A Prototype for Cosimulating SystemC and Coloured Petri Net Models

M. Westergaard[1]*, L.M. Kristensen[2], and M. Kuusela[3]

[1] Department of Computer Science, Aarhus University, Denmark.
Email: mw@cs.au.dk
[2] Department of Computer Engineering, Bergen University College, Norway.
Email: lmkr@hib.no
[3] OMAP Platforms Business Unit, Texas Instruments, Villeneuve-Loubet, France.
Email: m-kuusela@ti.com

Abstract. Semiconductor technology miniaturization allows designers to pack more and 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 validate functionality and evaluate the performance using executable models. The downside of this approach is that developing SystemC models can be very time consuming. We propose to use a coloured Petri net model to describe how IPs are interconnected and use SystemC models 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.

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 signal processors), audio and video accelerators, DMA (direct memory access) engines 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 [14], 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 they 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

*Supported by the Danish Research Council for Technology and Production.

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.

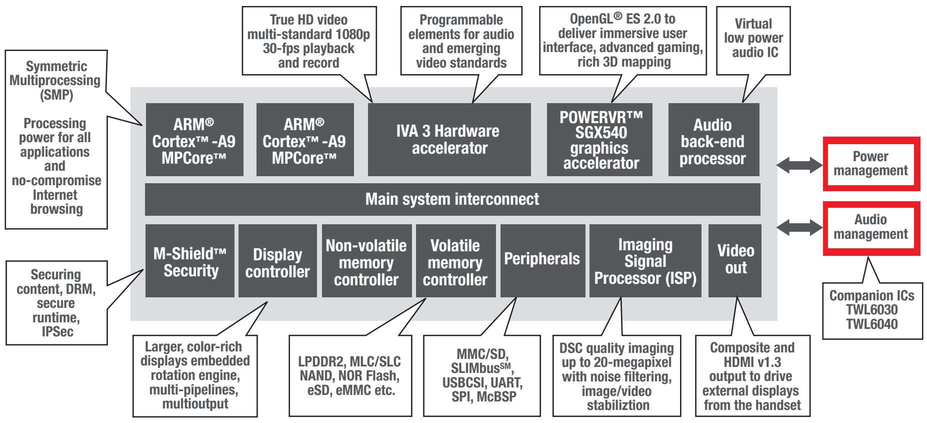

True HD video multi-standard 1080p 30-fps playback and record
Programmable elements for audio and emerging video standards
OpenGL® ES 2.0 to deliver immersive user interface, advanced gaming, rich 3D mapping
Virtual low power audio IC
Symmetric Multiprocessing (SMP)
Processing power for all applications and no-compromise Internet browsing
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
Securing content, DRM, secure runtime, IPSec
Companion ICs TWL6030 TWL6040
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

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

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 [5], 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 can semi-automatically be translated directly to microcode or even electrical circuits, making it possible to obtain an implementation of the final chip directly from the model. SystemC has weaknesses as well, as it has no formal semantics and therefore is not well-suited for performing formal verification. Furthermore, SystemC is not 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 an 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) [6] 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 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 main drawback of CP-nets is that the formalism is not widely used in the industry, meaning that only little expertise and few pre-existing IP models exist. In order to switch to using CP-nets for SoC modeling, one would have to make models of the obtained IPs or translating the CPN model to SystemC for simulation along with the IP models.

During development of the next generation SoC at Texas Instruments, some IPs were modeled with coloured Petri nets using CPN Tools [3, 12] instead of SystemC. Due to the next generation SoC 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 initial experiences with using CPN models for SoC modeling and verification. Firstly, a CPN model can be constructed 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 constructed 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 CPN 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 without doubling the effort put into the SystemC model of the memory system.

The above shows 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 created using 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 simulator in parallel with a number of SystemC simulators, what we call a *cosimulation*.

The reason for introducing our own notion of cosimulation instead of relying on, e.g., the High-Level Architecture (HLA) [4, 11], is mainly due to speed of development and a wish for a more decoupled architecture, which hopefully leads to faster 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 for a production-quality implementation. Finally, in Sect. 5, we sum up our conclusions and provide directions for future work.

An earlier version of this paper has been published as [17]. The changes made in this revision is that Sect. 2 has been rewritten for people with background in CP-nets. Section 3 has been expanded with Algorithm 2 and an improved description of Algorithm 1. Section 4 has been rewritten to tie the description of the architecture better to the algorithms from Sect. 3. Section 4 has also been expanded with a screenshot and a more detailed description of our prototype.

2 Background

In this section we introduce an example model of a simple stop-and-wait communication protocol over an unreliable network. We will use this example throughout the paper and introduce the SystemC formalism using the example. It is not crucial to understand the details of SystemC, but just to get an impression of SystemC models and their communication primitives.

2.1 Stop-and-wait Protocol CPN Model

We use the example hierarchical stop-and-wait protocol included with the CPN Tools distribution [3, 12]. We briefly recall the example with focus on how communication between the different pages takes place.

At the top level (Fig. 2) the model consists of three modules, a Sender, a Receiver, and a Network, here represented by *substitution transitions*. The sender sends packets via the network to the receiver. As the network can drop and deliver packets out of order, the sender attaches a sequence number to each packet and retransmits packets. The receiver acknowledges the receipt of packets to let the sender know when it is allowed to continue to the next packet. To make the example more interesting, we have attached a time stamp to each packet to allow us to simulate real world conditions, where packet delivery is not instantaneous, and where retransmission only takes place after a certain delay.

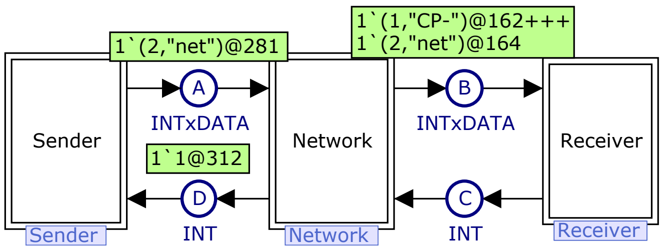


Fig. 2: Top level of network protocol.

Let us also take a closer look at the sender part of the model shown in Fig. 3 (left). We will not actually use a CPN version of the sender, but it may be useful to compare the CPN version with the SystemC version we present below. The sender is quite simple. The Send Packet *transition* reads a packet from the Send *place*, matches it against the NextSend counter, delays it for the amount of time read from Wait and transmits the packet to the out-buffer on A. When Receive Ack receives an acknowledgement from D, it updates the NextSend counter. Sending a packet takes 9 time units and processing an acknowledgement takes 7 time units. Figure 3 (right) shows the situation after Send Packet has been executed. The A and D places of the sender are *port places* (or just ports) that are assigned to the *socket places* (or just sockets) with the same names on the top page in Fig. 2. Ports are marked by a *port tag* showing the direction information flows (into and out of the sender module). Whenever a token is produced on or consumed from a port or socket place, it is also produced/consumed on the corresponding socket/port place, and we note that port places and the corresponding socket places indeed have the same markings in the right part of Fig. 3 (e.g., 1‘(2, "net")@281 on both A places), but apart from the shared names nothing in the graphical representation shows which ports correspond to which sockets.

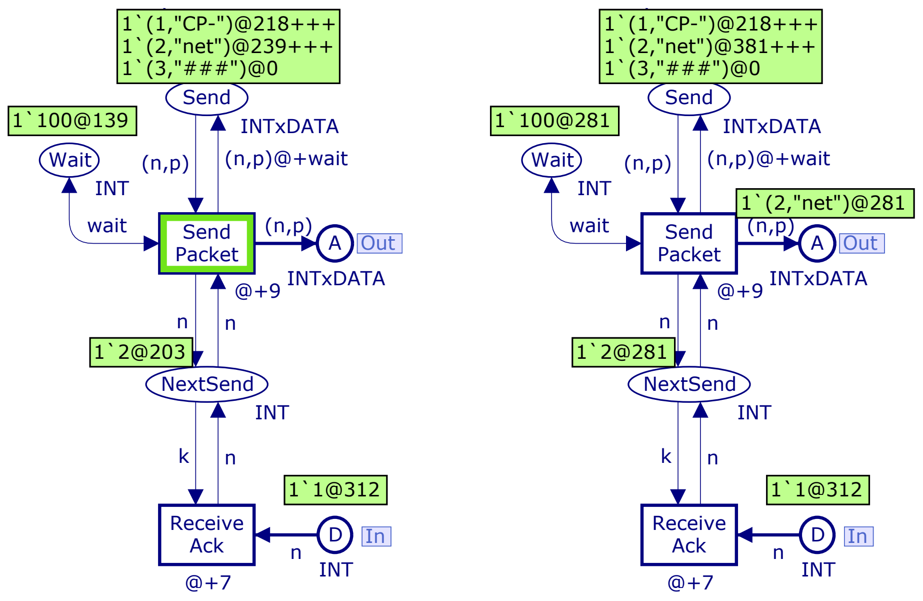


Fig. 3: Sender of network protocol before executing Send Packet (left) and after (right).

We will not go into details about the Network and Receiver modules; they are CPN modules that implement the aforementioned operations. The interested reader is invited to look at the model distributed with CPN Tools.

While neither of the modules shows it, the CPN model also has an associated *global clock*, which indicates what the current model time is in the execution of the model. The idea is that tokens are not available until the global clock reaches or exceeds the time stamp of the token; intuitively the execution of transitions take time and tokens are consumed immediately, but new tokes are only produced after the execution is done. In order to show this to the modeler, the token is shown immediately, but has an attached time stamp that indicates when it is available. In Fig. 3 (left) the global clock is 272 and in Fig. 3 (right) and Fig. 2 the global clock has the value 281 (the change is that Send Packet transmitted another copy of packet number 2 at time 272 and the packet is available 9 time units later, at time 281, due to its @+9 inscription).

2.2 SystemC

We wish to model the Sender module using SystemC instead of the CPN module shown in Fig. 3. SystemC models, like CPN models, consist of modules organized in a hierarchy. Modules have interfaces consisting of ports that can be connected to other ports using channels (note that in SystemC both ends of such an assignment are called ports). Modules can execute C++ code. Like CPN models, SystemC models have a global clock which allows us to model delays in transmission.

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, a and d (ll. 5–6). The names used in this module correspond to the names used in Fig. 3 except we use C++ naming conventions. The sender has some local data, a variable nextSend (l. 43) for keeping track of which packet to send next, and an array of all packets we intend to send, send (l. 44). These are set up in the constructor (ll. 9–13), where we also indicate (ll.15–16) 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 d (l. 17). The sendPacket thread (ll. 20–29) loops through all packets, writing them to a and delaying for sendDelay time units between transmitting each packet. The receiveAck thread (ll. 31–40) receives acknowledgements from d and updates nextSend, 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), where we have essentially removed the network part and just tied the sender directly to the receiver. The top module sets up two channels (ll. 7–8), packets and acknowledgements. 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 implementations of the network and receiver (not shown).

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 synchronize the

Listing 1: Sender.h

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

SC_MODULE (Sender) {
  sc_port<sc_fifo_out_if<INTxDATA> > a;
  sc_port<sc_fifo_in_if<int> >  d;

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

    SC_THREAD(sendPacket);
    SC_THREAD(receiveAck);
    sensitive << d;
  }

  void sendPacket(void) {
    sc_time sendDelay = sc_time(9,SC_NS);
    sc_time waitDelay = sc_time(100,SC_NS);

    while (nextSend < 3){
      wait(sendDelay);
      a->write(send[nextSend-1]);
      wait(waitDelay);
    }
  }

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

    while (true){
      newNo = d->read();
      wait(ackDelay);
      nextSend = newNo;
    }
  }

private:
  int nextSend;
  INTxDATA send[2];
};
```

Listing 2: sc_main.cpp

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

SC_MODULE (Top) {
  sc_fifo<INTxDATA> packets;
  sc_fifo<int> acknowledgements;

  Sender S;
  ReceiverTestBench RTB;

  SC_CTOR(Top): S("S"), RTB("RTB") {
    S.a(packets);
    RTB.b(packets);
    S.d(acknowledgements);
    RTB.c(acknowledgements);
  }
};

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

CPN and SystemC parts of the model after each step if we wish to simulate several seconds of activity. Instead, we only globally synchronize the clocks of models when needed, i.e., when one part has done everything it can do at one moment in time and needs to increase its clock in accordance with the other parts. We synchronize models pairwise whenever information is exchanged (which may enable further events at the current model time). In the following we refer to CPN and SystemC simulator as *components* in cosimulations, synchronization of global clocks as synchronization or time synchronization, and pairwise synchronization in the form of sending or receiving data to/from other components as information exchange.

Aside from requiring loose coupling between the components, we prefer a truly distributed algorithm in order to avoid having to rely on a coordinator. One goal of this work is to find out whether CPN/SystemC cosimulation is possible and feasible and can actually benefit modeling, and therefore we 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 and demanding task, and does not satisfy our requirement of development without investing too many resources before the viability of the solution can be judged. Finally, HLA relies on coordinators which conflicts with our desire for a distributed algorithm.

Our algorithm for simulation of the individual components 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 model 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, dealing with synchronization points, such as information exchange and time synchronization). The inner loop also transmits information to or from other components (here we have shown a single-threaded implementation that exchanges information after every step – but of course only if there is information to exchange – but we can of make a multi-threaded version or only transfer 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 the time increase requests by all components in the cosimulation.

Algorithm 1 The Cosimulation Algorithm

```
1: time ← 0
2: while true do
3:     while localStepIsPossibleAt( time ) do
4:         executeOneStepLocally()
5:         sendAndReceive()
6:     time ← distributedGlobalMin( desiredIncrease() )
```

We note that exchange of information takes place without global synchronization. Participants simply communicate directly as described by the model structure 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 time synchronization algorithm does not have to deal with information exchange.

Naturally, Algorithm 1 needs to be implemented for each kind of simulator we wish to be able to use for cosimulation. Our primary goal is to make implementations for CPN and SystemC models, but the algorithm is general and can in principle be implemented for any timed executable formalism as long as the formalism uses a compatible concept of time, i.e., a global clock. In order to implement the algorithm, we need to provide implementations of localStepIsPossibleAt, executeOneStepLocally, and sendAndReceive. The first two will typically be trivial when given a simulator, as executing steps and querying whether it is possible to execute steps is the main functionality provided by a simulator. The difficult part is the implementation of sendAndReceive, which requires that we hook into the simulator in some way to find out when values to send are produced, translate the value into an exchange format (such as JSON (JavaScript Object Notation) [7] or XML described using XML Schema [1]) agreed upon by the simulators. We must resolve the destination component, either directly or from an external binding, and transmit the encoded data to the receiving component. Only the latter part can be done independently of the component modelling language. When a value is received, we need to translate the exchange format to a format understood by the simulators of the component and modify the state of the simulator correctly. Again, these steps needs to be done for each simulator. For CPN models we regard each token as an individual exchanged value; SystemC only allows transmitting one value at a time on a port (though the channel may have a buffer), so sendAndReceive simply has to exchange single values over a channel. In [9] we describe how to embed types from Java in CPN models by basically translating simple values directly, translating between lists of SML and Lists of Java, translating between JavaBeans and Java Maps, and SML records, and translating between union data types (datatypes in SML) and Java Enums. A similar approach can be used to translate between data types of SystemC and CPN models.

Algorithm 1 does not specify how we calculate the global minimum required for synchronization. As we need to use the time specified by the components of the models, we cannot use something like, e.g., Lamport timestamps [8] to perform our time synchronization as they are only useful for ordering events according to a causal ordering. We do not only care about causal ordering but also for slowing down or halting simulation of components if the other components have not yet advanced their clocks as information exchange may make it possible to execute actions earlier than what was possible without information exchange. Therefore Algorithm 1 synchronizes every time a component wishes to increase its time stamp.

It is possible to do time synchronization without imposing any restrictions on the network structure, e.g., by using flooding, but making certain assumptions allows a simpler and faster implementation. As both CPN 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. Here we give an algorithm for distributedGlobalMin of Algorithm 1 where we assume that components are ordered in a tree, and we use normal tree terminology (root, parent, and child). 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 entire algorithm is shown as Algorithm 2.

Algorithm 2 consists of two procedures, a workerThread and the actual distributedGlobalMin procedure. We assume that each component has

Algorithm 2 DISTRIBUTEDGLOBALMIN for tree-structured components

```
 1: $ready \leftarrow$ false
 2: $requests \leftarrow$ QUEUE.EMPTY()
 3: $results \leftarrow$ QUEUE.EMPTY()
 4:
 5: **proc** WORKERTHREAD() **is**
 6:   **while** true **do**
 7:     $minRequest \leftarrow \infty$              // collect requests from children
 8:     **for** $i = 1$ to $children$.SIZE() $+ 1$ **do**
 9:       $minRequest \leftarrow min(minRequest, requests$.REMOVEHEAD())
10:     **if** $this = root$ **then**
11:       $result \leftarrow minRequest$              // we know the result locally
12:     **else**
13:       // propagate our request
          $result \leftarrow parent$.DISTRIBUTEDGLOBALMIN( $minRequest$ )
14:     **for** $i = 1$ to $children$.SIZE() $+ 1$ **do**
15:       $results$.ADD( $result$ ) // distribute time increase to children
16:     $ready \leftarrow$ true
17:
18: **proc** DISTRIBUTEDGLOBALMIN( $vote$ ) **is**
19:   **while** $ready$ **do**
20:     SKIP()          // wait for any previous ongoing calculations
21:   $requests$.ADD( $vote$ )
22:   **while** $\neg ready$ **do**
23:     SKIP()                        // wait for calculation to complete
24:   $result \leftarrow results$.REMOVEHEAD()
25:   **if** $results$.ISEMPTY() **then**
26:     $ready \leftarrow$ false   // if we are last, signal calculation is over
27:   **return** $result$
```

started a single instance of WORKERTHREAD in a separate thread. We also assume that calls to a parent component's DISTRIBUTEDGLOBALMIN conceptually happens in the thread of the caller (the parent starts a separate thread to handle each child or the child call communicates directly with the WORKERTHREAD of the parent). All variables defined in ll. 1–3 live in the context of the WORKERTHREAD. Now, the idea is that the calculation in each component has two stages: gathering of requests (a result is not ready for queries) and distribution of replies (a result is ready).

The *ready* variable keeps track of which stage we are currently in, initially gathering of requests (l. 1). We gather requests in a queue, which is initially empty (l. 2). When a child or the component itself (from l. 6 in Algorithm 1) calls DISTRIBUTEDGLOBALMIN, we first wait until we are in the request gathering stage (ll. 19–20). We then add our request to the queue of requests (l. 21). We then wait until replies are ready (ll. 22–23), and read the result (l. 24). If there are no more results available (l. 25), we indicate that (l. 26), and return the result (l. 27).

Meanwhile, the workerThread calculates the minimum received request (ll. 7–9). It knows that it will receive exactly *children*.size() + 1 requests, one for each child and one for the component itself. If the current component is also the root component (l. 10), the result is just this calculated minimum (l. 11), the worker requests an increase from the parent node of exactly this minimum (l. 13). The workerThread then makes exactly *children*.size() + 1 copies of the result, one for each caller (ll. 15–16), switches to the result distribution stage (l. 14), and restarts for another calculation (l. 6).

The algorithm can be improved in various ways. For 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. This can be done around line 9 in Algorithm 2, if we realize that *minRequest* is equal to *time* (l. 1 in Algorithm 1). We also need to change how we wait for responses, so we do not wait in lines 22–23, yet keep the stage correctly in lines 25–26. This can be done by counting the number of results returned instead of relying on *results* to be empty.

Another improvement can be made by observing that 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 is exchanged only up and down the tree). Here we need to change the calculation in lines 7–9.

We can also exploit additional knowledge about individual components. For example a component (or component sub tree) with only output ports can be allowed to continue indefinitely, as their processing will never be influenced by the calculation of other components.

4 Evaluation

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. Furthermore, a goal is to show that it is possible to make the integration without (or with very few) changes to the SystemC simulator, as there are multiple vendors with different implementations.

4.1 Prototype Architecture

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 simulator (left), an extended version of the ASCoVeCo State-space Analysis Platform (ASAP) [15] (middle), and a CPN simulator (right) with a library called Access/CPN [16] for easy interaction with the simulator. The yellow/light gray boxes are standard components (ONC RPC, C++, Java, Eclipse Platform, Eclipse Modeling Framework, SML runtime, and Standard ML), already part of a standard SystemC simulator (SystemC and SystemC model), ASAP (CPN model representation, CPN model loader, and CPN model instantiator), or CPN Tools' simulator process (CPN simulator and Access/CPN) and therefore does not have to be built from scratch.

At the top middle of the ASAP process, 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. The cosimulation action and cosimulation representation (marked in green/dark gray) are independent of the simulator used. The cosimulation action basically implements code to set up Algorithm 1 and Algorithm 2 as described by a cosimulation representation. The representation describes the hierarchy of components, a mapping of interfaces to modelling language-specific features (port places and exported ports in the cases of CP-nets and SystemC), and how interfaces are connected to each other (corresponding to port/socket assignments in CP-nets and channels in SystemC).

The two cosimulation jobs SystemC cosimulation job and CPN cosimulation job implement Algorithm 1. They share a common Java implementation of Algorithm 1 and Algorithm 2 and are just specializations in terms of LOCALSTEPIS-

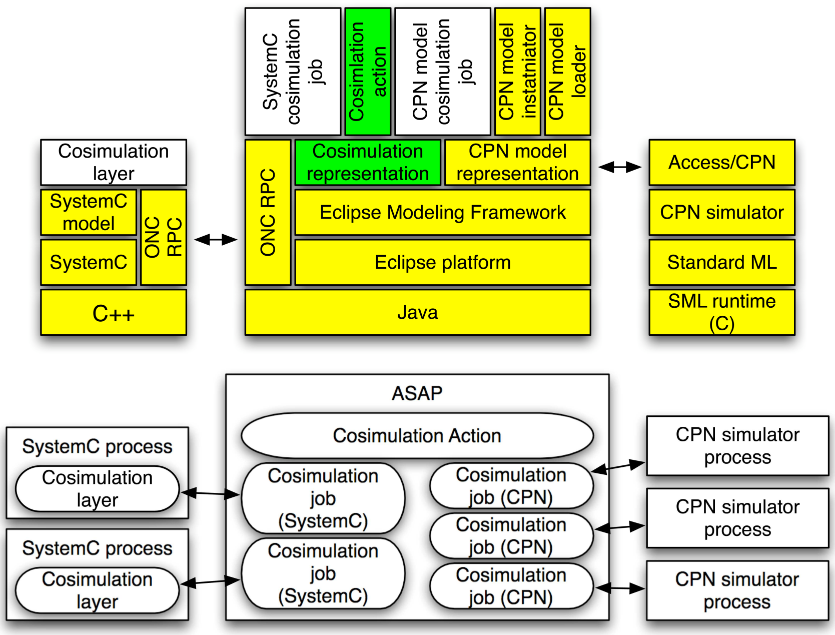


Fig. 4: The static architecture (top) and run-time architecture (bottom) of our prototype

PossibleAt, executeOneStepLocally, and sendAndReceive. The reason for this is that a common implementation allows us to do fast prototyping.

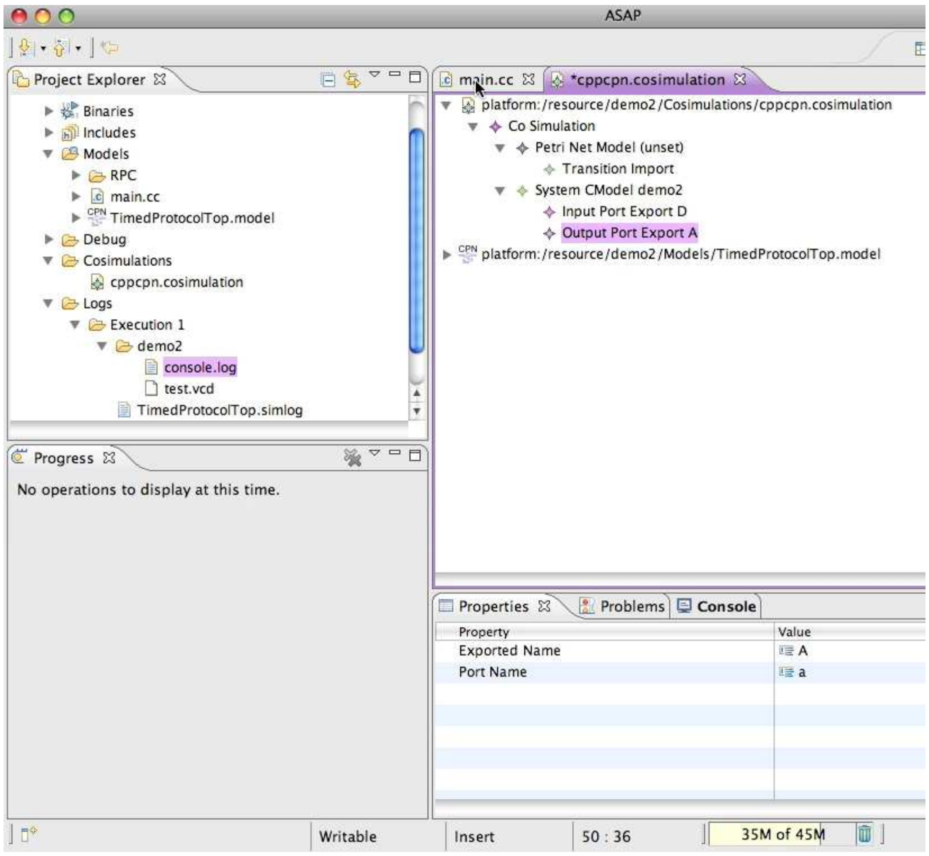
The CPN cosimulation job uses Access/CPN to implement the required function. Access/CPN allows us to check whether any transitions are enabled at the current model time (accounting for localStepIsPossibleAt), to execute a step (accounting for executeOneStepLocally) and to read and change the marking of all places, including port places (allowing us to implement sendAndReceive). All of this can be done in the ASAP process, so we do not have to change the CPN simulator process.

In order to implement the SystemC cosimulation job, we need to do a little more work, as we do not have something like Access/CPN available for SystemC. Instead, we have added a Cosimulation layer on top of the SystemC model, which basically plays the role of Access/CPN. The cosimulation layer provides stubs for modules that are external (such as a CPN model or another SystemC model) and implements the top level of the SystemC model (corresponding to Listing 2). Like with Remote Procedure Call (RPC) [13] systems, stub modules look like any other module to the rest of the system and takes care of communicating with other components. In the example in Sect. 2, the stub would consist of an implementation of ReceiverTestBench referred to in Listing 2 and the cosimulation layer would consist of code like Listing 2 along with a communication library. Currently, we need to write stubs and the top level code manually, but we are confident that both can be generated automatically, as the problem is very much like standard stub generation for RPC systems (we need to send/receive data, serialize it, and call the appropriate remote method). The stub communicates using ONC-RPC [13] (formerly known as Sun RPC and available on all major platforms) with the SystemC cosimulation job to implement localStepIsPossibleAt) and executeOneStepLocally (by communicating with the glue top level code) and sendAndReceive (by communicating with the stubs).

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 and Algorithm 2 within the ASAP process (this in particular means that the distributed algorithm runs within one process). We have implemented our algorithm in full generality using channel communication only, but as we were not overly concerned with speed in our prototype, decided against setting up a truly distributed environment.

4.2 Prototype

In Fig. 5 we see a screenshot from our prototype. The prototype runs on Linux and Mac OS X, and with a Windows version of ONC RPC port mapper also on Windows. The view basically consists of four parts, the project explorer at the top left, where we see all our models and related files, the progress area at the bottom left, where we see running components during execution, the editing area at the top right, where we describe our cosimulation jobs (as represented by a cosimulation representation), and an auxiliary area at the bottom right, currently showing properties of the currently selected object, but which can also show, e.g., the console of a running SystemC job.



Property	Value
Exported Name	A
Port Name	a

Fig. 5: Screenshot from prototype.

In the Project Explorer view, we see a top-level entries Binaries, Includes, and Debug, which are part of the C++ subsystem we have built our prototype upon (they contain compiled files, header files, and debugging files for SystemC models). A more interesting entry is Models, containing TimedProtocolTop.model, which contains the CPN model from Fig 2 as well as an implementation of Network and Receiver (but not Sender from Fig. 3). It also contains a SystemC model, main.cc containing the code from Listing 1 (implementing the sender in SystemC) along with an implementation of a hand-written communication top level and stubs for the rest of the model. The RPC sub-entry contained in the

Models folder contains the library to communicate with the SystemC cosimulation jobs. The Logs entry contains an entry for each time we have executed our cosimulation. We can see an entry Execution 1, containing a TimedProtocolTop.simlog containing the simulator log for the TimedProtocolTop.model component and an entry demo2 corresponding to the compiled name of our SystemC model. SystemC allows users to specify log files manually, and they are all gathered in this directory along with a log of the console while executing the model. If our cosimulation had consisted of more CPN and/or SystemC components, we would have an entry for each additional component here. The last thing we notice in the Project Explorer, is the Cosimulations entry, which contains just one file, namely cppcpn.cosimulation, which is a file containing our cosimulation representation.

In the editing area, we can see the overall structure of our cosimulation representation. A cosimulation description consists of a set of components (such as a CPN or SystemC components). Each component can expose an external interface (such as port places for CP-nets or ports for SystemC models), and can import other components (corresponding to substitution transitions in CP-nets and module instantiations in SystemC) and ties into the interface of imported modules (corresponding to port/socket assignments in CP-nets and channels in SystemC). In the example in Fig. 5, we have a cosimulation with two components, a Petri Net Model and a System CModel. The Petri net model is tied to TimedProtocolTop.model and the SystemC model is tied to the compiled name, demo2, which is our SystemC sender. We see that the SystemC model exports two ports, one input port and one output port. The ports have exported names (here we use names corresponding to the places they represent in the CPN models, but they could be anything) and describe which SystemC port they correspond to. In the Properties view, we can see that for Output Port Export A the exported name is A and the corresponding SystemC port is a. The Petri net model does not export any ports, but rather imports a module from the environment replacing a transition in the model. This import contains a link to the SystemC component (not visible in the figure) as well as assignments between exported port names and places (also not visible).

This information allows us to implement sendAndReceive for both kinds of jobs. For Petri net cosimulation jobs we can just read the marking of places, find the matching exported name and imported module, and transmit the data to that module when sending, and map an exported port name to a place when receiving data. For SystemC, we can set up channels listening on/transmitting to the specified ports. When we receive data on a channel, we invoke code transmitting it to the correct component. This code can be generated from information about the module structure (which we have) and information about the exported port name (which we also have). In the same manner, we can generate code to invoke when we receive data.

In our prototype, we have not focused on a real exchange format between the components, and just assume that transmitted values are strings that can be understood by the receiver.

4.3 Simplified Architecture for Production-quality Implementation

For a production-quality implementation we propose the simpler architecture in Fig. 6. In this architecture, we have removed the centralized process and instead moved the implementation of Algorithm 1 and Algorithm 2 to the Cosimulation layers for both SystemC and CPN (using instead a much faster in-process version of the ACCESS/CPN layer). We have also replaced ONC-RPC with Message Passing Interface (MPI) [10] which is an industry standard for very fast communication between distributed components. In order to use MPI, we have to embed a standard MPI implementation into the CPN simulator process and add code to interface with that from SML code used in the simulator. The run-time behaviour is as one would expect: Instead of having the communication being mediated by ASAP, ASAP is now only responsible for setting up a cosimulation by starting the autonomous component processes. After being set up, the components communicate directly with each other. ASAP can be used to process the results in a single user interface after simulation.

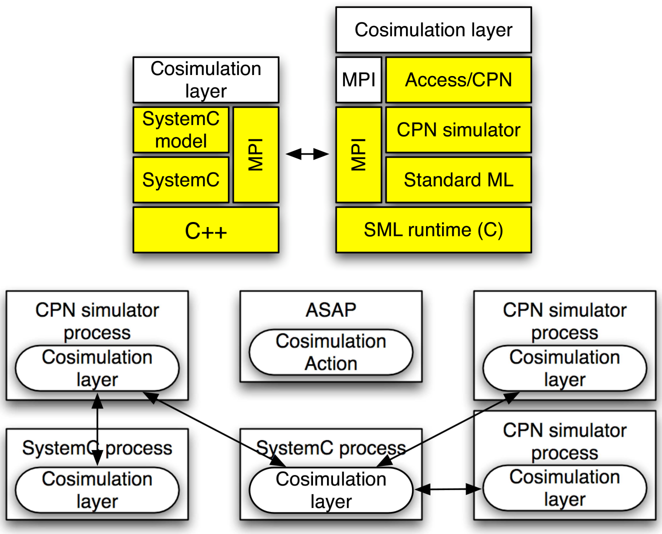


Fig. 6: The static architecture (top) and run-time architecture (bottom) of production-quality implementation

One of our design goals was that we did not want to change the SystemC simulator. Instead, we have created a cosimulation layer as a regular SystemC process, namely as stubs, so our prototype shows that it is feasible to achieve cosimulation without changing the SystemC simulator. For efficient implementation we may need to augment the CPN simulator, but that is less of a problem, since we have control over it.

Our implementation shows that our algorithm is able to provide cosimulation and we anticipate that the very loose coupling between components will allow it to perform very well. We believe 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, which is most interesting for the CPN models, to get graphical feedback, such as showing markings of the CPN models.

As a completely unrelated bonus, our prototype shows that it may be possible to do reasonable parallel or distributed simulation of timed CPN models. In fact, the current prototype is able to use as many processor cores as there are components in a simulation setup, which can potentially lead to faster simulation of timed CPN models on multi-core systems.

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 simulators and the practicality has been demonstrated using a prototype. We have have demonstrated that it is possible to cosimulate SystemC and CPN models without changes to either languages by introducing a cosimulation representation, external to the languages, which takes care of mapping between language specific features for composability. The current prototype is interesting and worth pursuing further as outlined below. The prototype has, in addition to our intended goal of demonstrating viability of cosimulation, also provided unforeseen benefits namely an idea for distributed simulation of timed CPN models.

The major problem currently is that we only have a prototype implementation and simple proof-of-concept examples. A natural next step is to implement an actual SoC model using the approach. This will most like lead to performance problems of the prototype, so future work includes making a production-quality implementation as proposed in the previous section. We have not currently implemented all of the optimizations to the distributed minimum calculation, and these should be implemented and evaluated.

It would be interesting to compare an implementation using the simplified architecture with an implementation using HLA for cosimulation of CPN and SystemC models, which would require making an implementation of HLA for CPN models. It would also be interesting to see if the proposed architecture architecture also allows faster simulation of timed CPN models by using multiple processor cores.

Until now, we have 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 [2] well when systems are loosely synchronized, which is indeed the case here.

References

1. P.V Biron and A. Malhotra. XML Schema Part 2: Datatypes. `www.w3.org/TR/2001/REC-xmlschema-2-20010502/`.
2. S. Christensen and L. Petrucci. Modular Analysis of Petri Nets. *The Computer Journal*, 43(3):224–242, 2000.
3. CPN Tools webpage. `www.cs.au.dk/CPNTools/`.
4. Modeling and Simulation High Level Architecture. IEEE-1516.
5. IEEE Standard System C Language Reference Manual. IEEE-1666.
6. K. Jensen and L.M. Kristensen. *Coloured Petri Nets – Modelling and Validation of Concurrent Systems*. Springer, 2009.
7. JSON: JavaScript Object Notation. `www.json.org/`.
8. Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.
9. K.B. Lassen and M. Westergaard. Embedding Java Types in CPN Tools. `http://westergaard.eu/personlig/publications/types.pdf`, 2006.
10. Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface. Online: `www.mcs.anl.gov/research/projects/mpi/mpi-standard/mpi-report-2.0/mpi2-report.htm`, July 1997.
11. K.L. Morse, M. Lightner, R. Little, B. Lutz, and R. Scrudder. Enabling Simulation Interoperability. *Computer*, 39(1):115–117, 2006.
12. A.V. Ratzer, L. Wells, H.M. Lassen, M. Laursen, J.F. Qvortrup, M.S. Stissing, M. Westergaard, S. Christensen, and K. Jensen. CPN Tools for Editing, Simulating, and Analysing Coloured Petri Nets. In *Proc. of ATPN'03*, volume 2679 of *LNCS*, pages 450–462. Springer-Verlag, 2003.
13. R. Srinivasan. RPC: Remote Procedure Call Protocol Specification Version 2. RFC 1831, August 1995.
14. Texas Instruments. OMAP^{TM} Applications Processors: OMAP^{TM} 4 Platform. Online: `www.ti.com/omap4`.
15. M. Westergaard, S. Evangelista, and L.M. Kristensen. ASAP: An Extensible Platform for State Space Analysis. In *Proc. of ATPN 2009*, volume 5606 of *LNCS*, pages 303–312. Springer-Verlag, 2009.
16. M. Westergaard and L.M. Kristensen. The Access/CPN Framework: A Tool for Interacting With the CPN Tools Simulator. In *Proc. of ATPN 2009*, volume 5606 of *LNCS*, pages 313–322. Springer-Verlag, 2009.
17. M. Westergaard, L.M. Kristensen, and M. Kuusela. Towards Cosimulating SystemC and Coloured Petri Net Models for SoC Functional and Performance Evaluation, 2009. Proc. of 21st European Modeling and Simulation Symposium (to appear).