on domain- or application-specific performance evaluation based on analytical models and simulation. The proposed solutions provide values for all system parameters, including configuration options for increasingly sophisticated multiprocessing, multithreading, prefetching, and cache hierarchy components.

Performance estimation is normally based on *mathematical analysis models* of system reliability, performance, and resource contention, as well as *stochastic traffic models* and *actual benchmarks* for common applications, e.g. commercial networking or multimedia. Performance evaluation must always take into account future traffic requirements arising from new applications, scaling of existing applications, or evolution of communication networks. In our case, we have simulated our simple transport layer protocol model for several million cycles with the STBus Type 1 channel on a Blade 1000 workstation. After

compiling all models with the highest optimization level, compiler switch –O3, we observed a simulation efficiency of ~100Kcycles/sec; simulation efficiency metrics are obtained by dividing the actual simulated cycles by CPU time. Furthermore, the simulation speed on customary buses, such as AMBA AHB or STBus Type 1, is almost independent of the number of initiators and targets, while for NoC, such as the STBus NoC, it is increasingly sensitive to the number of initiators, targets and computing nodes. This is due to the inherent complexity of the NoC architecture.

After extensive architecture exploration, ST Microelectronics exploits a reuse-oriented design methodology for implementing a generic interconnect subsystem. The system-level configuration parameters generated as a result of architectural exploration are loaded onto the Synopsys tools coreBuilder and coreConsultant. These tools integrate a preloaded library of configurable high-level (soft) IPs, such as the `STBus` interconnect; IP integration is performed only once using coreBuilder. CoreConsultant uses a user-friendly graphical interface to parameterize IPs, and automatically generate a gate-level netlist, or a safely configured and connected RTL view (with the correct components and parameters), together with the most appropriate synthesis strategy. Overall SoC design flow now proceeds normally with routing, placement, and optimization by interacting with various tools, such as Physical Compiler, Chip Architect, and PrimeTime.

Architecture exploration using OCCN allows the designer to rapidly assemble, synthesize, and verify a NoC through a methodology that uses pre-designed IP for each bus in the system. This approach dramatically reduces time-to-market, since it eliminates the need for long redesigns due to architecture optimization after RTL simulation.

5. Conclusions and Extensions

Network on-chip is becoming a critical determinant of system-level metrics, such as system performance, reliability, and power consumption. The development of SystemC as a general-purpose programming language for system modeling and design enables the application of C++ object-oriented technology for modeling complex MPSoC architectures, involving thousands of general purpose or application-specific PEs, storage elements, embedded hardware, analog front-end, and peripheral devices.

The OCCN project is aimed at developing new technology for the design of network on-chip for next generation high-performance MPSoCs. The OCCN framework focuses on modeling complex network on-chip by providing a flexible, state-of-the-art, object-oriented C++-based methodology consisting of an open-source, GNU General Public Licensed library, built on top of SystemC. OCCN methodology is based on separating computation from communication, and establishing communication layers, with each layer translating transaction requests to lower-level communication protocols. Furthermore, OCCN provides several important modeling features.

- Object-oriented design concepts, fully exploiting advantages of this software development paradigm.
- Efficient system-level modeling at various levels of abstraction. For example, OCCN allows modeling of reconfigurable communication systems, e.g. based on reconfigurable FPGA. In these models, both the channel structure and binding change during runtime.
- Optimized design based on system modularity, refinement of communication protocols, and IP reuse principles. Notice that even if we completely change the

internal data representation and implementation semantics of a particular system module (or communication channel), while keeping a similar external interface, users can continue to use the module in the same way.

- Reduced model development time and improved simulation speed through powerful C++ classes.
- System-level debugging using a seamless approach, i.e. the core debugger is able to send detailed requests to the model, e.g. dump memory, or insert breakpoint.
- Plug-and-play integration and exchange of models with system-level toolssupporting SystemC, such as System Studio(Synopsys), NC-Sim (Cadence), and Coware, making SystemC model reuse a reality.
- Efficient simulation using direct linking with standard, nonproprietary SystemC versions.
- Early design space exploration for defining the merits of new ideas in OCCA models, including high-level system performance modeling.

We also hope to develop new methodology and efficient algorithms for automatic design exploration of high-performance network on-chip. Within this scope, we hope to focus on system-level performance characterization and power estimation using statistical (correlated) macros for NoC models, considering topology, data flow, and communication protocol design. These models would eventually be hooked to the OCCN framework for SystemC simulation data analysis, and visualization.

While OCCN focuses on NoC modeling, providing important modeling components and appropriate design methodology, more tools are needed to achieve overall NoC design. For example, interactive and off-line visualization techniques would enable detailed performance modeling and analysis, by

- developing advanced monitoring features, such as generation, processing, dissemination and presentation,
- providing asynchronous statistics classes with the necessary abstract data types to support waves, concurrency map data structures and system snapshots, and
- combining modeling metrics with platform performance indicators which focus on monitoring system statistics, e.g. simulation speed, computation and communication load. These metrics are especially helpful in improving simulation performance in parallel and distributed platforms, e.g. through automatic data partitioning, or dynamic load balancing.

References

1. Albonesi, D.H., and Koren, I. “STATS: A framework for microprocessor and system-level design space exploration”. *J. Syst. Arch.* , 45, 1999, pp. 1097-1110.
2. Amba Bus, Arm, http://www.arm.com
3. Benini, L., and De Micheli, G. Networks on Chips: “A new SoC paradigm”, *Computer*, vol. 35 (1), 2002, pp. 70—781.
4. Bertozzi, D., Benini , L., and De Micheli, G. “Error control schemes for on-chip interconnection networks: reliability versus energy efficiency”. *Networks on Chip*, Eds. A. Jantsch and H. Tenhunen, Kluwer Academic Publisher, 2003, ISBN: 1-4020-7392-5.
5. Bertozzi, D., Benini , L., and De Micheli, G. “Low power error resilient for on-chip data buses, *Proc. Design Automation & Test in Europe Conf.*, 2002, pp.102-109
6. Brunel J-Y., Kruijtzer W.M., Kenter, H.J. et al. “Cosy communication IP's”. *Proc. Design Automation Conf.*, 2000, pp. 406-409.
7. Bolsens, I., De Man H.J., Lin, B., van Rompaey, K., Vercauteren, S., and Verkest, D. “Hardware/software co-design of digital communication systems”. *Proc. IEEE*, 85(3), 1997, pp. 391—418.

8. Carloni, L.P. and Sangiovanni-Vincentelli, A.L. “Coping with latency in SoC design”. *IEEE Micro, Special Issue on Systems on Chip*, Vol. 22-5, 2002, pp. 24—35.
9. Carloni, L.P., McMillan, K.L. and Sangiovanni-Vincentelli, A.L. Theory of latency-insensitive design. *IEEE Trans. Computer-Aided Design of Integrated Circuits & Syst.* Vol. 20-9, 2001, pp 1059—1076.
10. Caldari, M., Conti, M., Pieralisi, L., Turchetti, C., Coppola, M., and Curaba, S., “Transaction-level models for Amba bus architecture using SystemC 2.0”. *Proc. Design Automation Conf*., Munich, Germany, 2003.
11. Cierto virtual component co-design (VCC), Cadence Design Systems, see http://www.cadence.com/articles/vcc.html
12. Coppola, M., Curaba, S., Grammatikakis M.D. and Maruccia, G. “IPSIM: SystemC 3.0 enhancements for communication refinement”, *Proc. Design Automation & Test in Europe Conf.*, 2003, pp. 106—111.
13. Coppola, M., Curaba, S., Grammatikakis, M., Maruccia, G., and Papariello, F. “The OCCN user manual”. Also see http://occn.sourceforge.net (downloads to become available soon)
14. Coppola, M., Curaba, S., Grammatikakis, M and Maruccia, G. “ST IPSim reference manual”, internal document, ST Microelectronics, September 2002.
15. Coppola, M., Curaba, S., Grammatikakis, M. and Maruccia, G. “ST IPSim user manual”, internal document, ST Microelectronics, September 2002.
16. Dewey, A., Ren, H., Zhang, T. “Behaviour modeling of microelectromechanical systems (MEMS) with statistical performance variability reduction and sensitivity analysis”. *IEEE Trans. Circuits and Systems*, 47 (2), 2002, pp. 105—113.
17. Diep, T.A., and Shen J.R. “A visual-based microarchitecture testbench”, *IEEE Computer*, 28(12), 1995, pp. 57—64.
18. T. Dumitras, T., Kerner, S., and Marculescu, R. 'Towards on-chip fault-tolerant communication', *Proc.* ***Asia and S. Pacific Design Automation Conf.***, Kitakyushu, Japan, 2003.
19. De Bernardinis, F., Serge, M. and Lavagno, L. “Developing a methodology for protocol design ”. *Research Report SRC DC324-028*, Cadence Berkeley Labs, 1998.
20. Ferrari, A. and Sangiovanni-Vincentelli, A. “System design: traditional concepts and new paradigms”. *Proc. Conf. Computer De*sign, 1999, pp. 2—13.
21. Forsell, M. “A scalable high-performance computing solution for networks on chips”, *IEEE Micro*, 22 (5), pp. 46—55, 2002.
22. Gajski, D.D., Zhu, J., Doemer, A., Gerstlauer, S., Zhao, S. “SpecC: Specification language and methodology”. *Kluwer Academic Publishers*, 20003. Also see http://www.specc.org
23. Guerrier, P., and Greiner, A. "A generic architecture for on-chip packet-switched interconnections", *Proc. Design, Automation & Test in Europe Conf.*, 2000, pp. 250—256.
24. Grammatikakis, M.D., and Coppola, M. "Software for multiprocessor networks on chip", *Networks on Chip*, Eds. A. Jantsch and H. Tenhunen, Kluwer Academic Publishers, 2003, ISBN: 1-4020-7392-5.
25. Grammatikakis, M.D., Hsu, D. F. Hsu and Kraetzl, M. "*Parallel System Interconnections and Communications*", CRC press, 2000, ISBN: 0-849-33153-6,
26. Raghunathan, V., Srivastava M.B., and Gupta, R.K. “A survey of techniques for energy efficient on-chip communication”,. *Proc. Design Automation Conf.*, Anaheim, California, 2003.
27. Haverinen, A., Leclercq, M, Weyrich, N, Wingard, D. “SystemC-based SoC communication modeling for the OCP protocol”, white paper submitted to SystemC, 2002. Also see http://www.ocpip.org/home
28. Holzmann, G. J. “Design and validation of computer protocols”. Prentice-Hall International Editions, 1991
29. 15Klindworth, A. “VHDL model for an SRAM”. Report, CS Dept, Uni-Hamburg. See http://tech-www.informatik.uni-hamburg.de/vhdl/models/sram/sram.html
30. Krolikoski, S., Schirrmeister, F., Salefski, B. Rowson, J., and Martin, G. "Methodology and technology for virtual component driven hardware/software co-design on the system level", Int. Symp. Circ. and Syst. Orlando, Florida, 1999.
31. “IBM On-chip CoreConnect Bus”. Available from http://www.chips.ibm.com/products/coreconnect
32. Lahiri, K., Raghunathan, A. and Dey, S. “Evaluation of the traffic performance characteristics of SoC Communication Architectures”, Proc. Conf. VLSI Design, Jan. 2001.
33. Lahiri, K., Raghunathan, A., and Dey, S. "Design space exploration for optimizing on-chip communication networks", to appear, *IEEE Trans. on Computer Aided-Design of Integrated Circuits and Systems*.
34. Lahiri, K., Raghunathan, A., and Dey, S. "System level performance analysis for designing on-chip communication architectures", *IEEE Trans. on Computer Aided-Design of Integrated Circuits and Systems*, 20 (6), 2001, pp.768-783.
35. Networks on Chip, Eds. Jantsch, A. and Tenhunen, H. .Kluwer Academic Publisher, 2003, ISBN: 1-4020-7392-5.
36. Nussbaum, D., and Agarwal, A. “Scalability of parallel machines”. *Comm. ACM*, 34(3), pp. 56--61, 1991.
37. Paulin, P., Pilkington, C., and Bensoudane E., "StepNP: A system-level exploration platform for network processors", *IEEE Design and Test*, 2002, 19 (6), 17-26.
38. Poursepanj, A. “The PowerPC performance modeling methodology”. *Comm. ACM*, 37(6), 1994, pp 47—55.
39. Raw Architecure Workstation. Available from http://www.cag.lcs.mit.edu/raw
40. Rowson, J.A. and Sangiovanni-Vincentelli, A.L. “Interface-based design”. *Proc. Design Automation Conf.* 1997, pp. 178–183.

41. Salefski, B., and G. Martin, G. "System level design of SoC's", Int. Hard. Desc. Lang. Conf., 2000, pp. 3-10. Also in "*System On Chip Methodology and Design Languages*", eds. Ashenden, P.J., Mermet, J.P., and Seepold, R. Kluwer Academic Publisher, 2001.
42. Selic, B, Gullekson, G., and Ward P.T. "Real-time object-oriented modeling", J. Wiley & Sons, NY, 1994.
43. Sgroi, M. Sheets, M. Mihal, A. et al. "Addressing system-on-a-chip interconnect woes through communication-based design". *Proc. Design Automation Conf.*, 2001.
44. Scandurra A., Falconeri, G., Jego, B., 'STBus communication system: concepts and definitions", internal document, ST Microelectronics, 2002.
45. Scandurra A., "STBus communication system: architecture specification", internal document, ST Microelectronics, 2002.
46. Tanenbaum, A. "Computer networks". Prentice-Hall, Englewood Cliffs, NJ, 1999.
47. Turner, J. and Yamanaka, N. "Architectural choices in large scale ATM switches," IEICE Trans. on Communications, vol. E-81B, Feb. 1998.
48. Verkest, D., Kunkel, J. and Schirrmeister, F. "System level design using C++". *Proc. Design, Automation & Test in Europe Conf.*, 2000, pp. 74—83.
49. VSI Alliance, http://www.vsi.org/
50. Wicker, S. "*Error control systems for digital communication and storage*", Englewood Cliffs, Prentice Hall, 1995.
51. Zivkovic, V.D., van der Wolf, P., Deprettere, E.F., and de Kock, E.A. "Design space exploration of streaming multiprocessor srchitectures", IEEE Workshop on Signal Processing Systems, San Diego, Ca, 2002.
52. Zhang, T., Chakrabarty, K., Fair, R.B. "Integrated hierarchical design of microelectrofluidic systems using SystemC". *Microelectronics J.*, 33, 2002, pp. 459—470.

Biographies

Marcello Coppola received the Laurea degree in computer science from Pisa University, in 1992. Previously, he was in the architecture group at the INMOS in Bristol (UK). Currently, he is working for STMicroelectronics within a Research LAB. His current research interests are discrete event simulation, SoC modeling and architecture, programming languages, concurrent programming, RTOS and software/hardware engineering tools. He is responsible for R&D programs in real-time hardware, and software development techniques. He has published several papers in the areas of system modeling. He has chaired international conferences on SoC design and helped to organize several others. He is a member of OSCI and is contributing to MEDEA+ roadmap as well as the SystemC standardization.

Stephane Curaba was born in Grenoble, France, in 1973. He graduated from ISTG 3I engineering school in 1996. He worked for 3 years in Altran group on missions related to embedded software development. He joined the AST Grenoble Lab in June 2000. Since then he has been working on SoC modeling and co-developed an innovative SystemC-based C++ modeling and simulation environment called IPsim. called IPSIM. He now focuses on NoC modeling and is involved in the development of the open source OCCN framework.

Miltos D. Grammatikakis, received MSc (1985) and PhD (1991) in Comp. Sci. from the University of Oklahoma. He was a postdoctoral fellow in the parallel processing laboratory of ENS-Lyon (1991-1992), researcher at ICS, FORTH (1992-1995), assistant professor (C1) at the University of Hildesheim (1995-

1998) and Oldenburg (1998-1999), senior software engineer at INTRACOM (1999-2000) and ISD S.A. (2000-), and visiting associate professor at the University of Crete (2002-). While at ISD, he worked jointly with the AST Lab of ST Microelectronics on the proprietary SystemC-based IPSIM library for SoC modeling, and the open source OCCN NoC modeling framework. Since 2003, he is a tenured professor at TEI-Crete. He has participated in European ESPRIT, TEN TELECOM, TMR, EURESCOM, MEDEA+ and national R&D projects dealing with EDA, parallel architectures, distributed systems for telecom, and satellite networks. His professional interests include electronic system-level modeling for SoC and NoC, parallel and distributed systems, high-speed networks, discrete-event simulation, probabilistic modeling and testing. He has published over 40 technical articles, in international conference proceedings, edited books, and journals, and is the main co-author of ``Parallel Systems: Communications and Interconnects" published by CRC. Dr. Grammatikakis has been a member of IEEE, ACM and SIAM.

Riccardo Locatelli was born in Siena, Italy, in 1974. He received the M.S. degree (summa cum laude) in Electronic Engineering from the University of Pisa on February 2000. He is currently pursuing the Ph.D. at the Dept. of Information Engineering of the same University. His research interests include definition and prototyping of SoC architectures for video applications with emphasis on low power techniques and system communication. He also works in advanced signal processing schemes for VDSL application. Now he carries out a research stage at AST Grenoble Lab of STMicroelecronics on modeling future NoC architectures within the OCCN framework.

Francesco Papariello was born in Termoli, Italy, in 1972. He graduated from the University of Pisa in May 2000, with a specialization in Computer Science and Control Automation. He joined AST Grenoble Lab in September 2000. He worked for 2 years on the Design Space Exploration (DSE) project. Afterwards, he worked on the integration of an Instruction Set Simulator for system-level modeling. He is also involved in core processor architecture definition for Multiprocessor System on Chip (MPSoC).

Giuseppe Maruccia received his degree in Electronic Engineering from the University of Cagliari. His thesis work focused on the development of reusable IP components in VHDL for a SoC. He joined the AST Catania Lab of ST Microelectronics in May 2000. Since June 2000 he has been working on SoC modeling, and especially the development of a SystemC-based C++ modeling and simulation environment called IPsim. Since January 2003 he has been collaborating with AST Grenoble Lab, on the OCCN activity.