TOWARDS COSIMULATING SYSTEMC AND COLOURED PETRI NET MODELS FOR SOC FUNCTIONAL AND PERFORMANCE EVALUATION

M. Westergaard[a], L.M. Kristensen[b], M. Kuusela[c]

[a] Department of Computer Science, Aarhus University, Aarhus, Denmark.
[b] Department of Computer Engineering, Bergen University College, Bergen, Norway.
[c] OMAP Platforms Business Unit, Texas Instruments France, Villeneuve-Loubet, France.

[a]mw@cs.au.dk, [b]lmkr@hib.no, [c]m-kuusela@ti.com

ABSTRACT

Semiconductor technology miniaturization allows packing more transistors onto a single chip. The resulting System on Chip (SoC) designs are predominant for embedded systems such as mobile devices. Such complex chips are composed of several subsystems called Intellectual Property blocks (IPs) which can be developed by independent partners. Functional verification of large SoC platforms is an increasingly demanding task. A common approach is to use SystemC-based simulation to verify functionality and evaluate the performance using executable models. The downside of this approach is that developing SystemC models can be very time consuming, so we propose to use a coloured Petri net model to describe how IPs are interconnected and use SystemC to describe the IPs themselves. Our approach focuses on fast simulation and a natural way for the user to interconnect the two kinds of models. We demonstrate our approach using a prototype, showing that the cosimulation indeed shows promise.

Keywords: SystemC, coloured Petri nets, cosimulation, System on Chip

1. INTRODUCTION

Modern chip design for embedded devices is often centered around the concept of *System on Chip* (SoC) as devices such as cell phones benefit from the progress of the semiconductor process technology. In these platforms, complex systems including components such as general-purpose CPUs, DSPs (digital sound processors), audio and video accelerators, DMA (direct memory access) engines, graphics accelerators and a vast choice of peripherals, are integrated on a single chip. In Fig. 1, we see an example of an SoC, namely Texas Instruments' OMAP44x architecture (Texas Instruments 2009), which is intended for, e.g., mobile phones. Each of the components, called *Intellectual Property blocks* (IPs), can be contributed by separate companies or different parts of a single company, but must still be able to work together. The IPs are designed to be low-power and low-cost parts and often have intricate timing requirements, making the functional verification of such systems increasingly difficult. Therefore the IPs are modeled using an executable modeling language and simulation based validation is performed to ensure that, e.g., the multimedia decoder can operate fast enough to decode an incoming stream before it is sent to the digital-to-analog converter for playback.

When an IP is purchased for inclusion in an SoC, one often obtains a model of the component for inclusion in a whole-system simulation. Such a model is often created using SystemC (IEEE-1666), an industry-standard for creating models based on an extension of C++. SystemC supports simulation based analysis and is well-suited for making models that deal with intricate details of systems, such as electronic signals. SystemC has a couple of weaknesses as well, as it has no formal semantics and therefore is not well-suited for performing formal verification. Furthermore, SystemC is not very well-suited for modeling in a top-down approach where implementation details are deferred until they are needed, and SystemC is inherently textual, making it difficult to get and idea of, e.g., which parts of the chip are currently working or idle, unless a lot of post-processing of simulation results is performed. All of these traits make it tedious and time consuming to create models in SystemC, which postpones the moment where the modeling effort actually pays off by revealing problems in the design.

The coloured Petri nets formalism (CP-nets or CPNs) (Jensen and Kristensen 2009) is a graphical formalism for constructing models of concurrent systems. CP-nets has a formal semantics and can be analyzed using, e.g., state-space analysis (also known as state enumeration, reachability analysis, and model-checking) or invariant analysis. CPN models provide a high level of abstraction and a built-in graphical representation that makes it easy to see which parts of the model that currently process data. The draw-back of CP-nets is that it is not very well-suited for low-level processing as it has to be done either as graphical notation or directly as programming. Also, since many IP modules obtainable are modeled using SystemC, double effort has to be put into making models of the obtained IPs or translating the CPN model to SystemC for simulation along with the purchased models.

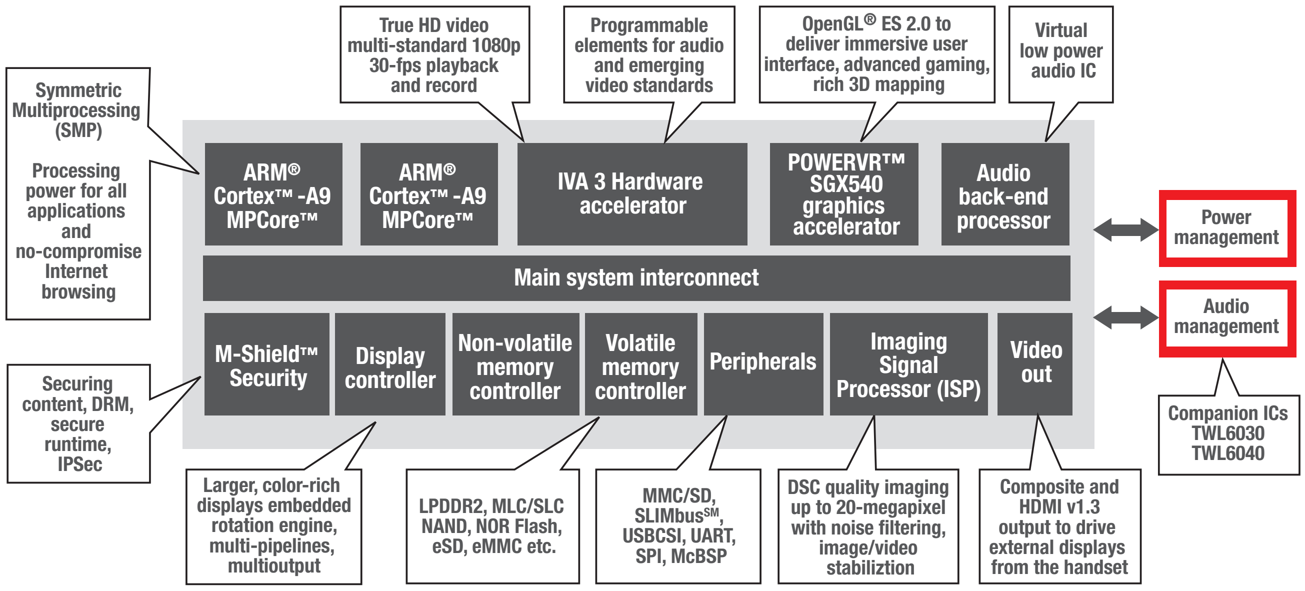
Symmetric Multiprocessing (SMP)
Processing power for all applications and no-compromise Internet browsing
True HD video multi-standard 1080p 30-fps playback and record
Programmable elements for audio and emerging video standards
OpenGL® ES 2.0 to deliver immersive user interface, advanced gaming, rich 3D mapping
Virtual low power audio IC
ARM® Cortex™ -A9 MPCore™
ARM® Cortex™ -A9 MPCore™
IVA 3 Hardware accelerator
POWERVR™ SGX540 graphics accelerator
Audio back-end processor
Power management
Main system interconnect
Audio management
M-Shield™ Security
Display controller
Non-volatile memory controller
Volatile memory controller
Peripherals
Imaging Signal Processor (ISP)
Video out
Companion ICs TWL6030 TWL6040
Securing content, DRM, secure runtime, IPSec
Larger, color-rich displays embedded rotation engine, multi-pipelines, multioutput
LPDDR2, MLC/SLC NAND, NOR Flash, eSD, eMMC etc.
MMC/SD, SLIMbus℠, USBCSI, UART, SPI, McBSP
DSC quality imaging up to 20-megapixel with noise filtering, image/video stabiliztion
Composite and HDMI v1.3 output to drive external displays from the handset

Figure 1: Block diagram of Texas Instruments' OMAP44x platform.

During development of the next generation SoC, some IPs were modeled using coloured Petri nets instead of SystemC. Due to the next generation being work in progress, we cannot go into further details about the specifics of the model nor the modeled architecture, but we can sum up some the experiences with using CPN models for SoC modeling and verification. Firstly, a CPN model has been made faster than a corresponding SystemC model, making it possible to catch errors earlier in the process and increase confidence in the new architecture. The model made it possible to catch a functionality error, and subsequent performance simulation provided input to making reasonable trade-offs between implementation of some sub-blocks in hardware or software. All in all, the model did provide interesting insights for a real-life example. Unfortunately, the model also had limitations. The biggest limitation is that the performance of the connection between the modeled block and the memory subsystem could not be evaluated even though a cycle accurate model of the memory system was available in SystemC.

We see that CPN models and SystemC models complement each other very well; one language's weaknesses are the other language's strengths. It would therefore be nice to be able to use the IP models in SystemC with a more high-level model created using CP-nets. In this way it is possible to have the SystemC models specify the low levels of the model and graphically compose the IPs using CP-nets, allowing us to have a high-level view of which IPs are processing during the simulation. In this paper we describe an architecture for doing this by running a number of CPN simulation kernels in parallel with a number of SystemC simulation kernels, what we call a *cosimulation*.

The reason for introducing our own kind of cosimulation instead of relying on, e.g., the High-Level Architecture (HLA) (IEEE-1516), is mainly due to speed of development and speed of execution; please refer to Sect. 3 for a more detailed discussion.

The rest of this paper is structured as follows: First, we briefly introduce SystemC and CP-nets using a simple example, in Sect. 3 we present the algorithm used to cosimulate models, and in Sect. 4 we describe a prototype of the cosimulation algorithm, our experiences from the prototype, and propose an architecture of a real implementation. Finally, in Sect. 5, we sum up our conclusions and provide directions for future work.

2. BACKGROUND

In this section we introduce the formalisms SystemC and coloured Petri nets using an example of a simple stop-and-wait communication protocol over an unreliable network. It is not crucial to understand the details of the languages nor the example, but just to give an impression of models and their communication primitives.

2.1. Coloured Petri Nets

At the top level (Fig. 2) the model consists of three modules, a Sender, a Receiver, and a Network. Before we explain the top level, let us look at the implementation of the receiver as a CPN model (Fig. 3). The Receiver consists of four *places* (named B, C, NextRec, and Received) with *types* written below them (INTxDATA for B, INT for C and NextRec, and DATA for Received) and one *transition* (named Receive Packet). Places can contain *tokens* written in rectangles above the places (in this example, NextRec contains one token with the value 2 and Received contains one token with the value "CP-"). C contains no tokens and B contains two tokens with the more complex values (1, "CP-") and (2, "net"). Tokens can have *time stamps* written after the @-sign to the right of the value of the token (in this example, the token (1, "CP-") on B has the time stamp 69, 2 on NextRec has time stamp 113, and "CP-" on Received

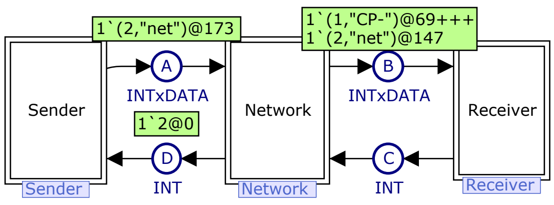

Figure 2: Top level of network protocol.

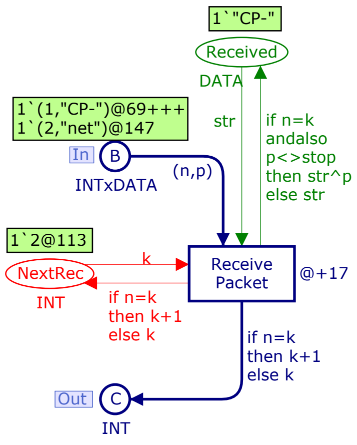

Figure 3: Receiver of network protocol.

has no time stamp). There are *arcs* between places and transitions. An arc from a place to a transition indicates that in order for the transition to be *enabled* tokens with the appropriate values must be available on the place(s) (called *input places* of the transition). When a transition is enabled, it can *occur*, and the result is that tokens are removed from its input places and tokens are produced on the places connected via arcs from the transition (the *output places*). Arcs contain expressions that describe which tokens can be moved, but we will not go into detail about that here. A CPN model has a global clock that governs when tokens are available; they are available if their time stamp is less than or equal to the value of the global clock. Transitions can have inscriptions after the @+-sign next to the transition that describe how long they take to occur (here Receive Packet takes 17 time units to occur). This influences the time stamp of produced tokens.

The gist of the receiver is that a packet (a pair of a packet number and some data) arrives on B, NextRec contains an internal counter that keeps track of which packet we expect, and Received contains the data we have received until now. In this case, Receive Packet is enabled. When it occurs, it consumes the packet from B. The serial number of the received packet (n) is compared with the number of the next expected packet (k). If the packet is the expected one, we append the data to Received, increment NextRec, and transmit an acknowledgement for the next packet onto C. If we receive another packet, we drop the packet, keep the value of the NextRec counter, and send an acknowledgement for the expected packet.

The top level of the protocol is shown in Fig. 2. It consists of four places and three *substitution transitions*, Sender, Network, and Receiver, drawn using a double outline. The idea is that a substitution transition represents a module. Here, the Receiver substitution transition represents the receiver module from Fig. 3. Communication between the top level and the module uses *port places*; in Fig. 3 we have two port places, B and C. We can distinguish port places from normal places by the inscription saying either In (on B) or Out (on C), which specifies that information flows into or out of the module. When a token is produced on the B place in Fig. 2, it appears on the B port place in Fig. 3; in the example, we see that both B places indeed have the same marking.

We will not go into details about the implementation of the Network module, but merely say that it implements a bidirectional network link that can drop and delay packets.

2.2. SystemC

We wish to model the Sender module using SystemC. SystemC models consist of modules that are organized in a hierarchy. Modules have interfaces consisting of ports that can be connected to other ports using channels. In Listing 1, we see a very simplistic SystemC version of the sender. We define a module Sender (l. 4) and give it two ports, p_in and p_out (ll. 5–6). The sender has some local data, a variable nextPacket (l. 39) for keeping track of which packet to send next, and an array of all packets we intend to send, allMes (l. 40). These are set up in the constructor (ll. 9–12), where we also indicate (ll.14–15) that our module has two threads, SendPacket, responsible for transmitting packets, and ReceiveAck, responsible for receiving and processing acknowledgements. We indicate that we are interested in being notified when data arrives on p_in (l. 16). The SendPacket thread (ll. 19–26) loops through all packets, writing them to p_out and delaying for sendDelay time units between transmitting each packet. The ReceiveAck thread (ll. 28–37) receives acknowledgements from p_in and updates nextPacket, so the next packet is transmitted. We see that the model basically is C++ code and despite its simplicity still comprises over 40 lines of code. We would normally split the code up in interface and implementation parts, but have neglected to do so here in order to keep the code simple.

We need to set up a complete system in order to run our sender. In Listing 2, we see how such a setup could look like. We basically have a module Top (l. 6) which is a simplified version of the top level in the CPN model (Fig. 2). The top module sets up two channels (ll. 7–8), D and A (using the same names as in Fig. 2). The constructor initializes the sender and receiver test bench (l. 13) and connects the ports via channels (ll.14–17). The main method initializes the top level (l. 22) and starts the simulation (l. 23).

Now, our goal is to use the code in Listing 1 as the sender module in the CPN top level (Fig. 2) with the CPN implementation of the network (not shown) and the CPN version of the receiver (Fig. 3) in a single simulation run.

3. ALGORITHM

As our primary goal is to be able to simulate real-life System-on-Chip (SoC) systems, which are typically modeled on the nanosecond scale, we need to be able to perform very fast simulation, and it is not feasible to tightly synchronize the CPN and SystemC parts of the model if we wish to simulate several seconds of activity. Instead, we try to only synchronize models when needed, i.e., when one part has done everything it can do at one time stamp and needs to increment its clock, or whenever information is exchanged. In the following we will refer to CPN and SystemC simulation kernels as *components* in cosimulations.

Aside from requiring loose coupling between the components, we prefer a truly distributed algorithm in order to avoid having to rely on a coordinator. As a goal of the project is to find out whether CPN/SystemC cosimulation is possible, feasible, and can actually benefit modeling, we also want to do relatively fast prototyping. For these reasons, we decided to make our own implementation of cosimulation instead of using an off-the-shelf technology such as HLA. HLA enforces a stricter synchronization than we need, so by making our own implementation, we believe we can achieve better performance. Furthermore, implementing a generic HLA interface for CPN models is a non-trivial task, and does not satisfy our requirement of fast development. Finally, HLA relies on coordinators which conflicts with our desire for a distributed algorithm.

Listing 1: Sender.h

```cpp
#include "systemc.h"
#include "INTxDATA.h"

SC_MODULE (Sender) {
  sc_port<sc_fifo_out_if<INTxDATA> > p_out;
  sc_port<sc_fifo_in_if<int> >  p_in;

  SC_CTOR(Sender) {
    nextPacket = 1;
    for (int i = 0; i < 2; i++)
      allMes[i].no = i + 1;
    allMes[0].mes = "CP-"; allMes[1].mes = "net";

    SC_THREAD(SendPacket);
    SC_THREAD(ReceiveAck);
    sensitive << p_in;
  }

  void SendPacket(void) {
    sc_time sendDelay = sc_time(9,SC_NS);

    while (nextPacket < 3){
      wait(sendDelay);
      p_out->write(allMes[nextPacket-1]);
    }
  }

  void ReceiveAck(void) {
    sc_time ackDelay = sc_time(7,SC_NS);
    int newNo;

    while (true){
      newNo = p_in->read();
      wait(ackDelay);
      nextPacket = newNo;
    }
  }
private:
  int nextPacket;
  INTxDATA allMes[2];
};
```

Listing 2: sc_main.cpp

```cpp
#include <systemc.h>
#include "Sender.h"
#include "ReceiverTestBench.h"
#include "INTxDATA.h"

SC_MODULE (Top) {
  sc_fifo<int> D;
  sc_fifo<INTxDATA> A;

  Sender S;
  ReceiverTestBench RTB;

  SC_CTOR(Top): S("S"), RTB("RTB") {
    S.p_out(A);
    S.p_in(D);
    RTB.p_in(A);
    RTB.p_out(D);
  }
};

int sc_main(int argc, char* argv[]) {
  Top SenderReceiver("SenderReceiver");
  sc_start();
  return 0;
}
```

Our algorithm is shown as Algorithm 1. Basically, it runs two nested loops (ll. 2–6 and 3–5). The inner loop executes steps locally as long as possible at the current time. A step is an atomic operation dependent on the modeling formalism; for CPN models a step is executing a transition and for SystemC a step can be thought of as executing a line of code (though the real rule is more complex). The inner loop also sends outgoing information to other components and receives information from other components (here we have shown a single-threaded implementation that exchanges information after every step, but we can of course make a multi-threaded version or only exchange information when it is no longer possible to make local steps). When we can make no more steps locally, we find the allowed time increase by calculating the global minimum of requested time increases from all components.

Algorithm 1 The Cosimulation Algorithm

```
1: $Time \leftarrow 0$
2: while true do
3:     while LOCALSTEPISPOSSIBLEAT( $Time$ ) do
4:         EXECUTEONESTEPLOCALLY()
5:         SENDANDRECEIVE()
6:     $Time \leftarrow$ DISTRIBUTEDGLOBALMIN(
                                DESIREDINCREASE() )
```

We note that exchange of information takes place without global synchronization. Participants simply communicate directly and if incoming information causes components to be able to execute more local steps they just do so, and reevaluate how much they want to increment time. This means that our synchronization algorithm does not have to deal with information exchange.

4. EVALUATION

Naturally, Algorithm 1 needs to be implemented for each kind of simulation kernel we wish to be able to use for cosimulation. Our primary goal is to make implementations of CP-nets and SystemC models, but the algorithm is general and can in principle be implemented for any timed executable formalism. In order to evaluate Algorithm 1 and whether CPN/SystemC cosimulation is feasible, we have developed a prototype to show that is is possible to integrate the two languages but also to show that it is possible to make the integration without (or with very few) changes to the SystemC kernel, as there are multiple vendors with different implementations.

Algorithm 1 does not specify how we calculate the global minimum required for synchronization. It is possible to do this without imposing any restrictions on the network structure, e.g., by using flooding, but making assumptions allows a much easier and faster implementation. As both CP-nets and SystemC models are naturally structured hierarchically with components containing nested components, optionally in several layers, making the assumption that components are structured in a tree is no real restriction.

The architecture of our prototype can be seen in Fig. 4. We first look at the static architecture from the top of Fig. 4. The prototype consists of three kinds of processes: a SystemC simulation kernel (left), an extended version of the of the ASCoVeCo State-space Analysis Platform (ASAP) (Westergaard, Evangelista, and Kristensen 2009), which contains a library called ACCESS/CPN (Westergaard and Kristensen 2009), making it easy to interact programmatically with the CPN simulator of CPN Tools (Ratzer, Wells, Lassen, Laursen, Qvortrup, Stissing, Westergaard, Christensen, and Jensen 2003) (right). The yellow/light gray boxes are already part of a standard SystemC simulation kernel, ASAP, or CPN Tools' simulator process and therefore does not have to be built from scratch. We have added a Cosimulation layer on top of the SystemC model. The cosimulation layer basically provides stubs for modules that are external (such as a CPN model or another SystemC model). Like with Remote Procedure Call (RPC) (Srinivasan 1995) systems, the stub module looks like any other module to the rest of the system and takes care of communicating with the other components. In the example in Sect. 2, the stub would consist of an implementation of ReceiverTestBench referred to Listing 2 and the cosimulation layer would consist of code like Listing 2 along with a communication library. Currently, we need to write such stubs manually, but we are confident that stubs can be generated automatically. The stub communicates using ONC-RPC (Srinivasan 1995) (formerly known as Sun RPC and available on all major platforms) with a SystemC cosimulation job in the middle ASAP process. For easy prototyping, we have selected to do most of the implementation in Java (ASAP is written in Java) rather than directly in SystemC (extension of C++) and CPN (written in Standard ML). The

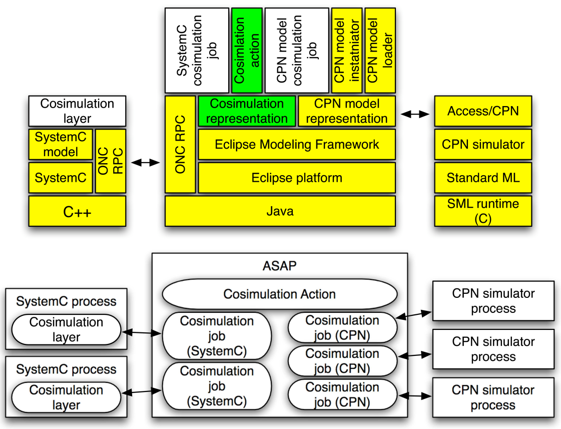

Figure 4: The conceptual architecture (top) and run-time architecture (bottom) of the prototype

SystemC cosimulation job and CPN cosimulation job from the ASAP process contain most of the implementation of Algorithm 1 specific for SystemC and CPN models. Between those, we have a Cosimulation action, which takes care of starting and connecting the correct components based on a Cosimulation representation which describes which components to use and how to compose them. We believe that the cosimulation action and cosimulation representation (marked in green, dark gray) can later be reused in a subsequent real implementation. Access/CPN abstracts away the communication between the ASAP and CPN simulator process, so we do not have to make changes to the CPN simulator.

At run-time, a cosimulation looks like Fig. 4 (bottom). Each rectangle is a running process, and each rounded rectangle is a task running within the process, corresponding to the blocks from the static architecture. We see that all simulators are external and can run on separate machines. We have implemented Algorithm 1 within the ASAP process (this in particular means that the distributed algorithm runs within one process). We have implemented the algorithm in full generality using channel communication only, but as we were not concerned with speed in our prototype, decided against setting up a truly distributed environment in the prototype.

The only non-trivial part is how to do the minimum calculation in line 6 of Algorithm 1. Our algorithm uses that the components are organized in a tree, and we will use normal tree terminology (root, parent, and children). Naturally, each node knows how many children it has and its parent. The idea is that each node requests a time increase from its parent. The parent then returns the allotted time increase. When a node wants to increase time, it waits for all its children to request a time increase. It takes the minimum of all of these votes (including its own) and requests this time increase from its parent. When it receives a response from the parent, it announces this increase to all children. The root just announces to all children without propagating to its (non-existing) parent. The algorithm can be improved in various ways. For

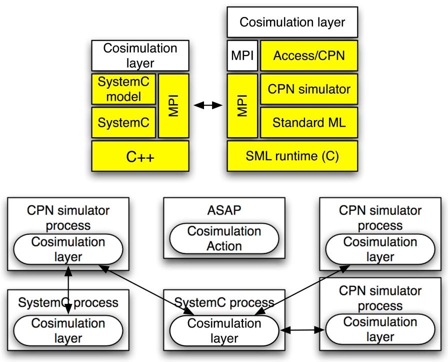

Figure 5: The conceptual architecture (top) and run-time architecture (bottom) of real implementation

example, as soon as a node realizes that only the minimum time increase can be granted (0 or 1 depending on whether we allow requesting a zero time increase), it can just announce the result to all children and continue propagating up in the tree. Also, a parent node need not actually announce the lowest time increase. It can announce the time increase requested by the node that has the second lowest request minus one, and sub-trees can then autonomously proceed (knowing that other sub-trees will not be able to proceed as they cannot receive data since information flows only up and down the tree).

For a real implementation we propose the much simpler architecture from Fig. 5. In this architecture, we have removed the centralized process and instead made real implementations of the Cosimulation layer for both SystemC and CPN. We have also replaced ONC-RPC with Message Passing Interface (MPI) (Message Passing Interface Forum 1997) which is an industry standard for very fast communication between distributed components. In order to use MPI, we will have to embed a standard MPI implementation into the SML run-time and add code to interface with that from SML code. The run-time behaviour is as one would expect. Instead of having the communication being mediated by ASAP, ASAP now just sets up a cosimulation, and the components communicate directly with each other, and ASAP can process the results after simulation.

One of our design goals was that we did not want to change the SystemC kernel. Instead, we have created a cosimulation layer as a regular SystemC process, so our prototype shows that this is feasible. For efficient implementation we probably need to augment the CPN simulator, but that is less of a problem, since we have control over it. Our implementation shows that the algorithm has the potential to provide efficient cosimulation and that it is possible to get meaningful results from the components of the model (currently we just extract log files, but it should be easy to map these back to the models, mostly for the CPN models to get graphical feedback). As a bonus, our prototype shows that it is possible to do reasonable distributed simulation of CPN models.

5. CONCLUSION AND FUTURE WORK

In this paper we have described an algorithm for cosimulation of CPN and SystemC models for verification of SoC platforms. The algorithm allows loose coupling between different simulation kernels, which we believe provide faster simulation. We believe that the prototype looks interesting and is worth pursuing further. The prototype has even provided some unforeseen benefits, such as distributed simulation of CPN models.

We have already mentioned that the High Level Architecture (HLA) provides similar features. HLA, however, has some problems that we wish to avoid. The main reason for not using HLA for the prototype was that it is too complex to get working code up and running fast. We also believe that the relatively tight coupling between the components of HLA is undesirable for performance reasons. Finally, our proposed system is completely distributed, whereas HLA has centralized components. Our distributed approach allows more decoupling, improves scalability, and provides a simpler implementation.

Future work includes making a real implementation as proposed in the previous section. We have not currently implemented all of the optimizations to the distributed minimum calculation described in the previous section, and these should be implemented and evaluated. It would be interesting to take compare a real implementation with an implementation using HLA for cosimulation of CPN and SystemC models, which would require making an implementation of HLA for CPN models. Finally, we have until now only dealt with simulation of composite models. It would be interesting to also look at verification, e.g., by means of state-spaces, which seems quite promising as modular approaches for CP-nets perform very well when systems are loosely synchronized, which is indeed the case here.

ACKNOWLEDGEMENTS

This work is in part supported by the Danish Research Council for Technology and Production.

REFERENCES

IEEE-1516, Modeling and Simulation High Level Architecture.

IEEE-1666, IEEE Standard System C Language Reference Manual.

Jensen, K. and Kristensen, L.M., 2009, *Coloured Petri Nets – Modelling and Validation of Concurrent Systems*, Heidelberg: Springer-Verlag.

Message Passing Interface Forum, 1997, *MPI-2: Extensions to the Message-Passing Interface*, http://www.mcs.anl.gov/research/projects/mpi/mpi-standard/mpi-report-2.0/mpi2-report.htm [Accessed 8 June 2009]

Ratzer, A.V., Wells, L., Lassen, H.M., Laursen, M., Qvortrup, J.F., Stissing, M., Westergaard, M., Christensen, S., and Jensen, K., 2003, CPN Tools for Editing, Simulating, and Analysing Coloured Petri Nets, *Proceedings of the Petri Net Conference*, pp. 450–462, June, Eindhoven, The Netherlands.

Srinivasan, R., 1995, *RPC: Remote Procedure CAll Protocol Specification Version 2*, RFC 1831.

Texas Instruments, 2009, *OMAP^{TM} Applications Processors: OMAP^{TM} 4 Platform*, http://www.ti.com/omap4 [Accessed 8 June 2009]

Westergaard, M., Evangelista, S., Kristensen, L., 2009, ASAP: An Extensible Platform for State Space Analysis, *Proceedings of the Petri Net Conference*, pp. 303-312, June, Paris, France.

Westergaard, M., Kristensen, L., 2009, The Access/CPN Framework: A Tool for Interacting With the CPN Tools Simulator, *Proceedings of the Petri Net Conference*, pp. 313-322, June, Paris, France.

AUTHORS BIOGRAPHY

Michael Westergaard is a PostDoc at Aarhus University, Denmark, where he obtained his PhD in Computer Science in 2007. He is working with modeling and analysis of concurrent systems using coloured Petri nets, and in particular analysis by means of state space exploration. He is involved in development of tools as well as algorithms to facilitate state space exploration for real-life systems, in particular CPN Tools and the ASAP model checking platform, and the ComBack state space method.

Lars Michael Kristensen is Professor in Computer Engineering at Bergen University College. He has more than 10 years of research experience in development and application of formal methods to concurrent systems. A main focus has been the theoretical foundation of state space methods and their implementation in computer tools, in particular in the context of Coloured Petri Nets. Important research contributions have been the development of the sweep-line state space space method, the development of two variants of the stubborn set partial-order method, and the development of the ComBack method. A strong focus has also been on evaluating the research results in industrial cooperation projects. He acted as researcher in the development of the state space tools of Design/CPN and CPN Tools, and he is the scientific leader of the ASCoVeCO project in the context of which the ASAP model checking platform is being developed. He has recently co-authored a new textbook on Coloured Petri Nets published by Springer-Verlag. In 2007 he received the Danish Independent Research Councils' Young Researcher's Award.

Maija Kuusela received her M.Sc. degree in Information Technology from Helsinki University of Technology, Finland in 1981 and her PhD degree in Mathematics from Duke University, North Carolina in 1986. During her professional carrier she has held positions at Nokia Research Center in Helsinki, Finland and at Texas Instruments France. Her current post as systems architect at OMAP Platforms Business Unit at Texas Instruments France includes modeling and performance evaluation of multiprocessor platforms and multimedia accelerator architectures.