A Timing-Accurate Modeling and Simulation Environment for Networked Embedded Systems

Franco Fummi† Paolo Gallo * Stefano Martini †

Giovanni Perbellini † Massimo Poncino † Fabio Ricciato *

† Università di Verona
Verona, ITALY 37134

*Telecom Italia Lab
Torino, ITALY 10148

ABSTRACT

The design of state-of-the-art, complex embedded systems requires the capability of modeling and simulating the complex networked environment in which such systems operate. This implies the availability of both a networking modeling environment and traditional system-level modeling capabilities. In this paper we present a modeling and simulation methodology based on a timing accurate integration of a system-level modeling language (SystemC) and a network simulation environment (NS-2). The efficiency of the proposed design environment has been demonstrated on a description of a 802.11 MAC layer.

Categories and Subject Descriptors

J.6 [**Computer Applications**]: Computer-Aided Engineering; C.2 [**Computer Systems Organization**]: Computer-Communication Networks; B.8 [**Hardware**]: Performance and Reliability

Keywords

Co-Simulation, Emulation, SystemC, Remote Debugging

General Terms

Algorithms, Performance

1. INTRODUCTION

Modern embedded systems exhibit an increasingly large quantity of communication capabilities; the devices should be casually accessible, mobile or embedded in the environment, and sometimes operate in hostile conditions and connected to a network structure.

The communication issue is also crucial within the embedded system itself, where the on-chip bus architecture is becoming the key factor for the embedded system performance success. As stated by International Technology Roadmap for Semiconductor [1], for technologies below 65nm it will be impossible to efficiently move synchronous signals across large dies and to keep the same clock synchronization along the whole chip due to delay and power problems. The synchronous approach will shift toward a *globally asynchronous and locally synchronous approach* (GALS) [2], where self-timed blocks will communicate as different computers in a network. This will bring a network-oriented paradigm into on-chip architectures [3].

The idea is to model these networked embedded systems in a hybrid style: part of the system is modeled in a conventional, system-level, hardware-oriented environment, while the network part is described with a network modeling tool. In this way, we can have preliminary feedbacks on the interaction between the embedded system to be implemented and the network environment. Figure 1 shows a typical context of application of this mixed-level simulation, where some embedded devices under project are progressively refined toward the system-level hardware domain.

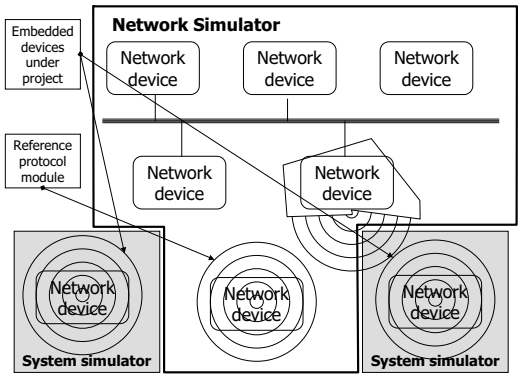

Figure 1: Example of Integrated Simulation Framework.

The refinement steps will be carried out on the system-level model, reusing the same network model at lower refinement levels. This is possible, however, only if the network simulator is timing-accurate and synchronized with the system simulator. In other words, the simulators should have a common notion of time, and they should have the possibility of exchanging data bidirectionally. This will yield a faster modeling of a system with respect to a description entirely

DAC 2003, June 2–6, 2003, Anaheim, California, USA.

specified as a system model, provided that the synchronization overhead is not too heavy.

Another issue is concerned with the data exchange format between the two simulators. A network simulator usually does not have bit-true models of data, and uses abstract data types to describe packets, frames and messages. The model of the embedded device must have bit-true data interfaces available, at least at lower refinement levels; some translation interfaces are therefore required.

A tool embedding a network simulator and an embedded system modeling environment can significantly speed up the design of a networked device, by producing the testbed in shorter times. In addition, it allows to verify and validate the embedded system communication structure by complementing its protocol stack with a reference protocol stack described at higher level with the network simulator.

This work describes a simulation framework consisting of the integration of two simulators: SystemC [4], as a system-level simulator, and NS-2 [5], as a network simulator. The integration is carried out at the level of the simulation kernels, to provide maximum efficiency, and it supports a timing-accurate synchronization of the two simulators. Results on an industrial case study, relative to the implementation of an 802.11 MAC Interface [6] design shows the high efficiency of the proposed integrated solution.

2. SIMULATION ENVIRONMENT

The issue of integrating different simulators is a well-known problem in the CAD community, and is known as *co-simulation*. Several co-simulation frameworks have been proposed in the literature [7, 8]. They typically differ in the abstraction levels targeted by the involved simulators (transactional, instruction-level, RTL), and in the type of communication primitives used for their synchronization.

In [9], Matlab is used to model the entire system (network and candidate hardware); this solution, however, does not provide a direct path to hardware. In [10], the system (an ADSL modem with the environment) is completely modeled in C++, thus making quite hard to model a complex network, in order to verify the device under design. The work of [11] directly connects the device model to a real network, without using a specific network model. None of these existing solutions explicitly targets the integration of network and system-level simulation by "connecting" effective simulators of both fields.

Our approach relies on two well-known simulation environments: SystemC [4], and the Network Simulator-2 [5]. SystemC is a C++ class library that can be used to create models of a system at different abstraction levels, from the system-level to the cycle-accurate one. NS-2 is one of the most popular tools for network protocol and algorithm analysys. It is based on two programming languages: C++ and OTcl, which separate the actual simulation kernel (written in C++) and the simulation configuration.

Both simulators are conventional event-driven engines; they schedule the execution of events in non-decreasing order of timestamp. What differs is the semantics of the events. In SystemC, events are associated to hardware-like entities such as signals and ports. The fact that SystemC explicitly supports the notion of a clock is purely conceptual, and it is only used to define the time granularity of the events. In NS-2, events are associated to asynchronous changes on communication channels such as sending or receiving a packet.

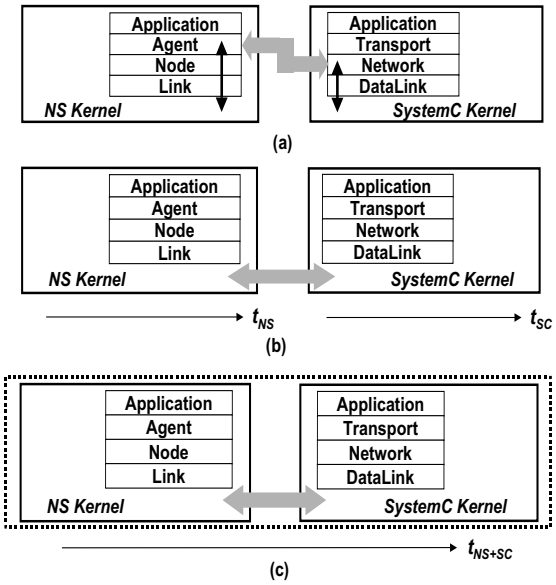

Figure 2: Simulation Abstraction Levels.

2.1 Simulation Interaction

When designing an integrated simulation framework, two are the main issues that affect the integration and define the semantics of the simulation. The first, and most important, is how tight the two simulators will be coupled. The options range from establishing a *communication* between the two simulators through a straightforward message-passing or shared memory interface, to a complete *integration* of the two simulation kernels. The second issue has to do with the level of abstraction at which the two simulators interact, and is meaningful for simulators that support various levels of abstractions. This may range from packet-level interfaces to low-level signals.

In our context, these two issues first require some insights on the features of NS-2 and SystemC. Both simulators consider the target description as a hierarchical entity, whose levels correspond to different abstraction layers and different granularities of the data manipulated by the simulators.

NS-2 views network descriptions as layered, protocol-like entities, consisting of four hierarchical objects: *links, nodes, agents,* and *applications*. These roughly map to the data link, network, transport and application layer of a simplified protocol stack, respectively. Agents, in particular, represent endpoints where network-layer packets are constructed or consumed, and provide support for all transport protocols primitives.

In SystemC, the abstraction layers are related to the refinement of the initial specification, and are labeled, in our context, with the conventional protocol layer names.

When simulating networked embedded systems, the most straightforward integration is shown in Figure 2-(a). Here, a NS-2 agent (and thus, a packet-level interface) is connected to a "network"-layer SystemC entity, corresponding to a structural description style. This solution simply implies the exchange of messages between these two layers, plus some sort of message translation interface. In this case, however, any synchronization of the two simulators requires an explicit message through a user-level API (e.g., UNIX sockets), which makes the communication overhead burden-

some. Furthermore, a consistent timing synchronization is not easy to achieve; since time is managed within the simulation kernel, any access to time information requires the interaction with the local kernels.

A direct, message-based integration at the simulation kernel level (Figure 2-(b)) may be a more efficient solution, since the communication overhead is limited to the messages exchanged between the kernels. However, as denoted by the two separate times scales, this approach does not solve the issue of timing synchronization automatically.

Figure 2-(c) shows the solution proposed in our work. The two kernels are tightly coupled (the dotted outer box), and the integration, though still message-based, allows for an easier synchronization of the respective simulation times. The high efficiency of this solution is presented in the experimental results section.

3. KERNEL SYNCHRONIZATION

The issue of keeping the two simulators synchronized in time is far from being trivial. As a matter of fact, one thing is to guarantee that the simulation times in each simulator are mutually consistent, which can be considered as a relatively easy task. A sensibly more difficult problem is to keep the two simulators consistent when data are exchanged between them. In this section we will discuss the technical issues related to this problem, and the relative implementation details.

3.1 Basic Timing Synchronization

Conceptually speaking, the synchronization of the two simulators can be realized according to two conceptual schemes. *Asymmetric*, where one of the two simulators (the *master*) explicitly controls the execution of the other (the *slave*). *Symmetric*, where no simulator has explicit control on the other, and the global time is kept consistent by reciprocally exchanging messages. Our approach follows the second scheme; in particular, the two simulators execute in lockstep: at any given time, the simulator with the smaller local time is executing, while the other one is blocked.

Figure 3 shows the pseudocode of the time synchronization procedure, which is executed by both kernels. In the figure, the subscript $k1$ refers to the kernel currently executing the code, and $k2$ to the other kernel.

```
1  SynchronizeTime () {
2      Setup phase;
       do {
3          Receive(T_k2); // from other kernel
4          if (T_k1 < T_k2) {;
5              schedule event;
6              Send(T_k1^next);
           } else {
7              Send(T_k1);
           }
8      } while (there are messages)
   }
```

Figure 3: Synchronization of Simulation Times.

We discuss the pseudocode with reference to one of the two simulators. After a setup phase (Line 2), which is executed only by the simulator that starts the synchronization, the simulator gets the "remote" simulation time from the other kernel T_{k2} (Line 3). Once read, it compares T_{k2} with its own, "local", simulation time T_{k1} (Line 4). If local time is smaller than the remote one, then the simulator is lagging behind and the event currently ready to be scheduled is executed (Line 5). Furthermore, the time of the next event in the scheduler queue T_{k1}^{next} is sent to the other simulator for synchronization (Line 6). Conversely, if the simulator is ahead of time with respect to the remote one it does not execute, and it send its current simulation time T_{k1} to the other simulator (Line 7). The above operations are repeated (Line 8) for the duration of the computation.

3.2 Handling Data Exchange

The relatively straightforward scheme of Figure 3 works fine as long as only the simulation times need to be kept consistent. In most of the cases, however, the interaction between the two simulators requires the exchange of some data, whose sequence over time must be also be kept consistent. This implies the modification of the data structures of the two kernels in order to handle the separation between events and associated data. Since both NS-2 and SystemC are event-driven, they both exhibit the classical structure with a conventional *Ready Queue* containing ready-to-run events, from which the corresponding scheduler picks the events, in order of non-decreasing timestamp, and dispatches it for the execution of the relative action.

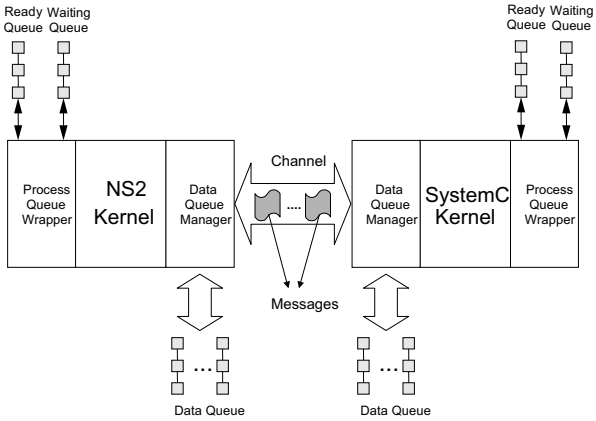

Figure 4: Kernel Architecture for Data Exchange.

Figure 4 shows the five fundamental entities involved in the synchronization. A *Waiting Queue* is present in both simulators. Each queue contains the "events" that are currently waiting for a certain data-driven condition to happen. This scheme is reminiscent of a conventional operating system, where the ready queue is separated from the queue of the processes waiting for some event to complete (I/O requests, or resources to become free).

The *Process Queue Wrapper* is responsible for interfacing the *Waiting Queue* and *Ready Queue* with each simulation kernel. However, the way the *Process Queue Wrapper* deals with the *Waiting Queue* is strictly related to the operations of the other two entities that are involved in the data exchange. When a data is received from the communication channel, the *Data Queue Manager* places it in a *Data Queue*. There is one *Data Queue* for each destination object. Processes in the *Waiting Queue* are waiting for some data-related conditions to happen. Therefore, for each newly received datum, the *Process Queue Wrapper* looks

into the *Waiting Queue* for processes waiting for events on the *Data Queue* affected by this newly received datum. Such processes are promoted to the *Ready Queue* for execution, and their timestamp is updated with the timestamp of the incoming data.

From this discussion, it is clear that it is essential to encapsulate the data moving between the two simulators with a number of information.

3.3 User Programming Paradigm

The protocol described in the previous section requires to specify at least: (i) the identity of the entities involved in the data exchange (NS-2 agent and SystemC module); (ii) the timestamp associated to the data by the sending entity. Such information are organized into messages, whose format is shown in Figure 5.

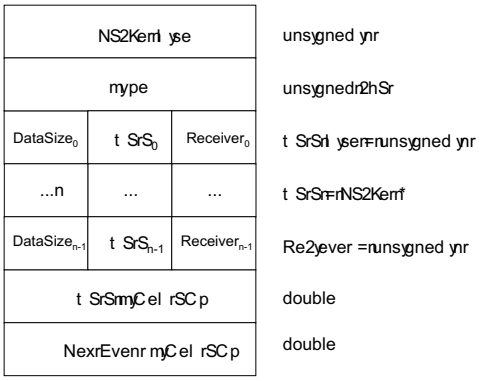

Figure 5: Message Format.

Messages are composed by the two kernels in a way that is transparent to the user. From the NS-2 side, the new class `SystemC` has been added. It is defined as follows:

```
class SystemC {
  void Recv (Packet *p, unsigned int size, int receiver) {
      SetKernelStateVariable(false);
      SaveDataPacketInKernel(p, size, receiver);
  }
  ...
}
```

The `Recv()` method allows a NS-2 object to pass to the kernel a packet of data (normally exchanged through NS-2 nodes) that must be sent to a SystemC process identified by the variable `receiver`. The Boolean variable `state` allows the kernel to know that a packet is ready to be sent to SystemC. Starting from such information, the NS-2 kernel is able to compose the message, which is sent to the SystemC kernel. For instance, the standard UDP agent modeled in NS-2, must be modified by the user as follow to communicate with SystemC:

```
class newUDP : public Agent {
  void sendMsg (...) {
      ...
      Packet *p;
      ...
      // original NS2 method to transfer a packet is removed
      // target_->Recv(p);
      // SystemC method to transfer a packet is added
      SystemC SC_;
      SC_->Recv(p, sizeof(Packet), receiver);
  }
}
```

Whenever a data message is received by NS-2, the kernel builds a packet and it calls, as usual, the `Recv()` method of the agent, which manipulates the packet as it would be received by another NS-2 agent.

From the SystemC side, the new ports `ns_in` and `ns_out` have been added to allow the user to send/receive a packet to/from a NS-2 object. They are derived by the template classes `sc_in` and `sc_out` and are managed by two methods `read()` and and `write()`, an extension of the standard methods managing `sc_in` and `sc_out` ports. Such ports do implement the concept of *"co-simulation external port"* described in [7]. Therefore, a SystemC process with a bidirectional communication with NS-2 is simply defined as follows:

```
SC_MODULE(m) {
    ns_in in_port; // port to receive a packet from NS-2
    ns_out out_port; // port to send a packet to NS-2
    ... // standard ports to communicate with other modules
    SC_CTOR(m) {
        SC_METHOD(proc1);
        sensitive << in_port;
    }
    void proc1();
}
```

To send (read) a packet sent (received) to (from) NS-2, the process must simply manipulate the ports as follows:

```
void m::proc1() {
    ...
    Packet * p;
    p = ... // the packet is explicitly built
    out_port.write(p, sizeof(Packet), receiver); // packet sent
    in_port.read(p,sizeof(Packet); // Read from NS-2
}
```

Whenever the SystemC kernel receives a data message, it generates an event, at the time specified in the message, on the `ns_in` port of the process specified in the message, by assigning the packet to the port. This wakes up the SystemC process that is waiting for the packet in the same manner as if the packet would be sent by another SystemC process. Conversely, whenever a SystemC process writes a packet on a `ns_out` port, the `write()` method stores it into a memory buffer monitored by the kernel. When the simulation control is passed to the kernel, this builds a message consisting of a packet, by following the algorithm described in the next section.

This implementation has two main advantages. First, both NS-2 and SystemC kernels wake up objects and processes by using their original primitives, respectively, the `Recv()` method and the assignment of a value to a port. In this way, no additional *Waiting Queues* are necessary, as in the general case depicted in Figure 4, since the standard queues of the two kernels are used. Second, both NS-2 objects and SystemC processes exchange data passing through the kernels that are responsible to use the communication channel. In this way, no events are generated, besides those necessary for an object or process to wake up for monitoring the communication channel. This issue directly translates into increased performance.

3.4 Kernels Extension for Synchronization

Figure 6 shows the pseudocode of the synchronization procedure, which is executed by both kernels. As for Figure 3, the subscript $k1$ refers to the kernel currently executing the code, and $k2$ to the other kernel; both kernels are modified so as to incorporate this procedure.

After the start phase, executed only by the simulator that starts the process (Line 2), the main synchronization loop (Lines 3–22) evolves around the reception of messages on the channel that links the two simulators (Line 4). Anytime a

```
1  Kernel_Scheduling () {
2     Start phase
3     do {
4        Receive(M_k2); /* from other kernel */
5        state = TRUE;
6        if (M_k2.Type == Data) { /* data messsage */
7           Call M_k2.Receiver_i.Recv();
         }
         T_k2 = M_k2.NextEventTimeStamp;
8        Get next event E_k1 from ReadyQueue;
9        T_k1 = TimeStamp(E_k1);
10       while (T_k1 ≤ T_k2 && state) {
11          do { /* events that are lagging behind */
12             dispatch event E_k1;
13             if (E_k1 requires sending data) {
                  /* setup M_k1 accordingly; */
14                M_k1.Data = D_k1; M_k1.Receiver_i = j;
15                state = FALSE;
               }
16             Get next event E_k1 from ReadyQueue;
               T_k1OLD = T_k1; T_k1 = TimeStamp(E_k1);
17          } while (T_k1 == T_k1OLD);
18       }
         Allocate a new message M_k1;
19       if (state) {
            M_k1.Type = Data;
            M_k1.Time_Data = T_k1OLD;
            state = true;
         } else M_k1.Type =Time;
20       M_k1.NextEventTimeStamp = T_k1;
21       Send(M_k1);
22    } while (there are messages);
   }
```

Figure 6: Data Exchange Synchronization.

message is received, it is first checked if it is a *Data* message type; in this case, it causes the invocation of the `Recv()` method (Lines 6–7) of the SystemC or NS-2 receiver, as described in Section 3.3.

Then, the timestamp T_{k2} of the remote event is extracted and compared to the timestamps T_{k1} of the local events (Lines 8–9). The loop of Lines 10–18 manages the processing of all the events in the local queue that are lagging behind the remote simulation time T_{k2}. If the generic event E_{k1} implies the transmission of data (Line 13), the current message M_{k1} for time T_{k1} is properly setup by specifying the corresponding data field D_{k1} and remote recipient j (Line 14). This condition is flagged (Line 15) for later use. For any new value of T_{k1} (yet still $< T_{k2}$), a new message M_{k1} is actually allocated (this operation is abstracted away in the pseudocode). The do while loop at Lines 11–17 allows the dispatching of all events with the same scheduling time. When the loop at Lines 10–18 exits it is time to send a message to the other kernel. If the flagged condition at Line 19 is true, the kernel sets up a data message, otherwise a time message is set up. When the message is ready it is sent to the other kernel.

Finally, notice that the timing synchronization procedure of Figure 3 is still visible in the above pseudocode, yet tightly intertwined with the rest of the synchronization steps.

4. EXPERIMENTAL RESULTS

The above simulation environment has been used to model a home networking scenario, in which a video flow is downloaded from a video server through a broadband network access and sent to a TV-set through a wireless LAN link based on the IEEE 802.11b protocol (Wi-Fi) [6]. The target was to simulate the system-level model of the chip in a complete network testbed. Because of the changes to the chip specifications and architecture, an accurate simulation phase was needed.

4.1 Simulation Models

The 802.11 MAC layer model included in NS-2 implements the basic access procedure described in the protocol, with the correct interframe spaces, the backoff procedure and the RTS-CTS procedure.

The physical layer in NS-2 is modeled at a very high level of abstraction: a propagation model is used to calculate the power received at every station for each frame transmitted on the wireless channel, but the data unit exchanged through the channel is a whole MAC frame instead of a bit or a group of bits. This assumption gives a MAC interface toward the lower layer entity that is quite different from the standard; in particular, the service data units between MAC layer and physical layer are MAC frames in NS-2, instead of bytes as required by the standard.

We developed a basic model for the IEEE 802.11 in SystemC. It is a simplification of the actual MAC protocol described in the standards. Although this model is currently used to describe only the basic medium access procedures (e.g., physical and virtual carrier sense, random backoffs, RTS-CTS procedure and frames acknowledgments), it can be easily extended to incorporate other functionalities, since it is derived from a complete description of the MAC protocol in ITU Specification and Description Language (SDL). The block diagram of the model implemented in SystemC is represented in the SystemC section of Figure 7.

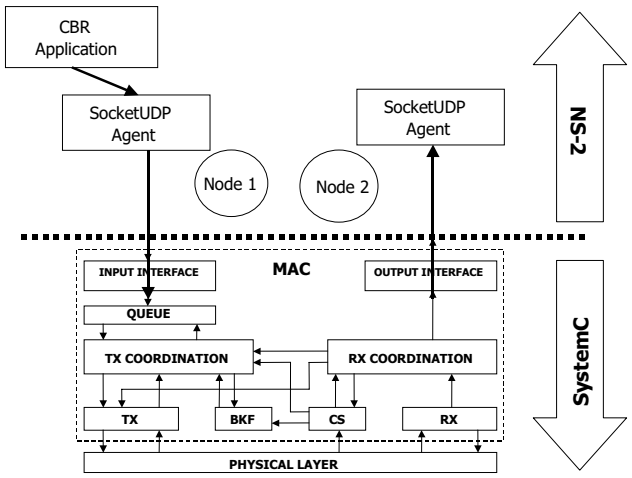

Figure 7: System Architecture.

4.2 Simulation Environment

The SystemC model was stimulated with CBR (Constant Bit Rate) traffic generated by two NS-2 nodes. Figure 7

shows the configuration used to verify the SystemC model. The two NS-2 nodes models two wireless stations: on the first node is attached a CBR Application on a `SocketUDP` agent. The second node models the receiver station with the peer `SocketUDP` agent. The SystemC input and output interface blocks are respectively connected to the transmitting and receiving `SocketUDP` agent.

4.3 Simulation Results

The above configuration allows to carry out a system-level exploration of the system parameters and of the architectural alternatives.

Figure 8 shows an example of analysis obtained by the proposed mixed-level simulation scheme. In the plot we observe that the traffic rate saturates the radio channel (802.11b 11 Mbit/s of modulation); it is thus possible to evaluate the maximum possible value of the average network bandwidth (using 802.11b) according to the value of some parameters like the packet length and the adoption of the RTS/CTS protocol.

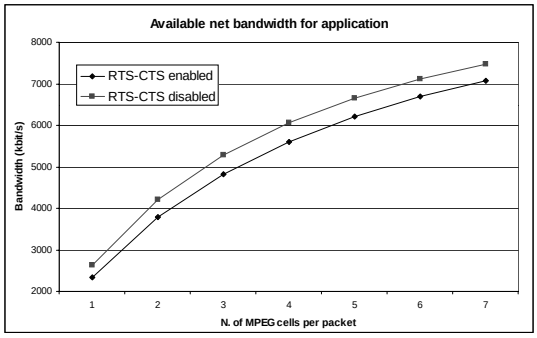

Figure 8: Simulation Analysis of the Traffic Rate.

An important issue of the proposed simulation environment is its efficiency. Clearly, the integration of the two simulators implies some communication overhead (e.g., the sockets used to transfer information between SystemC and NS-2) that may impact the overall simulation speed.

	Simulated Times (s.)			
	1.0	10.0	100.0	1000.0
`NS-2-only`	0.24	3.10	24.52	245.52
`SystemC+NS-2 Plain`	350.0	4650.0	$> 10^4$	$> 10^4$
`SystemC+NS-2 Proposed`	0.86	8.75	107.62	1106.93

Table 1: CPU times vs. Simulation Time.

Table 1 compares the efficiency of the our simulation scheme (`SystemC+NS-2 Proposed`) to a full NS-2 simulation (`NS-2-Only`), and to a straightforward implementation of the SystemC/NS-2 integration (`SystemC+NS-2 Plain`), realized by directly interconnecting via socket a NS-2 agent with a SystemC process without passing the packet through the respective kernels. In particular, we compared the relative slowdown incurred by simulation in simulating a given amount of real time. Simulations were run on a Pentium III @ 850 MHz, with 512 MB Ram, running Linux RedHat 8.0. The `SystemC+NS-2 Plain` produces a slowdown of more than two orders of magnitude with respect to a `NS-2-Only` simulation. On the contrary, it is evident the efficiency of the proposed cosimulation mechanism, which requires the same order of magnitude in CPU time of a `NS-2-Only` simulation.

5. CONCLUSIONS

The paper presented a design environment for the modeling and simulation of networked systems. The environment is based on the timing-accurate integration of the SystemC and NS-2 simulation environments. The two simulation kernels have been integrated by realizing an efficient synchronization methodology, where NS-2 is used for modeling the network environment, and SystemC is used to model the hardware part of the system under design. The simulation of a 802.11 MAC layer description has shown the effectiveness of the proposed simulation environment, in terms of solutions based on less integrated solutions. Furthermore, our simulation scheme exhibited marginal performance degradation with respect to NS-native simulation.

Future work will concern the identification of rules to guide the translation of NS-2 models into SystemC designs to extend the proposed refinement-based design environment for networked systems.

6. REFERENCES

[1] *International Technology Roadmap for Semiconductor 2001.* `http://public.itrs.net/Files/2001ITRS/Home.htm`

[2] J. Muttersbach, T. Villiger, H. Kaeslin, N. Felber, W. Fichtner. "Globally-Asynchronous Locally Synchronous Architectures to Simplify the Design of On-Chip Systems", *ASIC/SOC'99*, pp. 317–321, 1999.

[3] L. Benini, G. De Micheli. "Networks on chip: A New SoC Paradigm", *IEEE Computer*, Vol. 35, No. 1, pp. 70–78, Jan. 2002.

[4] Synopsys Inc. *SystemC User's Guide.* Version 2.1, 2002.

[5] L. Breslau *et al.* "Advances in Network Simulation," *IEEE Computer*, Vol. 33, No. 5, pp. 59–67, May 2000.

[6] ANSI/IEEE Standard 802.11. *Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications.* 1999.

[7] G. Nicolescu, S. Yoo, A. A. Jerraya. "Mixed-Level Cosimulation for Fine Gradual Refinement of Communication in SoC Design," *DATE'01*, pp. 754–759, 2001.

[8] A. Chatelain, Y. Mathys, G. Placido, A. La Rosa, L. Lavagno. "High-Level Architectural Co-Simulation Using Esterel and C," *CODES'01*, pp. 189–194, 2001.

[9] V. Aue, J. Kneip, M. Weiss, M. Bolle, G. Fettweis. "Matlab Based Co-Design Framework for Wirelss Broadband Communication DSPs," *ASSP'01*, Vol. 2, pp. 1253–1256.

[10] D. Desmet, M. Esvelt, P. Avasare, D. Verkest, H. De Man. "Timed Executable System Specification of an ADSL Modem Using a C++ Based Design Environment: a Case Study," *CODES'99*, pp. 38–42, 1999.

[11] R. Pasko, R. Cmar, P. Schaumont, S. Vernalde. "Functional Verification of an Embedded Network Component by Co-Simulation with a Real Network," *HLDVT'00*, pp. 64–67, 2000.