IPSIM: SystemC 3.0 Enhancements for Communication Refinement

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

ABSTRACT

Refinement is a key methodology for SoC design. The proposed IPSIM design environment, based on a C++ modeling library developed on top of SystemC 3.0, supports an object-oriented design methodology, separates IP modules into behavior and communication components and further establishes two inter-module communication layers. The Message Box layer includes generic and system-specific communication, while the driver layer implements higher level user-defined communications as illustrated in a design example.

1. Introduction

Due to steady downscaling in CMOS device dimensions, manufacturers are increasing the functionality on a single chip. By 2005 complex systems, called *System-on-Chip* (SoC), will contain hundreds of million transistors. This escalating gate count, desired heterogeneity in terms of hardware and software, and trend towards increased productivity and reduced time-to-market challenge traditional RTL-to-gates design methodology based on time-consuming HDL simulation. Electronic System Level methodology (ESL) focuses on the functionality and relationships of the primary system components, separating system design from implementation. Low level implementation details greatly increase the number of parameters and constraints in the design space, thus, complicating optimal design selection and verification. Similar to near-optimal combinatorial algorithms, e.g. travelling salesman heuristics, ESL models effectively prune away poor design choices and focus on closely examining feasible options. Most consortia address ESL using object-oriented C/C++ libraries. Popular libraries (or complete environments) used with C++ are classified into:

- open source approaches, such as SpecC [6], SystemC [13] and Superlog [12], and
- proprietary licensed tools, such as CynLib by CynApps [4] and N2C by Coware.

The Open SystemC Initiative (OSCI) has developed a collection of C++ classes describing hardware concepts and a simulation kernel implementing runtime semantics. SystemC supports design abstraction at the RTL, behavioral and system level, allows development and exchange of system models and provides seamless tool integration from a variety of vendors. SystemC collaborates with SpecC on synthesis.

Increased demand for accuracy and consistency in hardware modeling for SoC has led to development of STMicroelectronics' IPSIM design environment consisting of a SystemC-based, C++ modeling library, together with a simulation engine, runtime and test environment and communication refinement methodology. IPSIM supports real object-oriented methodology, a plethora of modeling objects compared to SystemC, and a fully documented design flow based on SystemC 3.0. IPSIM users may manually decompose their final design to RTL abstraction level, or use SystemC-based behavioral synthesis tools, such as the Synopsys Co-Centric tools.

In Section 2, we focus on general refinement methodology, also adopted by IPSIM, based on separation of system functionality and communication. This separation enables modeling at various abstraction levels, using appropriate computation and communication models. In Section 3, we introduce IPSIM's public objects, data types, concurrency and performance models. We also show how IPSIM enhances SystemC communication refinement by establishing inter-module communication layers. The *Message Box layer* consists of the IPSIM Message Box object that focuses on generic and system-specific inter-module communication protocols. *The communication driver layer* includes high-level communication routines

Proceedings of the Design,Automation and Test in Europe Conference and Exhibition (DATE'03)
1530-1591/03 $17.00 © 2003 IEEE

invoking the Message Box layer. In Section 4, we also explain IPSIM communication refinement and performance modeling using a transmitter-receiver model. In Section 5, we provide remarks and conclude with a list of References.

2. General SoC Refinement

A *virtual SoC prototype* models SoC by hiding, modifying or omitting system properties. Abstraction levels span multiple levels of accuracy, ranging from functional to transistor model. Each level introduces new model details. In practice, we usually consider the following models.

- *Functional models* have no notion of resource sharing or time, i.e. functionality is executed instantaneously and the model may or may not be bit-accurate.
- *Transactional behavioral models* are functional models mapped to a discrete time domain. Transactions are atomic operations with their duration stochastically determined.
- Except for asynchronous models, *transactional clock accurate models* map transactions to a clock cycle; thus, synchronous protocols, wire delays, and device access times can be accurately modeled.
- *RTL models* are mapped to a continuous time domain, including currents, voltages, noise, rise and fall times. Data types are bit-accurate, interfaces are pin-accurate, register transfer is accurate, and last but not least models are synthesizable. Propagation delay is back annotated from gate and transistor models.
- *Gate models* are RTL models with additional information, e.g. layout configuration.

A crucial part in the stepwise transformation of a high level behavioral model into actual implementation is *refinement*. Protocol refinement allows the designer to explore model functionality at different level of abstractions, thus trading between model accuracy with simulation speed. This continuous rearrangement of existing IP in ever-new composites is also a key process to new marketing ideas.

Object-oriented system design is based on virtual components (called IP), interacting with each other in a specific system environment. Thus, refinement is naturally based on separating each IP module into two components.

- *A behavior component* describes module functionality. It usually has an associated identity, state and an algorithmic process consuming or producing communication cells, synchronizing or processing data objects. Access to a behavior component is provided via a communication interface and explicit communication protocols.
- *A communication interface* consists of I/O ports transferring messages between one or more concurrent behavior components. The interface may support various communication protocols and is the **only** way to interact with the behavior. Thus, we fully de-couple behavior from inter-module communication.

Behavior and communication interfaces can be expressed at various levels of abstraction. Static, time-independent behavior is specified with untimed functional models, while dynamic, time-dependent behavior is based on complex control, e.g. hierarchical Finite State Machines (FSMs) or Threads. Similarly, communication can be abstract or close to implementation, i.e. using generic interfaces, `VCI` [14], or proprietary interconnects [10,11].

Furthermore, each communication port is connected using an interface to a *communication channel* object. The channel implements different communication protocols and **must** support an optimal design methodology based on top-down and bottom-up refinement.

- In top-down refinement, we focus on capturing desired system requirements by refining the abstraction level down to implementation, filling details and constraints.
- In bottom-up IP-reuse oriented refinement, we focus on the evaluation, composition and deployment of prefabricated IP.

Using refinement, behavior and communication interfaces are eventually mapped to hardware/software resources of a particular architecture. This categorization, known as *system partitioning*, is an essential element of co-design. Behaviors mapped to hardware logic are either synthesized, or selected from existing IPs, e.g. a processor core. Behaviors mapped to software are assigned to a software process or device driver. Similarly, communication protocols are mapped to hardware, or software depending on available semantics, e.g. shared memory, message passing, Ada-like rendezvous, or queuing structure. When the optimal mapping of behavior and communication components to architecture is reached, the designer may either manually decompose hardware components to the RTL level of abstraction, or use

available behavioral synthesis tools. Much of the hardware implementation can be reused from previous designs, including processor cores, bus interfaces and protocols, and large blocks, such as MPEG decoders. Similarly, on the software side, RTOS, device drivers and large blocks of code-like protocol stacks can also be reused.

3. IPSIM Refinement Methodology

3.1 The IPSIM Design Environment

IPSIM is a design environment consisting of a SystemC-based extensive C++ modeling library, together with a simulation engine, runtime and test environment [8] and communication refinement which further extends the general refinement methodology described in Section 2.

The IPSIM library provides basic building blocks (many not existing in SystemC) that support and simplify SoC modeling. IPSIM provides a fully documented design flow based on SystemC 3.0 methodology. The IPSIM library provides public access to the following objects.

- *module container*,
- *intra-module data type objects* allowing system-independent bit-accurate modeling,
- *intra-module memory objects*: Register, FIFO, LIFO, circular FIFO, Memory, Cache and IntMem (collection of memory objects),
- *intra-module communication and synchronization objects*: Mutex, Semaphore, Mailbox, Monitor, Event Flag, Timer and Watchdog Timer,
- *intra-module control objects*: hierarchical FSM, Thread (clocked Thread or asynchronous Thread),
- *intra-module simulation clock* (InClock),
- *inter-module communication object*: Message Box,
- *inter-module communication channel and interface*s supporting both generic protocols, such as Message Channel, Standard Channel, Request Acknowledge (`RA`), Request Valid and Interrupt Request, and proprietary protocols, such as Amba Bus and `STbus`.
- simulation clock (Clock), and
- simulation event scheduling environment.

IPSIM abstraction levels range from functional to clock cycle bit-accurate models. Signal and gate models are supported by general design flow, as seen in Section 2. Simulation can be performed using either the standard SystemC 3.0, or the IPSIM engine for faster simulation.

The IPSIM runtime and test environment includes support for compilation, debugging, simulation execution, model testing, FSM analysis and performance modeling.

3.2 IPSIM Performance Modeling

IPSIM supports performance modeling based on delay statistics for consecutive read/write operations, throughput for memory read/write access and possibly power (switching activity) estimates. In addition IPSIM provides average and instance size metrics for FIFO objects, and hit ratios for read and write access to Cache. The IPSIM statistical API is based on the `enable_stat_()` primitive. Furthermore, public IPSIM statistical classes, such as `stat_instant` and `stat_duration`, may be used either for advanced user-defined statistics, e.g. cell loss probability, or for creating new joint statistic classes.

3.3 IPSIM Communication Refinement

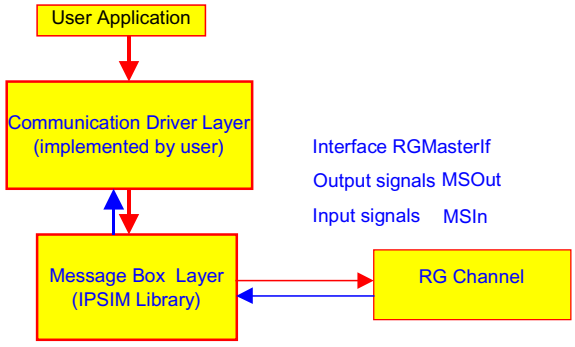

Figure 1. IPSIM Communication Interface

Layering is a common way to capture abstraction in communication systems. The advantage of layering is that methods of passing information from one layer to another are abstracted, thus changes within a protocol layer do not affect other layers. This simplifies design and maintenance of communication systems. As shown in Figure 1, IPSIM extends general SoC refinement by establishing distinct inter-module communication layers.

- The bottom layer, called IPSIM *Message Box layer* is based on two IPSIM objects: *Message Box*, and *communication channel*. The Message Box object, defined with a particular module, supports an extended message passing paradigm for implementing inter-module communication and synchronization, provides a generic API for mapping into various

communication channels, and provides the required buffers for inter-module data transfer. The communication channel establishes the transfer of module interface signals and data among Message Boxes according to a specific protocol (channel interface) chosen from the IPSIM channel library.

- The top layer, called *communication driver layer,* is a user-defined C++ class that translates inter-module transaction requests to the *Message Box layer*. The driver object simplifies specification and allows refinement when embedding communication protocols onto IP blocks or SoC. The driver layer may further be refined with additional communication layers.

IPSIM's two-level approach is similar to application- and system-level transactions in Cosy [2], which was based on concepts from VCC framework [1,3,5,7,9]. The system-level layer was responsible for selecting communication parameters and resolving latency-throughput tradeoffs, adopting eventually physical bus protocols. Similarly, drivers for multiprocessor SoC partition functionality into non re-entrant low-level drivers providing basic functionality and complex re-entrant high-level device drivers interacting with the application.

IPSIM refinement based on inter-module communication layers yields several important benefits:

- orthogonalizes behavior and communication, thus enabling plug-and-play system design by providing appropriate communication interface blocks, encapsulating and protecting IP cores,
- separates communication from architectural implementation, thus enabling co-design, and
- reduces ambiguity in writing communication drivers.

3.4 Implementation of IPSIM Refinement

Before we proceed with a detailed description of the layers, we define the *IPSIM module* and *communication channel* objects that are based on SystemC 3.0 elements.

The IPSIM Module, inherits from `sc_module`, and embeds a concurrent process modeled by one or more execution threads. In IPSIM methodology, we distinguish communication and synchronization into:

- *intra-module operations* based on a concurrent shared memory model supported by IPSIM Mutex, Semaphore, Event Flag and Mailbox objects, and
- *inter-module operations* based on an extended message passing paradigm supported by IPSIM Message Box.

The IPSIM communication channel is a template object of a pair of user-defined and/or protocol-specific C++ classes. Thus, in a point-to-point bus, it is parameterized through user-defined Master signals and data coming out of the Master (and going to the Slave), and Slave signals coming out of the Slave (and going to the Master). These signals define the Module interface, and may include for example address, operation code and data. In general, a channel may model both point-to-point communication, as well as multi-access channels, such as multicast switches, or bus systems. Thus, IPSIM supports a large variety of generic, as well as system-specific, channels.

3.4.1 The IPSIM Message Box Layer

The IPSIM Message Box Layer is based on the IPSIM Message box that generalizes the `sc_port` and (user-defined) `channel` elements in SystemC 3.0. The Message Box defines a C++ object API for implementing data transfer among modules that use compatible channel interfaces. Thus, it provides the required buffers for data transfer, as well as control for interface activation.

The channel interface is fixed at the declaration of a Message Box. The Message Box object is a template object with three parameters that build an appropriate IPSIM channel interface. The first parameter refers to the protocol, while the next two parameters associate the channel interface to a pair of user-defined or protocol-dependent C++ classes describing incoming and outgoing attributes (signals and data). For example, when using the RA protocol, we provide in the Master module:

```
Msgbox_Type(RaMas,MSOut,MSIn) MbxName;
```

Then, a Message Box `MbxName` for the Master component connected to an RA channel is created as:

```
Msgbox(MbxName, clk, len),
```

where `MbxName` is the name, Message Box `clk` defines the clock domain under which Message Box functions operate (`clk=0` refers to an asynchronous Message Box), and `len` specifies the data size, i.e. maximum number of bits, that we can send or receive through the channel in one clock cycle. For the `RA` protocol, `MSOut` and `MSIn` are user-defined attributes (signals and data) for inter-module communication, i.e. they are defined as follows.

```
struct MSOut {
```

```
    N_uint              sequence;
    IPSIM_memory_addr address; };
  struct MSIn {
    N_uint ack; };
```

3.4.2 The IPSIM Driver Layer

The IPSIM Communication Driver Layer implemented using IPSIM tasks, i.e. hierarchical FSM and Thread, allows the user to specialize the Message Box Layer in order to closely match communication requirements. Thus, Message Box functions are invoked by high-level communication drivers, e.g. for reception/transmission. While initially a high-level interaction between behaviors is implemented using driver and channel stubs, later detailed protocols use complex communication channels.

4. Case Study: Receiver-Transmitter

transmitter_drv.cc

```
#include "transmitter_drv.h"
// Driver constructor
tdrv::tdrv(InClock* clk, N_uint len)
 :Msgbox_type(MsgMas, MSIn, MSOut)::
  Msgbox(clk, 32, "mbox_tr")
{ // Message Box throughput for reads
  // in [0, 50000ns] with time window=1
  enable_stat_throughput_read(0, 50000, 1,
"Simulation Time","Average Throughput for Reads");
}
//Transmitting thread for driver
void tdrv::tx_driver(IPSIM_memory_addr addr,
                     N_uint seq, Bytes& data)
{ set_tx_attr(address) = addr;
  set_tx_attr(sequence) = seq;
  read(data); // data to send
  set_tx_start();
  IPSIM_printf("tdrv:tx_dr receives seq_no:%u
          to store at addr:%u", seq, addr);}
// Receiving thread for driver
void tdrv::rx_driver(N_uint *ack)
{ *ack = get_rx_attr(ack);
  //no data sent, i.e. no write(Bytes &tmp);
  set_rx_start();
  IPSIM_printf("t_driver: RX_proc sends
               ack no:%d", *ack);}
```

The code block (see transmitter_drv.cc) illustrates IPSIM driver implementation when the basic `Msg` channel is used. The driver API, i.e. the `tx_driver` and `rx_driver` functions, is based on the user-defined attributes: address, `seq` (sequence number), `data` (sequence of bytes) and `ack` (acknowledgment). The Message Box `set_tx` and `get_rx` functions set or get these Message Box attributes, while read and write transfer application data (sequence of bytes) to/from the Message Box object.

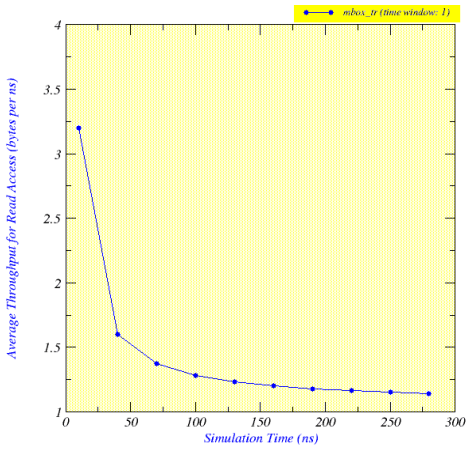

Figure 1. Time-driven statistics: Average Throughput for Read Access vs Simulation Time

Figure 2. Throughput graph for Message Box

In order to generate statistics for IPSIM components, `enable_stat_` function calls are made from within the module constructors. The highlighted code explains IPSIM methodology for collecting statistics from the communication driver Message Box. Figure 2 illustrates cumulative average throughput for read access to the Transmitter Message Box "`mbox_tr`".

Workstation	IPSIM Model (sim. cycles/sec)	SystemC Model (sim. cycles/sec)
Sun Ultra 60	253,000	84,000
Blade 1000	451,000	160,000

Table 1. Simulation efficiency for Case Study

We have simulated the Transmitter/Receiver model for over 4 million cycles with the communication driver based on the message channel (Msg) on two stand-alone workstations: Ultra 60 and Blade 1000. We compiled all models with the highest optimization level, compiler switch –O3, observing similar IPSIM and SystemC compilation time and slightly (20%) larger SystemC executable binary size. In Table 1 we compare IPSIM and SystemC simulation efficiency metrics obtained by dividing the actual simulated cycles by CPU time. IPSIM simulation is approximately 3 times faster than SystemC simulation. Similar results have been obtained for the Transmitter-Receiver model with complex communication channels (with a speedup of ~6). Current analysis of large-scale network models yields similar performance gains. IPSIM's simulation efficiency is due to its superior

simulation kernel based on an optimized calendar queue-based dynamic event scheduler and efficient implementations of IPSIM library objects.

The following code block (see transmitter_drv_STbus.cc) illustrates refinement for `tx_driver`, (`rx_driver` is omitted), if the proprietary `STbus` channel is used instead of the `Msg` channel. Notice that the Message Box type (line 5 in transmitter_drv.cc) must be changed to `STBusMas`, i.e. the Master interface for `STbus`. Observe that we do not modify Transmitter or Receiver modules, or test benches. Thus, IPSIM extends the 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.

transmitter_drv_STbus.cc

```
//Transmitting thread for driver
void tdrv::TX_driver(IPSIM_memory_addr addr,
                     N_uint seq, Bytes& data)
{
set_tx_attr(address) = addr;
set_tx_attr(source_id) = 0;
set_tx_attr(tid) =0;
set_tx_attr(lock)=0;
set_tx_attr(be)=0xF;
switch(data.get_size_in_bytes()) {
  case 1: set_tx_attr(opcode)= STBUS::STORE1;
          break;
  case 2: set_tx_attr(opcode)= STBUS::STORE2;
          break; // ... omitted
  }
  read(data);
  set_tx_attr(sequence) = seq;
  set_tx_start();
}
```

5. Conclusion

IPSIM refinement separates IP modules into behavior and communication interfaces and establishes two inter-module communication layers. The bottom layer, called *Message Box layer,* establishes inter-module transfer of interface signals and data according to a generic or system-specific protocol The top layer, called *communication driver layer* translates inter-module transaction requests to the Message Box layer. Layering simplifies specification and allows further refinement by introducing application-based abstractions.

In the case study, we have shown how IPSIM driver models exploit design abstraction, thus having the benefit of efficient simulation. Due to an extensive computation, control, communication and synchronization library, a fast simulation kernel and an efficient performance modeling methodology for collecting statistics from system components, we argue that IPSIM can achieve a higher degree of productivity than other SystemC-based C++ libraries present in the market.

IPSIM design methodology is being applied to large-scale design projects in application domains, such as high speed networks. IPSIM extensions towards asynchronous systems, efficient hardware/software partitioning via distributed simulation, and interoperability of tools will be promoted as potential standards for system design.

Acknowledgments

This research has been sponsored in part by EU Medea+ project ToolIP. We would also like to thank Synopsys for their support in integrating IPSIM into the Synopsys Co-Centric design tools.

References

[1] F. De Bernardini, M. Sgroi, and L. Lavagno “Developing a methodology for protocol design”. Cadence Berkeley Labs, SRC DC324-028.

[2] J. Y. Brunel, W. M. Kruijtzer, H. J. Kenter et al. “COSY Communication IP's”. Proc. Design Automation Conf., 2000, pp. 406-409

[3] Cierto Virtual Component Co-design (VCC), Cadence Design Systems, http://www.cadence.com/technology/hwsw/ciertovcc/

[4] CynLib, CynApps, http://www.cynapps.com

[5] A. Ferrari, and A.Sangiovanni-Vincentelli, “System design: traditional concepts and new paradigms”. *Proc. Int. Conf. Computer De*sign, 1999.

[6] D.D. Gajski, J. Zhu, A. Doemer et al. “SpecC: Specification language and methodology”. *Kluwer Academic Publishers*, 2000. Also see, http://www.specc.org

[7] J.A. Rowson and A. L. Sangiovanni-Vincentelli, A.L. “Interface-based design”. P*roc. Design Automation Conf*, 1997, pp. 178–-183.

[8] Runtests Lite, internal document, STM, June 2000.

[9] M. Sgroi, M. Sheets, A. Mihal et al. "Addressing SoC interconnect woes through communication-based design". *Proc. Design Automation Conf.*, 2001.

[10] STbus, internal document, STM, March 2001.

[11] STbus C++ classes, internal document, STM, Dec. 2000.

[12] Superlog, www.co-design.com

[13] SystemC, http://www.systemc.org

[14] VSI Alliance: http://www.vsi.org/.