On-Chip Communication Network: User Manual V1.0.1

Marcello Coppola[1], Stephane Curaba[1], Miltos Grammatikakis[2], Giuseppe Maruccia[1]and Francesco Papariello[1]
[1]ST Microelectronics, AST Grenoble Lab, 12 Jules Horowitz 38019 Grenoble, France
Emails: {marcello.coppola,stephane.curaba,giuseppe.maruccia,francesco.papariello}@st.com
[2]ISD S.A., K. Varnali 22, 15233 Halandri, Greece, Email: mdgramma@isd.gr

Abstract

The On-Chip Communication Network (OCCN) project provides an efficient, open-source, GNU-GPL licensed 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 communication 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) [18] and micro-electro-fluidic bio-chips (MEFS) [61] .

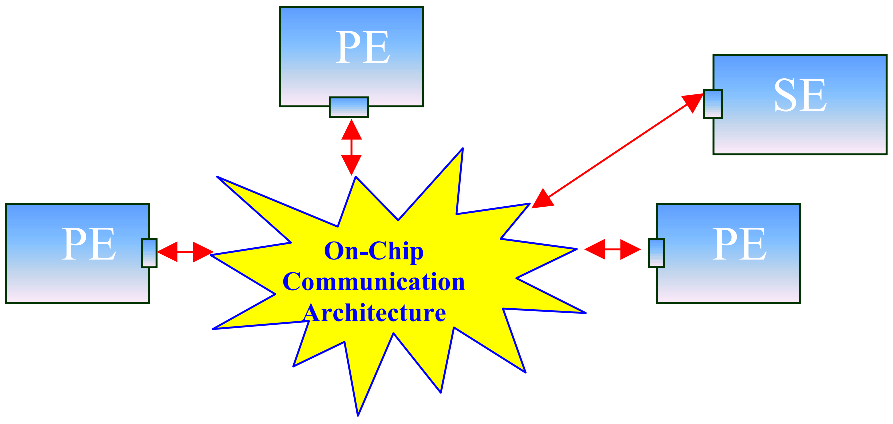

Figure 1. MPSoC configured with on-chip communication architecture, processing, and storage elements

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 [41]. 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 [51, 52], and Core Connect [35]. 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, 26, 41]. Innovative network on-chip architectures include LIP6's SPIN [26], MIT's Raw network [46], and VTT's Eclipse [24].

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. Furthermore, for achieving cost-effectively OCCA scalability, we must consider various architectural, algorithmic, and physical constraint issues arising from Technology [37]. 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 [27].

Realizability is associated to several network design issues that control system parallelism by limiting the concurrency level [28], 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,

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

- 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 [20]. 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, 19, 37, 38, 43, 45, 60].

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 [51, 52].

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 [58].

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.

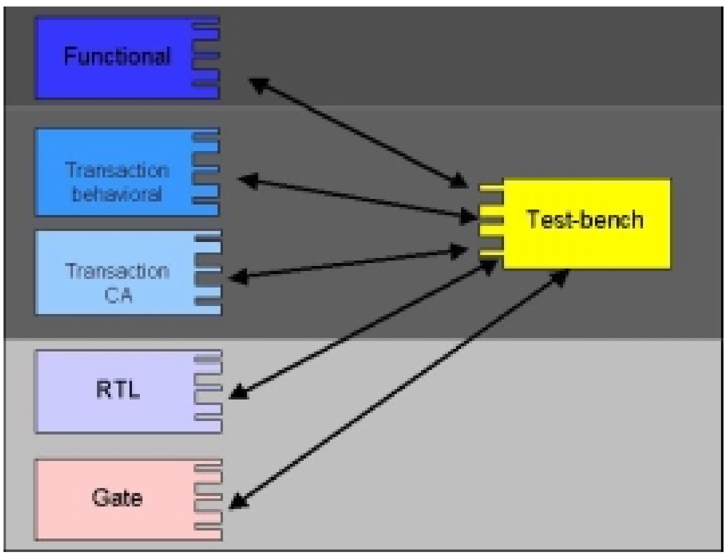

Figure 2. Modeling in various abstraction levels

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 [30]. 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.

As an example in Figure 3, we show a functional C++-based model of an SRAM. This high-level model is much simpler than a corresponding (300+ lines) VHDL model that includes complete timing parameters [32], thus decreasing design time and life cycle maintenance costs. Furthermore, notice that in the VHDL model description, it is not possible to distinguish code used specifically for SRAM control from code referring solely to the data path [57, 61].

```
/* core functionality written in C */
/* sram_func.c
#define SRAM_SIZE 128

static int32 sram[SRAM_SIZE]; // data element

static int32 sram_read(int32* ptr, int row) // control element
{ return *(ptr+row); }

void sram_write(int32* ptr, int row, int32 data) // control
{ *(ptr+row)=data; }

// Sram.h file – wrapper written in System C
class sram : public sc_module // function description
{ public:
    sram(sc_module_name name);
    sc_port<sc_fifo_in_if <int32> > in; // communication port
    sc_port<sc_fifo_out_if <int32> > out; // communication port
    SC_HAS_PROCESS(sram);
  private:
     void behavior(); };

//Sram.cpp file
sram::sram(sc_module_namename):sc_module(name) // SystemC wrapper
{ SC_THREAD(behavior); }
void sram::behavior()
{ struct {
    int32 data, addr;
    bool rnw } d;
  while (true) {
     in->read(d);
     if (d.rnw) out->write(sram_read(d.addr));
     else
     sram write(d.addr,d.data) }
```

Figure 3. Functional model abstraction of an SRAM

In order to provide system performance measurements, e.g. throughput rates, packet loss, or latency statistics, system computation (behavior) and communication models must be annotated with an abstract notion of time. Analysis using parameter sweeps helps estimate the sensitivity of high-level design due to perturbations in the architecture, and thus examine the possibility of adding new features in derivative products [8, 9]. However, architectural delays are not always cycle-accurate. For example, for computational components mapped to a particular RTOS or large communication transactions mapped to a particular shared bus, it is difficult to accurately estimate thread delays, which depend on precise system configuration and load. Similarly, for deep sub-micron technology, wire delays that dominate protocol timings can not be determined until layout time. Thus, one must include all necessary synchronization points and/or interface logic in order to avoid deadlocks or data races and ensure correct behavior independent of computation or communication delays.

All these abstract models can be analyzed, verified and validated in stand-alone manner. However, detailed VHDL or Verilog models are inadequate for system level description due to poor simulation performance, and a higher abstraction level is desirable. Thus, when high-level views and increased simulation speed are desirable, systems may be modeled at the transaction level using an appropriate abstract data type (ADT), e.g. processor core, video line, or network communication protocol-specific data structure. With transactional modeling, the designer is able to focus on IP functionality, rather than on detailed data flows or hardware details of the physical interface, e.g. FIFO sizes and time constraints.

2.2 Separation of Communication and Computation Components

System level design methodology is based on the concept of orthogonalization of concerns [25]. 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 [41]. 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` [51, 52], OCP [30], `VCI` interfaces [58], 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.

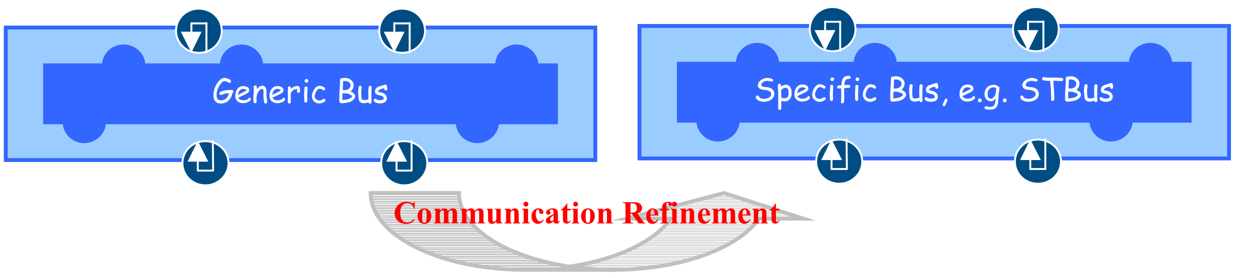

Figure 4. Enabling communication refinement

As shown in Figure 4, 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 [13, 21, 22, 34, 47, 48, 50]. 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 [14, 16, 17].

- 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).

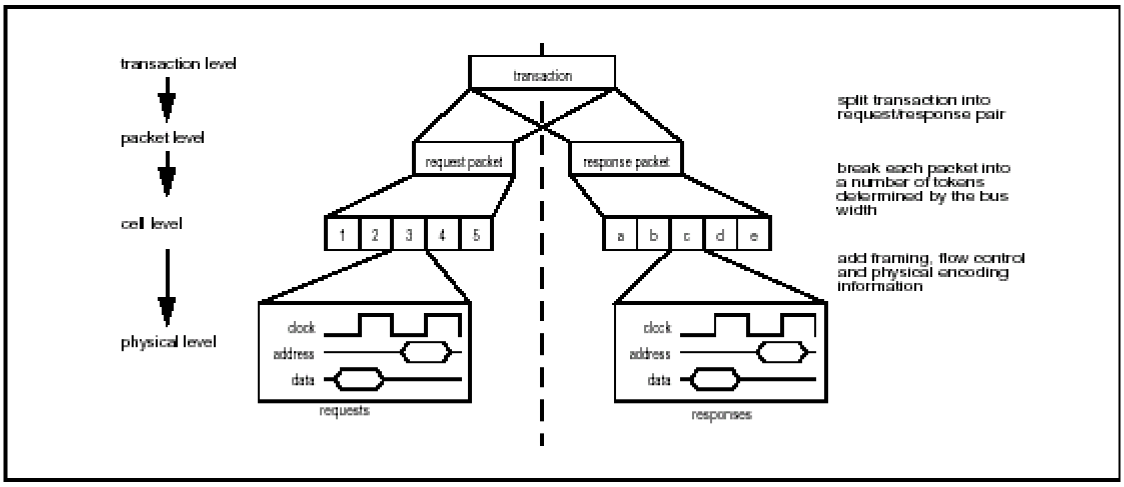
transaction level
packet level
cell level
physical level
transaction
request packet
response packet
1
2
3
4
5
a
b
c
d
e
clock
address
data
requests
responses
split transaction into request/response pair
break each packet into a number of tokens determined by the bus width
add framing, flow control and physical encoding information

Figure 5. STBus: An OSI-like layered approach to inter-module communications

An extension of this approach to include more communication layers is immediate. For example, as shown in Figure 5, the model for the communication protocol stack of ST Microelectronics' STBus, an on chip communication network, is based on the following four layers.

- The *transaction layer* defines communication between two system modules. Most communications involve data access. A data access may involve the transfer of a single packet or a sequence of packets.

- The *packet layer* breaks operations into one or more request/response packet pairs depending on the complexity of the protocol, the amount of data to be transferred, and the width of the interface. Each packet carries a fixed quantum of data directly mapped onto the interconnection architecture. Thus, for STBus, the amount of information to be transferred in the request phase is either the same (type 1/type 2), or differ (type 3 interface) from that transferred in the response phase
- The *cell layer* breaks these packets into a series of cells, each cell having the right width to match the packet route through the on-chip communication architecture.
- Finally, the *physical layer* is responsible for physical encoding of these cells, adding outbound framing and flow control information. Simple protocol verification may be performed for each layer in the hierarchy.

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 6. 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 6, 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.

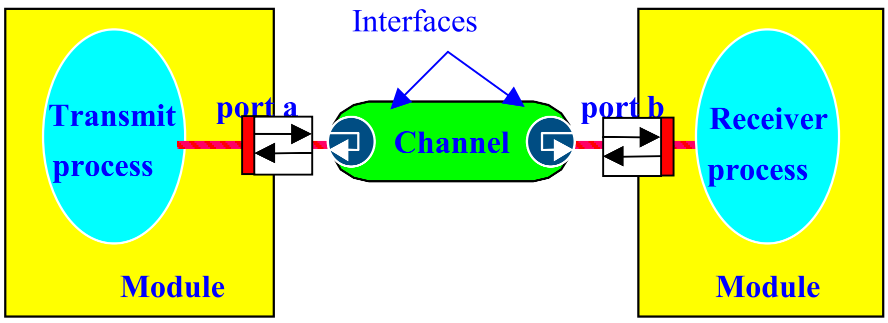

Figure 6. SystemC module components: behavior and inter-module communication

Access to a channel is provided through specialized ports (small red squares in Figure 6. 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 [49]. 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 7 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.

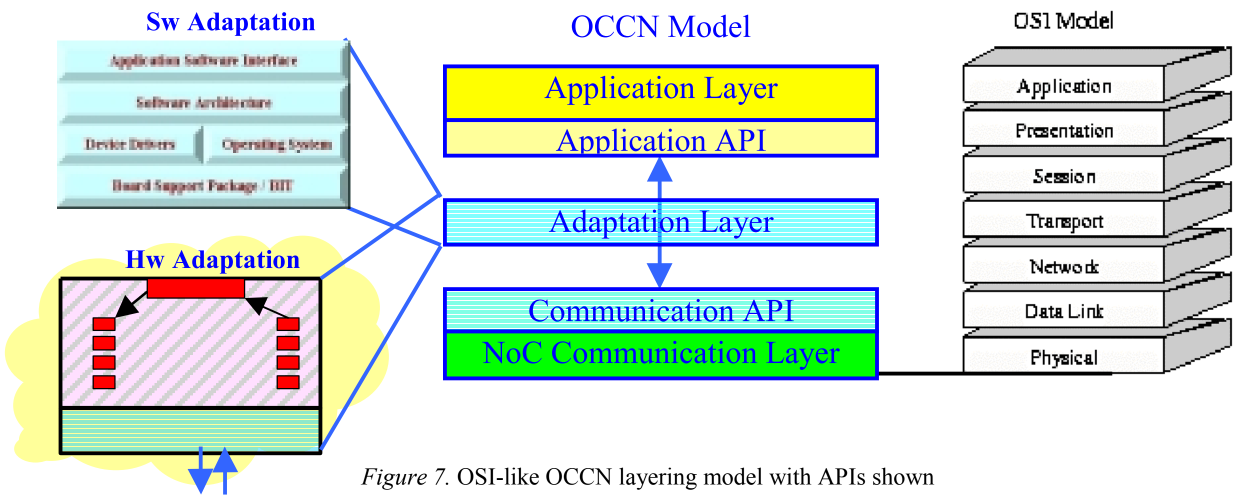

Figure 7. OSI-like OCCN layering model with APIs shown

An implementation of an adaptation layer includes software and hardware components, as shown in the left part of Figure 7. 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 a body of `length` many elements of `BodyUnitType`:
  `Pdu<BodyUnitType, length> 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.

For example, we can define a Pdu containing only a header as follows.

```
struct header {char addr, char opcode};
Pdu<header> pdu1;
```

Similarly, we can define a Pdu containing a body with a single character as follows.

```
Pdu<char> pk1, pk2; // we declare two Pdus containing body only
pk1='a'; // pk1 contains 'a'
pk2=pk1; // now pk2 is equal to pk1, copy operator
char x=pk1; // x assumes the value of 'a';
// since pk1 is equal to pk2 the cout statement is executed
if (pk1==pk2)
   cout << "pk1 == pk2" << endl;
```

Pdus may be duplicated as follows. This operation is useful for sending a Pdu while keeping a local copy.

```
Pdu<char> pk1, pk2;
pk2=pk1;
```

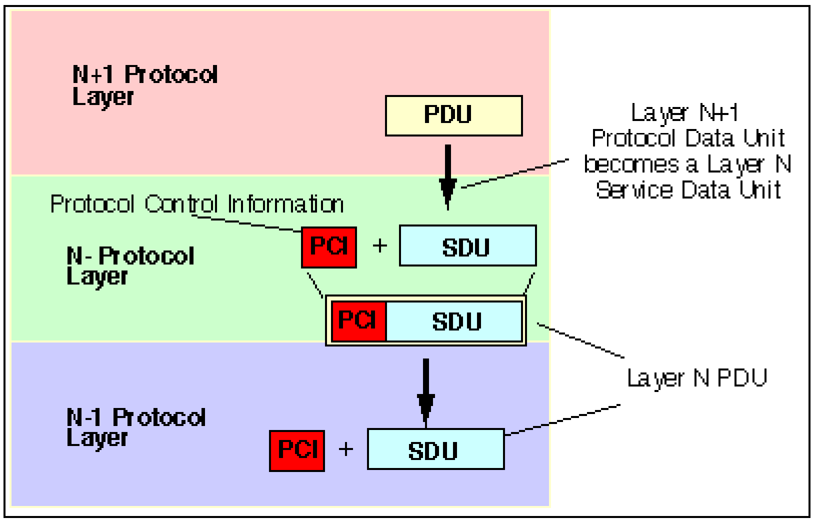
N+1 Protocol Layer
PDU
Layer N+1 Protocol Data Unit becomes a Layer N Service Data Unit
Protocol Control Information
N- Protocol Layer
PCI
+
SDU
PCI
SDU
Layer N PDU
N-1 Protocol Layer
PCI
+
SDU

Figure 8. Pdu encapsulation in communication layers

As shown in Figure 8, layered protocols may also encapsulate a Pdu from another layer.

The following example shows all operations that an N-layer protocol has to perform in order to build an N-layer Pdu, and subsequently an N+1-layer Pdu.

```
typedef Pdu<n1_pci,char> N1pdu  // declare a Pdu for layer N+1
N1pdu p1, p2;
typedef Pdu<n_pci,N1pdu> Npdu // declare a Pdu for layer N
NPdu p3,p4;
p1='a';// p1 contains 'a'
p3=p1; // p3 contains p1 (add operation)
p4=p3; // p4 is equal to p3 (copy operation)
p2=p4; // p2 assumes the value of the body (strip operation)
char tmp=pk1;
// since both the body of p1 and tmp are equal to 'a',
// then, the cout statement is executed
if (pk1=='a') && (tmp=='a')
   cout << "pk1 == b" << endl;
```

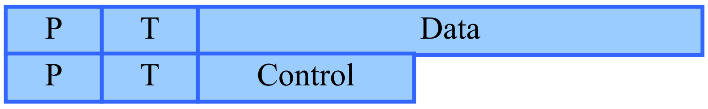

Figure 9. A simple token DS LINK (IEEE 1355)

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 9. 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;
```

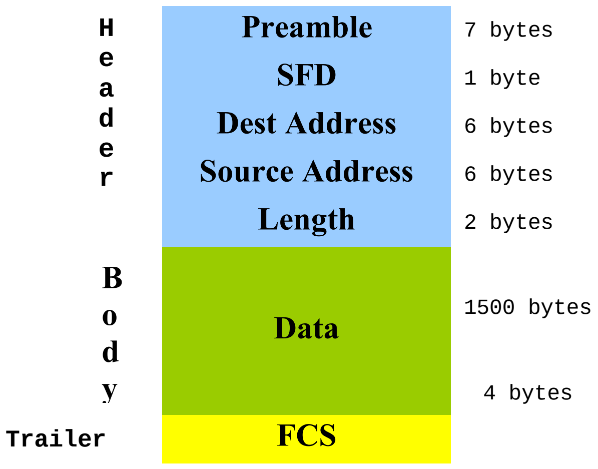

Figure 10. A complex Pdu for a CSMA/CD LAN

Alternatively, as illustrated in Figure 10, 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

The Pdu is a key element in the OCCN API, with its structure being determined by the corresponding OCCA model. In this Section we describe the transmission/reception interface of the OCCN API. The paradigm used for this interface is message-passing, 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. Thus, the same base functions are used for transmitting a Pdu for almost any OCCA, e.g. AMBA, STBus, Sonics, or Core connect.

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.

- In *synchronous blocking send/receive* one party may send a Pdu only if the other is prepared to receive it. A process that calls `send()` is blocked until its message is received,. Similarly, a process that calls `receive()` or `reply()` is blocked until it receives a message. If the channel is busy, waiting sender tasks compete to acquire the channel using a FIFO priority scheme.
- In *asynchronous blocking* send one party may send a Pdu whether or not the other is ready to receive it. Moreover, a process that calls `asend()`may execute other actions only after the dispatch of a Pdu, i.e. when the channel is free; if the channel is busy, waiting tasks compete to acquire the channel using a FIFO priority scheme. This

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

process may return later to check the status of the dispatched Pdu, e.g. by establishing a handshake.

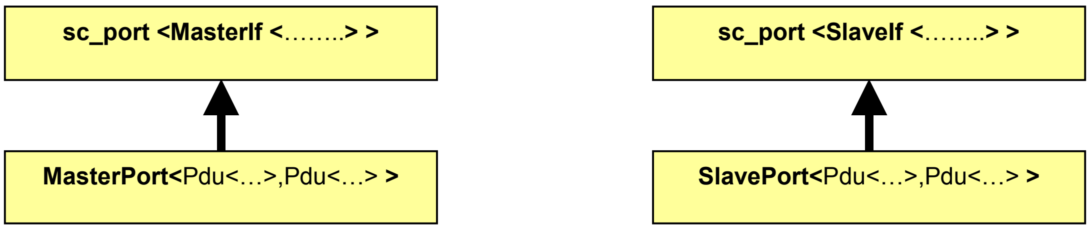
sc_port <MasterIf <........> >
MasterPort<Pdu<...>,Pdu<...> >
sc_port <SlaveIf <........> >
SlavePort<Pdu<...>,Pdu<...> >

Figure 11. `MasterPort` and `SlavePort` derived from `sc_port`

As shown in Figure 11, the transmission/reception function interface 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, i.e. 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. 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 [15].

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.

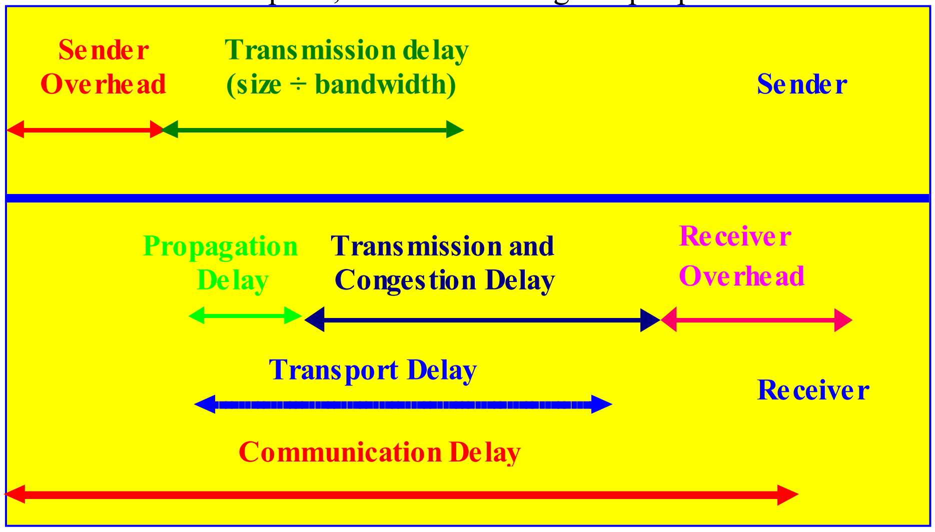

Figure 12. OCCN asynchronous blocking communication model

Assuming a global clock, Figure 12 illustrates time delays in a pair of communicating process (sender and receiver) under the packet routing model. Since timing information is protocol- and architecture-dependent, time delay is not a generic parameter.

Thus, from the side of the sender, we can define:

- *sender overhead* refers to sender overheads prior to Pdu transmission, such as a busy or non-functioning sender or next OCCA network node, or packetization in the send buffer,
- *transmission delay* refers to Pdu transmission, i.e. given the channel bandwidth and Pdu data size, this can be easily calculated; however, notice that this sender delay may also include extra time, due to packet admission control and traffic shaping policies.

Similarly, from the side of the receiver, we can define:

- *propagation delay* as the time that it takes one data unit to be transmitted from the sender to the receiver, and
- *congestion delay* refers to additional Pdu delay due to traffic congestion in the OCCN network, and
- *receiver overhead* refers to receiver overheads prior to or during Pdu reception, such as a busy or non-functioning receiver or previous OCCA network node, or de-acketization in the receiver buffer.

Figure 13 provides a graphical view of OCCN synchronous communication. A transmitter process in Module A initiates a Pdu send. Then, after a communication delay due to sender and receiver overheads, propagation, and congestion, the receiver process in Module B is able to issue a Pdu receive event. At the same time, or possibly after a specific delay reflecting receiver process overhead, an acknowledgement is sent back to Module A to confirm message arrival. Notice that the sender is blocked from the time the Pdu is sent to the time the Pdu acknowledgment is received.

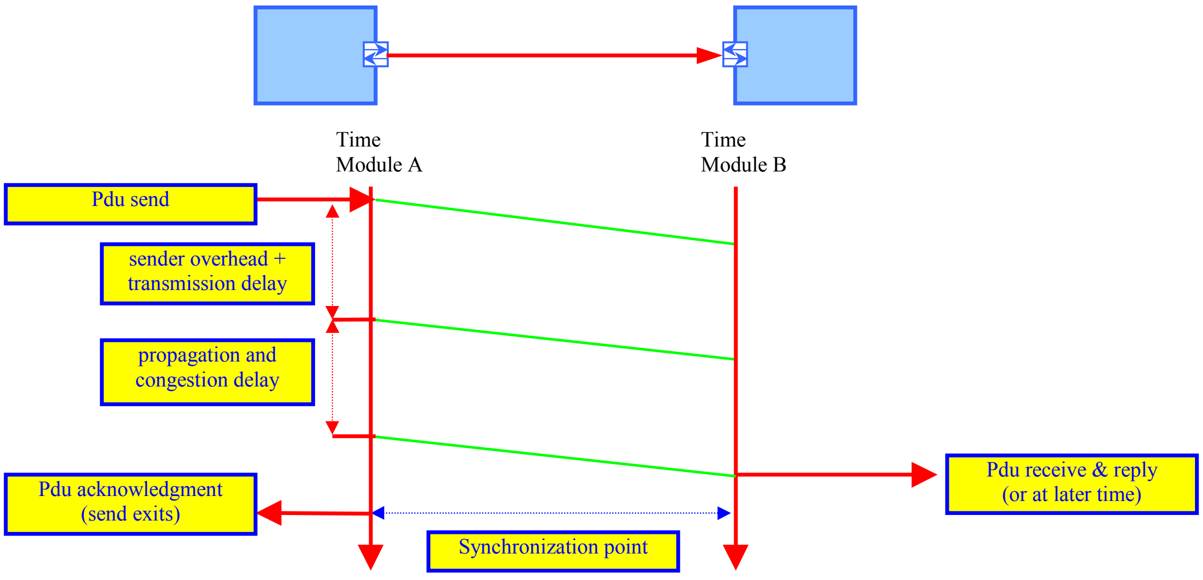
Time
Module A
Time
Module B
Pdu send
sender overhead + transmission delay
propagation and congestion delay
Pdu acknowledgment (send exits)
Pdu receive & reply (or at later time)
Synchronization point

Figure 13. OCCN synchronous communication model

Asynchronous blocking send is similar, except that a Pdu acknowledgment does not exist, and thus, `asend` exits soon.

In Figure 14, we show state changes during OCCN synchronous and asynchronous point-to-point communications. A process which calls `send()` is "*Send Blocked*" until its message is received,. A process which calls `receive()` is "*Receive-blocked*" until it receives a message. A process which calls `asend()` is never blocked, . The only condition to be blocked is when it tries to send something when the channel is already transmitting. A process which calls `areceive()` is "*Receive-blocked*" until it receives a message.

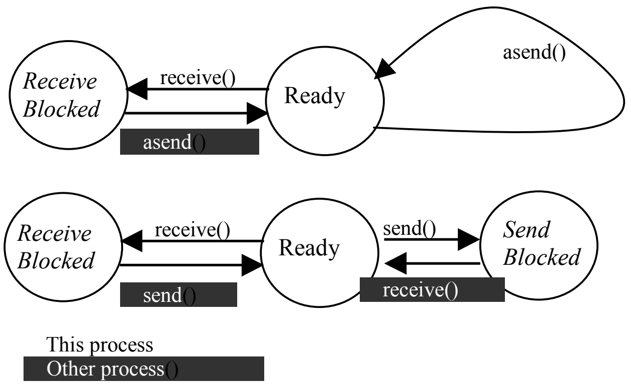

Figure 14. OCCN protocol `send/receive` transaction

Using the above send/receive functions and appropriate channel setup and control functions, any kind of on-chip communication protocol can be modeled. For instance, we can easily model a Request Grant Valid (RGV) protocol using the synchronous blocking paradigm for the request grant part and the asynchronous blocking scheme for the Valid part. RGV is a complex protocol shipping today in millions of SoC devices where high bandwidth peripherals (like MPEG decode hardware) share the system bus with a host CPU and many other peripherals of varying performance and system demands.

3.3 High-Level Performance Modeling

High-level system performance modeling is an essential ingredient in NoC and MPSoC design exploration and co-design. In this Section, we focus on the design of an object-oriented system performance modeling library for system-on-chip (SoC) and network-on-chip design based on SystemC. We discuss how OCCN enables automatic extraction of statistical metrics for throughput, delay, size, hit ratio and packet loss. We also consider advanced monitoring features, such as generation, processing, dissemination and presentation.

3.3.1 OCCN Statistics Interface

3.3.1.1 Automatic Extraction of Statistical Features

System-level modeling using SystemC is an essential ingredient of SoC design flow [53]. Data and control flow abstraction of the system hardware and software components express not only functionality, but also performance characteristics that are necessary to identify system bottlenecks. While for software components it is usually the responsibility of the user to provide appropriate performance measurements, for hardware components and interfaces it is necessary to provide a statistical package that hides internal access to the modeling objects. This statistical package can be used for evaluating SoC performance characteristics. The statistical data may be analyzed online using visualization software, e.g. the open source `Grace` tool, or dumped to a file for subsequent data processing, e.g. via an electronic spreadsheet or a specialized text editor.

For SoC modeling, we usually monitor simple dynamic performance characteristics, e.g. latency, throughput, and possibly power consumption (switching activity) for obtaining information on the effectiveness of intra-module computation, and inter-module communication and synchronization components. Although similar metrics for intra-module communication and synchronization objects can be defined, these components do not correspond to unique hardware components, since they may be implemented in various ways.

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. Notice that for a particular statistic, e.g. for read throughput, the basic graph layout (including the title and x, y axes) is similar, with the legends differentiating only across different objects. Thus, in order to distinguish the corresponding statistical graphs, the user must provide distinct names to all modeling objects, even if they exist within different modules.

Since complex systems involve both time-driven (instant) and event-driven (duration) statistics, we may provide two general monitoring classes, collecting instant and duration measurements from system components with the following functionality.

- In *time-driven simulation*, signals 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 method enables simple and precise measurement, it is 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, software probes are inserted into *the source code of library routines*, either manually by setting sensors and actuators, or more efficiently through the use of a monitoring segment which automatically compiles the necessary probes. Such software 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 average and instant throughput, latency, average and instant size, packet loss, and average hit ratio statistics can be based. 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.

We now describe our general methodology for collecting statistics from system components.

3.3.1.2 Overview of the OCCN Statistical Classes

Figure 15 illustrates the software components within the OCCN statistical package that include the following C++ statistical classes:

- top level, general classes, such as `BaseStat` and `GuiStat` (now implemented as a set of visualization functions),
- publicly available generic classes, such as `StatInstant` and `StatDuration`,
- derived time-driven and event-driven classes, such as `StatInstantAvg`, and `StatInstantCAvg`,
- application-specific functions enabling certain performance metrics, including `StatThroughput`, `StatDelay`, `StatSize` (and `StatInstantSize`), `StatProb` (and `StatInstantProb`) classes.

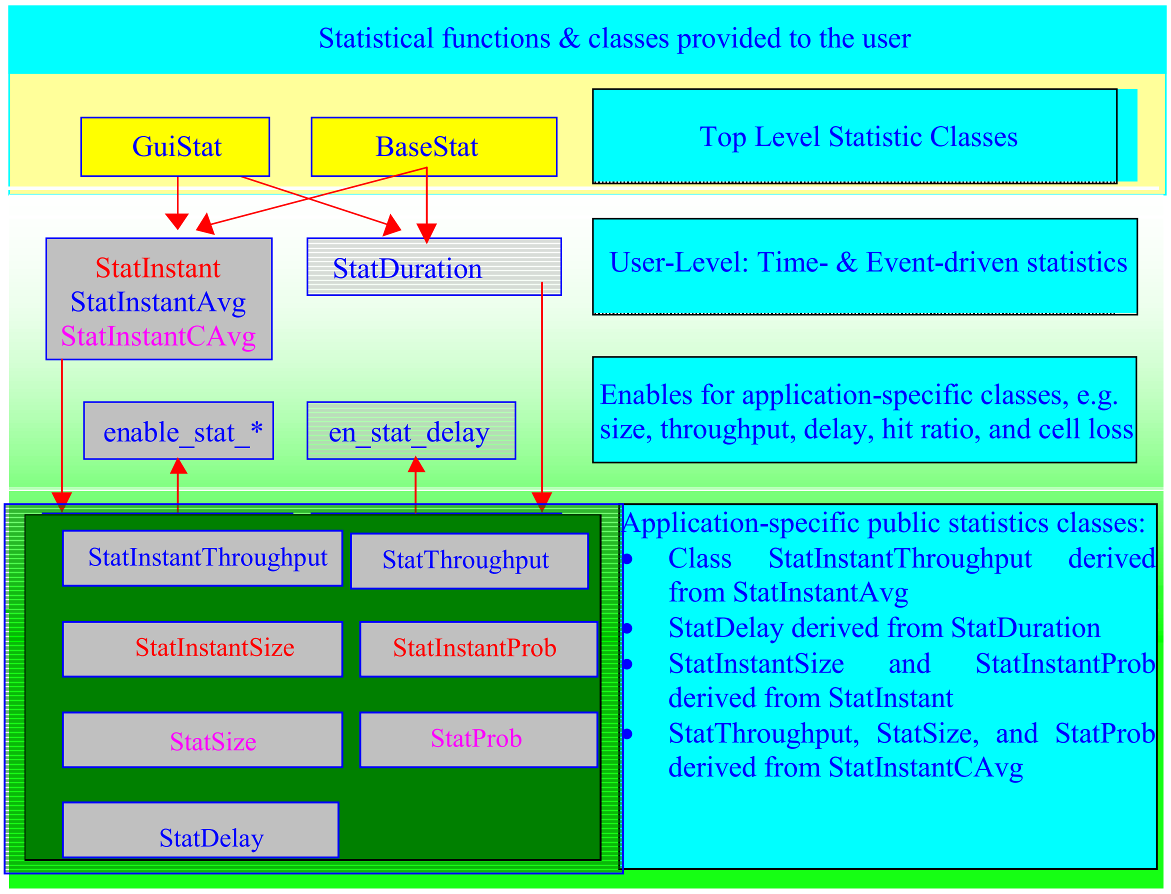
Statistical functions & classes provided to the user
GuiStat
BaseStat
Top Level Statistic Classes
StatInstant
StatInstantAvg
StatInstantCAvg
StatDuration
User-Level: Time- & Event-driven statistics
enable_stat_*
en_stat_delay
Enables for application-specific classes, e.g. size, throughput, delay, hit ratio, and cell loss
StatInstantThroughput
StatThroughput
StatInstantSize
StatInstantProb
StatSize
StatProb
StatDelay
Application-specific public statistics classes:
Class StatInstantThroughput derived from StatInstantAvg
StatDelay derived from StatDuration
StatInstantSize and StatInstantProb derived from StatInstant
StatThroughput, StatSize, and StatProb derived from StatInstantCAvg

Figure 15. Statistics Classes in OCCN library

3.3.1.3 Top Level Statistical Classes: Gui_Stat & Base_Stat

The `gui_stat` class is gui-specific, i.e. for the current OCCN version it is targeted to the Grace™ environment. This class includes substantial details regarding the parameters for the graph layout, e.g. title, axis, ticks, legends and viewport, and also the Grace™ socket interface. However, for the purpose of this document, it is sufficient to say that this class includes the necessary routines for visualizing both time-driven and event-driven statistics.

The `base_stat` class includes:

- *start- and end-time for statistics collection*; both times refer to the synchronous system clock; the warmup time implied by start-time is necessary sometimes, e.g. for starting measurements after reaching steady state in a dynamic system.
- *table of values for storing elements*, usually an array,
- *the number of statistical samples entered so far*,
- *a sum of values and/or a cummulative sum of values taken within a time window*,
- *user-defined boolean flags* representing statistical values on the monitored system parameter; these boolean flags select one or more flags from the following list of statistical metrics: *frequency statistics*, such as average, sum, max, min, current value, and standard deviation, and *cummulative statistics*, i.e. within a user-specified uniform time window T, such as e.g. cummulative average, sum, max, min, and standard

deviation; notice that sum is useful, e.g. in computing the total number of packets received or transmitted by a module,

- *a uniform time window* refering to cummulative statistics.
- *user-defined or possibly customized GUI labels*, e.g. for 2-dimensional throughput graphics, the corresponding titles, units, and tick sizes for x and y-axis; notice that this particular functionality for common graphs, such as frequency statistics for latency and throughput, may be provided to the user directly (via an extra user-defined flag).
- *a user-defined caption*, usually a string describing a particular IP component instantiated within a user-derived class, e.g. "`Mem01_test1`" for a statistic test on a memory named as "`Mem01`".
- *user-defined log file(s)*, i.e. for dumping statistical values; these file names may either be automatically generated from the user-defined caption above, or be provided explicitly to the GUI.
- *a cache_size*, as a performance enhancement for the statistics package; this refers to the maximum number of elements to be stored in the internal table of values before being transferred to the appropriate GUI or flushed to the file.
- *system lock* necessary for shared read access to values written by different threads in a multithreading, different processes in a multiprogramming, and/or different processors in a parallel or distributed computing environment.

Based on the above description, the C++ implementation of the `base_stat` class is as follows.

```
#define max_stat_params 11
#define max_string_size 90

typedef struct {
   double time, value;
} stat_entry; // internal table (StatInstant & StatDuration classes)

typedef struct {
   long int object_ptr;
   double time, value;
} stat_entry_lib; // internal cache StatDurationLib Class
                 // for implementing  event_driven functions

class base_stat {
 public:
   // functions defining start/end of statistics collection
   void start_stat_time(double _start);
   void end_stat_time(double _end);

   bool is_activated; // non-zero flag for user-selected statistics

   unsigned int get_id (); // get id of class instance

   // user-defined boolean flags for user-selected statistics
   enum {
     FREQ_STAT_AVG=2, FREQ_STAT_SUM=4, FREQ_STAT_CURR=4,
     FREQ_STAT_MAX=8, FREQ_STAT_MIN=16, FREQ_STAT_STDV=32,
     CUMM_STAT_AVG=64, CUMM_STAT_SUM=128, CUMM_STAT_MAX=128,
     CUMM_STAT_MIN=256, CUMM_STAT_STDV=512};

protected:
   double start_time, end_time; //start&end-time for stat collection
   unsigned int stat_param_flag; //OR of user-provided boolean flags
   unsigned int cache_size; // cache size for dumping data
```

```
    unsigned long no_samples; // number of statistical samples


    double avg_time,        // average time for averaging time values
           sum,             // sum
           cummulative_sum; // cummulative sum within time_window
    unsigned int time_window; // time_window for cummulative stats

    //constructor (user-defined cache_size & time_window)
    base_stat(unsigned int_cache_size, unsigned int _time_window);

    // destructor of base class
    ~base_stat();

    char *activate_stats(unsigned int _flag);//Ors flags for file&GUI
    char *string_gen_tbl[max_stat_params]; //table with filenames
    stat_entry *cache_table; // table (periodically flushed to file)
    int cache_table_ptr; // pointer to cache_table for next entry

  private:
     static unsigned int filename_counter; // unique file identity
    static unsigned int counter_id; // count number of instances
    unsigned int instance_id // id of class instance
    // In the base class we may also define:
    // mutex lock; // to enable shared access to variables
    // user-defined log file(s)
    // additional user-defined or possibly customized GUI labels
    // user-defined captions identifying IP components
};
```

3.3.1.4 Time-driven and Event-Driven Statistic Classes

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. Both derived classes are fully-accessible by the user and inherit from the base class `base_stat`. Notice that the constructor for the base_stat class is protected, thus the base class cannot be directly instantiated.

In *time-driven simulation*, signals usually have instantaneous values. that must be recorded during simulation. During simulation, these values are recorded by calling a library-provided public member function called `stat_write`.

- `stat_write (double time, double value)`.

In *event-driven simulation*, recorded statistics for events must include an arrival and a departure time. Since the departure time is known at some point later in time, we propose an interface which is based on two public `stat_event_start` and `stat_event_end` functions. Thus, first the user invokes an

- `a = int stat_event_start(double arrival_time)`

command to record the arrival time and save in variable `a` the unique location of the event within the internal table of values. Then, when the event's departure time is known, this is recorded within the internal table of values at the correct location, by invoking a call to

- `void stat_event_end(double departure_time, a)`.

Notice that the event duration can be readily obtained from the above two values. Using Grace™ this computation can be performed very efficiently using the `Edit data sets: create_new (using Formula)` option.

Since complex systems may involve both time-driven (instant) and event-driven (duration) statistics, we provide the `StatInstant` and `StatDuration` classes which inherit properties from the top-level classes and provide the functionality described above. The interface for these classes is described below. Implementation is slightly complicated due to the `_time_window` parameter. The precise implementation will be explained in the next release of this manual.

```
#define MAX_STRING_SIZE 64
class StatInstant : public base_stat {
  public:
    // constructor for time-driven (instant) stats
   StatInstant (unsigned int _cache_size,
                unsigned int _time_window);

    // class destructor
    ~StatInstant();

    // this function calls base_class::activate_stats
    // for setting parameter flag for user-selected statistics
    // it also appends a user-defined string to the filenames
    FILE* activate_stats(unsigned int _flag,
    const char *_function_name, const char *_user_string);

    // the following statistics update function stores at the
    // next available location in cache_table two values:
    // a) usually the simulation time and
    // b) the corresponding statistical value at this time
    void stat_write(double _time, double _value);

    // next function provides data to the gui or flushes
    // to a file (in cummulative form, if time_window > 1)
    void stat_write_to_gui(double _time, double _value);

    // other gui-specific functions (not implemented yet)
    // void enable_gui_instant();
    // void disable_gui_instant();

private:
    // file pointer used for dumping statistics (ASCII format)
    FILE *fp;
    // file name for printing statistics
    char fname[MAX_STRING_SIZE];
 };
```

```
class StatDuration : public BaseStat {

  public:
    // class constructor for event-driven (duration) statistics
    StatDuration(unsigned int _cache_size,
                 unsigned int _time_window);

    // class destructor
    ~StatDuration();
    // the next function calls base_class::activate_stats for
    // setting parameter flag for user-selected statistics and
    // appends a user-defined string to the filenames created
    FILE* activate_stats(unsigned int _flag,
               const char *_function_name,
               const char *_user_string);

    // the following function records the arrival, i.e. it
    // stores the arrival time in cache_table and returns
    // the corresponding _index within the cache_table
    int stat_event_start(double _time);

    // the following function records the arrival, i.e. it
    // stores the departure time at cache_table[_index]
    // and returns the associated  arrival time (if any)
    double stat_event_end(double _time, int _index);

    // other gui specific functions (not implemented yet)
    // void enable_gui_duration();
    // void disable_gui_duration();

  private:
    // file pointer used for dumping statistics (ASCII format)
    FILE *fp;
    // file name for printing statistics
    char fname[MAX_STRING_SIZE];

    // counter of pending events, i.e. events with recorded
    // arrival time, but unknown yet departure time
    unsigned int no_pending_events;
};
```

Finally, notice that for the `StatDuration` class there is a corresponding internal OCCN library `StatDurationLib` class with the same functionality. However, this class performs the basic `stat_event_start/end` operations using (unknown to the user) internal OCCN library object pointers. Thus, `stat_event_start/end` operations also take into account memory address in order to compute duration for subsequent, e.g. read/write accesses to the same memory address.

Thus, in this case, upon an event arrival, we invoke the command

- `void stat_event_start(long int MemAddr, double arrival_time);`

to record in the next available location within the internal table:

a) the current memory address (`MemAddr`),
b) the arrival time (`arrival_time`), and
c) an undefined (-1) departure time.

Then, upon an event departure, we invoke the command

- `void stat_event_end(long int MemAddr, double departure_time);`

to search for the current `MemAddr` in the internal table of values, and update the entry corresponding to the undefined departure time.

The StatDurationLib class allows for simple duration statistics based on the `enable_stat_delay` OCCN library function for various memory objects providing read/write access, e.g. FIFO objects, Memory, Cache and inter-module communication objects.

3.3.1.5 Statistic Classes for Throughput, Latency, Size, and Cell Loss

There are two very common performance metrics for evaluating system performance.

- Throughput refers to the average number of bytes routed through the component per clock cycle. Sometimes throughput is normalized by the maximum possible throughput.
- Latency refers to the average delay through the component, measured from arrival time of the first message bit to departure time of the last message bit. Latency may include various delays, such as link propagation delay, queuing time, as well as arbitration time.

As described above the application user interface for the statistical functions includes the base class called `base_stat` and `gui_stat`, as well as two inherited public classes: `StatInstant` and `StatDuration`.

```
class StatInstantAvg: public StatInstant {
public:
 StatInstantAvg(unsigned int _cache_size,
                unsigned int _time_window):
                        StatInstant(_cache_size, _time_window),
                        bytes_counter(0.0)
 {};

 void StatInstantAvg::stat_write(double _time, double _value) {
  // case 1: if current time < start time, then do nothing!
  if (_time < start_time) return;
  // keep point only if time > start_time
  if ((_time-start_time) != 0)
    StatInstant::stat_write(_time,
             (bytes_counter += _value)/(_time-start_time));
 };

 private:
    double bytes_counter;
};
```

```
class StatInstantCAvg: public StatInstant { // cummulative avg
public:
  StatInstantCAvg(unsigned int _cache_size,
                  unsigned int _time_window):
                          StatInstant(_cache_size, _time_window),
                          old_cum_sum(0.0),
                          total_no_samples(0)
  {};

 void StatInstantCAvg::stat_write(double _time, double _value) {
   // case 1: if time < start time, then do nothing
   if (_time < start_time) return;

   if (time_window <= 0) { // error: negative time_window
      fprintf(stderr, "Stats error: time_window (%d) is <0 \n",
                                               time_window);
      exit(1);
   }
   else if (time_window > (no_samples +1)) { // collect data
      no_samples++;
      cum_sum = cum_sum + _value;
      avg_time = avg_time + _time;
  } else if (time_window == (no_samples +1)) {
      // collect & output data (time_window finished)
      //compute and send to gui average time and value
      no_samples++;
      cum_sum = cum_sum + _value;
      avg_time = avg_time + _time;
      total_no_samples++;
      old_cum_sum = ((total_no_samples -1) * old_cum_sum +
                      cum_sum/ no_samples)/total_no_samples;
      stat_write_to_gui(avg_time/ no_samples, old_cum_sum);
      no_samples = 0;
      cum_sum = 0;
      avg_time = 0;
    } else { // errors: (possible case listed)
      fprintf(stderr, "Stats error, e.g. no_samples (%lu)",
              "> time_window (%d) \n", no_samples, time_window);
      exit(1);
    }
  };

 private :
    double old_cum_sum;
    unsigned int total_no_samples;
};
```

Using derived classes from `StatInstant` and the `StatDuration` class that represent time- and event-driven statistics, we can derive the application-specific classes *instant throughput* (cummulative average over only a time window), *cummulative throughput* (cummulative average over many consecutive time windows) and *latency* statistics: `StatInstantput`, `StatThroughput and StatDelay`. For these classes, we provide enable functions in order to initialize parameters, such as the absolute start and end time for statistics collection, the title and legends for the x and y axes, and the time window for window statistics. We also provide a boolean flag for stopping or resuming statistics collection during execution.

```
  class StatInstantThroughput : public StatInstantAvg {
  public:
    StatInstantThroughput(unsigned int _cache_size,
                          unsigned int _time_window)
               : StatInstantAvg(_cache_size, _time_window),
                 STATS_WANTED_INSTANT_THROUGHPUT(0)
    {};
    StatThroughput::~StatThroughput() {};
    void enable_stat_instant_throughput(const double start = 0,
                                        const double end = DBL_MAX,
                                        const long int time_window =1,
                               const char* x_axis = "Simulation Time",
                            const char* y_axis = "Instant Throughput",
                            const char* object_name = "Undefined Name");
    bool STATS_WANTED_INSTANT_THROUGHPUT;
  };
```

```
class StatThroughput : public StatInstantCAvg {

public:
    StatThroughput:: StatThroughput(unsigned int _cache_size,
                                    unsigned int _time_window) :
                           StatInstantCAvg(_cache_size, _time_window),
                           STATS_WANTED_THROUGHPUT(0),
                           bytes_counter(0.0)
    {};
    StatThroughput::~StatThroughput() {};
    void enable_stat_throughput(const double start=0,
                                const double end= OCCN_UINT64_MAX,
                                const long int time_window =1,
                                const char* x_axis = "Simulation Time",
                  const char* y_axis = "Cummulative Window Throughput",
                            const char* object_name = "Undefined Name");
    void StatThroughput::stat_write(double _time, double _value);
    bool STATS_WANTED_THROUGHPUT;

private :
    double bytes_counter; // for computing bytes per sec
};
```

```
  class StatDelay : public StatDuration {
  public:
    StatDelay(unsigned int _cache_size, unsigned int _time_window)
            : StatDuration(_cache_size, _time_window),
              STATS_WANTED_DELAY(0)
    {};

     void enable_stat_delay(const double start=0,
                            const double end= DBL_MAX,
                            const char* x_axis = "Arrival Time",
                            const char* y_axis = "Departure Time",
                            const char* object_name = "Undefined Name");
     bool STATS_WANTED_DELAY;
};
```

The `StatInstantSize` and `StatSize` classes may be used to provide application statistics for *instant size* (cummulative average over only a time window) and *cummulative size* (cummulative average over many consecutive time windows) of counter-based objects, such as FIFO, LIFO, and circular FIFO objects. Each class provides an enable function for initializing parameters, such as absolute start and end time for statistics collection, title and legends for the x and y axes, and time window for window statistics, and a boolean flag for stopping or resuming statistics collection during execution.

```
 Statclass StatInstantSize : public StatInstant {
 public:
   StatInstantSize(unsigned int _cache_size,unsigned int _time_window
                : StatInstant(_cache_size,_time_window),
                  STATS_WANTED_INSTANT_SIZE(0)
   {};

   void enable_stat_instant_size(const double start=0,
                                 const double end= DBL_MAX,
                                 const long int time_window =1,
                                 const char* x_axis = "Simulation Time",
                                 const char* y_axis = "Instant Size",
                            const char* object_name = "Undefined Name");

   bool STATS_WANTED_INSTANT_SIZE;
  };
```

```
class StatSize : public StatInstantCAvg {
public:
  StatSize(unsigned int _cache_size, unsigned int _time_window)
         : StatInstantCAvg(_cache_size, _time_window),
           STATS_WANTED_SIZE(0)
  {};

  void enable_stat_size(const double start=0,
                        const double end= DBL_MAX,
                        const long int time_window =1,
                        const char* x_axis = "Simulation Time",
                      const char* y_axis = "Cummulative Window Size",
                      const char* object_name = "Undefined Name");

  bool STATS_WANTED_SIZE;
};
```

The `StatInstantProb` and `StatProb` provide instant ratio (cummulative average over only a time window) and cummulative ratio (cummulative average over many consecutive time windows) of counter-based objects, with a binary (either 0 or 1) instantaneous counter value, as is common with hit ratio and cell loss probability application statistics. Both statistics classes define a corresponding enable function and a boolean flag used as described before.

```
class StatInstantProb : public StatInstant {
public:

  StatInstantProb(unsigned int _cache_size,unsigned int _time_window)
                : StatInstant(_cache_size,_time_window),
                  STATS_WANTED_INSTANT_PROB(0)
  {};

  void enable_stat_instant_prob(const double start=0,
                                const double end= DBL_MAX,
                                const long int time_window =1,
                              const char* x_axis = "Simulation Time",
                        const char* y_axis = "Instant Probability",
                        const char* object_name = "Undefined Name");

  bool STATS_WANTED_INSTANT_PROB;
};
```

```
class StatProb : public StatInstantCAvg {
public:
  StatProb(unsigned int _cache_size, unsigned int _time_window)
       : StatInstantCAvg(_cache_size, _time_window),
         STATS_WANTED_PROB(0)
  {};

  void enable_stat_prob(const double start=0,
                        const double end= DBL_MAX,
                        const long int time_window =1,
                        const char* x_axis = "Simulation Time",
              const char* y_axis = "Cummulative Window Probability",
                        const char* object_name = "Undefined Name");
  bool STATS_WANTED_PROB;
};
```

The use of the above classes is described in detail below.

At first, let us define a statistical object for measuring throughput during read access from a memory module called "stat_memory".

```
  StatThroughput ThroughputRead(1, 1); // cache size 1, time window 1
```

Then, an appropriate enable routine is called to initialize simulation and graphics parameters, such as the absolute start and end time for statistics collection, time window for window statistics, title and legends for the x and y axes. Furthermore, within this enable routine, the boolean flag (STATS_WANTED_THROUGHPUT) for statistics collection during execution is set to true, and also time_window may be overriden (see below).

```
  ThroughputRead.enable_stat_throughput(0, 1000, 2, // time window = 2
 "Simulation Time","Average Throughput for Read Access", "stat_memory");
```

An example of the use of the object ThroughputRead for statistics collection is shown below. The values for current time and number of bytes read from the memory module are recorded with the stat_write function.

```
  if (ThroughputRead.STATS_WANTED_THROUGHPUT)
    ThroughputRead.stat_write((N_uint64)
              (sc_time_stamp().to_default_time_units()), no_bytes_read);
```

A statistical object for measuring latency during successive write/read accesses from a memory module called "stat_memory" may be defined as follows.

```
 StatDelay WriteReadDelay(1, 1); // cache size 1, time window 1
     // equivalent representation: StatDuration stat_memory(1000,1);
```

Then, an appropriate enable routine is called to initialize simulation and graphics parameters, such as the absolute start and end time for statistics collection, time window for window statistics, title and legends for the x and y axes. Furthermore, within this enable routine, the boolean flag (STATS_WANTED_DELAY) for statistics collection during execution is set to true.

```
  WriteReadDelay.enable_stat_delay(0, 1000, "Arrival Time",
                                    "Departure Time", "stat_memory");
```

An example of the use of the object WriteReadDelay for statistics collection is shown below. Notice the use of the stat_event_start and stat_event_end functions in order to save the current time (and obtain the id) or current time and number of bytes read from the memory module are recorded with the stat_write function.

```
  // step 1: Calling stat_event_start
  int id;
  if (WriteReadDelay.STATS_WANTED_DELAY)
    id = WriteReadDelay.stat_event_start(
                (N_uint64)(sc_time_stamp().to_default_time_units()));
  // step 2: Calling stat_event_end
  if (WriteReadDelay.STATS_WANTED_DELAY)
    (void) WriteReadDelay.stat_event_end(
            (N_uint64)(sc_time_stamp().to_default_time_units()), id);
```

For other statistics classes, definition and use of the statistics is similar to the above two cases. For more details on the use of all statistics classes and the capability of online statistical graphs, the reader is referred to the corresponding statistical testbenches that accompany the OCCN library. However, notice that for the StatInstant (and

StatDuration) classes there are no enable functions, and instead the following base functions are used.

```
  StatInstant s1(1000,1);
  // similar for StatDuration
  s1.activate_stats(BaseStat::FREQ_STAT_CURR,"str1","uniq_number");
  s1.start_sim_time(0);
  s1.end_sim_time(100);
  // title and legends are defined in the appplication-dependent
  // online visualization function (see statistic testbenches)
```

In addition to the previously described classes which cover all basic cases, it is sometimes necessary to combine statistical data from different IP components, e.g. from two different memory units in order to compare average read vs. write access times. For this reason, we need new *joint or merged statistic classes* that inherit from the Instant and Duration classes. Parameters, e.g. necessary boolean flags, for these joint statistic classes will be examined in a future issue of this document.

Furthermore, an extension to an asynchronous environment would require introducing a new *asynchronous statistics class* with the necessary abstract data types to support waves, concurrency map data structures and system snapshots.

Finally, within the parallel and distributed domain, it is possible that simulation metrics must be combined together with platform performance indicators which focus on monitoring system statistics (e.g. simulation speed, computation and communication load) for improving system performance, e.g. through dynamic load balancing.

3.3.2 Advanced Monitoring Features

Conventional text output and source-code debugging are inadequate for monitoring and debugging complex and inherently parallel SoC models. Similarly, current tools, such as the Synopsys Cocentric Studio can generate vcd files for signal tracing, or build relational databases in the form of tables, for data recording, visualization, and analysis. However, extensive SoC performance-modeling environments may be based on advanced monitoring issues.

In this Section, we consider *monitoring activities* appropriate for system level performance modeling. Although these activities may correspond to distinct monitoring phases occurring in time sequence, potentially there is partial overlap between them.

- *Generation* refers to detecting events, and providing event and status reports containing traces (or histories) of system activity.
- *Processing* refers to functionality that deals with monitoring information, such as merging of traces, validation, filtering, analysis, correlation, and model updating. These functions mainly focus on converting raw and low-level monitoring data to the required format and level of detail.
- *Dissemination* concerns the distribution of selected monitoring reports to application users, managers, and processing entities.
- *Presentation* refers to processing and displaying monitoring information to the users in an appropriate form.

In addition to the above four main activities, implementation issues relating to intrusiveness and synchronization constitute a fifth dimension that is crucial to the design and evaluation of monitoring activities.

3.3.2.1 Generation of Monitoring Traces

A *status report* contains a subset of system state information, including object properties, such as time stamp and identity. This report can be generated either periodically, i.e. based on a predetermined FSM or Thread schedule, or on demand, i.e. upon receiving a request for solicited reporting. Notice that the request may itself be periodic, i.e. via polling, or on a random basis. Thus, appropriate status reporting criteria define the reporting scheme, the sampling rate, and the contents of each report.

Detection of event occurrences may be immediate, i.e. detected upon occurrence, or delayed. For example, signals on an internal bus of a node may be monitored in real-time, while alternatively, status reports may be generated, stored and used to detect events at some later time. Event detection may also be internal to the object, i.e. typically performed as a function of the object itself, or external, i.e. performed by an external agent who receives status reports and detects changes in the state of the object. Once the occurrence of an event is detected, an *event report* is generated. In general, an event report contains a variable number of attributes such as the event identifier, type, priority, time of occurrence, the state of the object immediately before and after event occurrence, and other application-specific state variables, time stamps, text messages, and possibly pointers to other detailed information.

In order to describe the dynamic behavior of an object or group of objects over a period of time, status and event reports are recorded in time order, as *monitoring traces*. A complete monitoring trace contains all monitoring reports generated by the system since the start of the monitoring session, while a segmented trace is a sequence of reports collected during a limited period of time. A trace may be segmented due to overflow of a trace buffer, or deliberate halting of trace generation that results in the loss, or absence of reports over a period of time.

Notice that within each trace we need to identify the reporting entity, the monitored object, and the type of the report, including user-defined parameters, e.g. start- and end-time, time-window, priority, and size. Thus, user-selected parameters may provide browsing or querying facilities (by name or content), dynamic adjustment (during runtime) of event occurrence interval, adjustment of intervals before or between event occurrences, and examination of the order of event occurrence. A monitoring trace may also be used to generate non-independent traces based on various logical views of objects or system activity.

3.3.2.2 Processing of Monitoring Information

A SoC model may generate large amounts of monitoring information. The data collection phase is only useful, if the data can be used to identify problems and provide corrective measures. Thus, the data collection phase is split into four different phases.

Validation of monitoring information provides validity checks, possibly specified in a formal language, ensuring that the system has been monitored correctly. This includes

- *sanity tests* based on the validity of individual monitoring traces, e.g. by checking for correct token values in event fields, such as an identity, or time stamp, and
- *validation of monitoring reports against each other*, e.g. by checking against known system properties, including temporal ordering.

Filtering minimizes the amount of monitoring data to a suitable level of detail and rate. For example, filter mechanisms reduce the complexity of displayed process communications by

- incorporating a variable report structure,
- displaying processes down to a specified level in the module hierarchy,
- masking communication signals and data using filter dialogs, and
- providing advanced filter functionality for displaying only tokens with predetermined values.

A*nalysi*s processes monitoring traces based on special user-selected criteria. Since analysis is application-dependent, it relies on sophisticated stochastic models involving combinatorics, probability theory, and Markov chain theory. *Analysi*s techniques enable

- simple data collection, such as computing average, maxima, and variance values of particular state variables,
- statistical trend analysis for forecasting using data analysis packages, such as Maple, Grace, SAS, or S from AT & T may be used, and
- correlation of event reports, e.g. for identifying system bottlenecks.

Correlation of monitoring information (also called merging, combination or composition) raises the abstraction level. Together with filtering, it prevents the users from being overwhelmed by an immense amount of detailed information. Thus, for example, events generated by sensors or probes may be combined using `and`, `or`, and `not` operators to provide appropriate high-level reliability metrics. Since, correlation of SoC monitoring information is a very challenging task, a relational database, such as mini-SQL, that includes selection, projection, and join operators is sometimes useful.

3.3.2.3 Dissemination of Monitoring Information

Monitoring reports would have to reach human users, manager, and processing entities. Dissemination schemes range from very simple and fixed, to very complex and specialized. Selection criteria contained within the subscription request are used by the dissemination system to determine which reports (and with what contents, format, and frequency) should be delivered. This requires implicit filtering, since only the requested reports are forwarded.

3.3.2.4 Presentation of Monitoring Information

Various visualization techniques may be provided by a user friendly monitoring interface.

Textual representation involves presentation of monitoring information in text form. This technique increases its expressive power, by providing appropriate indentation, color and highlighting to distinguish monitoring information at different abstraction levels. Events may be displayed in a causal rather than temporal order by including parameters, such as

- event type,

- name of process initiating the event,
- name of process(es) handling the event, and
- contents of transmitted messages.

A *time-process diagram* is a 2D diagram illustrating the current system state and the sequence of events leading to that state. The horizontal axis represents events corresponding to various processes, while the vertical one represents time. In synchronous systems, the unit of time corresponds to a period of actual time, while in asynchronous systems, it corresponds to an occurrence of an event. In the latter case, the diagram is called *concurrency map*, with time dependencies between events shown as arrows. An important advantage of time-process diagrams is that monitoring information can be presented either on a simple text screen, or on a graphical one.

An *animation* captures a snapshot of the current system state. Both textual and graphical event representations, e.g. input, output, and processing events, can be arranged in a 2D display window. Graphical representations use various formats, such as icons, boxes, Kiviat diagrams, histograms, bar charts, dials, X-Y plots, matrix views, curves, pie charts, and performance meters. Subsequent changes in the display occur in single step or continuous fashion and provide an animated view of system evolution; for online animation, the effective rates at which monitoring information is produced and presented to the display must be matched. For each abstraction level, animation parameters include enable/disable event monitoring or visualization, select clock precision, monitoring interval, or level of detail, and view or print statistics.

3.3.2.5 Other Important Monitoring Design Issues

Although the above monitoring techniques are useful, specifications for the actual graphical user interface of the monitoring tool depend on the imagination, experience, and available time of the designer. We present here some desirable features that a SoC monitoring interface should possess.

- *Visualization at different abstraction levels* enables the user to observe behavior at various abstraction levels, e.g. per object, per block, or per system. This stepwise refinement technique allows the user to start observation at a coarse level and progressively (or simultaneously) focus on lower levels. At each level, application and system metrics may be presented in appropriate easy-to-read charts and graphs, while the communication volume among subsystems (or power consumption) may be visualized by adjusting the width (or color) of the lines interconnecting the corresponding modules.
- A *history function* visualizes inherent system parallelism by permitting the user to
  - ➢ scroll the display of events forwards or backwards in time by effectively changing a simulated system clock, and
  - ➢ control the speed at which system behavior is observed using special functions for start/stop and pause/restart event display, single-step or continuous animation, and real-time or slow-motion animation.
- *Visibility of interactions* enables the user to dynamically visualize contents of a particular communication message, component data structure, system or network configuration, or general system data, such as log files, or filtering results.
- *Placement of monitoring information* greatly enhances visibility and aids human comprehension. Placement may either be automatic, i.e. using computational geometry algorithms, or manual by providing interface functions, e.g. for moving or

resizing boxes representing monitoring objects or system events, or moving coupled links representing process communication.

- *Multiple views* use multiple windows representing system activities from various points of view, thus providing a more comprehensive picture of system behavior.
- *Scalable monitoring* focuses on efficient computation and communication practices. Scalability refers to the ability of monitoring large-scale models, with tens or hundreds of thousands of objects as in telecommunication networks. Large-scale monitoring implementations could benefit from POSIX- or System V message queues, or non-blocking and blocking concurrent queues [33, 40] that achieve high degree of concurrency at low implementation cost compared to other general methods [44, 55].

In addition, the monitoring system designer should focus on the problems of intrusiveness and synchronization occurring in distributed system design.

- Intrusiveness refers to the effect that monitoring may have on the monitored system due to sharing common resources, e.g. processing power, communication channels, and storage space. Intrusive monitors, may lead not only to system performance degradation, e.g. due to increased memory access, but also to incorrect results due to event reordering which may in turn lead to data races and possible system deadlocks, e.g. when evaluating symmetric conditions which result in globally inconsistent actions.
- Distributed systems are more difficult to monitor than centralized systems because of parallelism among processes or processors, random and non-negligible communication delays, possible process failures, and unavailable global synchronized time. These features cause various interleavings of monitoring events which might result in different results from repeated runs of the same (deterministic) distributed algorithm, or different views of execution events from various objects of the execution events in the same distributed algorithm [11, 12, 23, 39]. Notice however, that these problems do not occur with SystemC scheduling, since current versions of the kernel are sequential, offering only simulated parallelism and limited event interleaving.

4. Case-Study: OCCN Communication Refinement

In this Section, we provide a producer/consumer and a simple, transport layer, inter-module data transfer protocol. For the transport layer model, after explaining 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 OCCN high-level system performance modeling. An example illustrating high-level system power modeling is under development.

4.1 The Producer/Consumer

The following example illustrates the methods discussed for `MasterPort` and `SlavePort` interfaces. It consists of a synchronous transmitter and receiver. This example counts the number of words in the standard input, separated by space, tab, and newline characters. The application is implemented using two threads. Both threads use the `new` function provided by the port to avoid making copies of messages between thread and channel.

- The *producer thread* generates a finite number of input characters and sends them through a channel to the consumer thread.
- Concurrently, the *consumer thread* takes characters out of the channel and counts the number of words until the input steam is exhausted.

The code for the transmitting and receiving thread are provided below.

producer.h

```
#include “occn.h” // include message definitions

class producer : public sc_module {
  public:
    producer(sc_module_name name);
    MasterPort<Pdu<char> > out; // communication port
    SC_HAS_PROCESS(producer);

  private:
     void read();
};
```

producer.cc

```
#include “producer.h”

producer::producer(sc_module_name name) : sc_module(name)
{SC_THREAD(read);}

void producer::read() {
  char c;
  Pdu<char>* msg;
  msg = new(&out) Pdu <char>; // channel allocation
  if (! msg)
    OCCN_end_sim(-1,"Error during Packet allocation");
  while (cin.get(c)) { // producer sends c
       *msg= c;
       out.send(msg);
  }
  delete msg;
} // after the send msg is not usable
```

consumer.h

```
#include "occn.h"
class consumer : public sc_module { // class description
  public:
    consumer(sc_module_name name);
    SlavePort<Pdu<char> > in; // communication port
    SC_HAS_PROCESS(consumer);

  private:
     void count();
     int counter;
};
```

consumer.cc

```
#include "consumer.h"
consumer::consumer(sc_module_name name) : sc_module(name),
                                          counter(0)
{SC_THREAD(count);}
void consumer::count() {
    bool word_complete=false;
    Pdu<char> * msg;
    while (1) {
        msg = *in.receive(); // receive will allocate the storage
        //cout << "consumer receives " << msg->body << endl;
        switch (msg) {
            case  ' ' :
            case '\t' :
              if (word_complete) {
                counter++;
                word_complete = false;
              }
              break;
            case '\n' :
              if (word_complete) counter++;
              cout << "Total: "<< counter << " words" << endl;
              OCCN_end_sim(0,"SUCCESS");
              break;
            default:
              word_complete = true;
              break;
        }
        in.reply();
    }
```

Notice that since the Pdu `msg` belongs to the transmitter, the `send` function makes a local copy before sending the Pdu through the channel. Since local copies are a drawback in terms of simulation speed, we let the channel manage creation and destruction of Pdus using the placement feature available in C++ for the standard `new` function. Therefore we provide an overload for the operator `new` that takes as second argument a MasterPort, using the function `void* operator new(size_t nbytes, MasterPort<…>& port)`.

This function allocates runtime storage returning a pointer to the newly allocated Pdu. Since in most cases, one caller manages transmission, the `new` function supports single-element dynamic allocation for efficient implementation. Thus, for allocating packet memory within a channel bound to a port called `out`, we call the operator new.

```
 Pdu<…> * pk = new(&out) Pdu<…>;
```

Furthermore, notice that the compiler calls the overloaded operator new with parameters `sizeof(Pdu<…>)` and `out`.

Since Pdus belong to the caller, their destruction is under the responsibility of the caller. Thus, the transmitter must destroy the `msg` using delete However, in order in order to deallocate a previously allocated `Pdu<…>` object within the channel, the operator `delete` is called automatically by the channel before the context switch of the current task.

4.1.1 Top-Level Module Instantiation and Port Binding

The standard channel interface (described in “`StdChannel.h`”) is the most basic channel for point to point, bi-directional inter-module communication. This channel essentially models circuit switching, since once a module prepares its signals and/or data, this information always becomes available to the connected module during the next clock cycle. The user currently has no control to change the number of cycles required for the transfer. Furthermore, there is **no** maximum data size on the Pdu to be transmitted through this channel.

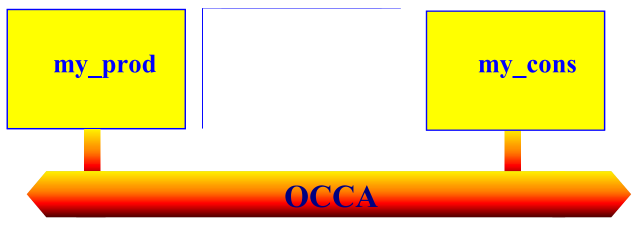

Figure 16. IP module components: behavior and inter-module communication

Here is an example of top-level module instantiation and port binding with the `StdChannel`. The binding represents the architecture shown in Figure 16.

main.cc

```
#include “occn.h”
#include "producer.h"
#include "consumer.h"
int main() {
    // clock, modules, and bus declaration
    sc_clock clock1("clk",10,SC_NS);
    producer my_prod("Master");
    consumer my_cons("Slave");

    // channel instantiation
    StdChannel<Pdu<char>,Pdu<char> > channel("channel");

    // module binding
    channel.clk(clock1);

    // bus configuration (operations dependent on the model)
    my_prod.out(channel);
    my_cons.in(channel);
    sc_start(-1);
    return -1;
}
```

4.2 A Transport Layer Inter-Module Data Transfer Protocol

This Section provides a case study for 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 11). 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);` The destination address `addr` identifies the target receiver and the `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.

NoC is becoming sensitive to noise due to technology scaling towards deep submicron dimensions [5]; Thus, we assume an unreliable channel and a simple stop-and-wait data link protocol with negative acknowledgments, called Automatic Repeat reQuest (ARQ), is implemented [31, 54]. Using this protocol, each frame transmitted by the sender (I-frame) is acknowledged by the receiver with a separate 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 [28];
- 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 [59], and
- a `source_id` identifying the transmitter for routing back an acknowledgment.

The ACK-frame sent by the receiver consists 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 through a special error code.

From the transmitter (Tx) side, the ARQ 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 ARQ 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 occurs during the acknowledgment exchanges. EDC is provided for a future more complex implementation.

4.2.1 Implementation using the OCCN StdChannel

Figure 17 illustrates OCCN implementation of our transport layer 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.

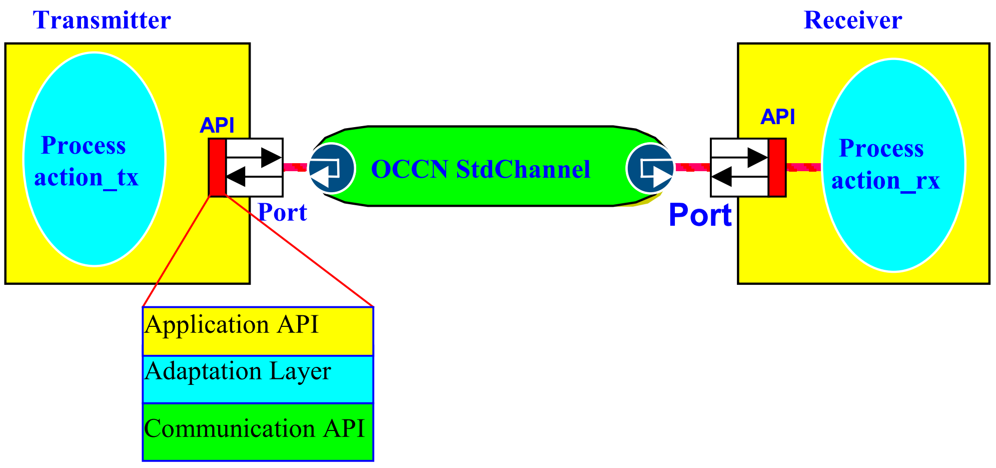

Figure 17. Transport Layer send/receive implementation with OCCN StdChannel

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 18). This SystemC-compliant approach allows design of the communication-oriented part of the application on top of the OCCN Communication API.

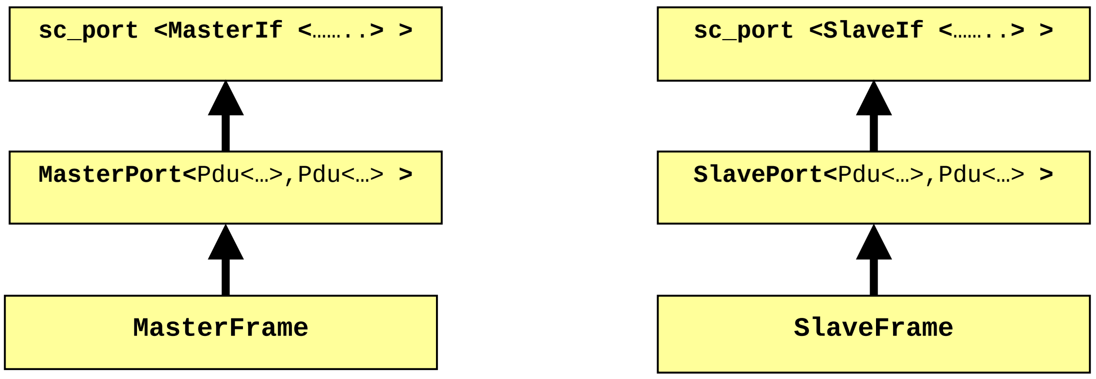
sc_port <MasterIf <…….. > >
MasterPort<Pdu<…>,Pdu<…> >
MasterFrame
sc_port <SlaveIf <…….. > >
SlavePort<Pdu<…>,Pdu<…> >
SlaveFrame

Figure 18. Inheritance of classes and specialized ports.

Comments are introduced within each code block to illustrate 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.

4.2.1.1 The Pdu Definition

With respect to data type definition, we define the buffer and frame data structures using the OCCN Pdu object. 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 19. For in-order transmission the sequence number can be represented with a single bit. However, a progressive number is assigned 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 a more general setting. Moreover, a reserved number (ERROR_CODE) in *ack* indicates an error, denoting among all non-acknowledgement conditions data corruption detected by EDC.

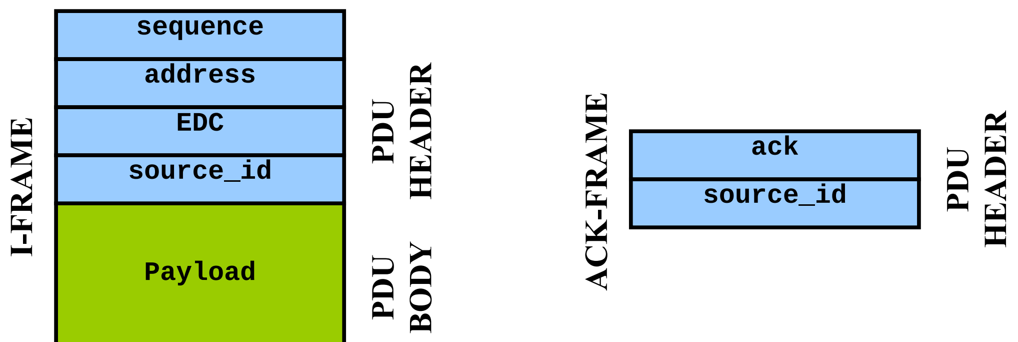

Figure 19. I- and ACK-frame data structures and corresponding OCCN Pdu objects.

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, source_id;
  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 (use dummy body of char)
typedef Pdu<ACK_frame_header, char> ACK_FrameType;
// The PDU for the buffer parameter of TransportLayer_send
typedef Pdu<char, BUFFER_SIZE> BufferType;
```

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 `I_FrameType` and `ACK_FrameType` that are used throughout the example.

4.2.1.2 The Transmitter Module

The Transmitter module implements the `SC_THREAD action_tx`. A buffer filled with random letters in ['A' ... '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 name and interface definitions, defined in "`MasterFrame.h`".
- the thread name and action (`action_tx`) routine, and
- 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);
    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);
  OCCN_end_sim(0,"SUCCESS"); }
```

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 ARQ 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(); }
 } 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;}
```

4.2.1.3 The Receiver Module

The code block “`Receiver.h`” describes the interface of the Receiver module. “`Receiver.cc`” implements a `SC_THREAD` process, called `action_rx`, which reads the buffer through the `API TransportLayer_receive` and prints it out. The channel is accessed through the `SlaveFrame` *`sap_rx`* specialized port.

Receiver.h

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

class receiver :  public sc_module {
public:
    receiver(sc_module_name name);
    ~receiver();
    SlaveFrame sap_rx;
    SC_HAS_PROCESS(receiver);
private:
     void action_rx();
     BufferType* buffer rx; };
```

Receiver.cc

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

// Receiver constructor and destructor
receiver::receiver(sc_module_name name)
        : sc_module(name),
          sap_rx()
{ SC_THREAD(action_rx);
  buffer_rx = new BufferType; }

receiver::~receiver()
{delete buffer_rx;}

// Receiver SC_THREAD process action_rx
void receiver::action_rx() {
    uint count = 0;
    do {
        buffer_rx = sap_rx.TransportLayer_receive();
        // Report
        cout << sc_time_stamp() << " : body of buffer_rx #" <<count;
        cout << " received with TL_receive" << endl << *buffer_rx;
        count++;
    } while(1);
}
```

The `TransportLayer_receive` API is a method of the `SlaveFrame` class; from a functional point of view it handles two main tasks:

- the reception of a I-frame according to the described ARQ protocol;
- the assembly of the valid frame payloads into the buffer.

The method that checks for errors by the EDC simply decides randomly on the correctness of the frame.

SlaveFrame.h

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

class SlaveFrame : public SlavePort<ACK_FrameType,I_FrameType> {
public:
    SlaveFrame ();
    ~SlaveFrame ();
    BufferType* TransportLayer_receive();
private:
    bool EDC_function_rx(I_FrameType frame); };
```

SlaveFrame.cc

```
#include "SlaveFrame.h"
SlaveFrame::SlaveFrame() {}
SlaveFrame::~SlaveFrame() {}

BufferType* SlaveFrame::TransportLayer_receive() {
    BufferType& buffer_rx = *(new BufferType);
    I_FrameType& frame_rx = *(new I_FrameType);
    ACK_FrameType& frame_ack_tx = *(new ACK_FrameType);
    const seq_num total_frames =
                  buffer_rx.sdu_size/frame_rx.sdu_size;
    Random rnd;
    seq_num sequence_expected = 0;
    do {
      frame_rx = *receive();
      if (EDC_function_rx(frame_rx)){ // frame_rx is not corrupted
      if (occn_hdr(frame_rx, sequence)==sequence_expected) {
        buffer_rx << frame_rx;
        sequence_expected++; }
      occn_hdr(frame_ack_tx, ack) = occn_hdr(frame_rx, sequence);
      occn_hdr(frame_ack_tx,source_id)=occn_hdr(frame_rx,source_id);
      } else { // frame_rx is corrupted
      occn_hdr(frame_ack_tx, ack) = ERROR_CODE; }
      reply();
      wait(rnd.integer(30),SC_NS); // random latency of slave
      send(&frame_ack_tx);
    } while (sequence_expected < total_frames);
    return &buffer_rx;
}

bool SlaveFrame::EDC_function_rx(I_FrameType frame) {// Error Model
  Random rnd;
  if (rnd.integer(1000)>995) return false; // Error detected
  else return true; } // No errors detected
```

4.2.1.4 The Top Level main.cc

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.

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 ARQ 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;
}
```

4.2.2 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

4.2.2.1 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.

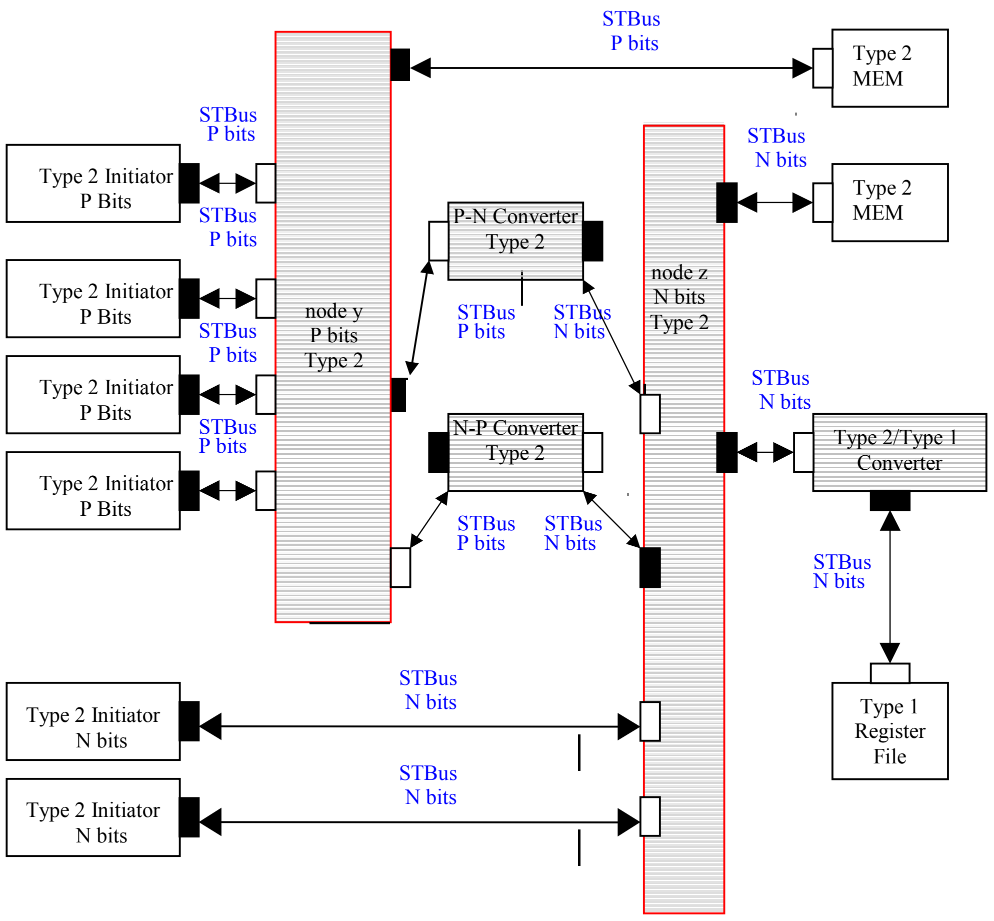

Figure 20. The StBus NoC, illustrating initiators and target components (shown as white boxes)

As shown in Figure 20, 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 [15, 51, 52].

4.2.2.2 Communication refinement with STBus

Figure 21 illustrates communication refinement, if the proprietary STBus NoC, described in Section 4.3.1, is used instead of the generic OCCN StdChannel.

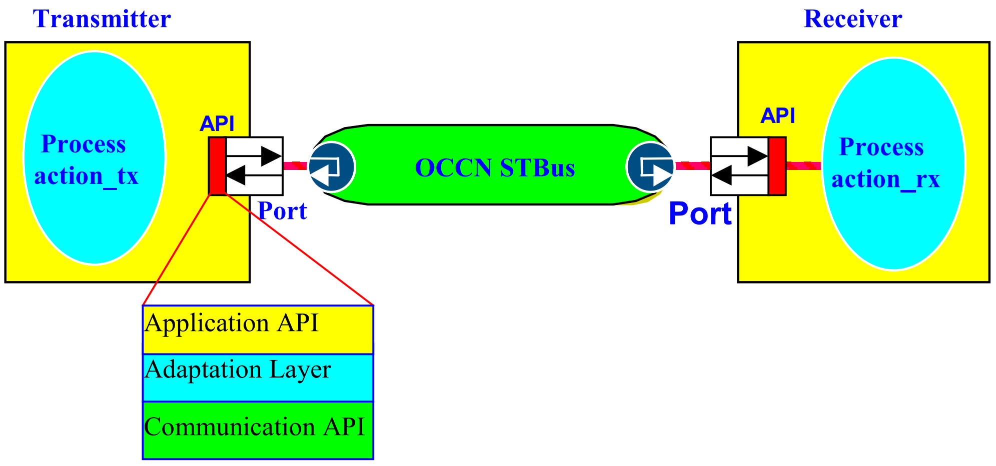

Figure 21. Transport layer protocol with STBus refinement

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 22 shows the simple handshake interface of STBus Type 1 [51, 52]. 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.

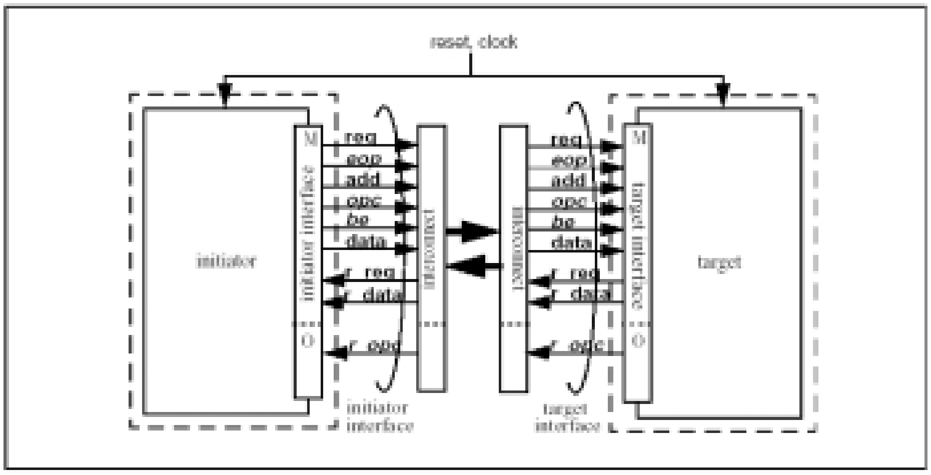

Figure 22. STBus Type 1 initiator and target interfaces

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 22, 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 22.

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 ARQ protocol, i.e. EDC with retransmission of erroneous data, to a real on-chip bus scheme [4, 29]. 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

4.2.3 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(0, 50000, 1, "Simulation Time",
                       "Average Throughput for Write Access", “buffer”);
```

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.

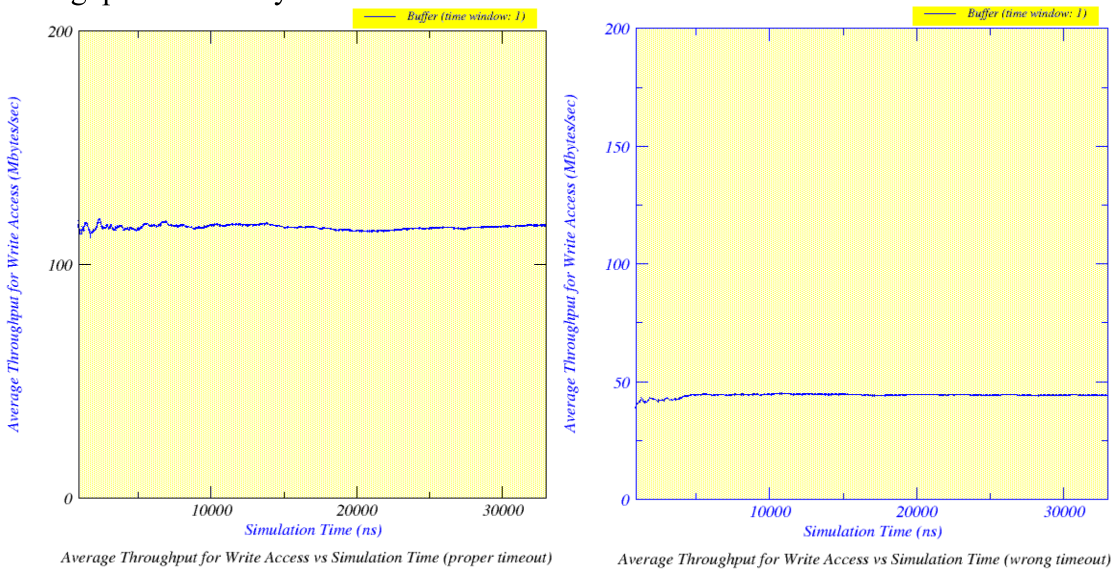

Figure 23. Performance results for transport layer protocol.

Figure 23 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. Observe how graph axes and title are composed from `enable_stat_throughput_read` function parameters. Also notice that units and legends are always provided automatically. Legends list the object name (computed automatically from the constructor) and time_window. Internal figure numbers are always included in the corresponding (*.agr) Grace™ file names; this helps in organizing multiple graphs when preparing a large document for desktop publishing.

Similarly, for obtaining delay statistics, we make the following function call:

```
enable_stat_delay(0,50000,"Arrival Time","Departure Time", “stat_box”);
```

Notice that delay statistics are only meaningful for objects which perform consecutive write and read operations; these objects essentially serve as transit points: they first accept information, and then, after a queuing delay, they propagate information. Thus, for delay statistics, a read access should always occur before the matching write access!

4.2.4 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 a generic, reusable, robust, and bug-free C++ statistical library for exploring system-level performance modeling. In addition, we have discussed statistical library extensions, including advanced monitoring features, such as generation, processing, dissemination and presentation.

Low power consumption is one of the crucial factors determining the success of modern, portable electronic systems. Reduction of chip overheating, which negatively affects circuit reliability, are the main factors driving massive investments in power conscious design solutions. New methodologies try to address power optimization from the early stages of the system design, leveraging the degrees of freedom available during architectural conception and hardware/software partitioning of the system. Thus, correlated power estimation is performed by using statistics on the bus resource's usage, e.g. by collecting data on the number and size of packets sent and received, and keeping track of the average bus utilization rate. One may use this information together with the bus topology to figure out the power and energy cost of the system, either in form of analytic equations or as lookup tables. Power consumption may be modeled at the Functional level, Transaction level (TLM), Bus Cycle Accurate level (BCA), or All Cycle Accurate (ACA) level; ACA provides cycle- and pin-accurate level description of all system modules. ST Microelectronics is currently developing a wide range of power macro-modeling for STBus, and AMBA AHB and APB buses. These models would eventually be hooked for SystemC simulation.

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. Chandy, K. M. and Lamport, L. "Distributed snapshots: determining global states of distributed systems". *ACM Trans. Comp. Syst.*, 3 (1), 1985, pp. 63-75.
12. Christian, F. "Probabilistic clock synchronization". *Distr. Comput.*, 3, 1989, pp. 146-158.
13. Cierto virtual component co-design (VCC), Cadence Design Systems, see http://www.cadence.com/articles/vcc.html
14. 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.
15. 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)
16. Coppola, M., Curaba, S., Grammatikakis, M and Maruccia, G. "ST IPSim reference manual", internal document, ST Microelectronics, September 2002.
17. Coppola, M., Curaba, S., Grammatikakis, M. and Maruccia, G. "ST IPSim user manual", internal document, ST Microelectronics, September 2002.
18. 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.
19. Diep, T.A., and Shen J.R. "A visual-based microarchitecture testbench", *IEEE Computer*, 28(12), 1995, pp. 57—64.
20. 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.
21. De Bernardinis, F., Serge, M. and Lavagno, L. "Developing a methodology for protocol design ". *Research Report SRC DC324-028*, Cadence Berkeley Labs, 1998.
22. Ferrari, A. and Sangiovanni-Vincentelli, A. "System design: traditional concepts and new paradigms". *Proc. Conf. Computer De*sign, 1999, pp. 2—13.
23. Fidge C. J.. "Partial orders for parallel debugging"*, Proc. ACM Workshop Parallel Distr. Debug.,* 1988, pp. 183-194.
24. Forsell, M. "A scalable high-performance computing solution for networks on chips", *IEEE Micro*, 22 (5), pp. 46—55, 2002.
25. 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
26. 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.
27. 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.
28. Grammatikakis, M.D., Hsu, D. F. Hsu and Kraetzl, M. "*Parallel System Interconnections and Communications*", CRC press, 2000, ISBN: 0-849-33153-6,
29. 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.
30. 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

31. Holzmann, G. J. “Design and validation of computer protocols”. Prentice-Hall International Editions, 1991
32. Klindworth, A. “VHDL model for an SRAM”. Report, CS Dept, Uni-Hamburg. See http://tech-www.informatik.uni-hamburg.de/vhdl/models/sram/sram.html
33. Hunt G. C., Michael, M., S. Parthasarathy and Scott, M.L.. “An efficient algorithm for concurrent priority queue heaps”. *Inf. Proc. Letters*, 60 (3), 1996, pp. 151-157.
34. 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.
35. “IBM On-chip CoreConnect Bus”. Available from http://www.chips.ibm.com/products/coreconnect
36. Lahiri, K., Raghunathan, A. and Dey, S. “Evaluation of the traffic performance characteristics of SoC Communication Architectures”, Proc. Conf. VLSI Design, Jan. 2001.
37. 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*.
38. 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.
39. Lamport, L. “Time, clocks and the ordering of events in distributed systems”. *Comm. ACM*, 21 (7), 1978, pp. 558-564.
40. Michael, M. and Scott, M.L.. “Simple, fast and practical non-blocking and blocking concurrent queue algorithms”. *Proc. ACM Symp. Princ. Distrib. Comput.*, 1996 , pp. 267-275.
41. Networks on Chip, Eds. Jantsch, A. and Tenhunen, H. .Kluwer Academic Publisher, 2003, ISBN: 1-4020-7392-5.
42. Nussbaum, D., and Agarwal, A. “Scalability of parallel machines”. *Comm. ACM*, 34(3), pp. 56--61, 1991.
43. 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.
44. Prakash, S., Yann-Hang, L. and Johnson, T. “A non-blocking algorithm for shared queues using compare-and-swap”. *IEEE Trans. Comput.*, C-43 (5), 1994, pp. 548-559.
45. Poursepanj, A. “The PowerPC performance modeling methodology”. *Comm. ACM*, 37(6), 1994, pp 47—55.
46. Raw Architecure Workstation. Available from http://www.cag.lcs.mit.edu/raw
47. Rowson, J.A. and Sangiovanni-Vincentelli, A.L. “Interface-based design”. *Proc. Design Automation Conf.* 1997, pp. 178–183.
48. 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.
49. Selic, B, Gullekson, G., and Ward P.T. “Real-time object-oriented modeling”, J. Wiley & Sons, NY, 1994.
50. Sgroi, M. Sheets, M. Mihal, A. et al. "Addressing system-on-a-chip interconnect woes through communication-based design". *Proc. Design Automation Conf.*, 2001.
51. Scandurra A., Falconeri, G., Jego, B., “STBus communication system: concepts and definitions”, internal document, ST Microelectronics, 2002.
52. Scandurra A., “STBus communication system: architecture specification”, internal document, ST Microelectronics, 2002.
53. SystemC, http://www.systemc.org
54. Tanenbaum, A. “Computer networks”. Prentice-Hall, Englewood Cliffs, NJ, 1999.
55. Turek, J., Shasha, D. and Prakash, S. “Locking without blocking: making lock-based concurrent data structure algorithms nonblocking”. *Proc. ACM Symp. Princ. Database Syst.,* 1992, pp. 212-222.
56. Turner, J. and Yamanaka, N. "Architectural choices in large scale ATM switches," IEICE Trans. on Communications, vol. E-81B, Feb. 1998.
57. Verkest, D., Kunkel, J. and Schirrmeister, F. “System level design using C++”. *Proc. Design, Automation & Test in Europe Conf.*, 2000, pp. 74—83.
58. VSI Alliance, http://www.vsi.org/
59. Wicker, S. “*Error control systems for digital communication and storage*”, Englewood Cliffs, Prentice Hall, 1995.
60. 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.
61. Zhang, T., Chakrabarty, K., Fair, R.B. “Integrated hierarchical design of microelectrofluidic systems using SystemC”. *Microelectronics J.*, 33, 2002, pp. 459—470.