ISSN 0105-8517

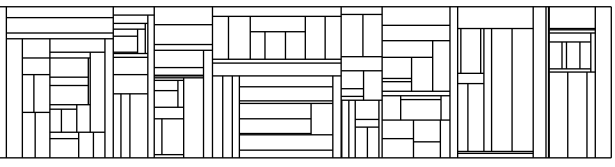
Tenth Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools

Aarhus, Denmark, October 19-21, 2009

Kurt Jensen (Ed.)

DAIMI PB - 590
October 2009

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF AARHUS
IT-parken, Aabogade 34
DK-8200 Aarhus N, Denmark



Preface

This booklet contains the proceedings of the Tenth Workshop on Practical Use of Coloured Petri Nets and the CPN Tools, October 19-21, 2009. The workshop is organised by the CPN group at Department of Computer Science, Aarhus University, Denmark. The papers are also available in electronic form via the web pages: www.cs.au.dk/CPnets/events/workshop09/.

Coloured Petri Nets and the CPN Tools are now licensed to nearly 9,000 users in 140 countries. The aim of the workshop is to bring together some of the users and in this way provide a forum for those who are interested in the practical use of Coloured Petri Nets and their tools. The submitted papers were evaluated by a programme committee with the following members:

Wil van der Aalst, Netherlands
João Paulo Barros, Portugal
Jörg Desel, Germany
Joao M. Fernandes, Portugal
Guy E. Gallasch, Australia
Jorge de Figueiredo, Brazil
Monika Heiner, Germany
Thomas Hildebrandt, Denmark
Kurt Jensen, Denmark (chair)
Ekkart Kindler, Denmark
Lars M. Kristensen, Norway
Charles Lakos, Australia
Johan Lilius, Finland
Daniel Moldt, Germany
Laure Petrucci, France
Rüdiger Valk, Germany
Lee Wagenhals, USA
Karsten Wolf, Germany
Jianli Xu, Finland

The programme committee has accepted 14 papers for presentation. Most of these deal with different projects in which Coloured Petri Nets and their tools have been put to practical use – often in an industrial setting. The remaining papers deal with different extensions of tools and methodology.

The papers from all CPN Workshops can be found via the web pages: www.cs.au.dk/CPnets/. After an additional round of reviewing and revision, some of the papers form the 1998-2007 workshops have been published in four special sections in the International Journal on Software Tools for Technology Transfer (STTT). For more information see: sttt.cs.uni-dortmund.de/. After an additional round of reviewing and revision, some of the papers from the 2008-2009 workshops will be published in Transactions of Petri Nets and Other Models of Concurrency (ToPNoC) which is new journal subline of Lecture Notes in Computer Science. For more information see: www.springer.com/lncs/topnoc.

Kurt Jensen
PC and OC chair

Table of Contents

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.

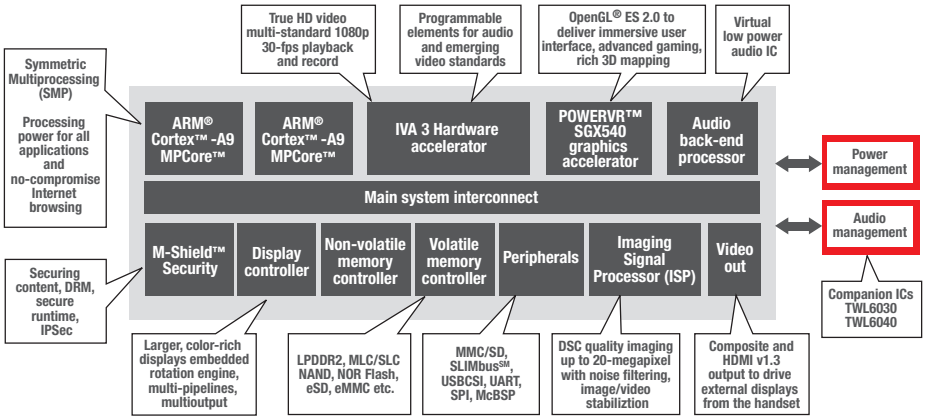


Fig. 1: Block diagram of Texas Instruments' OMAP44x platform.

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.

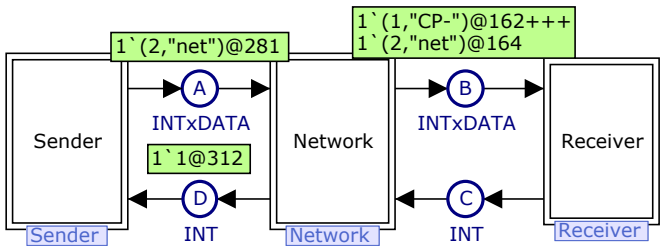


Fig. 2: Top level of network protocol.

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.

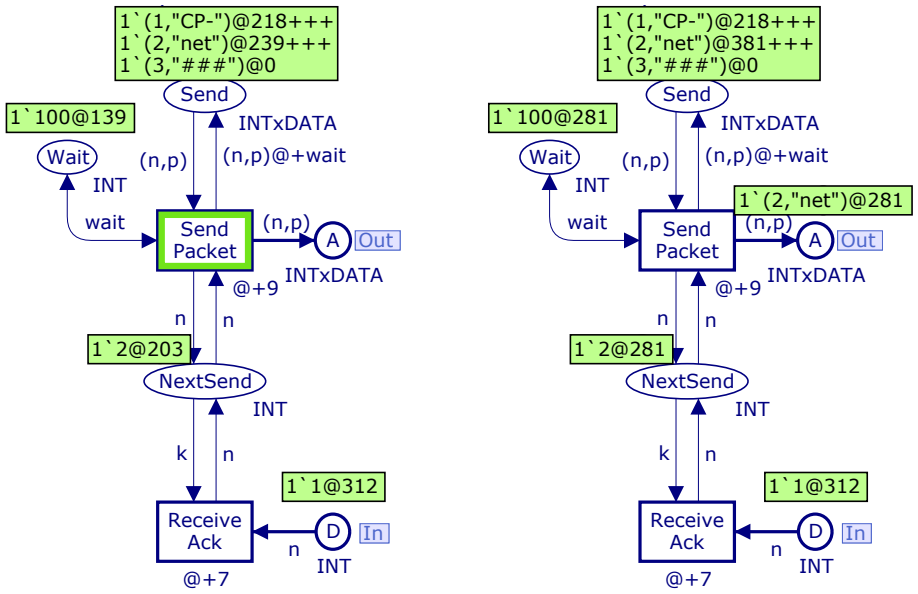


Fig. 3: Sender of network protocol before executing Send Packet (left) and after (right).

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 ← false
 2: requests ← QUEUE.EMPTY()
 3: results ← QUEUE.EMPTY()
 4:
 5: proc WORKERTHREAD() is
 6:   while true do
 7:     minRequest ← ∞                    // collect requests from children
 8:     for i = 1 to children.SIZE() + 1 do
 9:       minRequest ← min(minRequest, requests.REMOVEHEAD())
10:     if this = root then
11:       result ← minRequest             // we know the result locally
12:     else
13:       // propagate our request
          result ← parent.DISTRIBUTEDGLOBALMIN( minRequest )
14:     for i = 1 to children.SIZE() + 1 do
15:       results.ADD( result ) // distribute time increase to children
16:     ready ← 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 ¬ready do
23:     SKIP()                          // wait for calculation to complete
24:   result ← results.REMOVEHEAD()
25:   if results.ISEMPTY() then
26:     ready ← 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 $children.\text{SIZE}() + 1$ 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 $children.\text{SIZE}() + 1$ 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-

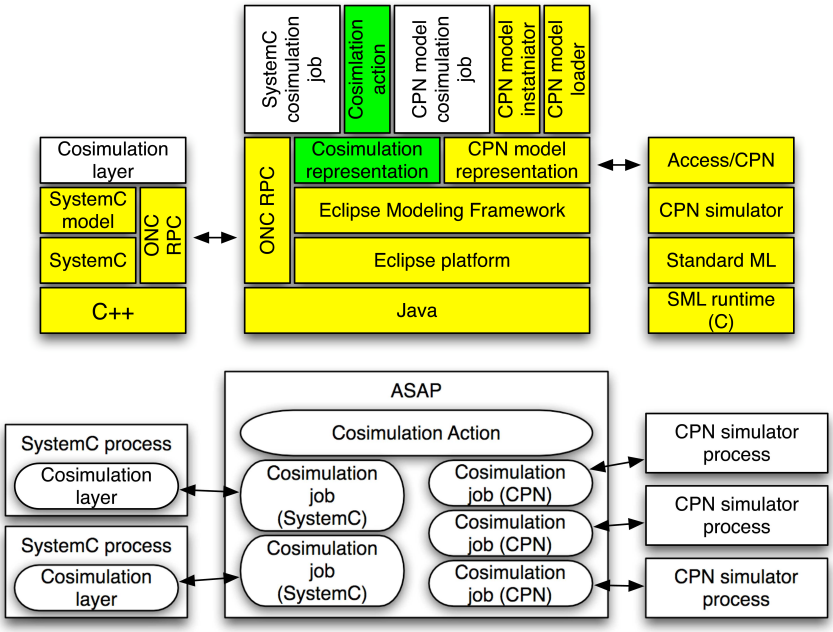


Fig. 4: The static architecture (top) and run-time architecture (bottom) of our prototype

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.

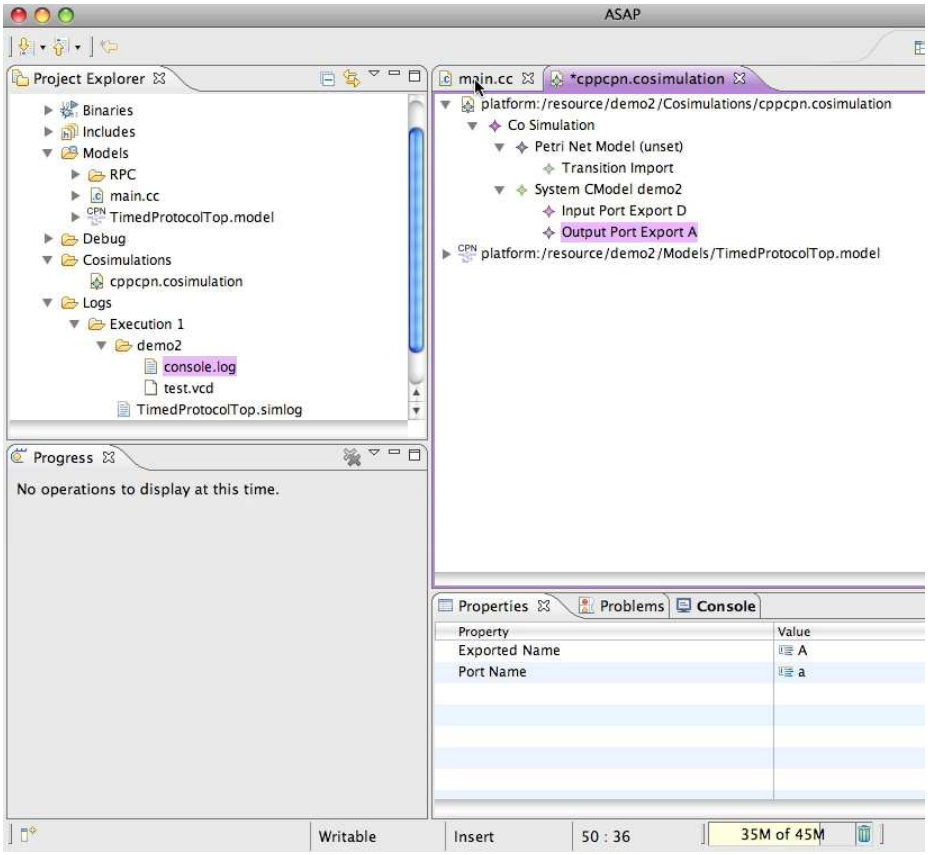


Fig. 5: Screenshot from prototype.

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.

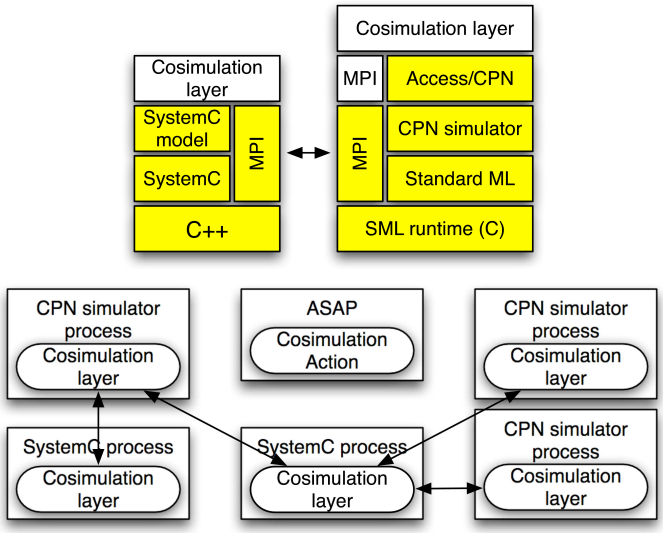


Fig. 6: The static architecture (top) and run-time architecture (bottom) of production-quality implementation

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™ Applications Processors: OMAP™ 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).

Seeking Improved CPN Tools Simulator Performance:
Evaluation of Modelling Strategies for an Army Maintenance Process *

Guy Edward Gallasch[1], Benjamin Francis[2], and Jonathan Billington[1]

[1] Computer Systems Engineering Centre
University of South Australia
Mawson Lakes Campus, SA, 5095, AUSTRALIA
Email: guy.gallasch@unisa.edu.au jonathan.billington@unisa.edu.au
[2] Land Operations Division
Defence Science and Technology Organisation
P.O. Box 1500, Edinburgh, SA, 5111, AUSTRALIA
Email: benjamin.francis@dsto.defence.gov.au

Abstract. The availability of military equipment during a campaign depends on many factors including usage rates, spares, policy, and importantly the composition and distribution of maintenance personnel. This paper presents a Coloured Petri Net model of an Army maintenance process that encapsulates these factors. On simulating the model with CPN Tools we identify a number of simulation performance concerns. Using Standard ML profiling we discover that they are related to the modelling of personnel. Two different representations of personnel are created and evaluated in terms of simulation time using CPN Tools. We demonstrate that a list representation can have dramatic performance gains over a multiset representation. We further demonstrate that simulation performance can be improved by unfolding the CPN with respect to the maintenance network topology. Finally, we canvass some ideas for further improvements to simulation performance.
Keywords: Army Maintenance Process, Simulation Performance, Models of Personnel.

1 Introduction

In order to support both preparedness levels and the actual conduct of military operations, Defence is required to maintain the wide variety of defence equipment. This involves the deployment of tradespeople at a number of military workshops, which may be widely geographically distributed. The defence maintenance system can thus be considered to be a distributed system. Australia's Defence Science and Technology Organisation (DSTO) is interested in understanding the maintenance system and its processes at a deep level in order to suggest improvements to them. As part of a broader research initiative, DSTO have contracted the Computer Systems Engineering Centre (CSEC) of the University of South Australia to work on a collaborative project to model aspects of Defence logistics [8]. CSEC and DSTO are developing an Army Maintenance System Analysis Tool to examine the effectiveness of a deployed maintenance capability for the Australian Army. The tool is required to validate the feasibility of plans and to explore "what if" scenarios. Due to the distributed nature of the maintenance system, the tool is based on a timed Coloured Petri Net (CPN) [17, 19] model.

There have been several previous attempts to model and analyse Defence maintenance processes, e.g. [4, 15, 16, 22, 23, 26]. More generally, there are models of the maintenance of fleets of aircraft or vehicles [9], maintenance of power systems [10], models of specific maintenance facilities [27] and models that schedule maintenance activities to minimise the disruption caused to the normal operation of that equipment [2, 5], e.g. equipment in a production line, or military vehicles performing missions. There are also models that examine key equipment items from a maintenance system perspective [6, 25]. Some models [15,26] make use of the discrete event simulation tool ARENA [20], while others use Bayesian and Markovian analysis techniques [5, 9, 16] or optimisation of mixed-integer linear programming models [2]. A genetic algorithm is used in [23] to optimise maintenance schedules.

CPNs have a long history of being applied to the verification of complex distributed systems, e.g. see [1, 3]. In the area of Defence, CPNs have been applied to the modelling of Defence logistics networks [12,13], the modelling and analysis of operations planning and the subsequent production of a prototype tool based on CPNs [21,29], and modelling and analysis of the Australian Defence Force planning process [24].

* This work was supported by Australia's Defence Science and Technology Organisation, Contract No. 4500498737 and Research Agreement No. 229146.

Our first prototype timed CPN model of the Army Maintenance Process was presented in [14]. This model captured a number of aspects of Army maintenance, including equipment usage rates and the composition and disposition of maintenance personnel across a distributed network of maintenance workshops, and the impact of these aspects on the operational availability of a deployed system of equipment (the amount of time the equipment is up and running). The model allows the user to specify a distributed network of maintenance workshops, rather than a single maintenance facility, and is not specific to any single piece or type of equipment. Personnel are explicitly modelled and are not assumed to have homogeneous skills. Hence, our CPN model is more extensive, flexible and general than any we have encountered thus far in the literature. However, modelling with greater fidelity generally requires greater computational resources.

Although serving well as a *specification* model (a formal specification of the Army maintenance process), analysis was problematic. Attempts to simulate realistic scenarios with CPN Tools [7,18] failed due to excessive run-time. The investigations reported in [14] revealed that poor simulation performance was primarily related to the representation of maintenance personnel within the model. This paper expands on the investigations reported in [14] into the performance of the CPN Tools Timed CPN simulator in the context of this model. It provides a more thorough investigation of simulation performance of two models of personnel, and also considers an 'unfolded' alternative for the modelling of network topology.

The contribution of this paper is threefold. Firstly, this paper presents a overview of the refined version of the CPN model of the Army's maintenance process from [14]. Fusion places are no longer used, the use of code segments has been eliminated to a large extent, and guards have been expressed more concisely. In particular, we present a significantly revised version of the Assign Transport Resources page, which is key to our analysis. Secondly, based on the results of [14], we consider two data structures to represent personnel within our model and two ways to represent the topology of the maintenance network: encoding topology within tokens (folded approach) and representing topology explicitly within net structure (unfolded approach). Thirdly, we provide a more comprehensive comparison and evaluation of the performance of the CPN Tools simulator using these different personnel modelling approaches, and extend this investigation to consider the impact of an unfolded network topology on the performance of the simulator. Although our model represents a military maintenance process, the observations made may equally apply to other (similar) large-scale CPN models.

The rest of this paper is organised as follows. A brief introduction to the Army maintenance system is given in Section 2. Section 3 provides a description of our model. Choices for modelling personnel are described in Section 4. The simulation performance of these different models is presented in Section 5 and discussed in Section 6. Finally, Section 7 provides some concluding remarks. It is assumed that the reader is familiar with Coloured Petri Nets and CPN Tools.

2 Army Maintenance System

The Army maintenance system can be described at one level as a simple tree hierarchy of maintenance nodes arranged roughly along four distinct *lines of support*: from the 1^{st} Line (closest to the front line) where equipment is used in the field, up to the 4^{th} Line, where deeper level maintenance and equipment pooling occurs. Typically, the *National Support Base* for a military campaign exists at the 4th line of support. Such a maintenance system is illustrated in Fig. 1, where circles represent maintenance nodes and arcs represent the movement of personnel and equipment. Nodes within the network represent maintenance workshops. Each contains a range of personnel of different trade types, forward repair/recovery capabilities, and authorisations to conduct particular repairs. A maintenance node also has an inherent capacity that governs the amount and nature of work that can be accepted at a node. In addition, a node may simultaneously be a source of maintenance liability as they also operate equipment which may require maintenance. A *strategic interface* exists between 4th line and the other lines of support, symbolising that 4th line typically exists within the country of origin and is relatively static, whereas the remaining lines are deployed.

These node characteristics, combined with the various equipment failures, determines where items can be repaired in the system, and hence the extent to which equipment is moved around the maintenance network. An item of equipment that requires maintenance must be co-located with suitable maintenance facilities so that the maintenance can be performed. The equipment may already be at a suitable maintenance workshop, or it may be that a team of maintenance personnel can be dispatched from a nearby workshop and travel to the equipment. More often, however, this means the equipment needs to be transported to a maintenance workshop.

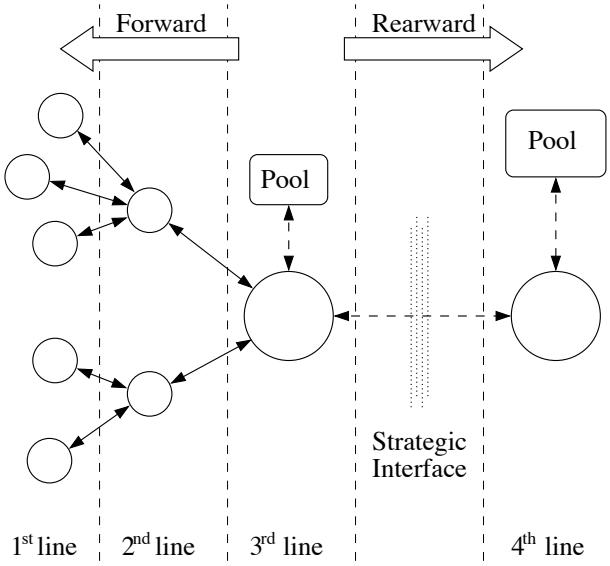


Fig. 1. Lines of Support within the Army Maintenance System.

In general, repair should be conducted as far forward as possible in order to reduce delays in returning equipment to an available state, thus increasing effectiveness. This may come at the cost of reduced efficiency due to additional delays in transporting personnel through the network. Alternatively, availability might be managed through releasing replacements from pools, which are in turn replenished with items that are repaired under lower operational tempo conditions.

The interaction between the degree of forward repair, the nature of repairs that can be conducted in the area of operations versus the National Support Base (i.e. 4th line) the disposition and type of personnel allocated to each line of support, and the use of replacement pools is a complex problem well suited to formal modelling and analysis. This allows force structure planners to holistically examine trade-offs involving the size and nature of the Army maintenance system through a review of performance over a wide range of operational scenarios. Typically, a realistic scenario will involve hundreds of personnel and thousands of pieces of equipment distributed over tens of nodes.

3 Model Description

Hierarchical Coloured Petri Nets are used to model the Army maintenance system. Figure 1 shows a *system*-oriented view of Army maintenance, with a geographically distributed set of nodes (workshops) with a given topology, where maintenance activities may be carried out at any of the nodes. Apart from the node at fourth line, the operations at each node follow a *maintenance process* that is very similar, regardless of the location of the node. This process differs only with respect to factors such as the capacity of the workshop, the grade of repair that can be carried out at the workshop, and the maintenance personnel located at the workshop. Because of these similarities, rather than taking a system-oriented view of Army maintenance in our CPN model, we have taken a *process*-oriented view. CPNs allow us to capture physical characteristics, such as network topology and individual node features, within its data structures, rather than within the net structure. We do so in our model, hence taking a process-oriented view avoids the need to duplicate the net structure relating to the maintenance process followed within each workshop. This approach eases maintenance of the model itself (e.g. in the event of a change to the maintenance process that we have captured, we need only implement the changes for one piece of net structure rather than a piece of net structure for every workshop). Our model is thus a hierarchical CPN model, representing a decomposition of the *processes* related to performing maintenance in a military environment.

The model consists of 14 pages arranged into three hierarchical levels, plus an additional layer required to initialise the model with a specific scenario. The hierarchical structure of pages within our model is illustrated in Fig. 2. The Process Overview and Workshop Maintenance pages serve as a flowchart of the

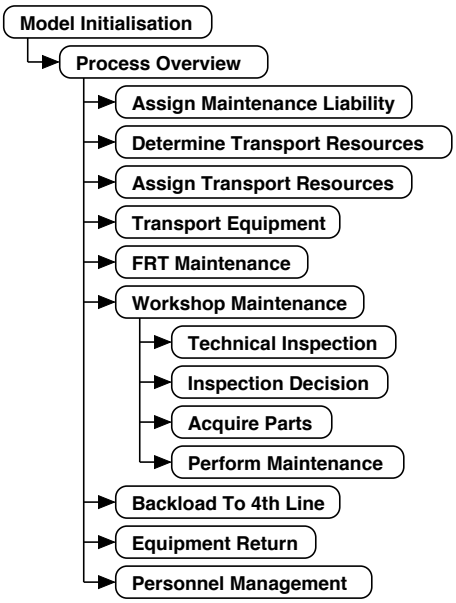


Fig. 2. The hierarchical structure of our CPN model.

process. We use these pages to describe the system and processes we are modelling. Size constraints prevent us from describing the model in detail in this paper. A full description is given in [11].

The model comprises 44 executable transitions and 14 *substitution* transitions. There are 27 distinct places in the model, with a total of 88 physical places due to port/socket place duplication. Further complexity is contained in extensive model inscriptions and function definitions, written in the programming language Standard ML [28], and comprising approximately 1600 lines of code.

Since it was first published [14], the model has been significantly improved. Originally, fusion places were used as part of the Model Initialisation page (see Section 3.2) to allow the initial marking of numerous places throughout the model to be given a scenario-specific pseudo-initial marking. Unfortunately, overcoming the limitation of CPN Tools not allowing a place to be both a fusion place and a socket place resulted in a mix of fusion places and port/socket places, which is undesirable from a hierarchical modelling perspective. The model no longer uses fusion places and hence the hierarchical structuring has been made clearer. Also, the previous model used code segments on a number of model pages to specify a binding of output arc variables. Where this was unnecessary, the code segments have been replaced by functions on output arcs. As will be seen in Section 4 we examine the use of two data structures for personnel within this model. The use of code segments has been eliminated completely from the first model (that considers each person as a separate token), an example of which is seen in Fig. 4 in Section 4.1. Code segments have only been retained in two instances in the second model (that considers people stored as values in lists) where pragmatic to do so.

3.1 Important Data Structures

Before we begin describing the model itself, we must first introduce three key data types used to describe the state information in the model. These are the `Equipment`, `Maintenance_Task` and `Personnel` colour sets. These three colour sets, combined in various ways, form the basis of the types of almost all places in the model.

Equipment The `Equipment` colour set is shown in lines 5-13 of Listing 1. It defines a record structure that describes individual items of equipment that will require maintenance. The equipment type, present location and home location (lines 5-7) are strings. The usage mode (line 8) designates whether an item of equipment is actively in use or is in an equipment pool (line 2). The last service type (line 9) records whether the last service was a major service or a minor service (line 3). Finally, the current usage meter reading, time of last service, usage meter reading at the last service and time of the last inspection (lines

Listing 1. The `Equipment` Colour Set.

```
1  colset Location = STRING;
2  colset Usage_Mode = with Active | Pool;
3  colset Service_Types = with Major | Minor;
4
5  colset Equipment = record equipment_type : STRING *
6                            present_location : Location *
7                            home_location : Location *
8                            usage_mode : Usage_Mode *
9                            last_service_type : Service_Types *
10                           usage_meter : INT *
11                           time_of_last_service : INT *
12                           usage_meter_at_last_service : INT *
13                           time_of_last_inspection : INT timed;
```

Listing 2. The `Maintenance_Task` Colour Set.

```
1  colset Maintenance_Type = with Service | Corrective | Inspection;
2  colset Grade = with L|M|H;
3  colset Trade = STRING;
4  colset ERT_by_Trade = INT;
5  colset Job = product Trade * ERT_by_Trade;
6  colset Job_List = list Job;
7  colset Priority = with Essential | Non_Essential;
8  colset Mobility = with wheeled_mobile | wheeled_not_mobile | not_wheeled;
9  colset Maint_Methods = with InSitu | SelfTransport | Distribution | Recovery | FRT;
10 colset Maint_Methods_Attempted = list Maint_Methods;
11
12 colset Maintenance_Task = record maintenance_type : Maintenance_Type *
13                                   next_occurrence_time : INT *
14                                   current_request_location : Location *
15                                   grade_required : Grade *
16                                   job_requirements : Job_List *
17                                   job_priority : Priority *
18                                   mobility : Mobility *
19                                   inspected : BOOL *
20                                   methods_attempted : Maint_Methods_Attempted *
21                                   assignment_timeout : INT timed;
```

10-13) are recorded as integers. Usage metrics could include e.g. distance travelled (odometer), operating hours, rounds fired, or calendar time.

Maintenance Tasks The `Maintenance_Task` colour set is shown in Listing 2. This is also a record (lines 12-21) that specifies the maintenance required and records the progression of individual items through the system. The type of maintenance required (line 12) can be either a regular service, corrective maintenance (in the event of a breakdown) or an inspection of a piece of equipment (line 1). The next occurrence time (line 13) records when the next 'future' maintenance event is due (unbeknownst to the system) to occur for a corresponding item of equipment. The location of the workshop that is currently considering the maintenance task is given by line 14. The grade of repair (line 15) required for a particular maintenance task will be either Light, Medium or Heavy (line 2) depending on the nature of the task, and hence will affect where and by whom the corresponding item of equipment can be maintained. The job requirements (line 16) specify the tradespeople required and the length of time for which they will be required (lines 5 and 6). Rather than using an enumerated type to specify the trade types, e.g. vehicle mechanic, our model uses strings (line 3). This is for extensibility and to overcome a CPN Tools limitation that prevents colour set declarations being specified externally and imported into the tool. The length of time a particular trade is required is given by the Estimated Repair Time (ERT) on line 4. The priority of each particular maintenance task (line 17) is categorised as either essential or non-essential (line 7) and is based on the type of equipment and the nature of the campaign. Line 18 specifies the mobility of the equipment to be maintained. According to line 8, the equipment can be either wheeled and mobile (e.g. a truck that can be driven), wheeled and not mobile (e.g. a truck with a broken engine) or not wheeled (e.g. a generator). Whether or not the equipment has undergone a technical inspection, revealing the current maintenance liability, is given on line 19. Line 20 specifies the methods of transport/maintenance that have been attempted in order to address this maintenance task. The methods are given on line 9 and will be described in Section 3.3. The assignment timeout (line 21) specifies how long a task will wait

Listing 3. The `Personnel` Colour Set.

```
1 colset Personnel_States =   with Ready | Working | Offline;
2
3 colset Personnel = record trade : Trade *
4                           home_location : Location *
5                           working_status : Personnel_States *
6                           last_came_online_time : INT timed;
7
8 colset Personnel_List = list Personnel timed;
```

for personnel to be assigned to it before some other alternative method of maintenance/transport is attempted.

Personnel Listing 3 describes the data structures used for personnel. `Personnel` (lines 3-6) records the trade (line 3), home location (line 4) and working status (line 5) of individual people. The working status specifies whether a person is ready to work, currently working, or *offline* (line 1). Offline means that the person is not currently available to be assigned to maintenance work (e.g. is sleeping or performing other duties). The time at which the person last came online (moved from Offline to Ready) is also recorded (line 6). Line 8 declares a list of personnel.

3.2 The Model Initialisation Page

The Model Initialisation page is used to populate the model with tokens representing the scenario under consideration. Because of our process-oriented modelling approach, no modification of the net structure is needed when analysing different scenarios. The Model Initialisation page initialises the following places:

- **Node Knowledge:** The Node Knowledge place is populated with tokens that describe the characteristics and capabilities of each maintenance workshop in the scenario;
- **Topology:** The Topology place specifies the topology of maintenance workshops (nodes);
- **Personnel:** The Personnel place is populated with tokens that represent the number and disposition of personnel to be distributed throughout the maintenance workshops;
- **Equipment Awaiting Maintenance Assignment:** This place is populated with all of the equipment in the system that will require maintenance;
- **Equipment Awaiting Parts and Equipment Ready for Maintenance:** As 'house-keeping', these places are populated with one token per maintenance workshop, allowing for a prioritised list of maintenance tasks to be stored in both places for each workshop, as will be described later.

This initialisation is achieved through the use of functions that read in initialisation data from text files on local storage. The text files may be populated by an external editor, e.g. from an Excel spreadsheet, thus allowing for increased modelling flexibility and expedient scenario generation and modification.

3.3 The Process Overview Page

The Process Overview page, shown in Fig. 3, is the highest level page that describes the maintenance process. The net structure of this page has been developed to explicitly represent the flow of equipment through the maintenance process, depicted by the bold arcs. This page consists mostly of substitution transitions, each of which represents a sub-process within the overall maintenance process. Each place on this page is typed by `Equipment` or by Cartesian products of `Equipment`, `Maintenance_Task` and `Personnel`. The names of these colour sets reflect their declarations (see [11]), e.g. `EquipmentXTask` is the product of `Equipment` and `Maintenance_Task`.

We now describe the processes represented on the Process Overview page.

Assign Maintenance Liability The starting point in the execution of our model is the assignment of a maintenance liability to each of the items of equipment in the Equipment Awaiting Liability Assignment place. This is done by the Assign Maintenance Liability page. Each item of equipment is paired with a 'future' liability (specific details of the next maintenance requirement, in the form of a maintenance task) as it is transferred to the Operational Equipment place, where it is considered to be in use until such

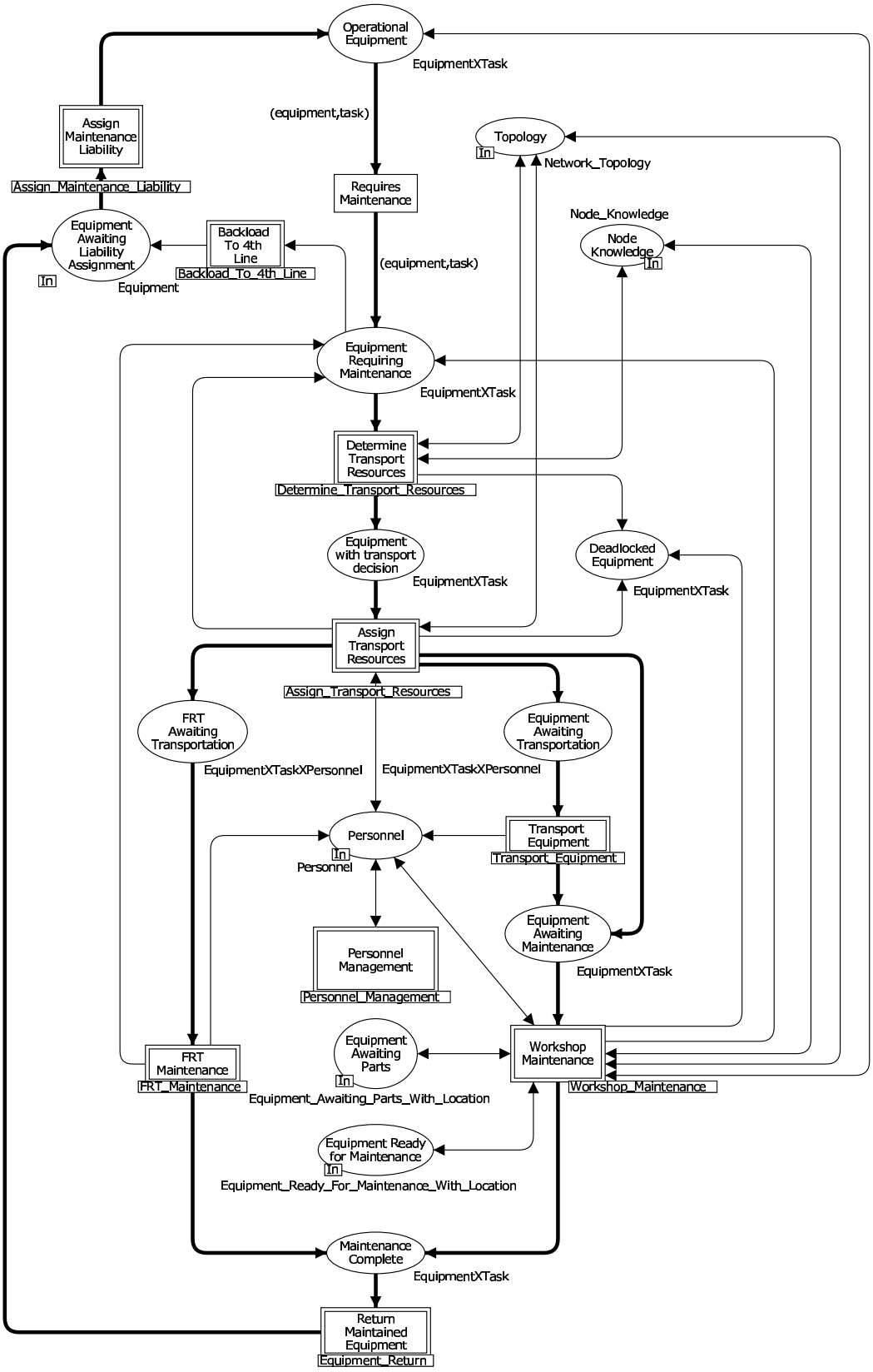


Fig. 3. The Process Overview page, showing the high-level structure and major flows of equipment (in bold).

time as its maintenance liability is realised. At this point, the Requires Maintenance transition moves the item of equipment from an operational state (Operational Equipment place) to a non-operational state (Equipment Requiring Maintenance place), where the item awaits maintenance. We use the timestamp mechanism of CPNs to control when this occurs.

Determine Transport Resources The first step in the maintenance process is to determine the transportation requirements. The approaches are:

- **In-situ:** the equipment does not need to move from its present location;
- **Self-transport:** the equipment moves itself to a suitable maintenance workshop;
- **Distribution:** the equipment is moved using a more general distribution network, e.g. road or railway, that services the demands of the military operation;
- **Recovery:** a team of personnel (a Recovery Team) moves from a maintenance workshop to the location of the equipment and brings the equipment back to the maintenance workshop; and
- **Forward Repair Team:** a team of personnel move from a maintenance workshop to the equipment location in order to effect the necessary maintenance.

The method is selected by the Determine Transport Resources subpage. All methods necessitate the use of a form of transportation with the exception of in-situ maintenance.

Assign Transport Resources Once the transport requirements have been identified, the Assign Transport Resources subpage ensures that the personnel requirements can be fulfilled and assigns the required personnel. In the event that the required personnel are not available, the equipment is passed back to the Determine Transport Resources page (the arc from Assign Transport Resources to Equipment Requiring Maintenance) and a different approach is selected. This process may be repeated a number of times both for the current location of the equipment and in the event that the equipment has been moved to a new location but that new location is not able to perform the necessary maintenance (hence the return arc from the Workshop Maintenance page to the Equipment Requiring Maintenance place). In the event that all options for transportation are exhausted and no further locations exist to which the equipment can be moved, this particular item of equipment is not able to be maintained by the system and is deposited into the Deadlocked Equipment place.

If the personnel requirements can be satisfied, there are three possible outcomes (the three bold arcs that exit the Assign Transport Resources page in Fig. 3). In the case of in-situ maintenance, no transportation is necessary, so the equipment item is passed directly to the Equipment Awaiting Maintenance place. Secondly, if a Forward Repair Team is required (defined by the type of item and location) the personnel comprising the FRT are passed to the FRT Awaiting Transportation place, along with the equipment and maintenance liability to which this FRT corresponds. Note that despite being present in the same token on the same place, the FRT and the item of equipment are still at geographically separate locations. Lastly, if an equipment item itself needs to travel within the maintenance network (self-transport, distribution or recovery), it is inserted into the the Equipment Awaiting Transportation place.

Transport Equipment An item of equipment that requires transportation may be able to transport itself, or it may require the distribution network or the the assistance of a Recovery Team to do so. All three of these scenarios are handled by the Transport Equipment page. Once all transportation is taken care of, equipment items will enter the Workshop Maintenance process from the Equipment Awaiting Maintenance place and any personnel required during the transportation will then be released to conduct other work.

FRT Maintenance and Workshop Maintenance Both the FRT and Workshop Maintenance processes result in maintained items of equipment. Workshop maintenance is described in Section 3.4. In the case of FRT Maintenance, the FRT travels to the equipment to maintain it, before returning to its assigned workshop and disbanding. However, it is possible that the FRT will fail to repair the equipment. This results in the equipment being reassessed by the Determine Transport Resources subpage and an alternative transportation approach chosen.

Backload to 4th Line In the case of either FRT or workshop maintenance, if an initial technical inspection reveals that an item of equipment is beyond the level of repair available, the equipment is 'strategically' recovered from the area of operation to the 4^{th} line of support. When this happens, a replacement item of equipment is issued from the 4^{th} line pool and inserted back into the area of operation at the location of the item it is replacing (after some transport delay). This process is denoted here as "Backload to 4th Line", and is modelled by the subpage of the same name. Like all equipment, the replacement must be given a "future" maintenance liability, hence the arc from Backload to 4th Line to Equipment Awaiting Liability Assignment.

Return Maintained Equipment Subject to the successful completion of the Workshop Maintenance or FRT Maintenance process, the equipment item is put into the Maintenance Complete place, and the maintenance personnel made available for another job, or moved into an off-line or resting state. The Return Maintained Equipment process will return the equipment to its original location. The equipment is inserted once more into the Equipment Awaiting Liability Assignment place, which permits the item to return to an operational state after having been assigned a new maintenance liability.

Personnel Management The Personnel Management page is responsible for moving personnel from a ready state to an offline state, and vice versa, at the correct times.

3.4 The Workshop Maintenance Page

The primary input to the Workshop Maintenance page is the Equipment Awaiting Maintenance place, containing equipment items (with their associated liability description) that are located at a maintenance workshop. The Workshop Maintenance page itself comprises four substitution transitions that describe four sequential sub-processes of workshop maintenance. A preliminary version of this page appears in [14]. These subpages are briefly described below:

Technical Inspection This process represents the inspection of equipment that reveals to the workshop the maintenance activities that must be undertaken for a particular item of equipment (the previously hidden maintenance liability).

Inspection Decision Firstly, this page determines whether to issue a replacement item of equipment to go into service while the other is undergoing maintenance. Secondly, it determines whether the current workshop can accept the new task (sufficient grade of repair, spare capacity and appropriately skilled personnel). Finally, accepted tasks and equipment are passed to the Acquire Parts process (below). If not accepted, the equipment is returned to the Equipment Requiring Maintenance place, so that it may undertake another iteration of the Determine Transport Resources process described in Section 3.3.

Acquire Parts The Acquire Parts process represents the delays inherent when spares are required but not presently in stock. Equipment items awaiting parts are stored in a queue prioritised by the expected arrival time of the parts.

Perform Maintenance Tradespeople are automatically assigned to the highest priority maintenance task at their location. Once assigned, the tradesperson continues to work on the repairs until they either complete their portion of the maintenance task (their job in the joblist of the maintenance task) or are required to rest. The completion of all required maintenance activities for a given item of equipment signals the end of the Perform Maintenance process, at which point the item of equipment is placed in the Maintenance Complete place, and the cycle begins again.

4 Modelling Personnel

Personnel enable transportation and maintenance activities. The Personnel place appears on 9 of the 15 model pages and the availability of tradespeople directly influences the enabling of 6 executable transitions across 4 model pages (Assign Transport Resources, Technical Inspection, Perform Maintenance and Personnel Management). The prominence of personnel within our model means that a large part of the computational effort of CPN Tools relates to the calculation of enabled binding elements involving personnel. The representation of personnel greatly influences simulation performance [14]. This section describes the two representations investigated in Section 5.

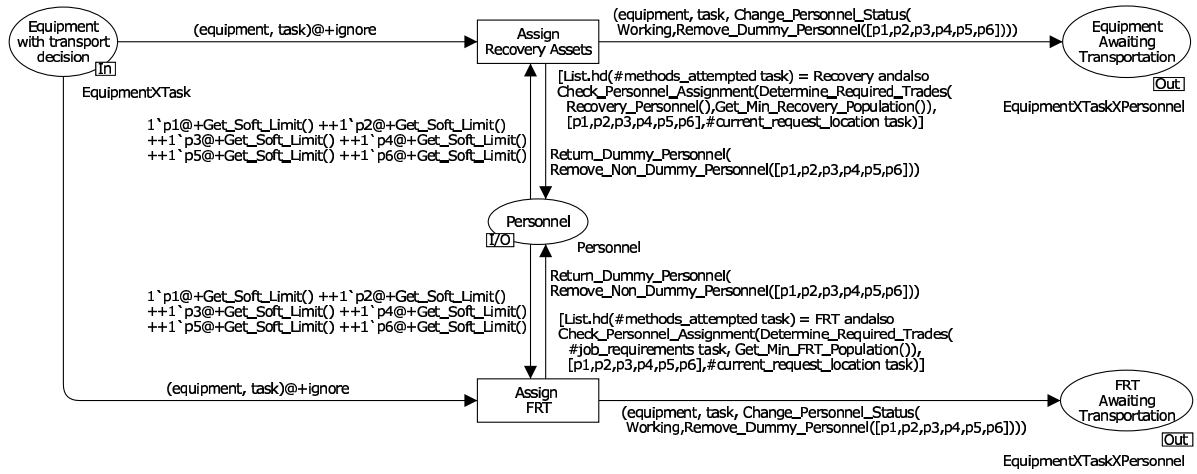


Fig. 4. A fragment of the Assign Transport Resources page of Model 1.

4.1 Personnel Modelled as Tokens (Model 1)

Our initial model represented personnel as individual tokens with timestamps, as per Listing 3. However, two difficulties were encountered: the number of personnel required may be different for different maintenance tasks; and the exact composition of the recovery team or FRT may not be known. This adds significant complexity to the problem of selecting personnel. It is possible to write an input arc inscription to select either a multiset of fully-specified values of varying size, or a multiset of unspecified or partially specified values of fixed size. The authors are yet to identify a direct way to select a varying number of partially known data values on an input arc.

A personnel selection mechanism was implemented to overcome this problem. This mechanism is shown in Fig. 4, which depicts a fragment of the Assign Transport Resources page, showing only the Assign Recovery Assets and Assign FRT transitions. To overcome the first problem of selecing a varying number of tokens we instead select a fixed number of tokens (six in this model) using one variable per person. These variables, `p1` to `p6`, can be seen on the input arcs from Personnel to Assign Recovery Assets and Assign FRT. The timestamp on personnel tokens is the time at which they are due to go offline. The `Get_Soft_Limit` function on these two input arcs returns a time value specifying how long people are allowed to work in the usual course of events, and so this allows selection of people before they are due to go offline. The number six was chosen as a reasonable maximum size for recovery and forward repair teams, but if needed the personnel selection mechanism can be extended to cope with larger teams. The idea is that the six people selected are compared against what is required and those not required are returned to the Personnel place. This works provided no Recovery or Forward Repair Team requires more than six people, and that the Personnel place always has at least six personnel tokens in it (e.g. if we require only two people, there still needs to be at least six tokens in the Personnel place). To ensure this, we introduced the notion of `Dummy` peronnel, who have both trade and location equal to the string `Dummy`. They do no work and cannot be assigned to Recovery or Forward Repair teams, and hence always reside in the Personnel place. The Model Initialisation page inserts six such dummy personnel into the Personnel place.

To overcome the second problem (selecting a team of personnel of unknown composition) we have introduced a string constant, `Unspecified`. When we require a person but don't care what trade they are, we specify a requirement for an `Unspecified` trade, and match any trade (except `Dummy`) to `Unspecified`.

In [14] this mechanism was implemented using a guard and code segment totalling 50 lines of ML code for each of the Assign Recovery Assets and Assign FRT transitions. This mechanism has been substantially refined in our current model and is now implemented using a guard only. The sizable guard has been expressed more concisely using list manipulation functions within the `Check_Personnel_Assignment` function given in Listing 4. We describe our new implementation with respect to the Assign Recovery Assets transition. The Assign FRT transition works in a similar way.

The purpose of the `Determine_Required_Trades` function (shown in Listing 5) in the guard of Assign Recovery Assets is to return a list of six trades, comprising the actual trades required (these

could be Unspecified) for the Recovery team, obtained by invoking the Recovery_Personnel and Get_Min_Recovery_Population functions, with the balance made up of Dummy trades. For example, in a particular scenario, Recovery_Personnel may specify that a Recovery Team must contain one Recovery Mechanic, but Get_Min_Recovery_Population may specify a minimum size of 2 for a Recovery Team. The additional person could be any trade, so is specified by the Unspecified trade. The remaining four trades in the list of six trades would be Dummy. The function Check_Personnel_Assignment (Listing 4) then takes this list of six trades, all six selected people (variables p1 to p6 in a list), and the location of the request for maintenance, and checks that the people selected match the trades required and are at the correct location, or else are dummy personnel. Note that the matching of Unspecified with any trade except Dummy is shown on line 11.

Functions Remove_Dummy_Personnel and Remove_Non_Dummy_Personnel (not listed) have been added to the model to replace the code segments previously associated with Assign Recovery Assets and Assign FRT. As can be seen in Fig. 4 these two functions are used on output arcs to separate real personnel from dummy personnel as appropriate.

As was reported in [14] this model, which we denote Model 1, was found to perform reasonably well on small scenarios (10's of people) when cycling personnel online and offline, but performed very badly when assigning personnel to teams, due to the nature of the assignment and the selection mechanism described above. Standard ML profiling of simulation runs indicated that the two transitions in Fig. 4 were responsible for this poor performance. Specifically, we discovered that the 'less than' function automatically defined by CPN Tools for ordering the Personnel colour set was being executed of the order of 10^7 times, even for small scenarios, every time the enabling of these two transitions was checked. Because the Personnel place is connected to these two transitions (amongst others), every time the marking of the Personnel place changed, a check for the enabling of these two transitions was instigated. This resulted in a four to five second delay between successive steps of the simulation, where both transitions contributed equally to this delay. Hence we were able to determine that the time was being taken in the enabling check subsequent to the firing of a transition that affected the marking of the Personnel place, rather than firing the transition itself. For scenarios of increasing size, the delay quickly became prohibitive.

4.2 Personnel Modelled in a List

In an attempt to improve model performance, rather than modelling each person as a token, a list of personnel was considered. To do this, each person 'token' was augmented with a time value modelled as part of the colour set as shown in line 5 of Listing 6. This time value was used to store the timestamp that the person would have had as a single, separate token. The *personnel list* tokens were then given a timestamp corresponding to the smallest time value over all personnel in the list, to inform the model of the earliest time that at least one person becomes 'ready'.

Mechanisms were implemented in the models containing personnel in lists to retain the nondeterministic selection of personnel for Recovery and Forward Repair teams, and the nondeterministic selection of the next eligible person to move to/from an offline state. It was realised by the authors that for the purposes of this modelling exercise it was not necessary to capture this degree of nondeterminism for selection of the next person to move to/from an offline state. Hence two variations were also considered in [14]: one in which the head of the personnel list was (deterministically) selected as the person to

Listing 4. The Check_Personnel_Assignment function.

```
1  exception ListLengthException;
2
3  fun Check_Personnel_Assignment([],[],current_request_location) = true
4    | Check_Personnel_Assignment(required_trades,[],current_request_location) =
5    raise ListLengthException
6    | Check_Personnel_Assignment([],selected_personnel,current_request_location) =
7    raise ListLengthException
8    | Check_Personnel_Assignment((trade_required,ert)::rest_of_required,
9      (selected:Personnel)::rest_of_selected,current_request_location) =
10   (trade_required = #trade selected orelse
11     (trade_required = Unspecified andalso #trade selected <> Dummy))
12   andalso (if ((#trade selected) = Dummy)
13            then true else (#home_location selected)=current_request_location)
14   andalso (#working_status selected) = Ready
15   andalso Check_Personnel_Assignment(rest_of_required,rest_of_selected,current_request_location);
```

Listing 5. The `Determine_Required_Trades` function.

```
fun Determine_Required_Trades(personnel_to_assign: Job_List, minimumRequired) =
  let
    val personnel_needed = 6 - List.length(personnel_to_assign)
    val additional_personnel = minimumRequired - List.length(personnel_to_assign)
  in
    if personnel_needed = 0
    then personnel_to_assign
    else if  additional_personnel <= 0
    then  personnel_to_assign^^ms_to_list(personnel_needed`(Dummy,0))
    else  personnel_to_assign^^ms_to_list((additional_personnel`(Unspecified,0))
          ++ ((personnel_needed - additional_personnel)`(Dummy,0)))
  end;
```

Listing 6. The revised `Personnel` colour set for use when representing personnel in a list.

```
colset Person = record trade : Trade *
                       home_location : STRING *
                       working_status : Personnel_States *
                       last_came_online_time : INT timed;
colset Personnel = product Person * INT timed;
colset Personnel_List = list Personnel timed;
```

cycle offline/online; and the other in which all eligible personnel were cycled offline/online in one action. Further discussion of the merits of investigating these variations can be found in [14].

A list imposes an ordering on its elements. Two possibilities were considered in [14] when ordering the list of personnel: ordering the personnel values in the list firstly according to their time values and secondly according to the automatically defined 'less-than' function for the `Personnel` colour set; and ordering the personnel in the list according to time value only. Both of these orderings result in the earliest available person being at the head of the list, hence searching for available personnel in the list becomes at worst a linear operation (it is a constant operation if we only wish to find a single available person, regardless of trade, as may occur with technical inspections).

In [14], seven variations (Model 1, plus three models of personnel in lists with two variations on the ordering of elements within the personnel lists) were analysed. The most promising list model variation in terms of simulation performance was the model which cycled offline/online all eligible personnel in one action and ordered elements in the personnel lists by time value only. This model was denoted Model 4B in [14]. In the rest of this paper, the simulation performance of Model 1 and Model 4B are examined and compared.

5 Simulation Performance

When simulating Model 1 for a simple scenario comprising 90 personnel and 50 items of equipment distributed over 18 locations, the model proceeded relatively quickly (cycling personnel offline and online) up to the point where the first maintenance liability became due and a team of personnel was required. At this point, the simulation was manually terminated after waiting more than 24 hours of real time for personnel to be allocated to the first team.

As already mentioned, two of the key involvements of personnel are the cycling of personnel online and offline and the assignment of personnel to either a Recovery Team or a Forward Repair Team. In this section, the simulation performance of Model 1 and Model 4B with respect to these two activities is evaluated and discussed. Further, we compare the simulation performance of models that represent network topology explicitly in net structure (unfolded models) rather than encoding topology information within tokens (folded models). All experiments have been carried out with the Timed CPN Simulator of CPN Tools version 2.2.0 on a 2GHz Intel Core 2 Duo processor with 2Gb of memory.

5.1 Baseline Scenario

As a baseline, we consider a scenario comprising five highly simplified nodes, where each node comprises:

- two tradespeople at each node:
  - one Recovery Mechanic; and
  - one Vehicle Mechanic;

– two tasks that require teams to be formed at each node:
  • one Recovery Team, requiring a Recovery Mechanic and any other person; and
  • one Forward Repair Team, requiring a Vehicle Mechanic and any other person.

For the baseline scenario, this gives a total of 10 personnel and 10 tasks that require teams to be formed. (In the baseline scenario there will be a conflict between assigning personnel to the Recovery Team and to the FRT.) The number of teams that require personnel reflects the number of maintenance tasks in the system, which in turn reflects the number of pieces of equipment in the scenario. This is a more sophisticated baseline scenario than was considered in [14].

5.2 Specifying Tests of Model Performance

In an attempt to better understand the cause of the observed performance results, a series of tests was devised. The tests were designed to investigate the two different aspects of the operation of the model already mentioned (cycling personnel offline and online and the assignment of personnel to teams) as different aspects of the size of the scenario were scaled up. There are three dimensions to the size of scenarios that we investigate below: the number of personnel; the number of teams that simultaneously require assignment of personnel; and the number of nodes (maintenance workshops) in the scenario.

The process of assigning personnel to teams is affected by all three dimensions. The process of cycling personnel online and offline is affected only by the number of personnel and the number of nodes in the scenario, and is independent of the number of teams that require personnel. Hence, five different tests were considered. Only three tests were considered in [14]: cycling personnel online and offline as the number of personnel increases; and assigning personnel to teams as the number of teams increase and the number of personnel increase.

To accurately gauge the individual contribution of these two processes to the performance of the simulation, the following tests consider these two components in isolation[1] from the rest of the model.

Scaling the Number of Personnel To scale the personnel, we applied a *personnel multiplier* to the base number of tradespeople. By default, the personnel multiplier is one, giving 10 tradespeople in total (2 per node). A personnel multiplier of two gives 20 tradespeople in total (4 per node), and so on. The base number of teams and nodes remains unchanged, at 2 teams per node (10 in total) and 5 nodes respectively.

Scaling the Number of Tasks that Require Teams Scaling the number of tasks that require teams was done in a similar manner, using a *team multiplier*. Again, the default value of this multiplier is one, giving 10 tasks that require teams in total. The number of people and nodes remains unchanged, at 2 tradespeople per node (10 in total) and 5 nodes respectively.

Scaling the Number of Nodes Scaling the number of nodes was not considered in [14]. This required changes to the Model Initialisation page, hence there is no simple *node multiplier* per se. Rather, we refer directly to the number of nodes in the scenario. When scaling the number of nodes, each node is as described in the baseline scenario (2 tradespeople and 2 teams per node) unless otherwise stated.

Folded and Unfolded Network Topology To further understand the nature of the performance problems, we also examine the impact of having the network topology represented in data (a folded model) versus the network topology represented in net structure (an unfolded model). Again, this was not considered in [14], so the investigation of the impact of topology representation on simulator performance is an entirely new aspect considered in this paper.

The maintenance process captured by our model occurs within all maintenance nodes in the maintenance network. The idea of representing network topology by data was to avoid duplication of the net structure representing the process carried out at each node. Hence, a folded model brings with it the advantage of compactness and provides flexibility in terms of the scenario to be analysed, as the topology of the scenario is encoded in tokens. An unfolded model that represents topology explicitly

[1] These two processes were isolated by deleting all unrelated model pages and executable transitions.

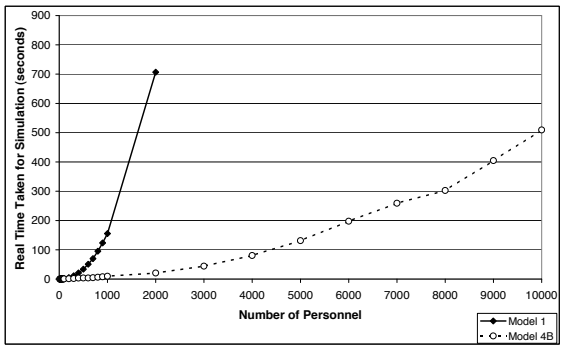


Fig. 5. Test 1: Performance of Personnel Cycling Offline and Online as No. of Personnel Increases.

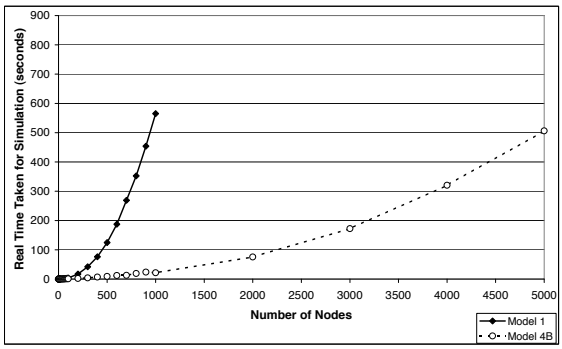


Fig. 6. Test 2: Performance of Personnel Cycling Offline and Online as No. of Nodes Increases.

in net structure requires modification of the net structure each time a new topology is analysed. However, unfolding the network topology has an advantage when specifying the maintenance network, as it provides direct spatial visualisation of the topology. This is helpful when the reader is trying to relate the model to the system (for validation) and for understanding materiel flows between different nodes. There is no similar benefit to be gained by unfolding the other data structures, such as equipment type, and doing so may result in a model that is less clear. Hence we decided to experiment with unfolding topology. From the analysis point of view, the motivation for producing an unfolded model is that the calculation of enabled binding elements becomes simpler. By unfolding the network topology, CPN Tools does not need to compute values for variables relating to different node locations, thus the number of computations is reduced.

To produce unfolded variants of the two sub-models considered in Section 5.3 we manually duplicated the page in question the number of times required (one page per node) and adjusted the Model Initialisation page accordingly. Hence, when scaling the number of nodes, we change the net structure. For this reason, we limit the number of nodes to range from 1 to 10 when analysing the unfolded models.

5.3 Performance of the Folded Models

For the five tests described above, we ran automatic simulations for 20 days of modelled time, whilst recording the real duration of the simulation. Results and discussion are presented below.

Test 1: Personnel Cycling Offline and Online when Increasing the Number of Personnel. Figure 5 shows a graph of the real time taken for each run as the personnel multiplier increases, for both models. Model 4B significantly outperforms Model 1 in this test (20.5 seconds vs 706.7 seconds at 1000 personnel, a factor of 34) confirming the result reported in [14]. The reason is most likely the ease with which an enabled binding element can be found for the two transitions that move personnel offline and online. There is only one list of personnel that are online, and one list of personnel that are offline, instead of a (potentially) large multiset of personnel from which to select.

However, we must be mindful of the fact that Model 4B's performance will deteriorate as the proportion of personnel eligible to move offline or online at the same model time decreases, e.g. as personnel availability become increasingly staggered due to performing maintenance work. There will be greater numbers of transition occurrences involving smaller numbers of personnel in each occurrence. The worst case corresponds to just the head of the personnel list being selected each time. Tests in [14] showed that this actually performs worse than Model 1. Hence, while Model 4B appears to be the clear winner in terms of performance when cycling personnel offline and online under the ideal circumstances of the test scenarios, it may deteriorate to perform worse than the original model.

Test 2: Personnel Cycling Offline and Online when Increasing the Number of Nodes. Figure 6 shows how the performance of the two models scales with the number of nodes when cycling personnel offline and online. The results look remarkably similar to those obtained when increasing the number of personnel (Fig. 5). At 1000 nodes, Model 1 took 565.1 seconds while Model 4B took 21.5 seconds - a factor of 26 improvement. This may be explained by considering what happens as the number of nodes

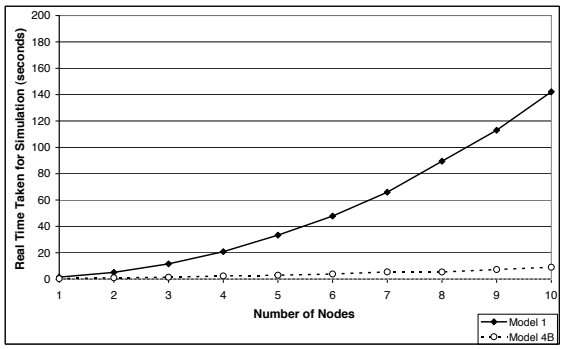


Fig. 7. Test 2*: Performance of Personnel Cycling Offline and Online as No. of Nodes Increases, with 100 Personnel per Node.

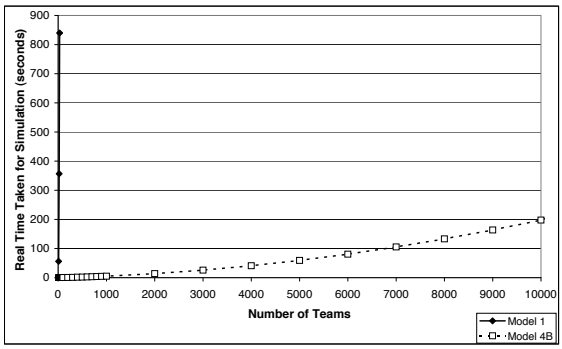


Fig. 8. Test 3: Performance of Assigning Personnel to Teams as No. of Teams Increases.

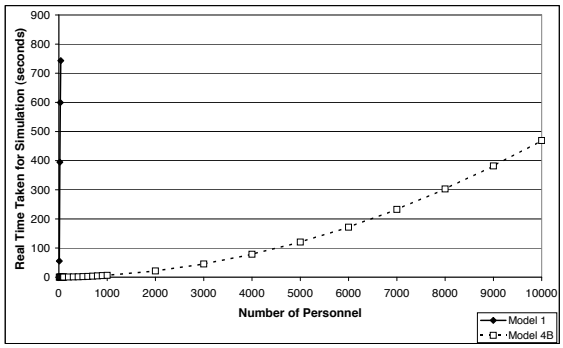


Fig. 9. Test 4: Performance of Assigning Personnel to Teams as No. of Personnel Increases.

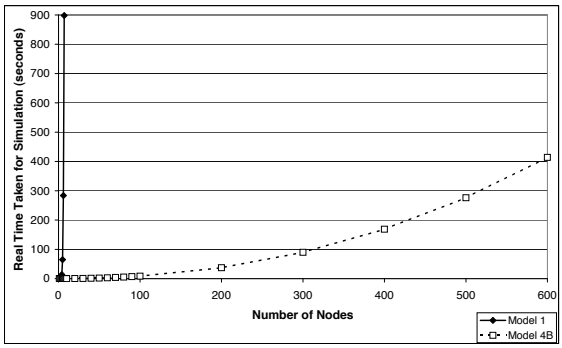


Fig. 10. Test 5: Performance of Assigning Personnel to Teams as No. of Nodes Increases.

increases. The nature of the model is such that although the personnel at each node may be conceptually geographically separate, they all appear on the same place in the model (the Personnel place) either as extra tokens (Model 1), or as extra elements in lists (Model 4B). Hence, the effect of increasing the number of nodes is the same as increasing the number of personnel.

This is not the case when considering the corresponding unfolded model, discussed later in Section 5.4. In order to allow direct comparison of results between the folded and unfolded models, we repeat Test 2 (as Test 2*) but only for the number of nodes from 1 to 10, where there are 100 personnel at each node instead of two (i.e. a personnel multiplier of 50). In this instance, because the range of nodes covered is small, we have increased the personnel multiplier so that the results obtained can be differentiated. These results are shown in Fig. 7, where we see that for 10 nodes, Model 4B (9 seconds) outperforms Model 1 (142.1 seconds) by a factor of 16.

Test 3: Assigning Personnel to Teams when Increasing the number of Teams. The first of three performance tests for assigning personnel to teams keeps the number of personnel fixed (personnel multiplier of 1) but increases the number of teams that simultaneously require personnel, by increasing the team multiplier. In order for the supply of available personnel at each node in the baseline scenario not to become exhausted before all teams are allocated (recall that there are only two tradespeople per node) personnel assigned to a team are immediately released back into the Personnel place, ready for assignment to another team. This is reasonable, given that we wish to analyse the process of assigning personnel to teams in isolation from the rest of the model, and hence we exclude the process of tradespeople performing maintenance before being released. The model was executed until all teams were assigned. Figure 8 shows a graph of the results obtained.

In this situation, Model 4B significantly outperforms Model 1. The three points shown for Model 1 are for 10, 20 and 30 teams. Assigning personnel to 30 teams took Model 1 839.9 seconds but Model 4B only 0.0625 seconds - a factor of 13400 improvement at 30 teams. (The factor of improvement is 6500

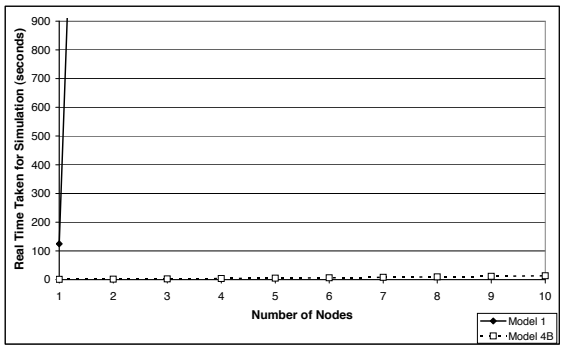


Fig. 11. Test 5*: Performance of Assigning Personnel to Teams as No. of Nodes Increases, with 200 Teams per Node.

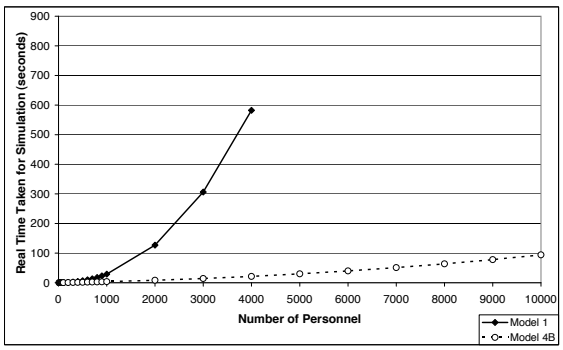


Fig. 12. Test 1^U: Performance of Personnel Cycling Offline and Online in an Unfolded Model as No. of Personnel Increases.

for 20 teams.) This may indicate how inefficient the mechanism for assigning a group of personnel of unknown composition and varying size implemented in Model 1 is.

Test 4: Assigning Personnel to Teams when Increasing the Number of Personnel. The second of the three tests for assigning personnel to teams keeps the number of teams requiring personnel fixed (at 10, 2 per node) but increases the number of personnel. A graph of this is shown in Fig. 9. Again, we see that Model 4B drastically outperforms Model 1. With 40 personnel to choose from, Model 1 took 743.2 seconds to assign personnel to all 10 teams, whereas Model 4B took only 0.101 seconds to do the same. This is a factor of 7350 improvement.

Test 5: Assigning Personnel to Teams when Increasing the Number of Nodes. Figure 10 shows the results obtained when scaling the number of nodes in the scenario. As in the previous two tests, the model of personnel using lists outperforms the baseline model of personnel as tokens. Model 1 took 898 seconds to assign personnel to teams in 7 nodes (2 teams per node), whereas Model 4B took only 0.0465 seconds - a factor of 19300 improvement at 7 nodes.

As was the case for Test 2, we also wish to repeat Test 5 for the purposes of comparison with the results from the equivalent unfolded model in Section 5.4. We again consider the number of nodes ranging from 1 to 10, but with 200 tasks requiring teams (requiring personnel) per node instead of 2 (a team multiplier of 100). We maintain the baseline of 2 tradespeople per node. The results of this test (Test 5*) are shown in Fig. 11. Only one point is shown on the graph for Model 1. This is for one node, where it took 125 seconds to assign personnel to all 200 teams. The next point on Model 1's curve is not shown, as for two nodes it took Model 1 5434 seconds (approx. 1.5 hours) to assign personnel to all 400 teams. Model 4B took only 1.67 seconds to assign personnel to all 400 teams, giving a factor of improvement of 3250 for two nodes. The increased number of teams has magnified the performance difference between Model 1 and Model 4B, contributing to the belief that Model 1's arbitrary personnel selection mechanism is highly sensitive to the number of tokens involved.

5.4 Performance of the Unfolded Models

Each test presented below is named according to the corresponding test in Section 5.3, but with a superscript 'U' to indicate that the test was carried out on the corresponding unfolded model. We have deliberately kept the X-axis scale of the graph the same as the corresponding graph in Section 5.3, to facilitate comparison. Note that in the case of Test 2^U the Y-axis scale has been reduced to 0-20 from 0-200 and in the case of Test 4^U the Y-axis scale has been reduced to 0-100 from 0-900 to avoid large areas of whitespace and to make the graphs easier to read.

Test 1^U: Personnel Cycling Offline and Online when Increasing the Number of Personnel. Figure 12 shows the results obtained when analysing the effect of increasing the number of personnel on the process of cycling personnel offline and online. Overall there was an improvement in computation time for both models for the unfolded case when compared to Fig. 5. For example, the unfolded Model

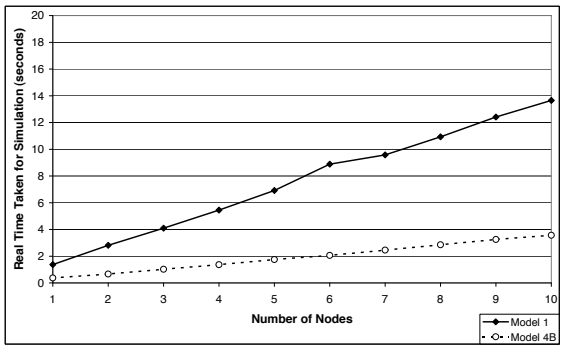


Fig. 13. Test 2^U: Performance of Personnel Cycling Offline and Online in an Unfolded Model as No. of Nodes Increases.

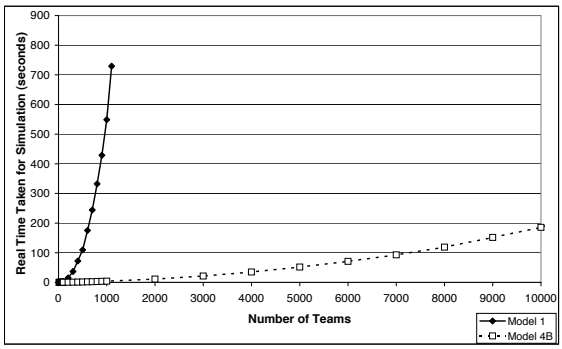


Fig. 14. Test 3^U: Performance of Assigning Personnel to Teams in an Unfolded Model as No. of Teams Increases.

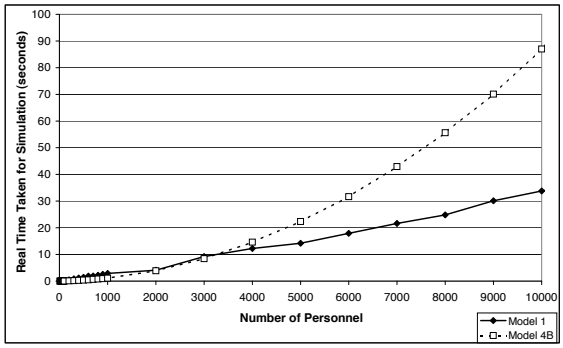


Fig. 15. Test 4^U: Performance of Assigning Personnel to Teams in an Unfolded Model as No. of Personnel Increases.

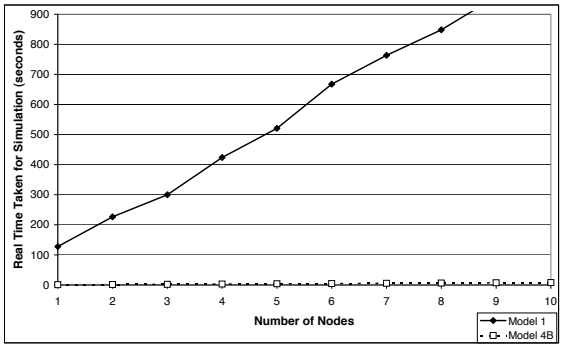


Fig. 16. Test 5^U: Performance of Assigning Personnel to Teams in an Unfolded Model as No. of Nodes Increases.

4B takes around 100 seconds to cycle 10,000 personnel offline and online for 20 model days, compared to 500 seconds for the folded Model 4B (Fig. 5) giving a factor of 5 improvement in performance due to unfolding. Model 1 has improved by approximately a factor of 6 at 2000 personnel. We suspect this is due to simpler calculations when determining enabled binding elements, but have yet to confirm this.

Test 2^U: Personnel Cycling Offline and Online when Increasing the Number of Nodes. This test reveals a major benefit when considering an unfolded model. Looking at Fig. 13, we see that the real time taken to cycle personnel offline and online scales linearly with the number of nodes. This is opposed to Fig. 7 in which real time taken appears to increase exponentially with the number of nodes. This linear growth is also evident in experiments that use a larger personnel multiplier (not shown in this paper). This linear behaviour makes sense intuitively, as each node is represented by a separate piece of non-interacting net structure. CPN Tools essentially repeats the same set of calculations for each separate node, hence one would expect the computational effort to be proportional to the number of nodes.

Test 3^U: Assigning Personnel to Teams when Increasing the Number of Teams. The results of this test are shown in Fig. 14. Unfolding the topology has not significantly affected model performance when modelling personnel in lists; indeed the curve for Model 4B shows only marginal improvement when compared to that in Fig. 8. However, whilst Model 4B still outperforms Model 1, there is a significant increase in the performance of Model 1. The gains in performance exhibited by Model 1 can be attributed to the fact that the number of tokens at any one place is reduced (distributed over multiple places), hence speeding up the calculations for enabling and firing of transitions. This effect is more pronounced for Model 1, where a reduction in the number of tokens to select from results in a more significant speed-up due to the inefficient mechanism for selection of personnel discussed in Section 4.1.

Test 4^U: Assigning Personnel to Teams when Increasing the Number of Personnel. This test (Fig. 15) reveals that, for the unfolded Model 1, the time taken to assign personnel to teams scales approximately linearly in the number of personnel. Figure 15 shows a significant improvement in performance when compared with Fig. 9. The folded Model 1 took over 700 seconds to assign personnel to teams when there were 40 personnel to choose from. The unfolded Model 1 took only 0.0785 seconds to do this. Indeed, there is a complete reversal of the performance trend, with Model 1 increasingly outperforming Model 4B as the number of personnel increases, despite Model 4B having improved in performance by a factor of 5 at 10,000 personnel.

Extending this experiment for Model 1 to 100,000 personnel (not shown) provides additional evidence of a linear relationship. This linear relationship may also be present for Model 1 in Fig. 9, however the increase in simulation time as the number of personnel increases is so severe that we cannot readily confirm this by experimentation. We conjecture that the mechanism in Model 1 for the selection of personnel is highly sensitive to the number of distinct personnel tokens from which to choose and the number of teams that require personnel, but is relatively independent of the multiplicity of the individual personnel tokens involved. Conversely, the list manipulation operations in Model 4B are affected by the length of the lists, i.e. the number of personnel under consideration. The length of the personnel lists increase regardless of whether the topology is folded or unfolded, the only difference being that when the topology is unfolded there are 5 lists of 'Ready' personnel (one per node) instead of a single list.

Test 5^U: Assigning Personnel to Teams when Increasing the Number of Nodes. In this test, the performance of both models scaled approximately linearly with the number of nodes. (Recall that each node has 200 teams that require personnel, a team multiplier of 100.) This is expected, since each new node is represented by its own piece of net structure. Figure 16 shows that Model 1 is also outperformed by Model 4B in this experiment. Despite this, Model 1 has shown dramatic improvement in performance. With one node, both the folded and unfolded models take approximately 125 seconds to assign personnel to all teams. With two nodes, the folded model took over 1.5 hours whereas the unfolded model took less than four minutes (225 seconds). Model 4B has improved, but not nearly as much. For 10 nodes the improvement is a factor of 1.68 (13.3 seconds folded vs. 7.94 seconds unfolded).

6 Discussion

We consider that model design for good model performance is difficult. The choice of modelling style, such as the choices for modelling personnel and topology discussed above, has a significant effect on the performance of the model. With respect to our original model (Model 1), we note that when there are complex interactions between personnel and tasks, the check for enabled binding elements results in a prohibitively large number of calculations. This has been greatly reduced for Model 4B, which performs well in most scenarios examined here. This is the case so long as personnel remain in step with respect to their scheduled offline and online cycle times. Otherwise, it may diminish performance rather than enhance it (compared to Model 1 - see [14] for more details).

When considering a folded network topology, modelling personnel as individual tokens results in highly inefficient simulations. A list-based representation appears to be computationally far superior without losing much of the desired behaviour of the token-based approach. Unfolding the network topology into net structure also improves the performance of the models, with Model 1 exhibiting the most significant improvement.

The above tests purposely cover extreme values of personnel, teams and nodes to elicit performance trends. For realistic scenarios we can see that Model 4B provides an acceptable level of performance for the case of cycling personnel and assignment of personnel to teams in isolation. However, this may not be the case when considering the model as a whole, as it is difficult to directly infer the performance of the complete model from these results. We are currently investigating the performance of both approaches to modelling personnel in the complete model.

Unfolding other aspects of the model may provide significant performance gains, but it is a significant undertaking and there comes a point at which the unfolded model becomes unmanageable. We must also consider the loss in flexibility that moving to an unfolded model would impose. For example, it would be useful to specify the tool facilities that could be developed to produce a new model for each distinct topology that needs to be simulated. Producing (partially) unfolded versions of the complete model for one or two carefully chosen scenarios may help to clarify the advantages and disadvantages.

We would also like to investigate a way of modelling personnel changing state from online to offline, and vice versa, that does not involve dedicated transition occurrences. Because the cycling of personnel is predetermined (in the absence of maintenance tasks) it follows that by knowing the most recent maintenance activity of all personnel, all future availabilities (until the next maintenance activity starts) can be derived. Therefore, explicit modelling of the transitions of personnel online and offline may not be necessary.

7 Conclusions

This paper has explored different modelling approaches to improve the performance of CPN Tools when simulating an industrial-scale CPN model that captures the Australian Army's maintenance process. Our aim was to improve simulation performance to a level that is suitable for timely evaluation of different maintenance scenarios.

In its usual operating environment, the maintenance system involves hundreds of personnel and thousands of pieces of equipment distributed over tens of locations. Under these conditions, simulations of our original model did not allow any results to be obtained within an hour. This led us to profile the model to determine where the bottlenecks in the simulation occurred. We found that the internal CPN Tools' function, 'less than', for the personnel colour set was being executed millions of times during the checking of the enabling of two transitions, both associated with assigning personnel to teams. Given this, we decided to explore different data structures for personnel. We have made some dramatic performance gains over our original model by using lists of personnel in the maintenance process, rather than the more natural use of multisets. Two components of the model (cycling personnel online and offline, and assigning personnel to teams) are now nearing the levels of performance required to make the model fast enough to be useful in supporting simulated military operations.

We have also considered models in which the network topology of the maintenance system is not encoded in tokens but rather is represented explicitly in the net structure (i.e. a partial unfolding of the CPN). This approach has also shown promise, opening up many possibilities to further enhance the performance of the model.

It is likely that the performance issues discussed in this paper will be present in many industrial-scale CPN models. We therefore believe that a more fundamental understanding of the relationship between the use of various modelling constructs and their impact on analysis and simulation performance in CPN Tools will be beneficial to the user community, particularly those concerned with industrial systems. We hope that the results in this paper will provide a starting point for the development of a set of guidelines for modelling complex systems that are more readily simulated by CPN Tools.

Acknowledgments

The authors would like to acknowledge their colleague, Mr. Christopher Moon, for his involvement in the early development of this model, and Dr. Nimrod Lilith, for support in developing some aspects of the model and for participation in many active discussions. The authors would also like to acknowledge Dr. Lisa Wells and Prof. Lars Kristensen for productive discussions and information regarding profiling. Finally, the authors would like to acknowledge the constructive feedback received from the anonymous reviewers of [14].

References

1. W.M.P van der Aalst (guest editor). *Transactions on Petri Nets and Other Models of Concurrency II, Special Issue on Concurrency in Process-Aware Systems*. Lecture Notes in Computer Science. Springer, 2009.
2. J. Ashayeri, A. Teelen, and W. Selen. Production and maintenance planning model for the process industry. *International Journal of Production Research*, 34(12):3311–3326, 1996.
3. J. Billington, M. Diaz, and G. Rozenberg, editors. *Application of Petri Nets to Communication Networks*, volume 1605 of *Lecture Notes in Computer Science*. Springer, 1999.
4. L. Briskin and W. S. Demmy. An overview of the network repair level analysis model [military systems]. In *Proceedings of the IEEE 1988 National Aerospace and Electronics Conference (NAECON 1988)*, volume 4, pages 1414–1420, 1988.
5. C. R. Cassady, W. P. Murdock, and E. A. Pohl. Selective maintenance for support equipment involving multiple maintenance actions. *European Journal of Operational Research*, 129(2):252–258, 2001.

6. T. N. Cook and R. C. DiNicola. Modeling Combat Maintenance Operations. In *Proceedings of the Reliability and Maintainability Symposium*, pages 390–396, 1984.
7. CPN Tools. `http://www.cs.au.dk/CPNTools`.
8. DSTO. *Coloured Petri Net Modelling of Defence Logistics*. Australian Government, Department of Defence, Contract No. 4500498737, 1 February 2006.
9. W. W. Fisher. Markov process modelling of a maintenance system with spares, repair, cannibalization and manpower constraints. *Mathematical and Computer Modelling*, 13(7):119–125, 1990.
10. O. Fouathia, J.-C. Maun, P.-E. Labeau, and D. Wiot. Stochastic approach using Petri nets for maintenance optimization in Belgian power systems. In *Proceedings of 8th International Conference on Probabilistic Methods Applied to Power Systems*, pages 168–173, 2004.
11. G. E. Gallasch and J. Billington. Army Maintenance System Analysis Tool Enhancement: CPN Model Documentation. Technical Report CSEC-39, Computer Systems Engineering Centre Report Series, University of South Australia, September 2009.
12. G. E. Gallasch, N. Lilith, and J. Billington. Extending a Coloured Petri Net Model of a Defence Logistics Network. Technical Report CSEC-29, Computer Systems Engineering Centre Report Series, University of South Australia, September 2007.
13. G. E. Gallasch, N. Lilith, J. Billington, L. Zhang, A. Bender, and B. Francis. Modelling Defence Logistics Networks. *International Journal on Software Tools for Technology Transfer, special section on CPN'06*, 10(1):75–93, 2008.
14. G. E. Gallasch, C. Moon, B. Francis, and J. Billington. Modelling personnel within a defence logistics maintenance process. In *Proceedings of 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems (SIMUTools 2008)*, March 2008. (10 pages).
15. T. Gossard, N. Brown, S. Powers, and D. Crippen. Scalable Integration Model for Objective Resource Capability Evaluations (SIM-FORCE). In *Winter Simulation Conference Proceedings*, volume 2, pages 1316–1323, 1999.
16. A. Jacopino, F. Groen, and A. Mosleh. Modelling imperfect inspection and maintenance in defence aviation through Bayesian analysis of the KIJIMA type I general renewal process (GRP). In *Proceedings of Reliability and Maintainability Symposium (RAMS'06)*, pages 470–475, 2006.
17. K. Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use. Volumes 1 to 3, Basic Concepts, Analysis Methods, and Practical Use*. Monographs in Theoretical Computer Science. Springer, 2nd edition, 1997.
18. K. Jensen, L. M. Kristensen, and L. Wells. Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems. *International Journal on Software Tools for Technology Transfer*, 9(3-4):213–254, June 2007.
19. K. Jensen and L.M. Kristensen. *Coloured Petri Nets: Modelling and Validation of Concurrent Systems*. Springer, 2009.
20. W. D. Kelton, R. P. Sadowski, and D. T. Sturrock. *Simulation with Arena*. McGraw-Hill Higher Education, 3rd edition, 2004.
21. L. M. Kristensen, P. Mechlenborg, L. Zhang, B. Mitchell, and G. E. Gallasch. Model-based Development of a Course of Action Scheduling Tool. *International Journal on Software Tools for Technology Transfer, special section on CPN'06*, 10(1):5–14, 2008.
22. O. Maimon and M. Last. Information-efficient design of an automatic aircraft maintenance supervisor. *Computers & Operations Research*, 20(4):421–434, 1993.
23. V. Mattila and K. Virtanen. A simulation-based optimization model to schedule periodic maintenance of a fleet of aircraft. In *Proceedings of European Simulation and Modelling Conference 2005 (ESM'2005)*, pages 479–483, 2005.
24. B. Mitchell, L. M. Kristensen, and L. Zhang. Formal Specification and State Space Analysis of an Operational Planning Process. In *Fifth Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools*, pages 1–17. Department of Computer Science, University of Aarhus, 2004. Available via `http://www.daimi.au.dk/CPnets/workshop04/cpn/papers/`.
25. S. R. Parker. Combat Vehicle Reliability Assessment Simulation Model (CVRASM). In *Winter Simulation Conference Proceedings*, pages 491–498, 1991.
26. T. Raivio, E. Kuumola, V. A. Mattila, K. Virtanen, and R. P. Hamalainen. A simulation model for military aircraft maintenance and availability. In *Proceedings of Modelling and Simulation 2001, 15th European Simulation Multiconference (ESM'2001)*, pages 190–194, 2001.
27. M. Ramadass, J. Rosenberger, B. Huff, S. Gonterman, and R. N. Subramanian. Simulation Modelling for a Bus Maintenance Facility. In *Winter Simulation Conference Proceedings*, pages 1222–1225, 2004.
28. J.D. Ullman. *Elements of ML Programming*. Prentice-Hall, 2nd edition, 1998.
29. L. Zhang, L. M. Kristensen, B. Mitchell, G. E. Gallasch, P. Mechlenborg, and C. Janczura. COAST - An Operational Planning Tool for Course of Action Development and Analysis. In *Proceedings of the 9th International Command and Control Research and Technology Symposium (ICCRTS), Copenhagen, Denmark.*, 2004.

Towards Automatic Code-generation from Process-partitioned Coloured Petri Nets

K.L. Espensen[1], M.K. Kjeldsen[1], L.M. Kristensen[2], and M. Westergaard[1*]

[1] Computer Science Department, Aarhus University, Denmark.
Email: {espensen,keblov,mw}@cs.au.dk
[2] Department of Computer Engineering, Bergen University College, Norway.
Email: lmkr@hib.no

Abstract. Constructing an abstract description in the form of a model can give useful insight into a given system, e.g., to investigate important properties of the system either through simulation or state space analysis, and to use the model as inspiration for subsequent manual implementation. The problem is that a manual implementation may introduce errors in the code not present in the model. Automatic code generation from the model saves resources spent on writing code, and eliminates errors introduced during implementation. This is difficult for coloured Petri nets models as their rich structure translates badly to common programming languages. Approaches either severely restrict the input accepted or generate code that is difficult to extend and modify. In this paper we introduce process-partitioned coloured Petri nets, which is an attempt to restrict the input accepted as little as possible while still allowing automatic inference of the control structure of the model to generate code that can be manually modified afterwards. We illustrate our approach using a simple example and demonstrate the viability of the approach by demonstrating that it can be applied to a model of a real-life system, the Dynamic MANET On-demand (DYMO) routing protocol.

1 Introduction

Software development is a challenging process, and writing a program of substantial size without errors is difficult. A major part of software development is therefore concerned with finding and eliminating errors. Testing is widely used as a technique to detect errors, but the programmer does not know whether the absence of failed test cases means a missing test case or that the software is free of errors. It is especially difficult to write exhaustive test cases for concurrent systems, e.g., for a communication protocol where several process instances are executing at the same time.

Building an abstract representation of the system in the form of a *model* is a way to detect errors early. A model can be used to verify properties of a system, e.g., that a system does not contain deadlocks, or that a communication protocol behaves correct when operating over an unreliable network. A typical

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

way of using models in software development is to build a model from a system specification written in plain text. After verifying that the model has the desired properties, it can be used as a basis for an implementation. A problem with this approach is that there may be a mismatch between the specification, the model, and the actual implementation. This is because the translation from one to another is done manually and hence can introduce errors. A way to reduce this problem is to use the model as the specification and automatically generate the implementation from the model. Details abstracted away in the model will of course also lack in the implementation, but eliminating errors in the verified parts of the system leads to more reliable software with fewer errors.

The aim of this paper is to develop a technique to automatically generate code from coloured Petri nets (CPNs or CP-nets) [7]. The code should be readable and intuitive such that the user can read, modify and extend the generated code. We also want the model to be clearly recognizable in the generated code since the people working with the generated code are typically also familiar with the model. The technique should allow different target languages to be used, e.g., C, Java, SML or Erlang [3]. However, the target language should be invisible in the model and the usual inscription language should be used in the model.

To achieve this aim, we use a sub-class of CP-nets called process-partitioned CP-nets (PCP-nets or PCPNs). PCPN models preserve much of the general-purpose strength of CP-nets as we show by constructing a model of the Dynamic MANET On-demand (DYMO) protocol [1]. We have developed a technique that translates from the class of PCP-nets to the Erlang programming language, and have created a prototype of the technique. The prototype is able to generate readable code from the DYMO model, and we validate that the generated code has the same behaviour as the model.

Related Work. There are different approaches to automatically generate code from Petri nets. The chosen strategy has a large impact on the properties of the final code. The approach should preserve the behaviour of the model, but the code generated using one approach might be very efficient while the code generated by using another approach may be very readable and extensible.

In [6] and [13] approaches to automatic code generation is divided into the four categories *simulation-based*, *structure-based*, *state space-based* and *decentralised*. Our approach falls into the structure-based approach.

Simulation-based. The basic idea in simulation-based approaches is to have a central component which controls the flow of the program on the basis of the state of the environment. This is done by a scheduler which from the current state computes which state to proceed to. This process corresponds to finding enabled transitions in CPN models.

A simulation-based approach is used by Philippi to generate Java code from a high-level Petri net in [13]. The idea is to make a class diagram which outlines the classes and method signatures of the program. From this diagram, classes with attribute definitions and methods with empty bodies are generated. The empty bodies are filled with the simulator code made from the formal model.

Simulation-based approaches are also used in the projects described in [12] and [8] where the generated simulator code made from a CPN model (by CPN Tools [2]) is used directly in the final implementation. The simulation kernel is generated from a CPN model and after undergoing automatic modifications, e.g., linking the code to external code libraries, the generated code is used in the final implementations.

One advantage of simulation-based approaches is that code execution follows a simulation of the model very closely, making it easier to establish that the behaviour of the generated code is the same as the behaviour of the model. Naturally, such approaches do not put any limitations on the class of nets to generate code for. The main disadvantage of these approaches is that the generated code is not very natural and often inefficient.

Structure-based. The code generated using a structure-based approach contains no central component to control the execution of the program. Instead the control flow of the program is distributed across the program, e.g., to function calls in functional programming languages. The key idea of these approaches is to recognise *structure* (regular patterns) in the model. Structure is then mapped to well-known programming constructs like sequences, loops, and case constructs. It is not in general possible to recognize such structure in coloured Petri nets mainly because they provide much more opportunities for constructing different control flow structures than common programming languages [13]. Because of this, it is necessary to restrict the class of nets when using a structure-based approach.

A structural approach is found in [6]. In this approach the focus is on identifying processes in a Petri net, i.e., parts of the net that work independent of one other or only have few synchronisation points. Afterwards local variables (i.e., information only used by one process) and communication channels are found.

In [14] the authors translate a class of CP-nets, called coloured workflow nets (CWNs), into BPEL, an XML-based workflow implementation language. CWNs are quite restricted, and mainly focus on the flow of data and not much on data processing, making the approach basically a graphical way to describe control structure instead of a natural way to make CPN models. Furthermore, the BPEL language is not aimed at general application development. [10] improves on this by translating directly to Java by adding a data processing component, but it is very restricted and does not allow the use of general functions in the data processing part. Furthermore, the approach is limited to emitting Java code.

The advantage of using structure-based approaches is that the code obtained is more readable than code obtained with a simulation-based approach. The coding style is more natural and looks more like it is written by a human programmer. The generated code also has a tendency to be more efficient because it does not rely on a central component. The main disadvantage is that the requirements on the modeling language may make the models unnatural.

State space-based. The idea of state space based approaches is to use the state space of the model to compute the next state. In the state space, we have all

successor states computed for each reachable state which alleviate the overhead of computing the successors each time. Relying on the full state space to be generated is a huge drawback because of the state space explosion problem, and therefore we do not find this method worth pursuing.

Decentralised. The opposite of centralised simulation-based approaches are decentralised approaches. The idea is to implement each place and transition of the net as processes. Here the program does not directly reflect the structure or state of the system. This approach has the advantage that parallelism in the net is preserved, but it also introduces an overhead because of the administration needed, e.g., for locks and message passing.

The rest of this paper is structured as follows: In the next section, we introduce our net class, process-partitioned CP-nets via a simple example. In Sect. 3, we describe our translation algorithm, and in Sect. 4 we describe our experiences with application of our prototype to a model of a real-life protocol made before the definition of the net class. Finally, we sum up our conclusions and provide directions for future work. Part of this work has been published as [4]. The main change in this version is that the presentation has been improved and shortened.

2 Process-partitioned Coloured Petri Nets

We use a simple producer-consumer system as example. The example can be seen in Fig. 1. The system consists of a number of producers that produce data and send it to the consumers (the top part of the model), and a number of consumers consuming the data (the bottom part of the model). Producing data is split up into producing the data and transmitting the data to the consumers. The producers have local Data, which contains the next data value to produce. When a data item is produced, it is transmitted to Produced Data for transmission. Each producer sends its data to a specific consumer, getting the identity of the consumer from the place Next Consumer, and transmitting it onto Buffer. Currently the identity of the receiving consumer is hard-coded to consumer c(1), but we can easily replace this by a load balancer. Consumers receive data from Buffer, which is a simple model of a network, transmit it to the Received Data place, where it will be consumed.

A general coloured Petri net is not limited to regular control flow structures in the same way common programming languages are. For this reason it is not easy to capture the behaviour of a CPN model by using common programming constructs, e.g., sequences, loops, and case-statements. We note that while the model is indeed a CPN model, it is modelled slightly differently from how one would normally go about it. Most notably, we see that we always both consume and produce tokens on places with data in the name and that we explicitly bind the consumer on the Send Data transition. This is because the model is created using the sub-class *process-partitioned coloured Petri nets* (PCPNs or PCP-nets). PCPNs are defined in a way that makes it possible to recognise

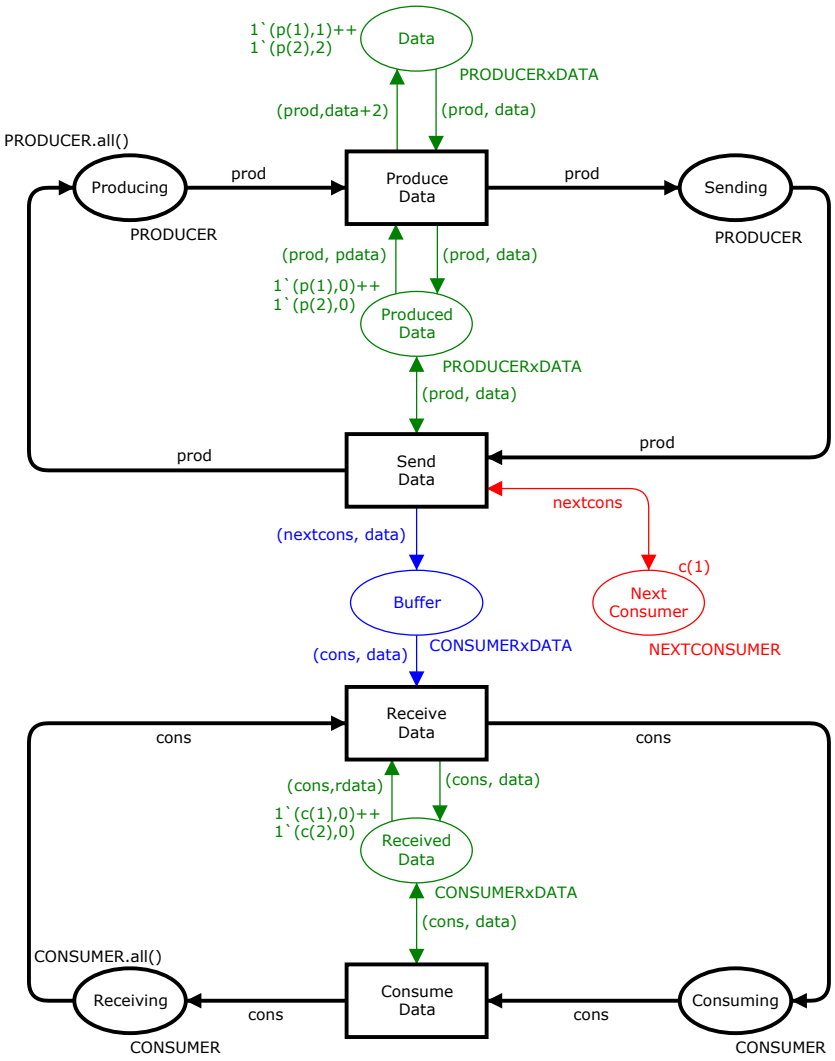


Fig. 1: The producer-consumer CPN model.

control structures and thus to generate code from the model. The definition of PCP-nets is inspired by the definition by Kristensen and Valmari from [9].

A main property of PCP-nets is that they are partitioned into processes that can be executed in parallel without influencing the behaviour of each other except for distinguished synchronisation points. Another important property is

that the control flows of processes are explicit in the net structure, so the state of the model always reflects where the process is in the control flow. Furthermore, access to stored values local to each process partition is also explicit in the model, allowing us to determine the local state of processes.

Here we introduce PCP-nets using the producer-consumer example; for a formal definition, please refer to [4]. The model has two *process partitions*, one modelling producers (top) and one modelling consumers (bottom). Process partitions can be connected by either *buffer* or *shared* places, but are otherwise disjoint. In Fig. 1, the producers and consumers are connected only by the buffer place Buffer. Intuitively, a process partition models the state and actions of one or more *process instances* running the same program code, e.g., producer process partition models two producer process instances running the same program code in the example. Transitions in a PCP-net belong to a unique process partition, e.g., the transition Send Data in Fig. 1 belongs to the producer process partition.

There are four kinds of places in PCP-nets: *process places*, *local places*, *buffer places* and *shared places*. In Fig. 1, process places are black (Producing, Sending, Receiving, and Consuming) and represent the control flow of processes. Process places have distinguished *process types*, here PRODUCER or CONSUMER, and we impose the restriction that every token from a process type, a *process token*, must reside on exactly one process place. We ensure this by requiring that transitions are always connected to exactly one input and one output process place and that the arc expression must be a variable (allowing a double arc instead of two arcs with the same inscription). We call the variable used the *process variable* of the transition. We require that initially, all process tokens of a given type reside on the same place, corresponding to all processes starting at the same point. For example, initially all producer tokens reside on Producing.

Local places in Fig. 1 are green (Data, Produced Data, and Received Data), and represent variables local to a process. We require that local places have a type that is a product between a process type and a data type. For example, Data has type PRODUCERxDATA, the product of PRODUCER and DATA. We require that if a transition has an arc from a local place, it must also have an arc leading to the place (and vice versa), arc expressions must be pairs where the first component is the process variable of the transition, and each local place must initially have exactly one token for each process token (together ensuring that local places always have exactly one value for each process instance). Finally, the second component of the expression on the arc from a local place must always be a variable only bound on that arc (this ensures that reading a variable from a local place never disables a transition).

Buffer places are blue in Fig. 1 (Buffer) and represent a communication channel between two processes. Like local places, the type of a buffer place must be a product of process type and a data types. Buffer places may contain any number of tokens, but the initial marking is required to be an empty multi-set (corresponding to the communication channel containing no data). Buffer places are allowed to have any number of arcs as long as outgoing arcs have expressions that are pairs of the process variable and an otherwise free data variable (like

for local places), but we impose no special requirements on arcs going into buffer places.

Shared places are red in Fig. 1 (Next Consumer) and represent data shared between multiple processes, corresponding to shared memory. Shared places can have any type that is not a process type (which is why we use NEXTCONSUMER on Next Consumer instead of just CONSUMER). The reason for that is to be able to distinguish shared places from the other kinds of places. We require that a shared place has an initial marking of size one (corresponding to the variable having exactly one value), and we preserve this by always requiring that any transition with an arc from a shared place also has an arc to the shared place.

We require that all arc expressions evaluate to multi-sets of size one to preserve the flow in process partitions. We also require that except for process variables, all variables exist at most once in all expressions on input arcs around one transition. This is to make the enabling calculation simpler in the generated code. It is still possible to make equality tests in the guards, however. We do not allow free variables on output arcs or in guards, as this would correspond to drawing random numbers in programs. Randomness can still be introduced by explicitly calling a random number generator. Variables used in the guard must be bound from local places, i.e., we do not allow input from shared or buffer places, as this would introduce race conditions as we shall discuss later.

3 Translation Algorithm

In this section, we explain the techniques developed for translating a PCPN model into program source code. The producer-consumer system is used to illustrate each phase of the translation. The translation from PCPN models to the target language is divided into five phases. The idea is to move closer and closer to the target language in small steps. Figure 2 illustrates the phases of the translation. The three first phases (top) are independent of the target language, i.e., they make no assumptions about the target language. The first phase consists of decorating the different parts of the PCPN model to allow us to distinguish, e.g., process places and shared places. The second phase translates from the decorated PCPN model into a control flow graph (CFG) for each process partition, extracting the control flow from the model. In the third phase the CFG is translated into an abstract syntax tree (AST) for a simple language designed to be abstract enough that it can be translated into most programming languages. The control flow represented by the CFG is made explicit by, e.g., goto statements in the AST. The last two phases of the translation are shown at the bottom of Fig. 2. These are language dependent, i.e., the phases are specific to a target programming language. We have shown two possible target languages: Erlang and Java. In case of Erlang, the AST is translated into an Erlang syntax tree (EST), mapping the generic concepts of the AST to language specific constructs for the Erlang language. In case of JAVA, the AST is translated into a Java syntax tree (JST) and then into Java source code. The last phase pretty-prints a syntax tree for the concrete target language under consideration.

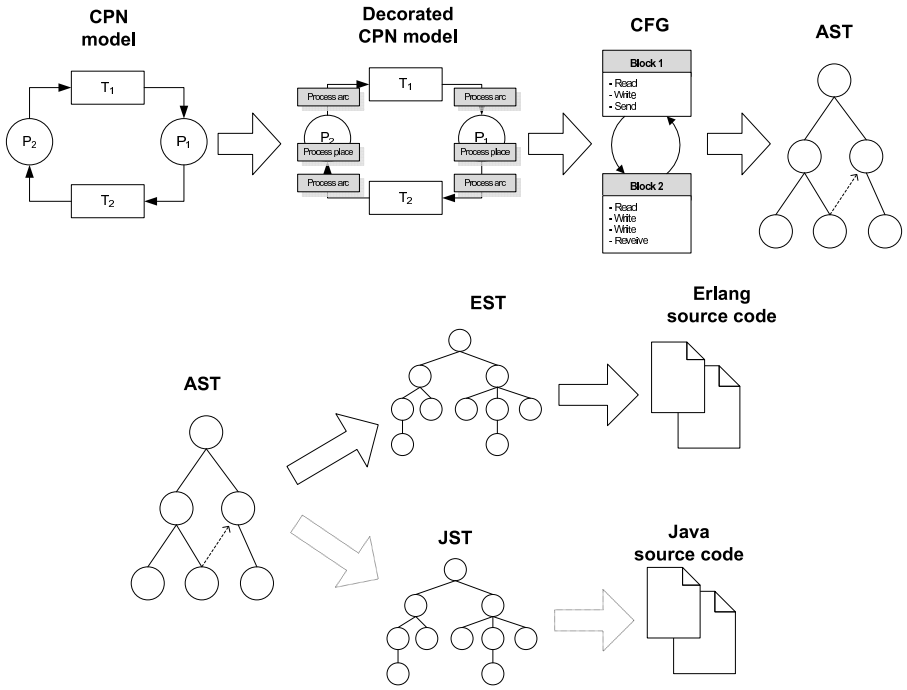


Fig. 2: The first three phases (top) and last two (bottom) of the translation.

Phase 1: Decorating the PCPN Model. The purpose of this phase is to identify different parts of the PCPN model and decorate them with information such as whether, e.g., places are shared places or process places. This phase uses properties of the PCPN net class to perform the identification. In our prototype, we actually perform this step, but it is likely that one would create an editor that would automatically provide this information, and ensure that the constructed model is indeed a PCPN model. For this reason, we will not give the full details of the decoration, but refer the interested reader to [4]. The main idea is that a user must identify process types, allowing us to find the process partitions. Local places can be identified as places only connected to transitions from a single partition with types that are products of the correct process type (that of the process variables of connected transitions) and another type. Buffer places can be recognized as places connected to multiple process partitions, and finally shared places are identified as places whose type is not a product containing a process type.

Phase 2: Translating the Decorated PCPN Model to a CFG. The main purpose of this phase is to extract the control flow from the decorated PCPN model and make it explicit in a control flow graph (CFG). This phase

also identifies common program constructs, e.g., processes, variables, and access to variables. Furthermore, the phase finds synchronisation points, i.e., messages passing between processes. The CFG we use is a directed graph in which arcs correspond to control flow and nodes corresponds to a sequence of statements to be executed. A CFG is constructed for all process partitions in the model, thus in the producer-consumer system two CFGs are generated: one for the producer process partition and one for the consumer process partition. In Fig. 3, we see the translated CFG for the producer process partition. Transitions are translated into *basic blocks* in the CFG. In the producer process partition, the transition Produce Data is translated into the basic block `produce data`. The contents of basic blocks depends on the places connected to the transition. There is a special `start` basic block for each CFG representing the start point of the process. In Fig. 3 it points to the basic block `produce data` as all tokens initially are on the Producing place, which is input to Produce Data.

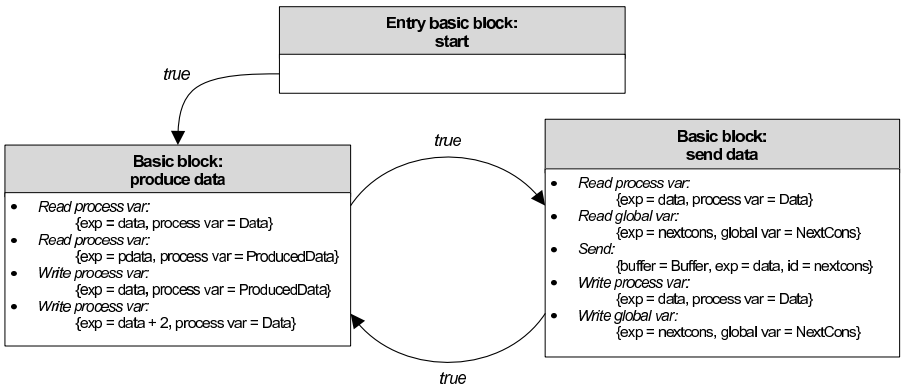


Fig. 3: The CFG of the producer process.

Process places in the PCPN model are not explicitly translated into nodes in the CFG, but are represented as edges between basic blocks. In Fig. 3 we see that the basic block `produce data` has an edge to `send data` signifying that after executing `produce data` control should flow to the basic block `send data`. The edge condition `true` indicates that control flows unconditionally.

Local places are used to store data. In a programming language this corresponds to reading/writing a process local variable. The producer process partition contains the two process variables `Data` and `ProducedData` corresponding to the two local places Data and Produced Data in the PCPN model. The initial values for the variables are extracted from the initial markings of the corresponding places. The initial marking of the local place Produced Data contains 0 for both instances and thus this is the initial value of this variable for both instances. An input arc with an arc expression on the form `(pid, var)` from a local place to a transition corresponds to process instance `pid` reading a vari-

able `var`. The input arc with expression `(prod, data)` from Data to Produce Data is translated to *Read process var* with the expression `data` as shown in Fig. 3. Output arcs corresponds to *Write process var*. `(prod, data)` on the arc from Produce Data to Produced Data is translated to *Write process var* with the expression `data`.

Buffer places are used to send data to a particular process instance. In a programming language this corresponds to sending to and receiving from a shared buffer, thus a buffer place is translated into a *buffer* in the CFG. Input arcs from a buffer place to a transition with arc expression `(pid, var)` correspond to a process `pid` receiving a message which is put into a variable `var`. This kind of input arc can be found in the consumer part of the producer-consumer system on the arc from Buffer to Receive Data (with expression `(cons, data)`) and is translated into *Receive* with the expression `data` meaning that the value of the received data should be read into the variable `data` for later use. Output arcs with expression `(pid, exp)` correspond to sending a value `exp` to a process instance `pid`. The output arc expression `(nextcons, data)` from Send Data to Buffer is translated to *Send* the expression `data` to the receiver process instance `nextcons`.

Shared places in the PCPN model are used to share data between multiple process instances. In a programming language this corresponds to a global variable, e.g., in shared memory. Shared places are translated into *global* variables in the CFG. In the producer-consumer system there is one global variable corresponding to the shared place Next Consumer. Initial markings of shared places are extracted and used as initial values. Input arcs with arc expression `var` from a shared place to a transition corresponds to a process reading a variable with a global scope. In the producer-consumer system, there is an input arc expression `nextcons` to the transition SendData from the shared place Next Consumer. This is translated to a *Read global var*. In a similar fashion, outgoing arcs are translated to *Write global var*.

Phase 3: Translating the CFG to an AST. The main purpose of this phase is to take the control flow given in the structure of the control flow graph (CFG) and translate it into a tree form consisting of nodes representing common programming constructs, e.g., jump statements. Furthermore, read and write expressions contained in the CFG are parsed and translated into trees in order to make them independent of the inscription language.

Figure 4 shows a sub-tree of the AST for the producer-consumer example where only the nodes from the `produce data` block of the `producer` process are shown. When building the AST, a process is created for each CFG process. Figure 4 shows the program contains two processes, namely `producer` and `consumer`. The program node also contains the global variable `NextConsumer`. In the CFG each process contains a number of variables. These variables are translated into process variables and contain an initial expression for each instance of the process. Expressions are parsed using a simplistic SML parser. An AST process also has a number of AST blocks which are created from the basic

blocks of the CFG. The `producer` process contains three such blocks: `produce data`, `send data`, and `start`. The AST contains nodes for reading and writing process and global variables as well as for sending and receiving via buffers. The rightmost node in the `produce data` block is an *unconditional goto statement* containing a pointer to the block it jumps to, in this case the `send data` block. Jump statements are translated from edges between basic blocks in the CFG. An AST can have two types of goto statements, a unconditional jump and a conditional jump with a boolean expression that must evaluate to true for the jump to take place.

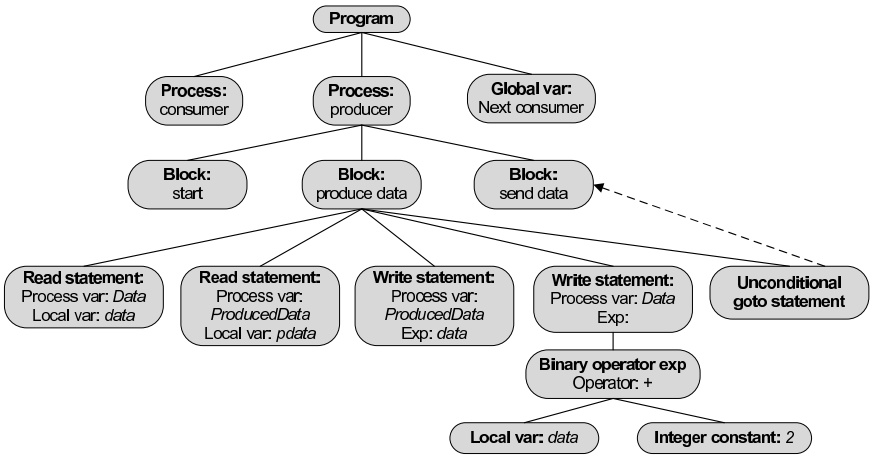


Fig. 4: The AST for the "produce data" part of the producer.

The AST can either be used as is or we can further transform it by extending the abstract syntax with, e.g., loops, if statements, or exceptions. These structured programming paradigms must be inferred from the control flow of the program. For example, we can identify an infinite loop in the producers of producing and transmitting. By doing this at the AST level, any new kind of control structure recognized can be used by all target languages.

Phase 4: Translating the AST to an EST. This phase generates an Erlang syntax tree (EST) based on an AST. The purpose of this phase is to translate the abstract representation of a program into an Erlang program represented as a tree. We have chosen Erlang as implementation because it is a functional language not too far from SML and it has been developed specifically for handling concurrent and critical systems, such as telephone centrals. The control flow, represented by goto statements in the AST, is translated into the functional language paradigm equivalent *function calls*. Since functional languages are stateless the state is passed along with the function calls. Processes are na-

tive in the Erlang language, thus each process in the AST is simply translated into a module. The generated modules are spawned in a special *system* module.

To give an impression of the translation from an AST to an EST we take a look at the producer-consumer system. Figure 5 shows how a part of the producer is represented in the generated EST (top) and the details of `produce_data` (bottom). The `producer` module basically consists of a function, `start` which is exported to the environment to allow the environment to start up processes, a declaration of the environment of the process containing all process variables, and functions for each basic block. Process variables are represented using an environment in the form of a record, so reading and writing translates to record operations. Global variables in the AST are used to share data between processes. There is no native equivalent to global variables in the Erlang language, so instead we construct a module which can be used to spawn processes that acts like global variables. Receiving messages via a buffer is a native construct in the Erlang language. To handle the translation of more advanced control flow constructs, a buffer with extra functionality is needed. For this reason we construct an explicit `buffer` process for each receiver to receive messages. *Goto* statements in the AST are divided into *unconditional* and *conditional* jumps.

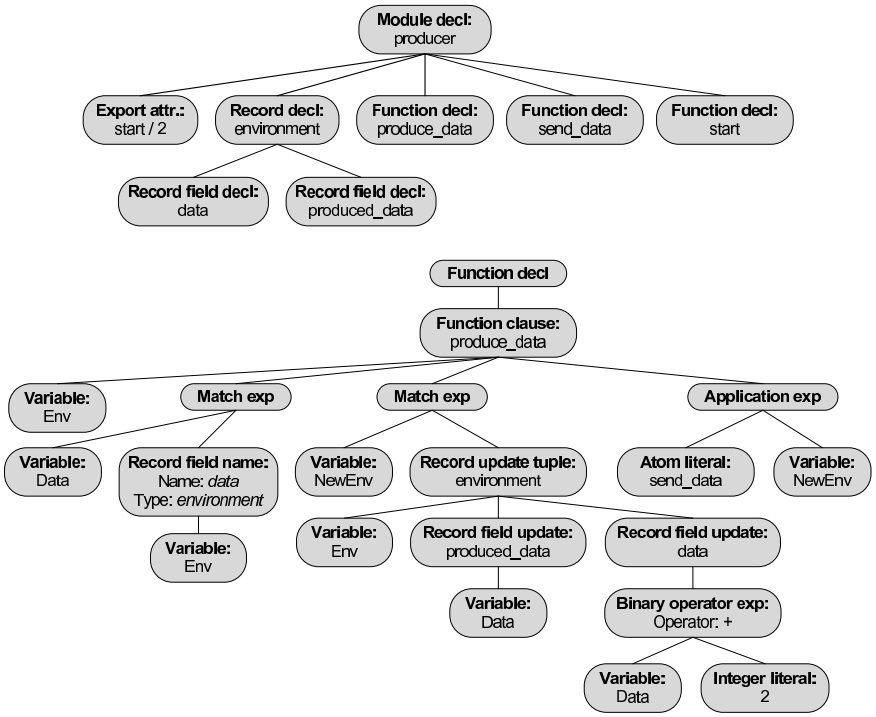


Fig. 5: The producer module (top) and function declaration `produce_data` (bottom) of the EST.

Unconditional goto statements are simply translated into function application expressions in the EST. In the example, we see an application of `send_data` at the far right.

We have a dedicated module to spawn an Erlang process for each process instance. It also spawns `buffer` processes and a process for each global variable. In the producer-consumer system this sums up to: two producers, two consumers, two buffers belonging to the consumers, and a shared instance for `NextConsumer`.

Phase 5: Translating the EST to Erlang Code. The last phase is translating the Erlang syntax tree (EST) into a textual representation. The EST is a concrete representation of Erlang, so the task is to traverse the tree and print a textual representation of each node to a file. A traversal is made for each module declaration because they represent different output files. Listing 1 shows a part of the generated Erlang code for the producer module. In Fig. 5 at the top we see the EST of the module declaration and at the bottom, we see the `produce_data` function declaration. We first declare our module (l. 1) and export the `start` function (l. 2). We then define our environment type (ll. 3–5) and move on to the definition of the `produce_data` function (l. 7–11) starting by reading `Data` from the environment (l. 8) and creating a new environment with updated values of the variables (ll. 9–10). Finally, we hand over control to the `send_data` function with the new environment (l. 11).

Listing 1: Part of the generated code for the producer module

```
1  - module(producer).
2  - export([start/2]).
3  - record(environment, {
4    produced_data,
5    data}).

7  produce_data(Env) ->
8    Data = Env#environment.data,
9    NewEnv = Env#environment {produced_data = Data,
10                             data = Data + 2},
11   send_data(NewEnv).
```

Advanced Control Flow Issues. While covering most of the constructs found in PCP-nets, the producer-consumer model does not contain a branch of the control flow. Here, we describe how branches of control flow are handled in the translation. In the producer-consumer model process tokens residing on a process place are only available to a single transition but the definition allows process tokens to be available to multiple transitions, i.e., a control flow branch. Control flow branches introduce an additional challenge when the target transitions have input arcs from buffer places. These buffer places have to be taken into consideration when choosing the flow of control. Making a function call without looking at the buffer may introduce a deadlock in the program that did not exist in the model. In Fig. 6 we see a part of a PCPN model with one process place Process Place, two transitions T1 and T2, and two buffer places Buffer1 and Buffer2. The process token can be removed by either T1 or T2, and

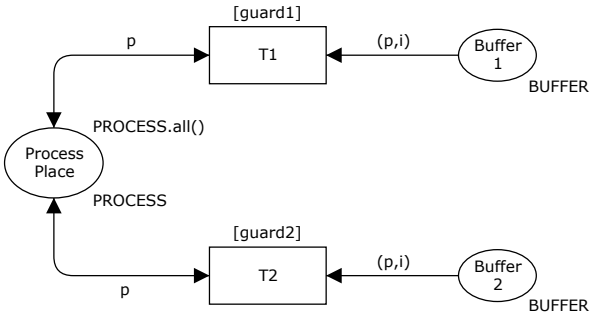


Fig. 6: A process partition with a control flow branch.

are in both cases put back on Process Place. T1 is enabled if guard1 evaluates to `true` and there is a token on Buffer1, and analogously for T2. This means that the generated process can proceed to either T1 or T2 depending not only on the guards but also on presence of tokens on buffer places.

Translation to a CFG. Given the PCPN model shown in Fig. 6, we generate the CFG shown in Fig. 7 (top). It contains an entry basic block `start` which has an edge with condition `guard1` to the basic block `T1`, and an edge with condition `guard2` to the basic block `T2`. The flow of control from `T1` is either to itself or to `T2` depending on the value of `guard1` and `guard2`, and analogously for `T2`. `T1` contains a receive statement from `Buffer1` and `T2` contains a receive statement from `Buffer2`.

Translation to an AST. The CFG is translated to the AST shown in Fig. 7 (bottom). The `Process` node contains two blocks `T1` and `T2`. Taking a look at `T1` it contains a receive statement which has a pointer to `Buffer1` where the incoming messages are stored. The receive statement also contains a local variable `i` into which a message from the buffer is read. `T1` also contains two conditional statements; one holding the condition expression `guard1` and pointing to `T1`, and one holding the condition expression `guard2` and pointing to `T2`. The block `T2` is similar to `T1`.

Translation to Erlang Source Code. The translation becomes more complex when allowing branches. Jumping to the first block were the guard evaluates to `true` could introduce a deadlock in the program if that block contains a receive statement from a buffer that will never have an element added. For instance, in the PCPN model shown in Fig. 6, it could be the case that both guard1 and guard2 evaluates to `true`. Assume that Buffer1 is empty, and that there will never be added a token to it. Assume also that Buffer2 already contains a token. If the program was to jump to the function corresponding to the transition T1 the program would stop on the blocking receive expression. This is not desirable

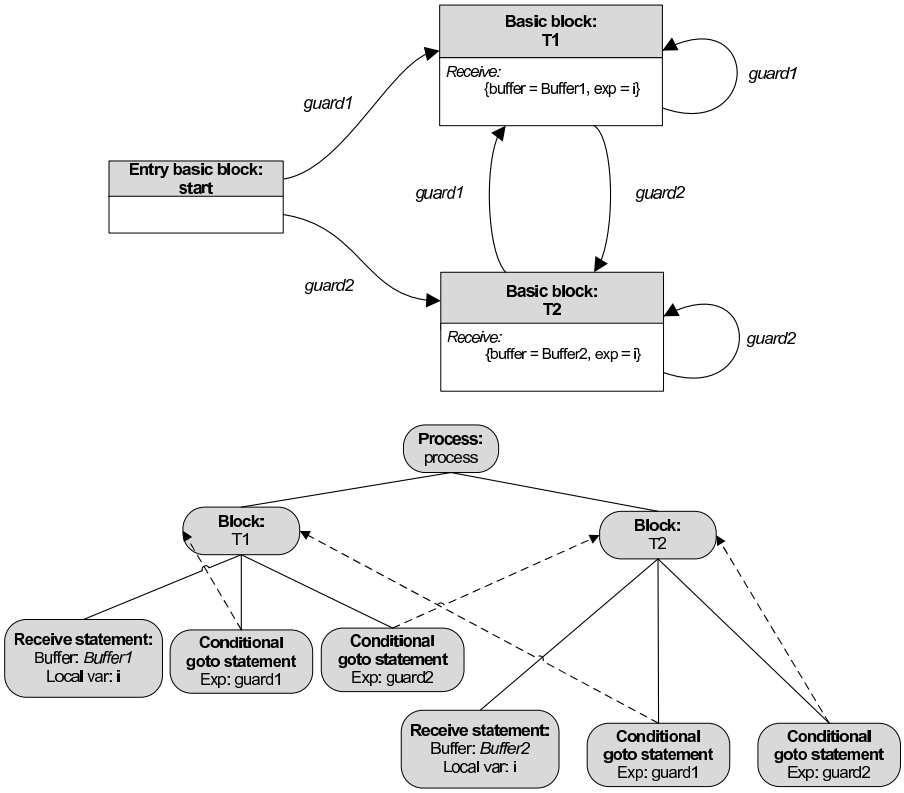


Fig. 7: The CFG (top) and AST (bottom) of the control flow branch.

since T2 is enabled in the PCPN model, thus calling the function corresponding to T2, would not make the program stop.

A solution to this problem is to only jump to a function with a receive expression if there is an element in the buffer. Since buffers are local to a process instance, the element will remain in the buffer until removed by the process instance. For this reason, we have introduced an explicit Erlang buffer module with operations for checking if an element is available. An example of this can be seen for our industrial example in Listing 2 in the next section.

4 Using the Method on a Real-life Example

We have used our prototype to generate code for the producer-consumer example, part of which is shown in Listing 1. We have validated this code by manual inspection and by adding logging code verifying that it is possible to reproduce runs of the generated code in the model.

We have also run our prototype on a model of the real-life dynamic on-demand MANET routing protocol (DYMO) [1]. The first versions of our model of this protocol were made before starting this work and the translation effort required to get running code is therefore realistic for other projects as well. Earlier versions of the model have been used to find and fix problems in the protocol specification [5]. The protocol is responsible for establishing routes in a *mobile ad-hoc network* (MANETs), i.e., a network with no preexisting infrastructure. The protocol establishes routes on-demand, i.e., when they are actually needed. The model models the two parts *route discovery* and *route maintenance* of DYMO. Route discovery establishes routes by multicasting *route request* messages. Each intermediate node records a route back to the originator and forwards the message. When the request arrives at the destination, a route is established and used to unicast a *route reply* back to the originator. Route maintenance actively monitors links and uses timeouts to discover loss of connection, and when this happens a *route error* is multicasted to all neighbours.

The CPN model of DYMO has been translated into PCPN from the earlier versions in approximately 20 man hours and consists of 8 modules, 49 places, and 18 transitions, and is thus fairly complex. We have shown part of the route discovery in Fig. 8. The module is responsible for initiating route discovery. We have used the same color conventions as in Fig. 1, i.e., process places are black, local places are green, shared places are red, and buffer places are blue. The module can either create a new route request or cancel the request if we have retried too many times.

Generating code from the DYMO model yields the modules shown in Table 1. We have listed lines of code (L.O.C.) for each module – in total we generated 563

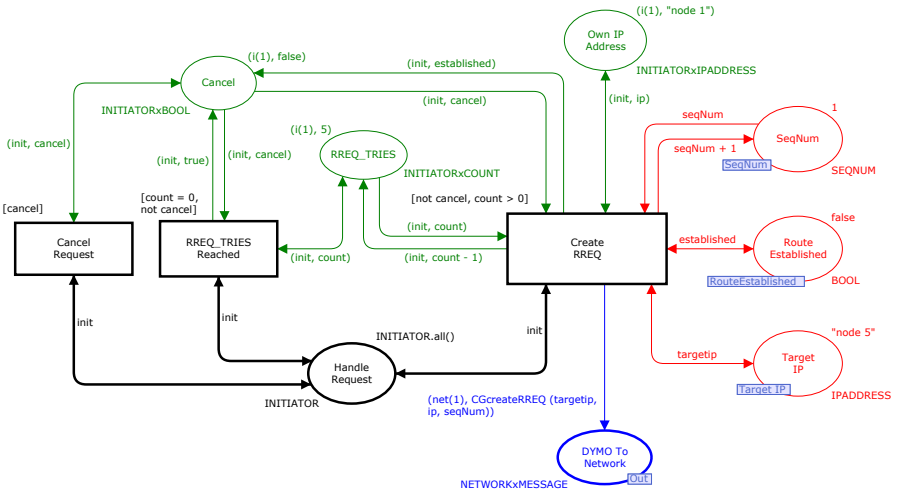


Fig. 8: The Initiator module of the PCPN DYMO model.

Table 1: Generated Modules from the DYMO Protocol.

Module name	L.O.C.	Functions to implement
system.erl	20	0
buffer.erl	36	0
shared.erl	16	0
initiator.erl	116	1
receiver.erl	116	7
processer.erl	111	4
establishchecker.erl	126	0
network.erl	22	0
Total	563	12

lines of code. Since we do not support automatic translation from SML to Erlang, we had to manually implement various Erlang expressions and functions on the basis of the corresponding SML code. These SML expressions are carried along as comments in the generated code. The comments are placed where the expression should be used, thus the structure of the program is preserved. Implementing the functions (12 in total) in Erlang is an easy task, because of the similarity between Erlang and SML. In total, we spent approximately 12 man hours modifying the generated code, including removal of unused extracted values. The time spent could be eliminated by implementing automatic translation from SML to our AST and adding rudimentary liveness calculation for variables.

In Listing 2, we see the generated code for the Create RREQ from the Initiator module. This code is pretty typical for the generated code as it starts by gathering parameters, then performs a calculation, distributes results, and finally passes on control. The code consists of picking the needed values of the environ-

Listing 2: Generated code for the Create RREQ transition of the Initiator module.

```
1  create_rreq(Env) ->
2      Count = Env#environment.rreq_tries,
3      Cancel = Env#environment.cancel,
4      Ip = Env#environment.own_ip_address,
5      is_route_established ! {get, self()}, receive Established -> Established end,
6      target_ip ! {get, self()}, receive Targetip -> Targetip end,
7      seqnum ! {get, self()}, receive Seqnum -> Seqnum end,
8      NewEnv = Env#environment {rreq_tries = Count - 1, cancel = Established},
9      is_route_established ! {set, Established},
10     target_ip ! {set, Targetip},
11     seqnum ! {set, Seqnum + 1},
12     Id1 = 1,
13     Receiver1 = list_to_atom("network_ID" ++ integer_to_list(Id1) ++ "_dymo_to_network"),
14     Receiver1 ! {send, %% CGcreateRREQ (targetip, ip, seqNum)
15                   undefined},
16     Guard_cancel_request_cancel = NewEnv#environment.cancel,
17     Guard_rreq_tries_reached_count = NewEnv#environment.rreq_tries,
18     Guard_rreq_tries_reached_cancel = NewEnv#environment.cancel,
19     Guard_create_rreq_count = NewEnv#environment.rreq_tries,
20     Guard_create_rreq_cancel = NewEnv#environment.cancel,
21     Guard_create_rreq_ip = NewEnv#environment.own_ip_address,
22     if
23       %% cancel
24       undefined -> cancel_request(NewEnv);
25        %% count = 0, not cancel
26       undefined -> rreq_tries_reached(NewEnv);
27        %% not cancel, count > 0
28       undefined -> create_rreq(NewEnv)
29    end.
```

Listing 3: The `createRREQ` function.

```
1 createRREQ(Target, N, Seqnum) ->
2    #message {src = N, dest = 'LL_MANET_ROUTERS', target_addr = Target, orig_addr = N, orig_seqnum = Seqnum,
3              hop_limit = 5, dist = 1, msg_type = 'RREQ'}.
```

ment (ll. 2–4) and reading variables from shared variables (ll. 5–7), gathering of parameters. Then the code constructs a new environment (l. 8), and sets all shared values (ll. 8–15), calculation and distribution of results. In lines 14–15 we see that the original declaration from the CPN model has been commented out and replaced by `undefined`. We need to manually translate this function, and remove the comment and undefined. We then extract values for use in evaluation of the guard (ll. 16–21), and dispatch based on the result of evaluating the guard (ll. 22–28). We also need to translate the expressions in the switch in lines 22–28, but due to similarity between SML and Erlang, this mostly consists of renaming the variables shown. To illustrate how easy the modifications are, we have shown the manually implemented function in Listing 3.

In order to execute more than one node running the generated DYMO implementation, we use the *distributed Erlang system* which is a mechanism in Erlang allowing a number of Erlang systems to communicate over a network. It consists of a number of independent Erlang runtime systems, each called an *Erlang node*. Each node executes the same generated DYMO code. The processes running the DYMO implementation on different Erlang nodes do not communicate directly with each other. Instead they communicate via a *network simulator* process running on a separate Erlang node. The stub code for the network simulator is generated directly from the `network` process partition of the DYMO PCPN model. The network simulator process implements a simple MANET with a static topology where both unicast and multicast is supported.

To monitor the behaviour of the program, each node prints its own routing table, which can then be manually inspected to verify that the expected routes were established. To make sure that every part of the generated code at some point has been executed, we have executed the generated DYMO code with several different MANET configurations designed to exercise all parts of the code. The generated code established the correct routes in all cases, which provides confidence in the generated code.

5 Conclusion and Future Work

In this paper, we have introduced a sub-class of coloured Petri nets, process-partitioned coloured Petri nets (PCPNs or PCP-nets), which separates control flow from data, and we have shown a structure-based approach to automatically generating (Erlang) code from models created using this sub-class. The approach first extracts a control flow graph from the model, and from the control flow graph infers the higher level control structures and constructs an abstract syntax tree for an abstract language. From the abstract syntax tree, we generate a

syntax tree specific to the target language, translating generic control structures into language specific control structures.

We have validated that the translation and net class works for real-life examples, and have shown that a real life model with 8 pages, 49 places and 18 transitions can be translated to the net class in acceptable time (20 man hours), which indicates that the net class is not too far from general CP-nets, at least for network protocols. The generated code can be edited relatively easily, and even though we do not translate more complex SML functions automatically, the adaptation and setup for testing of the code was done in only 12 man hours, which indicates that the generated code is indeed fairly natural. Using manual inspection and logging, we have validated that traces in the generated code can be reproduced in the model and that the calculated results are correct according to the model.

For future work, we would like to extend the PCPN class to allow using variables from buffer or shared places in guards, which would reduce the complexity of the PCPN DYMO model. This complicates the calculation of guards, as the value may change as other process instances modify/receive values, requiring introduction of a locking mechanism. It would also be interesting to look at dynamic instantiation of processes, which should not be too difficult since we already instantiate processes in our setup process. Furthermore, our current implementation does not allow hierarchical PCP-nets, so they have to be flattened, e.g., using CPN Tools prior to translation. Automatic flattening, or, even better, use of the module concept in CP-nets to further improve the translation would be preferable.

It would also be interesting to improve the control structure recognition, from only using goto and conditional jumps to also include inlining of unconditional unique jumps, recognition of loops, and binary if statements. It would also be interesting to add Java code emission to validate that our approach is indeed output language agnostic. It is also a very interesting option to translate to GCCs GENERIC [11] intermediate language instead, as this would give us direct emission of binary code and allow us to directly link with code written in the plethora of languages supported by GCC.

Tool support can also be improved to, e.g., check that a model is indeed a PCPN model (or to only allow construction of valid PCPN models), and to add a parser to automatically generate AST nodes for all SML functions to eliminate this manual step.

An obvious weak point of the current implementation is that the validation is done in an ad-hoc manner, which increases our confidence in the implementation, but does not deliver the promised certainty that the code is correct.

References

1. I.D. Chakeres and C.E. Perkins. Dynamic MANET On-demand (DYMO) Routing, version 14, June 2008. Internet-Draft. Work in Progress.
2. CPN Tools webpage. `www.cs.au.dk/CPNTools/`.

3. Erlang specification. `www.erlang.org/doc.html`.
4. K.E. Espensen and M.K. Kjendsen. Automatic Code Generation from Process-Partitioned Coloured Petri Net Models. Master's thesis, Dept. of Computer Science, Aarhus University, 2008.
5. K.L. Espensen, M.K. Kjeldsen, and L.M. Kristensen. Modelling and Initial Validation of the DYMO Routing Protocol for Mobile Ad-Hoc Networks. In *Proc. of ATPN'08*, volume 5062 of *LNCS*, pages 152–170. Springer-Verlag, 2008.
6. C. Girault and R. Valk. *Petri Nets for System Engineering: A Guide to Modeling, Verification, and Applications*. Springer-Verlag, 2003.
7. K. Jensen and L.M. Kristensen. *Coloured Petri Nets – Modelling and Validation of Concurrent Systems*. Springer, 2009.
8. L.M. Kristensen, P. Mechlenborg, L. Zhang, B. Mitchell, and G.E. Gallasch. Model-based Development of a Course of Action Scheduling Tool. *STTT*, 10(1):5–14, 2007.
9. L.M. Kristensen and A. Valmari. Finding Stubborn Sets of Coloured Petri Nets Without Unfolding. In *Proc. of ATPN'98*, volume 1420 of *LNCS*, pages 104–123. Springer-Verlag, 1998.
10. K.B. Lassen and S. Tjell. Translating Colored Control Flow Nets into Readable Java via Annotated Java Workflow Nets. In *Proc. of 8th CPN Workshop*, volume 584 of *DAIMI-PB*, pages 127–146, 2007.
11. J. Merril. GENERIC and GIMPLE: A New Tree Representation for Entire Functions. `gcc.gnu.org/pub/gcc/summit/2003/GENERIC%20and%20GIMPLE.pdf`.
12. K.H. Mortensen. Automatic Code Generation Method Based on Coloured Petri Net Models Applied on an Access Control System. In *Proc. of ATPN'00*, volume 1825 of *LNCS*, pages 367–386. Springer, 2000.
13. S. Philippi. Automatic code generation from high-level Petri-Nets for model driven systems engineering. *Journal of Systems and Software*, 79(10):1444–1455, 2006.
14. W.M.P. van der Aalst, J.B. Jørgensen, and K.B. Lassen. Let's Go All the Way: From Requirements Via Colored Workflow Nets to a BPEL Implementation of a New Bank System. In *Proc. of OTM Conferences (1)*, volume 3760 of *LNCS*, pages 22–39. Springer, 2005.

Towards Verification of the PANA Authentication and Authorisation Protocol using Coloured Petri Nets

Steven Gordon

Sirindhorn International Institute of Technology
Thammasat University, Thailand
steve@siit.tu.ac.th

Abstract

The Extensible Authentication Protocol (EAP) allows a server to request authentication information from a client. In order to transport EAP messages over an IP network, the Protocol for Carrying Authentication for Network Access (PANA) has been developed. This paper applies a protocol engineering methodology using Coloured Petri nets (CPNs) as a step towards formally verifying the design of PANA. State space analysis of a simple PANA configuration shows that the current specification has removed deadlocks discovered in previous PANA versions. Furthermore, state space and language analysis of PANA for different client retransmission limits leads to two important conjectures: the state space size (number of nodes, arcs) can be expressed as a polynomial in terms of the retransmission limit; and the protocol language is independent of the retransmission limit. The results suggest parametric verification is applicable to PANA. Finally, ideas for automatically validating the CPN model against the original specification are discussed.

1 Introduction

The Extensible Authentication Protocol (EAP) [1] is a framework for performing authentication in computer networks (see Figure 1). A typical usage scenario, as illustrated in Figure 2, involves a server (known as *authenticator* in EAP) initiating an authentication request to a *peer*. The peer responds to this, and any subsequent requests, until the authenticator determines the procedures to be a success (the peer is authenticated for network access) or failure (the peer is denied access to the network). In practice a third entity, the *authentication server*, may be utilised for storage of credential information. EAP is designed to support different authentication methods (e.g. MD5, TLS) and to operate over different (non-IP-based) network technologies. For example, a laptop can authenticate with a wireless LAN access point using IEEE 802.11i, or a home PC can authenticate with a dial-in server using EAP over the Point-to-Point Protocol (PPP).

In order to allow EAP to be carried over IP networks, PANA has been developed and released as IETF Request For Comments (RFC) 5191. The Protocol for Carrying Authentication for Network Access [8] is a lower layer for EAP, and PANA itself uses UDP as a lower layer. In addition to the protocol definition in [8], the PANA Working Group has developed a state-table model of PANA published as RFC 5609 [7]. Although the state-table model is for informative purposes, combined with the protocol definition, it provides a detailed explanation of the behaviour of PANA. However, as with many distributed protocols, it is important that the PANA specification is accurate and unambiguous. This is particularly important for an authentication protocol, where small errors or an ambiguous specification may lead to implementations with potentially damaging security flaws.

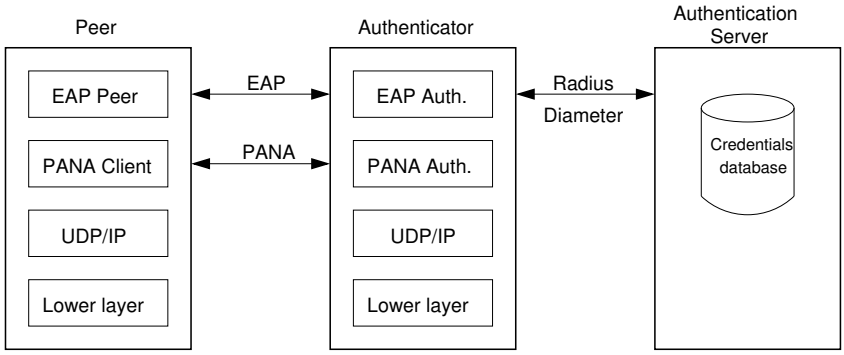


Figure 1: EAP framework

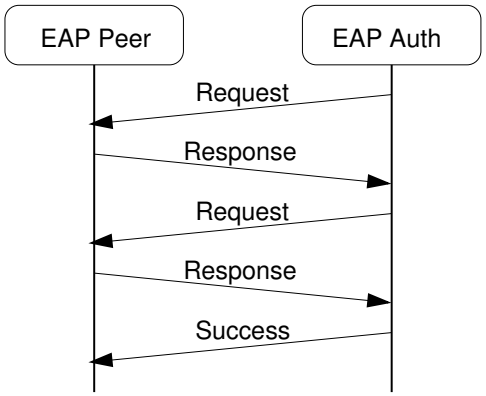


Figure 2: Typical EAP message sequence

Most research on PANA involved its application to wireless networks [18, 17, 5], especially performance analysis of PANA re-authentication during handovers [4, 9]. Little effort has been directed to the formal analysis of PANA, including security analysis. The overall aim of this research is to verify the design of PANA to ensure a complete and correct specification is available. To do so a protocol engineering methodology [2] is applied utilising Coloured Petri nets (CPNs) [14]. There are numerous examples of a protocol engineering methodology applied to other protocols (e.g. [19, 16, 11, 12]). The steps followed in this paper include:

1. Modelling of the PANA protocol specification using CPN Tools [15].

2. Simulation of the PANA CPN model to investigate specific scenarios. CPN Tools is used to step through sequences of events, and combined with BRITNeY [21] to automatically generate message sequence charts.

3. Functional property verification from state space analysis. CPN Tools is used to inspect terminal states and identify possible deadlocks, as well as bounds on communication channels.

4. Generation and inspection of the PANA protocol language (i.e. the possible ordering of interactions between PANA and the higher layer, EAP). Obtaining the protocol language is necessary in verifying that PANA is a faithful refinement of the service that EAP assumes is provided by the lower layer. However in this work, there is not yet a formal definition the PANA service offered to EAP. Hence only manual inspection of the protocol language is used at this stage.

Note that this paper does not attempt a formal security analysis (from a cryptographic viewpoint) of PANA. In fact, such analysis depends largely on EAP and other authentication methods, as PANA is only a protocol that carries EAP messages.

In previous work [13] a CPN model and initial analysis of PANA was presented. This was based on RFC5191 and draft version 6 of the PANA state tables. Using state space analysis of the simplest configurations of PANA (no retransmissions, no optimisation of the initiation procedure), a problem with aborting sessions was identified. Since then the PANA state tables have been updated (from version 6 through to 13, which is now published as RFC 5609). This paper uses an updated CPN model of the latest version of PANA. In addition, new configurations are analysed, in particular with retransmissions and initiation optimisation.

The remainder of this paper is organised as follows: Section 2 describes PANA and EAP in further detail. Section 3 provides an overview of the CPN model of PANA. Results from the formal analysis of the PANA Authentication and Authorisation Phase are presented in Section 4. Discussion of the approach, results and ideas for future work are given in Section 5.

2 EAP and PANA

2.1 EAP

EAP is a request/response protocol where only a single packet is in-flight at once, i.e. the authenticator cannot send a new request until the response from the previous request is received. The requests contain authentication challenges to the client. EAP assumes the lower layer (in this paper, PANA) will provide in-order delivery of packets, however it does not require the lower layer to be reliable, provide security or remove duplicates. A typical scenario, as illustrated in Figure 2, involves one or more EAP Request/Response exchanges (always initiated by the authenticator) followed by a final EAP Success or EAP Failure message, depending on the authentication information supplied by the client.

2.2 PANA

The role of PANA is to transport EAP messages between peer (referred to as PANA Client or PaC) and authenticator (PAA). PANA uses UDP at the transport layer, and hence the service provided to PANA may have packet losses, duplication and re-ordering. An exchange of messages in PANA is a session, which is divided into four phases:

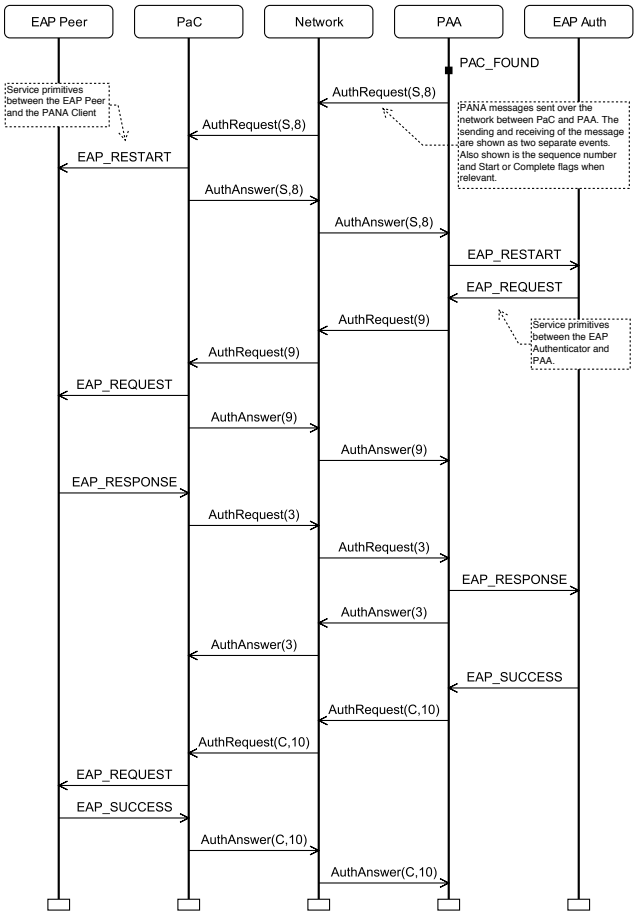
Authentication and Authorisation At the start of a PANA session this phase involves the exchange of EAP messages to perform authentication.

Access Once authentication is successful network access is provided. In this phase either PaC or PAA may test for the liveness of the session (which has a limited lifetime).

Re-authentication May be performed to maintain the session liveness.

Termination Either PaC or PAA may terminate a session. If a session isn't terminated gracefully, then a timeout on the PANA session will result in the termination.

PANA communications are implemented as a series of request and answer messages. To explain the Authentication and Authorisation phase consider the example scenario in Figure 3(a) (which is generated from our CPN model in Section 3). It shows the



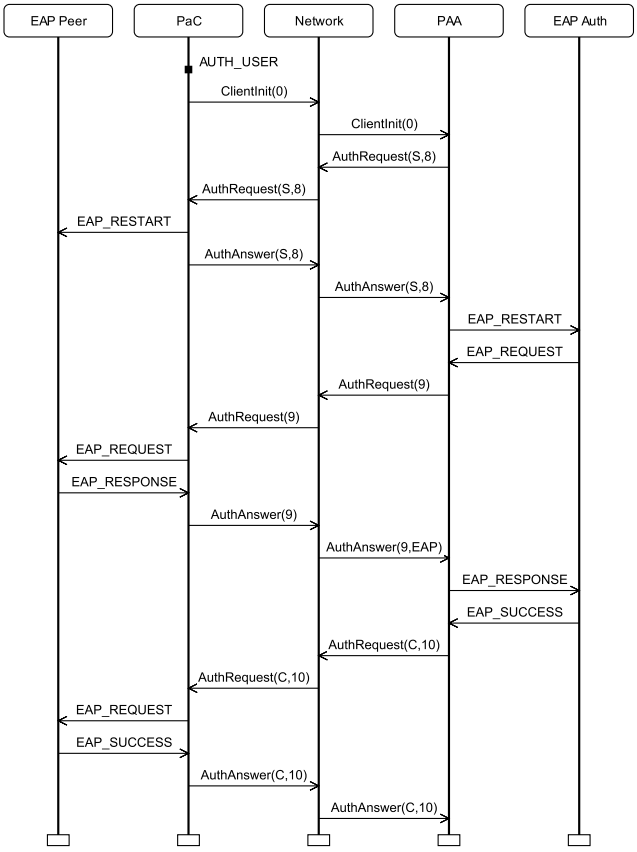


Figure 3: Message sequence chart of PANA Authentication phase: (a) no piggybacking; (b) with piggybacking

communication between EAP entity and corresponding PANA entity, as well as between PANA entities across the network.

The PANA session can be initialised by either the PAA or PaC (Figure 3(a) shows an example initiated by PAA, whereas Figure 3(b) illustrates PaC initiation, as well as piggybacking). The methods for each entity learning about the presence of the other is out side of the scope of PANA. For a PAA-initiated-session, after it discovers the presence of a PaC, it sends an AuthRequest message to start the session (the 'S' flag indicates this message is to start the session). This initial AuthRequest is used to force a restart of the EAP session at the Peer. The PaC responds with an AuthAnswer, which results in the EAP session at the Authenticator restarting.

The EAP Authenticator then initiates the authentication with an EAP Request. This triggers the PAA to send an AuthRequest carrying the EAP Request method (e.g. the authentication challenge). Upon receipt of the AuthRequest, the PaC passes the EAP Request method to the EAP Peer and replies with an AuthAnswer.

The PaC sends the response to the challenge in an AuthRequest (which is also acknowledged by the PAA with a AuthAnswer). This sequence of EAP requests and responses (and AuthRequest and AuthAnswer messages) may repeat until the authentication is complete. Finally the EAP Authenticator will send a Success or Failure method indicating the result of authentication. The EAP Success is shown in Figure 3(a), which is carried in an AuthRequest with the Complete flag set. Once the AuthAnswer is received by the PAA, both PAA and PaC have the PANA session established and the Access phase is entered. Note that the AuthAnswer messages can be considered as acknowledgements of the AuthRequest messages. They do not necessarily carry the answer to the authentication challenge (i.e. the EAP response). Other relevant details of PANA include:

- 32-bit sequence numbers are used to maintain ordering and perform error detection. The sequence numbers at PAA and PaC are independent. An outgoing request message contains a sequence number, and the corresponding answer message must have the same sequence number.
- Request messages are retransmitted if an answer is not received within a specified time. The session is terminated if too many retransmissions occur.
- PANA messages contain 16 bytes of fixed size header (e.g. flags, message type, sequence number, session identifier) as well as a variable number of Attribute-Value Pairs (AVPs). AVPs include: the actual EAP message; authentication data; session lifetime; and other security related information.
- Optional piggybacking of messages allows either PaC or PAA to send a single PANA message that represents both an answer and a request. For example in Figure 3(a) without piggybacking, PaC sends an AuthAnswer(8) followed by AuthRequest(3). With piggybacking turned on (Figure 3(b)), the PaC sends a single message, AuthAnswer(9) which also includes the EAP response.

2.3 EAP/PANA Interface

In order to verify if the PANA protocol correctly interacts with EAP, it is necessary to understand the interface between the two layers. The EAP state-machines [20] specify the variables used for communication between EAP and a lower layer. This information is summarised in Figure 4. As an example, when the EAP Authenticator sends an EAP Request, the eapReq flag will be set to true and the Request method will be included in eapReqData.

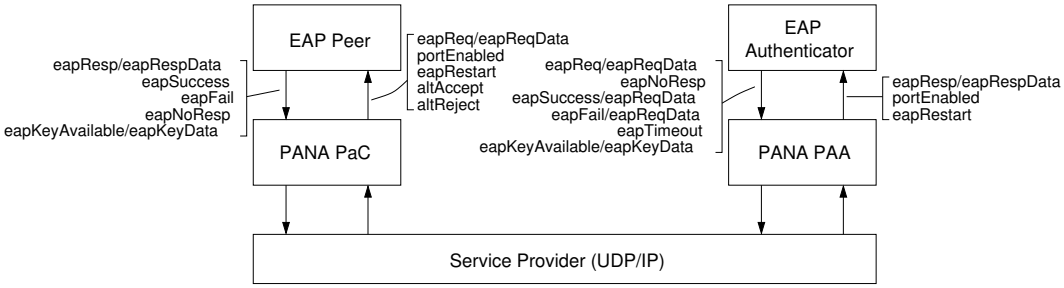


Figure 4: EAP/PANA interface based on [20]

In addition to the EAP-defined interface, the PANA state-tables [7] describes its own set of variables and procedures used for communication between PANA and EAP. As an alternative to the two separate set of variables defined by EAP and PANA, *service primitives* can be used to describe the interface between the two layers. Table 1 is an attempt to define the service primitives that correspond to information found in the EAP standard [20] and PANA standard [8, 7]. The table lists the EAP-defined variables, the PANA-defined variables, as well as the newly defined service primitives. However from the available specifications it is difficult to define all valid sequences of primitives, and hence orderings are not yet defined. In Section 4 the service primitives are used in determining the PANA protocol language, i.e. the possible sequences of service primitives.

Table 1: EAP/PANA interface variables and service primitives

No.	Entity	EAP	PANA	Primitive
1	Peer/PaC	-	AUTH_USER	CAuthUser
2	Peer/PaC	eapRestart	EAP_RESTART	CRestart
3	Peer/PaC	eapReq	EAP_REQUEST	CRequest
4	Peer/PaC	eapResp	EAP_RESPONSE	CResponse
5	Peer/PaC	eapSuccess	EAP_SUCCESS	CSuccess
6	Peer/PaC	eapFail	EAP_FAILURE	CFailure
7	Peer/PaC	-	-	CTimeout
8	Peer/PaC	-	ABORT	CAbort
9	Auth/PAA	-	PAC_FOUND	APacFound
10	Auth/PAA	eapRestart	EAP_RESTART	ARestart
11	Auth/PAA	eapReq	EAP_REQUEST	ARequest
12	Auth/PAA	-	-	AResponse
13	Auth/PAA	eapSuccess	EAP_SUCCESS	ASuccess
14	Auth/PAA	eapFail	EAP_FAILURE	AFailure
15	Auth/PAA	-	EAP_TIMEOUT	ATimeout
16	Auth/PAA	-	ABORT	AAbort
17	Peer/PaC	-	DISCARD	CDiscard
18	Auth/PAA	-	DISCARD	ADiscard

3 Coloured Petri Net Model of PANA

3.1 Model Hierarchical Structure

A CPN model of PANA has been created based on the state tables in [7]. The model consists of 23 pages, 63 transitions and 7 places (however not all transitions are relevant for the Authentication and Authorisation phase of PANA considered in this paper).

The model focuses on the components of the protocol important for functional verification, i.e. the ordering of exchange of messages. Where possible, abstraction is used so that details of message content and format can be omitted. This makes analysis easier, but at the expense of a complete protocol specification. The structure of the model fol-

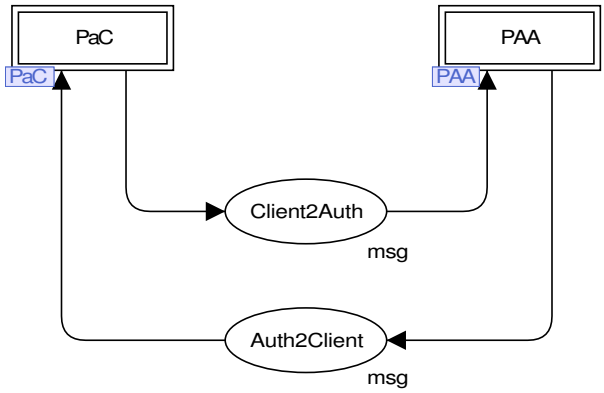


Figure 5: Top-level page of PANA CPN Model

lows a state-based approach, where transitions are used to model actions (each row) in the PANA state-tables. Further discussion of this approach is given in Section 5.

The PANA CPN is hierarchical, with the PaC and PAA modelled on separate pages, and then each state of the PaC/PAA modelled on separate pages. This is achieved using substitution transitions and port/socket places. At the highest level (Figure 5) there are two transitions (PaC and PAA) and two places modelling the communication channel between PaC and PAA (and vice versa). The channel is assumed to be reliable (no message loss), but allows re-ordering and delay of messages. As UDP is used as the lower layer by PANA, the assumption of no message loss is not always valid. However assuming no message loss is a useful starting point for the analysis, since modelling loss may hide deadlocks in the protocol. Studying the impact of message loss is part of future work.

Both the PaC and PAA transitions at the top-level contain detailed models on their respective sub-pages. These sub-pages (Figures 6 and 7) contain transitions that model the events at each state and a place to model the current state. For example, the place Client stores the current state of PaC, C_INITIAL, and state variables such as eap_piggyback=false. Places LastClientMsg and LastAuthMsg are used to store the previous message sent in case a retransmission is necessary.

3.2 Modelling State Tables

Each transition on the PAA/PaC sub-pages is further decomposed to individual pages that model the events, conditions, actions and next states as presented in the state-tables. To explain, the case of the PAA in the INITIAL state will be used as an example. The PANA state-table from [7] for the PAA INITIAL state is given in Figure 8.

For a given state, the state-tables in [7] specify:

- An exit condition, i.e. the conditions that must occur. For example, in Figure 8 there is an exit condition called EAP REQUEST, which indicates a request is received from the higher layer EAP entity.

- Exit actions, i.e. the actions that will be executed upon the conditions being met. For the EAP REQUEST condition, the actions are to transmit a PANA AuthRequest message (`Tx:PAR[S]`) and then start the retransmission timer (`RtxTimerStart()`. Note that the contents of the AuthRequest message depends on the type of EAP method received from the higher layer.

- The exit state, i.e. the next state of the entity. After the EAP REQUEST and corresponding actions, the PAA will re-enter (i.e. remain in) the INITIAL state.

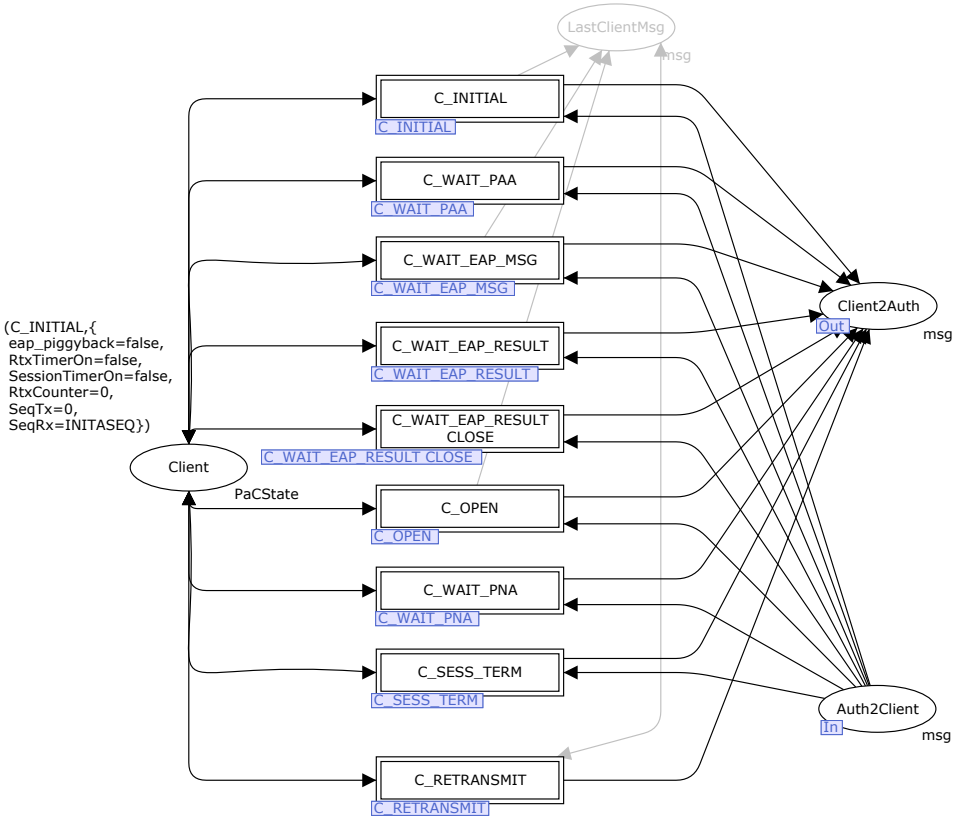


Figure 6: CPN Model of PaC

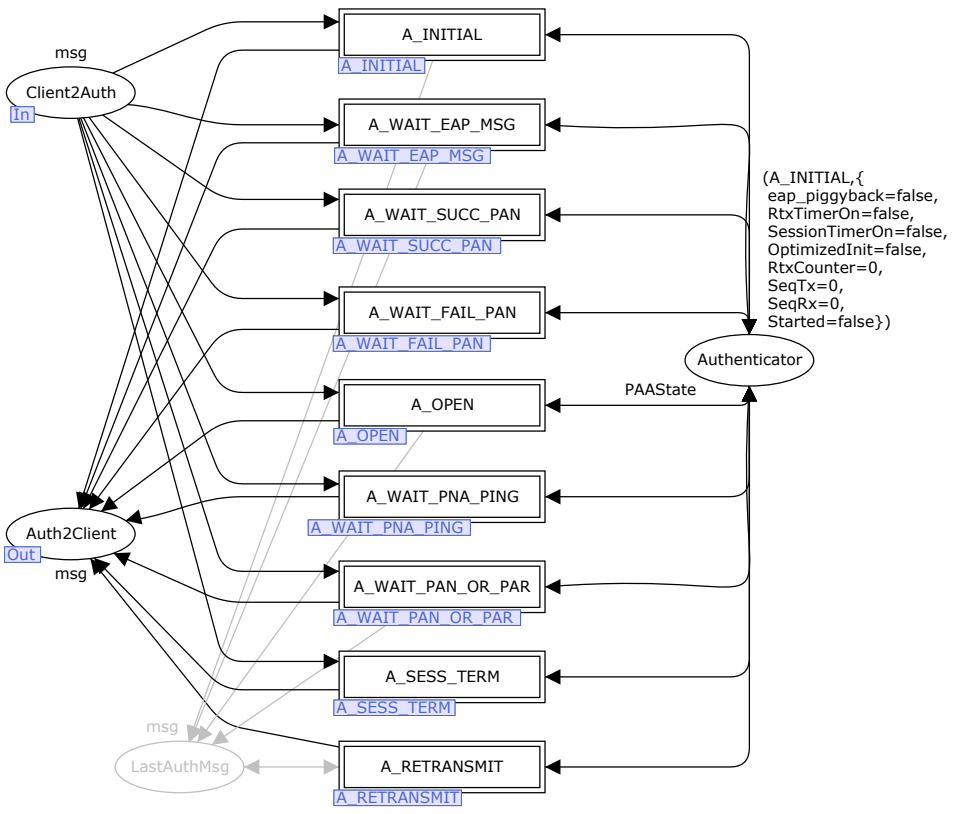


Figure 7: CPN Model of PAA

Ideally, each entry in the state table could be modelled by a single transition in the CPN. However for some entries multiple transitions are used. This is because the exit actions in the state table sometimes contain *conditions*. Consider the action for the exit condition `(Rx:PCI[] || PAC_FOUND)` in Figure 8. If `OPTIMIZED_INIT` is set then one sequence of actions are taken, and if not set another sequence of actions are taken. Hence this entry in the state table is modelled as two transitions. Furthermore, there are in fact two distinct exit conditions: `Rx:PCI[]` and `PAC_FOUND`. These are therefore modelled as separate transitions (this is necessary as the two conditions correspond to different interactions with the higher layer). As a result, the one state table entry in this case is modelled with four transitions. In Figure 9 the four transitions are: PAC_FOUND Optimum; PAC_FOUND NoOptimum; RxPCI Optimum; and RxPCI NoOptimum.

```
------------------------------
State: INITIAL (Initial State)
------------------------------

Initialization Action:

  OPTIMIZED_INIT=Set|Unset;
  NONCE_SENT=Unset;
  RTX_COUNTER=0;
  RtxTimerStop();

Exit Condition           Exit Action                Exit State
-----------------------+--------------------------+------------
 - - - - - - - - (PCI and PAA initiated PANA) - - - - - - - - -

(Rx:PCI[] ||             if (OPTIMIZED_INIT ==      INITIAL
 PAC_FOUND)                  Set) {
                           EAP_Restart();
                           SessionTimerReStart
                            (FAILED_SESS_TIMEOUT);
                         }
                         else {
                           if (generate_pana_sa())
                                Tx:PAR[S]("PRF-Algorithm",
                                   "Integrity-Algorithm");
                           else
                                Tx:PAR[S]();
                         }

EAP_REQUEST              if (generate_pana_sa())    INITIAL
                             Tx:PAR[S]("EAP-Payload",
                                "PRF-Algorithm",
                                "Integrity-Algorithm");
                         else
                             Tx:PAR[S]("EAP-Payload");
                         RtxTimerStart();
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

- - - - - - - - - - - - - - (PAN Handling)  - - - - - - - - - -
Rx:PAN[S] &&             if (PAN.exist_avp          WAIT_EAP_MSG
((OPTIMIZED_INIT ==         ("EAP-Payload"))
  Unset) ||                TxEAP();
PAN.exist_avp            else {
  ("EAP-Payload"))         EAP_Restart();
                           SessionTimerReStart
                            (FAILED_SESS_TIMEOUT);
                         }

Rx:PAN[S] &&             None();                    WAIT_PAN_OR_PAR
(OPTIMIZED_INIT ==
  Set) &&
! PAN.exist_avp
 ("EAP-Payload")
 - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

Figure 8: State Table for PAA in INITIAL state from [7]

Each transition has or may have:

- An input arc from a place containing the current state, and related state information, e.g. sequence numbers, flags. Consider the bottom transition in Figure 9 as an example. The transition is only enabled when PAA is in the A_INITIAL state.
- An input arc from the communication places (Client2Auth or Auth2Client, depending on the entity) if the event involves receiving a message. (The example transition is only enabled when a AuthAnswer has been sent by the PaC).
- A guard for the conditions related to the event. (AuthAnswer message must have the Start flag set, not contain EAPPayload, and PAA must be using OptimizedInit)
- An output arc to the communication places if the action involves sending a message. (No message is sent for the example transition).
- An output arc to the place containing state information, where the next state is stored. (The new state is A_WAIT_PAN_OR_PAA for the example transition—there are no changes to the state variables).

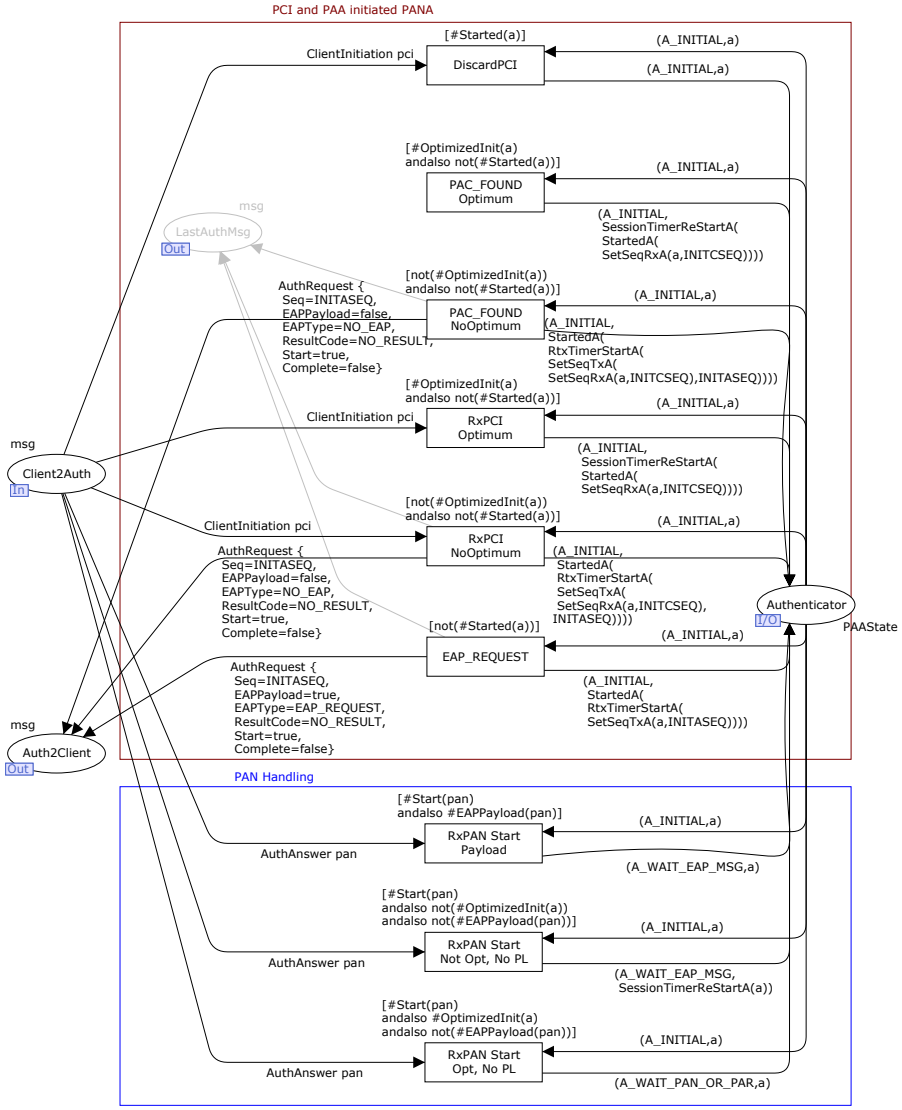


Figure 9: CPN Model of A_INITIAL state

3.3 Message Sequence Charts

To illustrate the message flow between PANA and EAP entities, BRITNeY [21] and the message sequence chart library are used in conjunction with CPN Tools. Each transition modelling state table entries has a code segment indicating the transfer of messages or special events (if any). For clarity, the code segments are omitted from the figures in this paper. The code segment for the bottom transition in Figure 9 is:

```
input (pan); output (); action (if mscOn then (
msc.addEvent("Network","PAA",concat ["AuthAnswer[S,"
  Int.toString (#Seq(pan)),")"])) else ());
```

The firing of the transition will draw an arrow indicating an AuthAnswer message being sent from the *Network* to the PAA. Message sequence charts, such as that illustrated in Figure 3, are useful for testing during model development, and exploring the protocol behaviour under specific conditions.

4 Analysis Results

4.1 Approach and Assumptions

The state space is generated with CPN Tools and examined to determine the presence of unexpected terminal states (deadlocks) in PANA, as well as the integer bounds on the communication places.

To investigate the protocol language (i.e sequence of interactions between PANA and higher layer), CPN Tools was used to translate the state space into a FSA, where states were mapped to their node number and each binding element was mapped to an integer depending on the service primitive it represented. The integer mappings are given in Table 1. All other binding elements map to 0. In addition, terminal states in the state space mapped to halt states. Then using AT&T's FSM Library[1] and GraphViz[2], the minimised deterministic FSA is created, representing the PANA protocol language. The protocol language is studied in order to identify unexpected sequences of events in PANA.

In this paper four protocol parameters are of interest:

1. Piggybacking (PB_{PaC} and PB_{PAA}): this can be either on (true) or off (false). If on, the PaC/PAA may combine two messages into one, thereby reducing the number of messages sent over the network.

2. Optimised Initiation ($OptInit$): this can be either on or off. If on, the PAA can send an EAP Request in the initial AuthRequest message.

3. PaC Maximum Retransmission Count (MRC_{PaC}): an integer indicating the maximum number of retransmissions of request messages by PaC.

4. PAA Maximum Retransmission Count (MRC_{PAA}): an integer indicating the maximum number of retransmissions of request messages by PAA.

The analysis assumes only a single EAP Request is sent by the Authenticator. In addition, 4-bit sequence numbers are used (instead of 32-bit), and the initial sequence numbers are randomly set at 3 and 8 for PaC and PAA, respectively.

In Section 4.2 analysis results for a simple configuration with no retransmissions are presented. Then the effect of retransmissions is considered in Section 4.3.

[1] http://www.research.att.com/~fsmtools/fsm/

[2] http://www.graphviz.org/

Table 2: State Space Analysis of PANA CPN with No Retransmissions

Piggyback	OptInit	States	Arcs	Terminal States	Client2Auth	Auth2Client
Off	Off	15531	34047	6866	5	3
On	Off	3436	7212	1265	3	2
Off	On	12079	26360	5292	5	3
On	On	2085	4233	775	3	2

4.2 Analysis of Simple Configuration: No Retransmissions

Statistics from the PANA state space are shown in Table 2. In all cases, $MRC_{PaC} = MRC_{PAA} = 0$. Also, for simplicity piggybacking is either on for both PaC and PAA *or* off for both PaC and PAA. The integer bounds of the communication places, Client2Auth and Auth2Client, are reported.

The state space is significantly larger when *Piggyback* is off. This is because without piggybacking, both the PaC and PAA must send a separate AuthRequest message to carry EAP responses. As illustrated in Figure 3, after receiving an EAP_REQUEST, the PaC sends an AuthAnswer(9) acknowledging the receipt of the EAP_REQUEST, and then sends a AuthRequest(3) containing the EAP_RESPONSE. With piggybacking, the EAP_RESPONSE can be sent in the AuthAnswer(9), omitting the need for the AuthRequest(3). Figure 10 shows the partial state space illustrating the sequences from Figure 3.

The integer bound of the communication places Client2Auth and Auth2Client indicates the number of messages an entity can send before it has to wait for a response from the peer entity. Normally an entity sends a AuthRequest and then waits for an AuthAnswer before sending another message. However, with piggybacking off for example, the PAA can send an AuthRequest message containing an EAP Request, and then if the EAP Authenticator returns a result to PAA, the PAA can send anotherAuthRequest message containing the EAP result. After this the PAA must wait for an answer. Hence the upper bound on the messages in Auth2Client is 2. Similar scenarios occur for the other configurations.

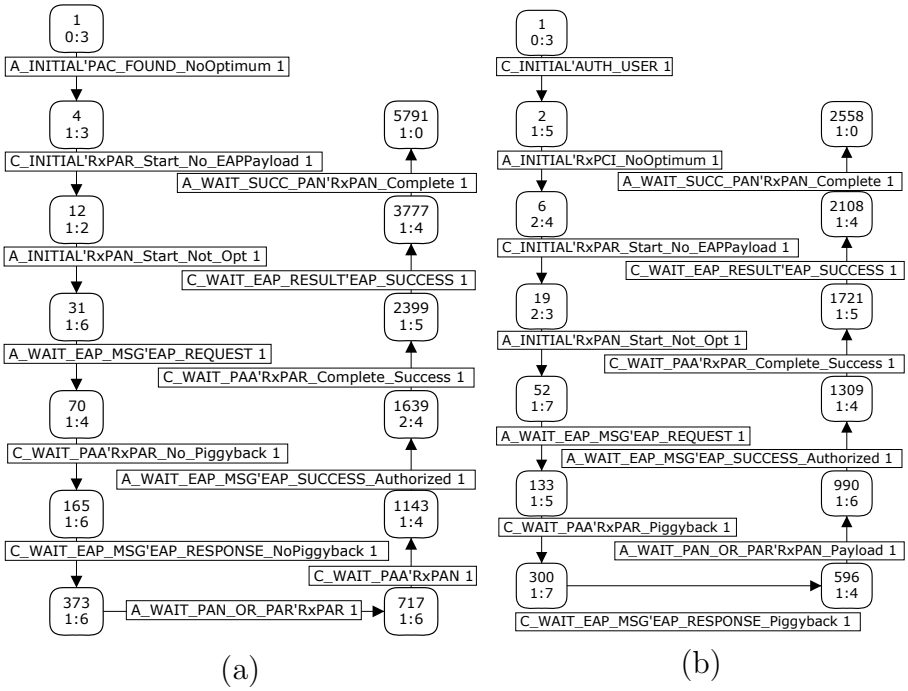


Figure 10: Selected state space showing sequence corresponding to those in Figure 3

4.2.1 Terminal States

Closer inspection of the terminal states is necessary to determine if any are unexpected. However first the expected terminal states must be defined. From previous analysis of PANA [13], the expected terminal states are classified by the state that the PaC and PAA finish in. Recall that only the Authentication and Authorisation phase of PANA is analysed: if this phase is successful the PANA session will be OPEN, and both PaC and PAA enter the Access phase. If unsuccessful, both PaC and PAA should enter the CLOSED state (PANA session is closed). Therefore expected terminal states are defined as those where the PaC/PAA entity is in a valid state. The four expected groups of terminal states are:

1. C_OPEN and A_OPEN: both PaC and PAA have opened a PANA session, i.e. authentication was successful.

2. C_CLOSED and A_CLOSED: both PaC and PAA have closed a PANA session, e.g. after a failed authentication attempt or abort due to too many retransmissions.

3. C_OPEN and A_CLOSED: PaC successfully opens a session, however the PAA aborts before opening the session thereby leaving it in the CLOSED state. This is considered valid because the PaC will enter the Access phase and eventually the PANA session will timeout (and close) after receiving no responses from the PAA. Rather than explicitly modelling the timeout event which is part of the Access phase, these terminal markings are considered valid.

4. C_CLOSED and A_OPEN: PAA successfully opens a session, however the PaC aborts. Following the same reasoning as above, this is a valid terminal state.

In [13] a fifth, unexpected group of terminal states was discovered. After the PaC responded to the initial AuthRequest message (with Start bit set), it entered the C_WAIT_PAA state. If the PAA aborted before receiving the AuthAnswer from PaC, then the PAA entered A_CLOSED. This was an invalid terminal state because as the PANA session had not yet started by the PaC, the session timer would never expire, leaving PaC in C_WAIT_PAA. This problem arose because, with no explicit abort messages, an entity only knows the peer has aborted if a time out occurs. In specific cases, such as described above, a timer is not started, and hence no timeout will occur.

In the updates of the PANA state table from version 6 to version 13 this behaviour has been fixed. That is, the session timer is started after PaC sends the initial AuthRequest message (rather than after entering the C_OPEN state). Therefore if PaC is waiting in the C_WAIT_PAA state, while the PAA has aborted, eventually PaC session will time-out. Similar behaviour can lead to a valid terminal state with PaC in C_WAIT_PAA and PAA in A_OPEN.

Finally, there is another special terminal state when optimised initiation is used (this case was not analysed in [13]). Note that both PaC and PAA may initiate the Authentication and Authorisation phase. If both entities initiate before receiving a message from the peer, then the PAA takes precedence. That is, the PAA will discard any ClientInitiation messages received from PaC. Similar to the above issues, if the PaC then aborts, the PAA may remain in the A_INITIAL start. However, as the session timer has been started, this is considered a valid terminal state (as eventually, the PAA will timeout and close the session).

Therefore, we define three more valid groups of terminal states: (5) C_WAIT_PAA and A_OPEN; (6) C_WAIT_PAA and A_CLOSED; and (7) C_CLOSED and A_INITIAL.

Using CPN Tools queries, the terminal states of each state space were inspected to determine if they matched any of the above 7 valid groups. Table 3 shows the count of terminal states for each group. No unexpected terminal states were discovered. Further discussion on the terminal states is given in Section 5.2.

Table 3: Terminal States with No Retransmissions

Piggyback	OptInit	1	2	3	4	5	6	7	Unexpected	Total
Off	Off	61	6090	164	456	11	84	0	0	6866
On	Off	41	871	98	61	8	186	0	0	1265
Off	On	59	4660	164	333	7	68	1	0	5292
On	On	23	534	59	38	4	116	1	0	775

4.2.2 Language Analysis

There is currently no formally defined service language for PANA. Table 1 lists the possible service primitives to be exchanged between PANA and the higher layer, but does not define any ordering. Hence language analysis of PANA is not used for verification (i.e. comparing a protocol language to a service language), but instead to gain confidence in the correct operation of the protocol and to investigate a possible service language. However, as will be shown shortly, the size of the PANA protocol language is too large for manual inspection. Therefore an abstraction is applied where the protocol languages for PaC and PAA are produced separately. That is, the *PaC Only* protocol language shows the sequence of primitives exchanged between the PaC and the higher layer. This is useful in validating that at least the PaC and PAA are operating in a normal manner.

Table 4 gives the number of states/arcs/final states in the minimised deterministic FSA, as well as the number of sequences in the language. Results are shown for the complete PANA protocol language, PaC only and PAA only. Figures 11 and 12 show the PaC only and PAA only protocol languages when piggybacking and optimised initiation are both on. Visual inspection of the PaC and PAA languages reveal no obvious unexpected sequences: the ordering of Requests then Responses is as expected; there are no Requests or Responses after a Success or Failure; and the PAA only sends a single Request. The sequence of primitives as seen by the PaC shown in Figure 3 are captured in the sequence from states 0–2–5–8–12–19 in Figure 11. Unfortunately, the full PANA protocol language is too large for inspection. Analysis of the full PANA language, and eventually determining a PANA service language, are left for future work.

Table 4: Protocol Language of PANA CPN with No Retransmissions. (S = States; A = Arcs; FS = Final States; Seq = Sequences)

		PANA				PaC Only				PAA Only			
PB	OptInit	S	A	FS	Seq	S	A	FS	Seq	S	A	FS	Seq
Off	Off	95	369	7	17862	14	36	6	71	7	23	2	70
Off	On	100	383	6	14742	20	56	8	60	7	23	2	56
On	Off	88	350	6	13604	14	35	5	65	9	31	3	64
On	On	93	340	6	11468	20	47	6	48	9	28	3	54

4.3 The Effect of Retransmissions

The simple configuration assumed no retransmissions from either PaC or PAA. However in PANA the PaC (or PAA) may retransmit an AuthRequest (or ClientInitiation) message up to MRC_{PaC} (or MRC_{PAA}) times if a corresponding AuthAnswer has not been received.

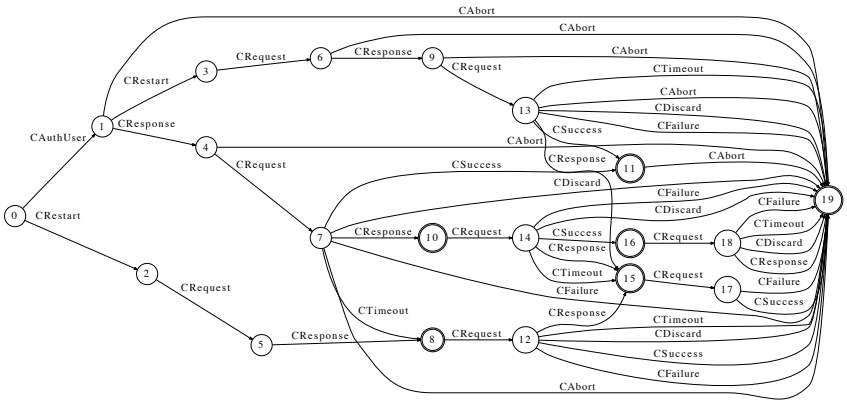


Figure 11: PaC language (piggybacking on, optimised initiation on)

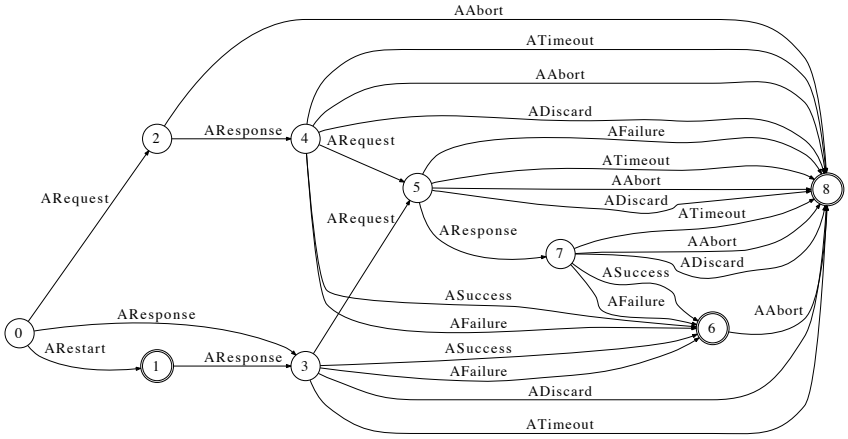


Figure 12: PAA language (piggybacking on, optimised initiation on)

Table 5 lists state space and language statistics for the PANA CPN for increasing values of MRC_{PaC}. In all cases, $MRC_{PAA} = 0$ [3].

First consider the results when piggybacking is on. For both cases of Optimised Initiation on and off, the increase in MRC_{PaC} leads to a quadratic increase in the size of the state space, as well as number of terminal markings. For values 0 to 5 of MRC_{PaC}, the number of states (S), arcs (A) and terminal markings (T) can be expressed as Equations (1) to (3) for the case of Optimised Initiation off:

$$S_{MRC_{PaC}} = 441(MRC_{PaC})^2 + 2958(MRC_{PaC}) + 3436 \quad (1)$$

$$A_{MRC_{PaC}} = 1190.5(MRC_{PaC})^2 + 7029.5(MRC_{PaC}) + 7212 \quad (2)$$

$$T_{MRC_{PaC}} = 71.5(MRC_{PaC})^2 + 813.5(MRC_{PaC}) + 1265 \quad (3)$$

and Equations (4) to (6) for the case of Optimised Initiation on:

$$S_{MRC_{PaC}} = 270.5(MRC_{PaC})^2 + 3721.5(MRC_{PaC}) + 2085 \quad (4)$$

$$A_{MRC_{PaC}} = 699(MRC_{PaC})^2 + 4150(MRC_{PaC}) + 4233 \quad (5)$$

$$T_{MRC_{PaC}} = 45.5(MRC_{PaC})^2 + 502.5(MRC_{PaC}) + 775 \quad (6)$$

[3]Initial results reveal the increase in state space size as MRC_{PAA} is increased is much larger than in the case of MRC_{PaC}. More detailed analysis of MRC_{PAA} is left for future work

Table 5: State Space Analysis of PANA CPN with PaC Retransmissions (Term = Terminal States; C2A = Client2Auth; A2C = Auth2Client; Seq = Sequences)

Parameters				State Space				Bounds		Language
PB	$OptInit$	MRC_{PaC}	MRC_{PAA}	States	Arcs	Term.	Time	C2A	A2C	Seq.
Off	Off	0	0	15531	34047	6866	91	5	3	17862
Off	Off	1	0	47046	112620	18943	1549	6	3	17862
Off	Off	2	0	101624	257908	38064	5129	7	3	17862
Off	Off	3	0	186485	494481	65729	15118	8	3	17862
Off	On	0	0	12079	26360	5292	52	5	3	13604
Off	On	1	0	38691	93267	15174	665	6	4	14675
Off	On	2	0	85589	219656	30983	2248	7	4	14675
Off	On	3	0	159568	428774	54100	11413	8	4	14675
Off	On	4	0	268212	747212	85947	33286	9	4	14675
On	Off	0	0	3436	7212	1265	5	3	2	14742
On	Off	1	0	6835	15432	2150	14	4	2	14742
On	Off	2	0	11116	26033	3178	24	5	2	14742
On	Off	3	0	16279	39015	4349	50	6	2	14742
On	Off	4	0	22324	54378	5663	127	7	2	14742
On	Off	5	0	29251	72122	7120	146	8	2	14742
On	Off	10	0	77116	196557	16550	997	13	2	14742
On	On	0	0	2085	4233	775	2	3	2	11468
On	On	1	0	4163	9082	1323	6	4	2	11468
On	On	2	0	6782	15329	1962	14	5	2	11468
On	On	3	0	9942	22974	2692	28	6	2	11468
On	On	4	0	13643	32017	3513	51	7	2	11468
On	On	5	0	17885	42458	4425	83	8	2	11468
On	On	10	0	47210	115633	10350	342	13	2	11468

The equations hold for $MRC_{PaC} = 10$, giving increased confidence that they are true for any value of MRC_{PaC}. Similar relationships between retransmission limits and state space size have been observed with other protocols [11, 12].

A much larger growth in the state space size is seen when considering no piggybacking (see Figure 13). Further state space results are necessary to determine the exact relationship between state space size and MRC_{PaC} when piggybacking is off.

When piggybacking is on, the PaC can send EAP responses in AuthAnswer messages. That is, the PaC does not send any AuthRequest messages (instead, the PaC sends AuthAnswer messages in response to AuthRequest messages sent by the PAA). Therefore the only message that can be retransmitted is the ClientInitiation which is used by the PaC to start the session.

When piggybacking is off, the PaC can retransmit a ClientInitiation message as well as an AuthRequest message that contains the EAP response (since the EAP response cannot be piggybacked in an AuthAnswer). The ability to retransmit the AuthRequest results in significant increase in the number of possible states, as illustrated in the partial state space in Figure 14.

The integer bound of the communication place Client2Auth increases linearly as the retransmission limit MRC_{PaC} increases. This is expected as the retransmission mechanism simply means an additional MRC_{PaC} messages can be sent, and stored in Client2Auth before the PaC must wait for the PAA to receive a message and respond.

Closer inspection of the PANA protocol language reveals the language is independent of MRC_{PaC} (with one exception, explained shortly). In other words, retransmissions by the PaC do not result in additional interactions between PAA and EAP Authenticator, nor between PaC and EAP Peer. The reason is that retransmissions are only used in two possible instances by the PaC: ClientInitiation in the C_INITIAL state and AuthRequest

carrying EAP response in the C_WAIT_EAP_MSG state. In both cases, when the PAA receives one of these messages (either original or retransmitted) it will process that message and ignore any subsequent duplicates. An exception is the protocol language with piggybacking off, and optimised initiation on. With no retransmissions there are less sequences than with retransmissions. The reason for this is retransmitted messages in this case can cause a different ordering of interactions between PAA and EAP Authenticator.

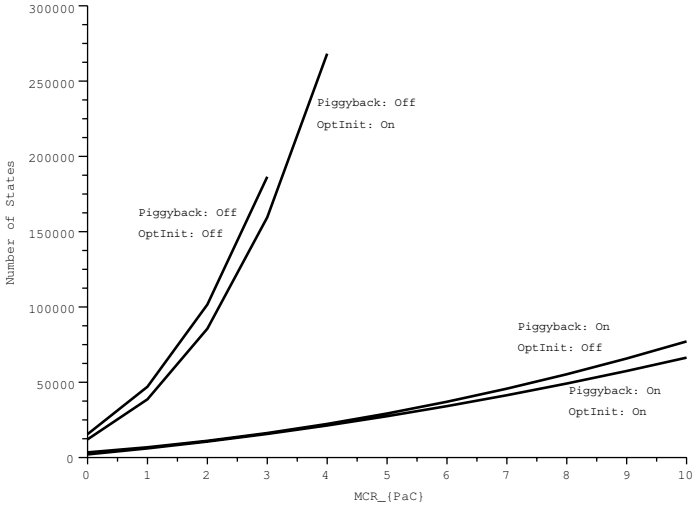


Figure 13: Impact of PaC retransmissions on state space size (in number of states)

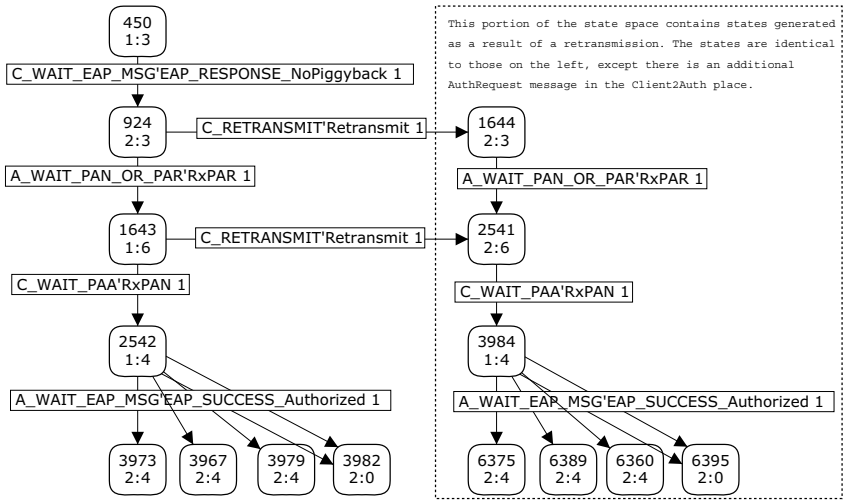


Figure 14: Partial state space showing effect of retransmission on state space size

In summary, the analysis with increasing MRC_{PaC} shows that retransmissions by the PaC do not adversely affect the operation of PANA. However, currently it is assumed only a single EAP Request is sent and messages cannot be lost. Further work is needed to analyse the impact of relaxing these assumptions.

5 Discussion

Steps from a protocol engineering methodology utilising Coloured Petri nets have been applied to PANA, a protocol for carrying authentication information between clients and servers. In previous work, initial CPN modelling and analysis of an earlier version of

PANA revealed an undesirable state when the authenticator aborted a session. This paper modelled and analysed the latest version of PANA, showing the undesirable state is now avoided. In addition, new PANA configurations have been analysed, in particular when Optimised Initiation is used by the authenticator, and when the client can retransmit messages. The results so far show no unexpected behaviour in the latest version of PANA.

The PANA CPN model has been developed over a period of about two years, over which time the PANA protocol specification and state tables have progressed from versions 14 to 18 and 5 to 13, respectively. The modelling, maintenance and analysis has amounted to 3-4 months of effort from a single person new to CPN Tools (but experienced with CPNs and Design/CPN). The following discusses lessons learned in the current work, as well as ideas for future work.

5.1 Modelling Approach

A state-based approach is used for modelling PANA as a CPN. The aim is for the CPN model to closely follow the PANA state tables, which is advantageous during the development of the protocol (and corresponding CPN model). In many cases there is a direct mapping from a state table row to a CPN transition. However the approach has limitations [3]. For example in the PANA CPN there are many transitions that model the same behaviour (but in different states) and could be folded together. In addition, by closely following the protocol state tables, little consideration is given in optimising the model for state space analysis.

Despite the close relationship between the original state tables and CPN model, as changes to PANA are made within IETF it is still time consuming to ensure those changes are accurately reflected in the CPN model. This was especially difficult with PANA, as there was an official PANA RFC with informal description of the protocol, as well as a separately maintained (and often out-of-date) Internet Draft for the state table description. With the IETF protocol descriptions, the difference between versions can be visually highlighted using *diff*-based tools. Similar functionality would be useful in CPN Tools: for example, highlighting CPN elements that have been changed since a previous model. Another method to assist in validating the PANA CPN model is to generate state tables directly from the model. Using the state-based modelling approach this is possible and is currently work-in-progress.

5.2 State Space Analysis

A property of interest for many communication protocols is the absence of deadlocks. With state space analysis of a CPN model, this requires specifying the expected terminal states. In this paper, the simplest possible definition of an expected terminal state is used: the state name of the PaC/PAA is expected (e.g. PaC and PAA both in the OPEN state). A more precise definition would also consider the state information stored by the PaC/PAA (e.g. value of session timer, current sequence number) as well as messages remaining in the communication places. Also, as only the Authentication and Authorisation phase of PANA is analysed in this work, the expected set of terminating conditions is quite large. As discussed in Section 4.2.1, there are 7 different valid states of the PaC/PAA, a number of which are valid only because it is expected in later phases a valid terminal state will be reached (because a time-out will occur). In the current CPN model the timeout in the subsequent phase is not explicitly modelled. This CPN model design decision was chosen to keep the analysis of PANA phases separate, however it leads to extra complexity when defining/analysing the terminal states. If all PANA phases were

analysed at once, the number of valid terminal states would be less (e.g. only the case when PaC/PAA are both CLOSED).

The state space results as the number of PaC retransmissions increases suggests, if the PANA service language is completed, parametric verification may be applicable to analyse PANA properties [10]. In addition, the CPN model can be optimised for state space analysis, and Figure 14 indicates equivalence classes may be promising as a state space reduction technique.

Acknowledgements

The detailed comments from the reviewers are highly appreciated, not only for improving this paper, but also for providing interesting ideas for future work.

References

[1] B. Aboba, L. Blunk, J. Vollbrecht, J. Carlson, and H. Levkowetz. Extensible Authentication Protocol. IETF RFC 3748, June 2004.

[2] J. Billington, G. E. Gallasch, and B. Han. A Coloured Petri net approach to protocol verification. In *Lectures on Concurrency and Petri Nets, Advances in Petri Nets*, pages 210–290. Springer-Verlag, 2004.

[3] J. Billington and S. Vanit-Anunchai. Coloured Petri nets modelling of an evolving internet standard: the Datagram Congestion Control Protocol. *Fundamenta Informaticae*, 88(3):357–385, 2008.

[4] P. Chamuczynski, O. Alfandi, H. Brosenne, C. Werner, and D. Hogrefe. Enabling pervasiveness by seamless inter-domain handover: Performance study of PANA preauthentication. In *Proc. Sixth Annual IEEE Intl. Conf. Pervasive Computing and Communications*, pages 372–376, Hong Kong, China, 17–21 Mar. 2008.

[5] V. Fajardo, Y. Ohba, and S. Das. Network service provider selection and security bootstrapping using PANA. In *Proc. of the 2nd Intl Conf. Testbeds and Research Infrastructures for the Development of Networks and Communities*, Barcelona, Spain, 1–3 Mar. 2006.

[6] V. Fajardo, Y. Ohba, and R. Lopez. State machines for protocol for carrying authentication for network access (PANA). IETF Internet Draft draft-ietf-pana-statemachine-06 (work in progress), Oct. 2007.

[7] V. Fajardo, Y. Ohba, and R. Lopez. State machines for protocol for carrying authentication for network access (PANA). IETF RFC 5609, Aug. 2009.

[8] D. Forsberg, Y. Ohba, B. Patil, H. Tschofenig, and A. Yegin. Protocol for carrying authentication for network access (PANA). IETF RFC 5191, May 2008.

[9] B. Gaabab, D. Binet, and J.-M. Bonnin. Authentication optimization for seamless handovers. In *Proc. 10th IFIP/IEEE Intl. Symp. Integrated Network Management*, pages 829–832, Munich, Germany, 21–25 May 2007.

[10] G. E. Gallasch and J. Billington. A parametric state space for the analysis of the infinite class of stop-and-wait protocols. In *Proc. 13th Intl. SPIN Workshop on Model Checking of Software*, pages 201–218, Vienna, Austria, 30 March - 1 April 2006.

[11] G. E. Gallasch and J. Billington. Parametric language analysis of the class of stop-and-wait protocols. In *Proc. 29th Intl. Conf. Application and Theory of Petri Nets and Other Models of Concurrency*, Xi'an, China, 25-27 June 2008.

[12] S. Gordon. *Verification of the WAP Transaction layer using Coloured Petri nets.* PhD thesis, Institute for Telecommunications Research, University of South Australia, Adelaide, Australia, Nov. 2001.

[13] S. Gordon. Formal analysis of PANA authentication and authorisation protocol. In *Proc. Ninth Intl. Conf. Parallel and Distributed Computing, Applications and Technologies*, Dunedin, NZ, 1-4 Dec. 2008.

[14] K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 1, Basic Concepts.* Monographs in Theoretical Computer Science. Springer-Verlag, Berlin, 1997.

[15] K. Jensen, L. M. Kristensen, and L. Wells. Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems. *Intl. J. Software Tools for Technology Transfer*, 9(3-4):213–254, June 2007.

[16] L. Liu and J. Billington. Verification of the Capability Exchange Signalling protocol. *Intl. J. Software Tools for Technology Transfer*, 9(3-4):305–326, June 2007.

[17] P. S. Pagliusi and C. J. Mitchell. PANA/GSM authentication for Internet access. In *Proc. Joint First Workshop on Mobile Future and Symposium on Trends in Communications*, pages 146–152, Bratislava, Slovakia, 26–28 Oct. 2003.

[18] T. Tanizawa, M. Goto, V. I. Fajardo, and Y. Ohba. A wireless LAN architecture using PANA for secure network selection. In *Proc. IEEE Intl. Conf. Wireless And Mobile Computing, Networking And Communications*, pages 111–118, Montreal, Canada, 22–24 Aug. 2005.

[19] S. Vanit-Anunchai. Towards formal modelling and analysis of SCTP connection management. In *Proc. Ninth Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools*, Aarhus, Denmark, 20-22 Oct. 2008.

[20] J. Vollbrecht, P. Eronen, N. Petroni, and Y. Ohba. State machines for Extensible Authentication Protocol (EAP) peer and authenticator. IETF RFC 4137, Aug. 2005.

[21] M. Westergaard. BRITNeY: Basic Real-time Interactive Tool for Net-based animation. URL: http://wiki.daimi.au.dk/britney/, Sept. 2006.

A New Coloured Petri Net Methodology for the Security Analysis of Cryptographic Protocols

Yongyuth Permpoontanalarp and Panupong Sornkhom
Logic and Security Laboratory, Department of Computer Engineering
Faculty of Engineering
King Mongkut's University of Technology Thonburi, Bangkok, Thailand
E-mail: yongyuth.per@kmutt.ac.th

Abstract

Cryptographic protocols are protocols which use cryptographic techniques to achieve certain tasks while preventing malicious parties to attack the protocols. The design and analysis of cryptographic protocols are difficult to achieve because of the increasingly attacking capabilities and the complex requirement of the applications. Therefore, attacks in many cryptographic protocols have been found later after they have been designed and even implemented. In this paper, we propose a new Coloured Petri Net methodology for the security analysis of cryptographic protocols. Our approach offers a simple but effective way to analyze multiple sessions of protocol execution. To demonstrate the practical uses of our approach, we apply our method to analyze Micali's contract signing protocol and TMN authenticated key exchanged protocol. Surprisingly we found many new attacks in those protocols.

1. Introduction

Cryptographic protocols are protocols which use cryptographic techniques to achieve certain tasks while preventing malicious parties to attack the protocols. There are many applications of cryptographic protocols, for example, authenticated key exchange protocols, web security protocols, e-payment protocols, e-banking protocols, e-voting protocols, etc.

The design and analysis of cryptographic protocols are difficult to achieve because of the increasingly attacking capabilities and the complex requirement of the applications. Therefore, attacks in many cryptographic protocols have been found later after they have been designed [1, 2, 3] and even implemented eg. [4, 5]. Note that in this paper we focus on only message replay attacks [6].

A lot of works on Petri nets [7] have been applied to analyze cryptographic protocols in [8-15]. They can be classified into two kinds. The first kind [8-14] offers a modeling and analysis method to find attacks. The second kind [15] provides a theoretical semantics for cryptographic protocols which can be used to prove properties of protocols, rather than to find attacks. We focus on the former kind due to its practical use. All the works in the first kind offer an analysis of a single session of protocol execution only.

In this paper, we propose a new Coloured Petri Net (CPN) methodology for the security analysis of cryptographic protocols. We adopt the CPN approach [16,17] due to the intuitive way to model cryptographic protocols by using the graph representation. Our approach offers a simple but effective way to analyze multiple sessions of protocol execution. Our new CPN methodology is based on our group's previous works [18-21]. We argue that our new CPN methodology improves on all existing CPN and Petri Net methods [8-14] for security protocols on several issues. In particular, our CPN method is the first CPN method which offers a security analysis methodology of multiple concurrent sessions of protocol execution. Furthermore, it offers a systematic method to analyze attacks in protocols. Also, it can detect more attacks with a better efficiency than all existing CPN methods for security protocols.

In essence, our new methodology offers four important concepts. Firstly, we use decomposition and multi-session scheduling techniques to analyze multiple sessions of protocol

execution. The decomposition allows us to construct a state space of one specific instance of multiple sessions of protocol execution at a time to reduce the size of the output state space. In general, such a specific instance means a setting or a configuration which specifies all required information for the execution of each concurrent session, for example, the parties who are involved in the protocol, all secrets, all nounces and all attackers in a session. The multi-session scheduling allows us to build a state space which contains only one alternating execution of multiple sessions of protocol run, instead of all possible alternating executions. As a result of these two techniques, the state space obtained is small and fast for analysis.

Secondly, we characterize attack states in the computed state space by using the concept of vulnerability events. Vulnerability events are events which may lead to a compromise of protocols, and such events are protocol dependent. The concept of vulnerability events provides a general method to characterize attack states intuitively and comprehensively. Thirdly, we develop an efficient method to extract attack traces without the need for any further computation on the state space. An attack trace describes how an attacker carries out their attacks successfully step by step. In other CPN works for security protocols, such an attack trace is computed by extracting a path from an initial state to an attack state. Fourthly, we propose a way to classify systematically a huge amount of found attack traces by using attack patterns. In general, an attack pattern describes the core of an attack, and it contains a minimal attack trace which leads to the attack. To the best of our knowledge, our four concepts are novel for the CPN methodology for analyzing cryptographic protocols.

To demonstrate the practical uses of our approach, we apply our methodology to two case studies which are Micali's contract signing protocol [22] and TMN authenticated key exchange protocol [23]. Surprisingly, we found many attacks in the two protocols. For Micali's contract signing protocol, we found new attacks in a single session of protocol execution in [19]. In [20], we also found many new multi-session attacks in both the original Micali's protocol and a modified version of Micali's protocol. For TMN authenticated key exchange protocol, in [21] we found six new attacks in the single session of protocol execution and two new attacks in multiple sessions of protocol execution. In fact, new attacks that we found in TMN protocol are quite surprisingly since TMN have been analyzed quite extensively [24-27].

In section 2, we provide the background on Micali's ECS1 and TMN protocol. In section 3, we compare our new CPN method with existing related works. In section 4, we present our new CPN methodology and apply it to two case studies which are Micali's ECS1 and TMN protocol.

2. Background

We use the following notations throughout the paper. $S \rightarrow R : M$ means that user S sends message M to user R. $SIG_X(M)$ represents party X's signature on a message M and we assume that M is always retrievable from $SIG_X(M)$. The encryption of a message M with party X's public key is denoted by $ENC_X(M)$. Also, $H(C)$ stands for the hash of message C, and $E_K(M)$ means symmetric encryption on message M by key K.

2.1. Micali's ECS1 Protocol [22]

Micali proposed an efficient optimistic fair exchange protocol for contract signing. The protocol aims to ensure that two exchanging parties get each other commitment on an agreed contract or neither of them does. There are three kinds of parties in the protocol : Alice as an initiator of the protocol, Bob as an responder of the protocol and a third trusted party who resolves a dispute between Alice and Bob during the exchange.

We denote Alice, Bob and a trusted party by A, B and TTP, respectively. It is assumed that both Alice and Bob have already agreed on a plaintext contract C before the exchange. Alice is committed to contract C as an initiator if Bob has both $SIG_A(C,Z)$ and M where $Z=ENC_{TTP}(A,B,M)$ and M is random. On the other hand, Bob is committed to C as a responder if Alice has both $SIG_B(C,Z)$ and $SIG_B(Z)$. However, there is no need for Alice to verify Z to prove Bob's commitment.

The following is the detail of the protocol.

A1: 1) $A \rightarrow B$: $SIG_A(C,Z)$

B1: 2) $B \rightarrow A$: $SIG_B(C,Z)$, $SIG_B(Z)$

A2: If Bob's signatures in step 2 are both valid, then

3) $A \rightarrow B$: M

B2: If Bob receives valid M such that $Z=ENC_{TTP}(A,B,M)$
then the exchange is completed
else Bob requests TTP to resolve a dispute by the following step
4) B→TTP: A, B, Z, $SIG_B(C,Z)$, $SIG_B(Z)$
TTP1: If Bob's signatures in step 4 are both valid and $Z=ENC_{TTP}(A,B,M)$ then
5a) TTP→A: $SIG_B(C,Z)$, $SIG_B(Z)$
5b) TTP→B: M

Note that the request to TTP at step 4) contains identities of initiator and responder which have the dispute. Also, to resolve the dispute, TTP sends required information to related parties.

In [28], Bao et al. analyzed ECS1 manually and found three message replay attacks in ECS1. It can be argued that many of these attacks are caused by an ambiguity at how TTP should resolve a dispute. Then, Bao et al. modify ECS1 to solve the ambiguity problem in the original ECS1, and they found one attack in the modified version. In other words, Bao showed that a simple modification on TTP's behavior to resolve a dispute is not adequate. The modified ECS1 is similar to the original ECS1 except that in step TTP1, there is no check on $Z=ENC_{TTP}(A,B,M)$ and messages in steps 5a) and 5b) are also sent to identities A and B which are in step 4 and are obtained from the decryption of Z. In other words, in the modified ECS1, TTP resolves the dispute even when Z is not correct, ie. $Z \neq E_{TTP}(A, B, M)$.

In [29], Zhang and Liu applied a model checking technique to analyze the security of ECS1. They found three new attacks. We will discuss about their new attacks and the comparison with our new attacks in section 3.2.

2.2. TMN authenticated key exchange protocol [23]

TMN is a cryptographic key exchange protocol for mobile communication system. TMN allows user A to exchange a session key with user B by the help of server J. The user A is called an initiator, but the user B is called a responder. The detail of TMN is described as follows.

1) $A \rightarrow J : (B, ENC_J(K_{aj})), A$
2) $J \rightarrow B : A$
3) $B \rightarrow J : (A, ENC_J(K_{ab})), B$
4) $J \rightarrow A : B, E_{Kaj}(K_{ab})$

Where K_{ab} is an exchanged session key and K_{aj} is A's secret which is used to transport the session key at the last step. Note that the session key is created by user B. In [23], it is suggested that the Vernam cipher (or one-time pad) and RSA public key algorithm are used as the underlying symmetric encryption $E_K(M)$ and the public key encryption, respectively.

TMN has been analyzed extensively by many formal method approaches [24-27], and many attacks were found. But it was analyzed manually also in [23] by Simmon. Simmon found an attack to TMN by using the homomorphic property of the underlying public key cryptographic algorithm. An attacker can learn an exchanged session key easily, and the server cannot detect any message replay. The attack involves two concurrent sessions of protocol execution. We will discuss about the analysis of TMN by formal methods in section 3.2.

3. Related works

3.1. Petri Nets for Cryptographic Protocols

A lot of works on Petri nets [7] have been applied to analyze cryptographic protocols in [8-15]. They can be classified into two kinds. The first kind [8-14] offers a modeling and analysis method to find attacks. The second kind [15] provides a theoretical semantics for cryptographic protocols which can be used to prove properties of protocols, rather than to find attacks. We focus on the former kind due to its practical use.

All the works in the first kind [8-14] offer an analysis of a single session of protocol execution only. Moreover, the works in [8-11] employ some extended Petri nets to analyze security protocols but those Petri nets are less expressive than CPN. In [12], CPN is applied to analyze a denial of service attack on cryptographic protocols in a single session of protocol execution. The work [12] analyzes protocols by using the simulation technique but our work analyzes protocols by

using the state space computation. Furthermore, while their work analyzes the denial of service attack, we analyze attacks on the confidentiality, the authentication, key exchange, fair exchange properties. So the kinds of attacks are different.

There are two works [13, 14] which are closest to ours. They apply CPN to analyze cryptographic protocols on similar kinds of attacks to ours. However, both works do not really provide an analysis of multiple concurrent sessions of protocol execution, but just two sequential sessions of protocol execution. Even though [13] offers an analysis of two sequential sessions of protocol execution, it has to analyze one session at a time. In other words, the analysis of two sequential sessions is non-compositional. In particular, after an execution of the first session is finished, the second session is then executed separately with information obtained from the first session. So, it is considered as a single session analysis strictly. However, the work [14] offers a compositional analysis of two sequential sessions of protocol execution according to the detected attack reported in the work. The analysis is compositional in that two sequential sessions can be analyzed at a time. Since it is a sequential composition of two sessions, then the work is essentially a single session analysis. Moreover, the attacker cannot initiate a new session with any user. So, only attacks where a legitimate user is a session initiator can be detected, and these are very limited. More importantly, a methodology to deal with a huge amount of alternating executions between multiple sessions has not been addressed at all in both works. But the work [14] deals with the alternating executions between a user and an attacker, since their attacker model can be executed independently of the execution of protocol steps. Such alternating executions occur in a single session only.

It is important to note that the analysis of multiple concurrent sessions of protocol execution is important since many crucial attacks, for example the man-in-middle attack [30] and parallel-session attack [31] are carried out in the multiple concurrent sessions where protocol runs are alternated in a non-sequential manner between multiple sessions.

Recently in [18-21], our group has developed a new CPN method to analyze cryptographic protocols. Our new CPN method improves on all existing CPN and Petri Net methods for security protocols on several issues. In particular, our CPN method is the first CPN method which offers an analysis methodology of multiple concurrent sessions of protocol execution. Furthermore, it offers a systematic method to analyze attacks in protocols. In particular, our method offers decomposition and multi-session scheduling for the computation of state spaces, an intuitive approach to characterize attack states, an efficient way to extract attack traces and a systematic way to classify a large amount of attack traces. In addition, our method can detect more attacks with a better efficiency. In particular, our attacker model allows session initiation, receiver impersonation and message dropping capabilities all of which are missing in [14]. As a result, more kinds of attacks can be analyzed. Also, our method is more efficient due to more restricted underlying models of users and attackers. In particular, in the user model there is a strict message validation for every received message, and each type of cryptographic messages, eg. public-key ciphertext and symmetric-key ciphertext, is defined individually instead of mixing them into one type Also, our attack model generates messages by taking both the message validation and the individual types of cryptographic messages into account. As a result, a state space generated is smaller than [14] since invalid messages and inappropriate cryptographic messages are eliminated.

3.2. Other formal methods for analyzing Cryptographic protocols

There are a huge amount of formal methods for analyzing cryptographic protocols. A survey and discussion on the comparison between them can be found in [2,3]. Here we discuss only formal methods which are applied to analyze contract signing protocols, ECS1 and TMN protocols.

There are at least three main works which analyze general contract-signing protocols. In [32], Chadha, Kanovich, and Scedrov proposed an inductive proof method to analyze a variant of contract-signing protocol proposed by Garay, Jakobsson and MacKenzie. Their method aims to prove fairness and abuse-free properties of the protocol, rather than to find attacks. In addition, their method is manual. In [33], Shamatikov and Mitchell applied a model checking system called Murφ to analyze two contract signing protocols. A protocol is modeled as an automata by using a programming language. Their method is automatic and some new attacks on the protocols are discovered. In [34], Gurgens and Rudolph analyzed a number of fair exchange non-repudiation protocols using asynchronous product automata (APA) and the simple homomorphism verification

tool (SHVT). Similar to Murφ, a protocol is modeled as an automata but by using a text-based description on states and state transitions. Their method is automatic and some new attacks are found.

In [29], Zhang and Liu applied a model checking technique to analyze ECS1. They found one new single-session attack in Micali's ECS1 and two new multi-session attacks in Bao et. al. 's modified version of ECS1 [28]. Independently, we found two new single-session attacks in Micali's ECS1 in [19], and found two new multi-session attacks in Micali's ECS1 and five new multi-session attacks in Bao's modified version of ECS1 in [20]. In fact, at the writing time of our works in [19, 20], we were unaware of Zhang and Liu's work. However, after we look into the details, we found that one of our single session attacks is a variant form of Zhang and Liu's attack, and two of our multi-session attacks are variant forms of Zhang and Liu's attack. Therefore, to summarize the new attacks found by our CPN method we found one new single-session attack of Micali's ECS1, two new multi-session attacks in Micali's ECS1 and three new attacks of Bao's modified version of ECS1.

There are seven formal method approaches which are applied to analyze TMN protocol. In [24], three formal method approaches, namely NRL, Interrogator and Inatest, to analyze cryptographic protocols are compared, and the TMN protocol is chosen as a case study. All of them take the state exploration approach to analyze protocols. Both NRL and Interrogator detect an attack in a single session of protocol execution. However, Inatest can only reproduce Simmon's attack [23] which is an attack in multiple sessions. In [25], Murφ, a general model checker, is applied to analyze the TMN protocol. Murφ can reproduce Simmon's attack, and it detects a new multiple session attack which allows an attacker to learn an exchanged session key between two legitimate parties. However, both legitimate parties do not commit on the session key after the completion of the protocol. In [26], Lowe and Roscoe applied Communicating Sequential Processes (CSP) and its model checker FDR to analyze the TMN protocol. Not only all previously known attacks to the TMN protocol can be reproduced, but also two new attacks have been found. The first new attack occurs in a single session of protocol execution, and an attacker can impersonate user A and learn the exchanged session key created by user B. The second attack which occurs in multiple sessions is that both A and B commit on the same session key after the completion of the protocol, but the key is known by the attacker. In [14], Al-Azzoni et. al. applied CPN to analyze the TMN protocol. Their CPN method can only detect a variant form of the attack found by Murφ [25]. Note that the attack occurs in two sequential sessions of protocol run. In [27], Zhang and Liu employed a model checking technique to analyze the TMN protocol. They found some variant forms of Lowe and Roscoe's attacks [12] in both a single session and multiple sessions. Even though TMN protocol has been analyzed very extensively, surprisingly we found new attacks in the protocol by using our CPN methodology.

In summary, we argue that the CPN methodology offers an advantage over those formal methods discussed previously in that it provides a simpler and more intuitive way to model a protocol by using the graph representation.

4. Our Model

4.1. Our New Methodology

Our new CPN method here is based on the CPN approaches that we have developed earlier in [18-21]. There are five steps. The detail of our methodology is shown as follows.

First, we build a CPN model to represent message exchange by all user parties and to represent attacker behavior. Each user party is modeled according to the protocol. In general, an attacker in our model can eavesdrop, modify and drop messages during the transmission. Also, the attacker can send new messages. We will discuss about assumptions of the protocol and the abilities of the attacker in details later.

Second, an automata or a state space of the protocol with attackers is generated by using the state space tool in CPN Tools [17]. In general, the state space represents all possible behaviors of every party, including attacker, in the protocol. Instead of generating the state space of all possible instances of multiple concurrent sessions of protocol execution at once, we generate a state space of one specific instance of multiple concurrent sessions at a time to reduce the size of the output state space. We call this a decomposition technique. In general, such a specific instance means a setting or a configuration which specifies all minimally required information for the protocol execution, for

example, the identities of initiator and responder, the role of attackers, all secrets and all nounces in each concurrent session, and a schedule of the execution of the multiple concurrent sessions. The schedule ensures that the output state space contains one alternating execution of multiple concurrent sessions of protocol run only, instead of all possible alternating executions. Exploring all possible alternating executions within a state space is expensive and causes a huge state space. Usually there are many possible configurations. As a result of the decomposition and the multi-session scheduling techniques, the state space obtained is small and fast for analysis.

Third, we create a query function in CPNML language, which is based on ML functional programming language, to search for attack states in the state space. Attack states are characterized by vulnerability events. Vulnerability events are events which may lead to a compromise of protocols, and such events are protocol dependent. In ECS1, there is one vulnerability event where one party, who is either initiator or responder, gets another party commitment, but the latter does not get the former commitment. In other words, this event describes exactly an unfair state. In TMN protocol, there are two main vulnerability events. Firstly, the attacker learns a secret key, for example, the exchanged session key. Secondly, a session key which may be a fake key is committed by a user. Based on these two vulnerability events, several combined vulnerability events are created in order to characterize many meaningful attack states in TMN protocol. The concept of vulnerability events provides a general method to characterize attack states intuitively and comprehensively. Also, queries can be built easily to detect such combined vulnerability events.

Fourth, after attack states are discovered from the state space, we extract attack traces. Conceptually, an attack trace describes how an attacker carries out an attack successfully step by step. Given an attack state in a state space, a path from the initial state to the attack state contains a sequence of actions by all parties, including attackers, which leads to the attack. So, this sequence contains an attack trace. But the sequence contains superfluous and redundant information about an attack trace since it contains all CPN transitions in an execution which lead to an attack state, and many of those transitions are redundant or irrelevant to the essence of the attack. Our method offers an efficient new approach to extract attack traces from an output state space without the need for any further computation. In our CPN model, as the protocol execution proceeds, an attack trace which is a record of all exchanged messages between parties so far is embedded into an output state. More specifically, the attack trace is stored in a global fusion place when a message is sent from one user to another. Thus, when an attack state is found, the attack trace can be extracted from the state immediately and efficiently.

Fifth, after attack traces are obtained, we classify them into each group. In general, there can be a huge amount of attack states and traces found in a state space. For example, in ECS1 we found 7,000 attack states (and traces) in a configuration. Thus, to ease the analysis of a large amount of attack traces, we develop an attack classification by using attack patterns. In general, an attack pattern describes the core of an attack, and it contains a list of minimal protocol messages that are the cause of each attack. Attack traces that produce the same attack pattern are classified into the same group of attacks. In general, it requires human intervention to create each attack pattern from an attack trace.

In the next two sections, we apply our CPN method to analyze two case studies. To illustrate the effectiveness of our method, we focus our analysis on TMN. Our CPN framework for ECS1 is similar to the framework for TMN.

4.2. Our CPN Analysis for TMN Protocol

In this section, we discuss the analysis of TMN by using our new methodology. In section 4.2.1, we provide definitions and more concrete concepts of our methodology. In section 4.2.2, we briefly show some of our CPN graph model. We explain our queries and our attack classification technique in sections 4.2.3 and 4.2.5, respectively. Also, new attacks and the performance of our method are discussed in sections 4.2.6 and 4.2.7, respectively.

4.2.1. Our CPN framework for TMN

In this section, we discuss the assumptions of our protocol analysis. We also describe vulnerability events of TMN. Finally, we provide a definition of a configuration of the protocol execution.

Definition 1 : The assumptions of the protocol execution
The following are the assumptions of the execution of the TMN protocol.

1. There are three users who are an initiator, a responder and a server. And all the users follow the protocol specification strictly and honestly.
2. There is one attacker whose abilities are defined below.
3. The underlying encryption is perfect in that nothing can be inferred from a ciphertext without the knowledge of the correct key. Also, we consider a general public key encryption scheme, rather than any specific scheme.
4. We consider the execution of two concurrent sessions of the protocol where such execution can be performed in either a sequential or a non-sequential but alternating style.
5. Both initiator and responder involve in the protocol execution as if there is one session of execution only, but the server may involve in more than one session.

The assumptions 1) and 2) mean that those parties are all that are involved in the protocol execution. An initiator means a user who initiates a new protocol session, and a responder means a user who responds to an existing session with an initiator to perform the key exchange.

The first part of assumption 3) is also known as Dolev and Yao's assumption [35]. As stated in the second part of assumption 3), in this paper we consider a general public key encryption scheme rather than RSA scheme. This assumption is sufficient to illustrate the effectiveness of our new CPN approach to analyze TMN.

In assumption 4), we mean that in addition to a sequential execution of two sessions, the protocol execution can alternate between the two concurrent sessions. Thus, the result of the execution is a non-sequential or interleaving manner. We will discuss about the execution of two concurrent sessions of the protocol more specifically later.

In assumption 5), we mean that both initiator and responder think that they involve in only one session of the protocol execution, and their goals are to exchange a session key between them. However, the server may involve in more than one session of the execution. In fact, this assumption is reasonable since a server just responds to any request and it maintains a state only during a request at step 1 and a respond at step 4. After the step 4 occurs, the state is destroyed to minimize the resource.

Definition 2 : The attacker abilities
The attacker in our model is capable of the following:

1. The attacker can eavesdrop, modify and drop messages during the transmission between users.
2. The attacker can send a message to a user.
3. The attacker can either initiate a new session with users or take part in an existing session with users.
4. The attacker can impersonate any user.
5. The attacker can perform any cryptographic computation by using known keys, known messages and known ciphertexts with a limited but reasonable power, eg. encryption and decryption.
6. The attacker has its own storage with a finite and reasonable amount.
7. The attacker does not attack himself.
8. There is at most one attacker who performs the attack in 1) on a protocol step in a session at a time.

Note that 2) means the ability to send any message to any user where the message and the user may or may not be according to the protocol specification. Thus, it is different from 3).

The assumption 4) means that the attacker can impersonate an initiator A, a responder B or a server J. If the impersonated user is a responder or a server, then the attacker must be able to intercept an input message and then to send a fake respond message at a next step. But if the impersonated user is an initiator, the attacker must be able to initiate a new session, and to generate fake responds.

In 5), the attacker also has the ability to perform any computation in addition to the cryptographic one.

In 7), any message that is sent to an attacker who may impersonate a user will not be modified by anyone during the transmission. Note that if there is any modification, then the attacker is attacked. In 8), any message that is sent from an attacker will be delivered to the intended receiver

intact. If there is anyone else who modifies the sent message further, then the message is modified by two attackers.

Definition 3 : The basic goals of the attacker
There are two basic goals of the attacker for TMN
1. The attacker aims to disclose a secret key which is a session key or A's secret.
2. The attacker aims to impersonate a user which is an initiator or a responder

These are two basic and general goals of the attacker for TMN. We focus on the first goal, but still consider the second goal, since the first goal incurs a worse damage than the second one. In addition, the attacker has more aims to compromise the protocol by achieving the combined vulnerability events which will be discussed later. Those events can be considered as concrete and advanced goals of the attacker.

Attack states are characterized by vulnerability events. For the TMN protocol, there are two basic events, and five combined events.

Definition 4 : The first basic vulnerability events
The attacker learns a secret key. There are two cases.
1. The attacker learns the exchanged session key K_{ab}. *(K_K_{ab})*
2. The attacker learns A's secret K_{aj}. *(K_K_{aj})*

Definition 5 : The second basic vulnerability events
There are three cases for a session key which is committed by users.
1. Both A and B commit on K_{ab}. *(AB_K_{ab})*
2. A commits on K_i or K_{aj}. *(A_K_i, A_K_{aj})*
   where K_i is attacker's secret key.
3. B commits on K_{ab}. *(B_K_{ab})*

Note that 1) does not look like a vulnerability event on its own, but when it is combined with another vulnerability event, it becomes a vulnerability event clearly. We will discuss about this later. Also, it is not possible to fool user B to commit to other key than K_{ab} since B is the creator of the session key K_{ab} and then the key is fixed for the communication with A. Based on the two basic events, the following combined and interesting vulnerability events can be created.

Definition 6 : The combined and interesting vulnerability events.
There are five combined vulnerability events.
1. The attacker learns K_{ab}, and both A and B commit on K_{ab}. *[K_{ab}][K_{ab}][K_{ab}]*
2. The attacker learns K_{ab} and K_{aj}, and both A and B commit on K_{ab}. *[K_{ab},K_{aj}][K_{ab}][K_{ab}]*
3. The attacker learns K_{ab}, and A is fooled to commit on K_i but B commits on K_{ab}. *[K_{ab}][K_i][K_{ab}]*
4. The attacker learns K_{ab} and K_{aj}, and A is fooled to commit on K_i but B commits on K_{ab}. *[K_{ab},K_{aj}][K_i][K_{ab}]*
5. The attacker learns K_{ab} and K_{aj}, and A is fooled to commit on K_{aj} but B commits on K_{ab}. *[K_{ab},K_{aj}][K_{aj}][K_{ab}]*

We use the notation *[KB_1][KB_2][KB_3]* to describe each combined vulnerability event where KB_1 stands for keys that are known by the attacker, and KB_2 and KB_3 stands for and keys that are committed by users A and B, respectively, at the completion of the protocol.

Clearly, in the event 1 the attacker will then learn all later communication between A and B, because the attacker obtains the session key which are exchanged and agreed by both A and B. We do not find any attack instance in this event.

The event 2 is similar to the event 1 but the attacker in the event 2 learns an additional key which is A's secret. Lowe and Roscoe's multi-session attack [26] is in the category of this event. In this event, we found 10 attack patterns, and many of them are interesting variant forms of Lowe and Roscoe's attack. For example, in some variant attacks, server J cannot detect the replay attack occurred in the two sessions. Its detail will be discussed later.

In event 3, the attacker learns the session key, and fools A to commit to a fake key which is the attacker's secret K_i. The event 4 is similar to the event 3, but the attacker learns A's secret key in

addition the session key. In event 5, the attacker learns the session key and A's secret key, and fools A to commit to a fake key which is A's secret key K_{aj}. The result of the events 3, 4 and 5 can be seen as a kind of the man-in-the-middle attacks. In events 3 and 4, the attacker can impersonate B to A by using key K_i, while the attacker can impersonate A to B by using key K_{ab}. In the event 5, the attacker can impersonate B to A by using key K_{aj}, while the attacker can impersonate A to B by using key K_{ab}. We found 10 attack patterns in the events 4 and 5, but do not find any attack instance in the event 3. It is the attacks in the events 4 and 5 that are novel.

Note that there are other two combined vulnerability events which are $[K_{aj}][K_i][K_{ab}]$ and $[K_{aj}][K_{aj}][K_{ab}]$. However, both events are not interesting since the attacker does not learn the session key K_{ab}. Moreover, they are just variant forms of single session attacks. So, we do not consider them here.

Definition 7 : The computation of the five combined vulnerability events

The following shows how to compute the five combined vulnerability events.

1. $[K_{ab}][K_{ab}][K_{ab}] = (K\_K_{ab} \cap AB\_K_{ab})$
2. $[K_{ab},K_{aj}][K_{ab}][K_{ab}] = (K\_K_{ab} \cap A\_K_i \cap B\_K_{ab})$
3. $[K_{ab}][K_i][K_{ab}] = (K\_K_{ab} \cap K\_K_{aj} \cap AB\_K_{ab})$
4. $[K_{ab},K_{aj}][K_i][K_{ab}] = (K\_K_{ab} \cap K\_K_{aj} \cap A\_K_i \cap B\_K_{ab})$
5. $[K_{ab},K_{aj}][K_{aj}][K_{ab}] = (K\_K_{ab} \cap K\_K_{aj} \cap A\_K_{aj} \cap B\_K_{ab})$

Each combined vulnerability event can be computed by applying the intersection on the relevant basic vulnerability events.

In the following, we provide the definition of a configuration for a state space computation of our CPN framework to analyze the TMN protocol.

Definition 8 : A configuration of our CPN framework for TMN

A configuration of our CPN framework consists of $((S_1, S_2,\ldots,S_n), Sch, Tr)$ and $S_i = (s,I,R,T,K,N)$ for $1 \le i \le n$ where

1. S_i is a session information which consists of
   - 1.1. s is a session identity
   - 1.2. I is an initiator identity
   - 1.3. R is a responder identity
   - 1.4. T is a server identity
   - 1.5. K is keys for each party (including attacker) which consists of
     - a) Pair of public and private keys
     - b) Shared key with a specific party
   - 1.6. N is nounces used by each party
2. Sch is a multi-session schedule which contains a list of session identities to be executed in that order
3. Tr is an attack trace which consists of a vulnerability event and a list of protocol traces which leads to the vulnerability event

In the configuration, each S_i and Sch are input parameters to the CPN state space computation while Tr is the desired output from the state space computation.

For example, the 1st session information S_1 which is $(1,A,B,J,(K_1,K_2,K_3,K_4),\_)$ means that A, B and J are identities for initiator, responder and server, respectively, and K_1, K_2, K_3 and K_4 are keys for A, B, J and In respectively. Also, "_" means that there is no information about the nounces. Let $K_1 = (\_,\{(K_{aj},\_)\})$, $K_2 = (\_,\{(K_{ab},A)\})$, $K_3 = (\{(PK_J,SK_J)\},\_)$ and $K_4 = (\_,\{(K_i,\_)\})$. K_1 means that A has no public and private keys, but has one shared key K_{aj} which can be used with anyone. K_2 means that B has no public and private keys, but has one shared key K_{ab} which is intended to share with A. K_3 means that J has a public key PK_J and a private key SK_J, but J has no share key. K_4 means that the attacker has no public and private keys, but has one shared key K_i which can be used with anyone.

Let Sch be $[1,1,1,1,2,2,2,2]$. In this schedule, "1" and "2" mean a complete execution of one protocol step in the first and second session, respectively. In the execution, both message sending and receiving in the step are performed. This means that a sender has sent a message, and a receiver has received the message. So the schedule is just the sequential execution of the first session and then the

second session. Consider another schedule *[1,2,2,1,1,2,2,1]*. This schedule is a non-sequential but concurrent execution which corresponds to the man-in-the-middle attack [30]. The man-in-the-middle attack means that the attacker situates in the middle between two sessions, and replays messages between them. The figure 1 illustrates the flow of messages for the man-in-the-middle attack in TMN where the first session is between A and *In*, and the second session is between *In* and B. Thus, the schedule means an alternating execution between two sessions in that the first protocol step of the 1st session is executed first, and then two protocol steps of the 2nd session are executed, and so on.

Let *Tr* be *([K_{ab},K_{aj}], [K_i], [K_{ab}], LTr)* and *LTr* be *[(1,1,A,J, ((B,{K_{aj}}PK-J),A)), (1,1,In,J,((X2,{K_i}PK-J),X1)), ...]*. The attack trace *Tr* consists of a vulnerability event *([K_{ab},K_{aj}] [K_i] [K_{ab}])*, and a list *LTr* of protocol traces which leads to the attack for the vulnerability event. As for a simple example, the list *LTr* of protocol traces is only partial and contains two steps only. The first protocol trace *(1,1,A,J, (B,{K_{aj}}PK-J),A))* means that in the first session and at the first protocol step, user *A* sends message *(B,{K_{aj}}PK-J),A* to server *J*. The second protocol trace *(1,1,In,J,(X2,({K_i}PK-J),X1))* means that in the first session and at the first protocol step, the attacker *In* intercepts the message sent in the first trace, and modifies it to *(X2,{K_i}PK-J),X1* which is delivered to *J*.

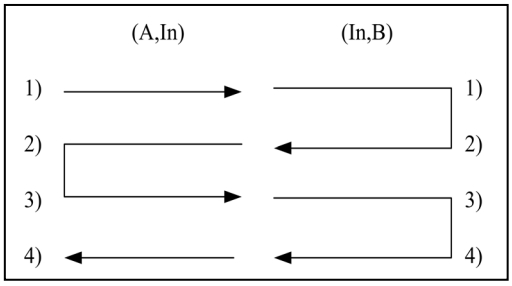


Figure 1: The flow of messages for the man-in-the-middle attack

To analyze the attacks in TMN, we consider only the schedule *[1,2,2,1,1,2,2,1]* which is clearly the execution of non-sequential but alternating sessions of the protocol. It suffices to consider only this schedule to illustrate the effectiveness of our CPN method. Other schedules can be considered too as different input parameters.

We consider four kinds of two concurrent sessions which are as follows.

1. *(1,A,B,J,(K_1,K_2,K_3,K_4),_) & (2,In,In,J,(K_1,K_2,K_3,K_4),_)*
2. *(1,A,In,J,(K_1,K_2,K_3,K_4),_)& (2,In,B,J,(K_1,K_2,K_3,K_4),_)*
3. *(1,In,B,J, (K_1,K_2,K_3,K_4),_)& (2,A,In,J,(K_1,K_2,K_3,K_4),_)*
4. *(1,In,In,J, (K_1,K_2,K_3,K_4),_)& (2,A,B,J, (K_1,K_2,K_3,K_4),_)*

where K_1, K_2, K_3 and K_4 are discussed previously.

In session *(1,A,B,J,(K_1,K_2,K_3,K_4),_)*, the attacker behaves as an external observer on the communication amongst A, B and J. In session *(1,A,In,J,(K_1,K_2,K_3,K_4),_)*, the attacker impersonates a responder to user A and server J. In session *(1,In,In,J,(K_1,K_2,K_3,K_4),_)*, the attacker impersonates both A and B to server J. Thus, there are three explicit roles of our attacker : an external observer and impersonators for initiator and responder. The role of server impersonation is implicitly enabled, but it is disabled in the session where both A and B are impersonated.

These four concurrent sessions are all possible sessions regarding to the goal of the user impersonation attack.

4.2.2. Our CPN graph model

Our CPN graph model for TMN extends the CPN graph model in [14] on many issues to provide the new methodology discussed in section 4.1. The CPN graph model consists of four levels: top, entity, sub-entity and control. The top level shows the interaction between all parties including the attacker. The entity level shows the detail of the overall behaviour of each party according to the protocol, and the sub-entity level shows the detail of a specific behaviour of a party. The control level controls the execution of the model according to an input schedule.

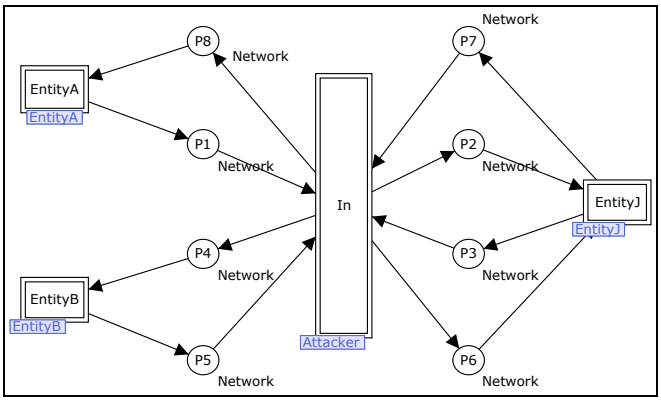


Figure 2: Top level

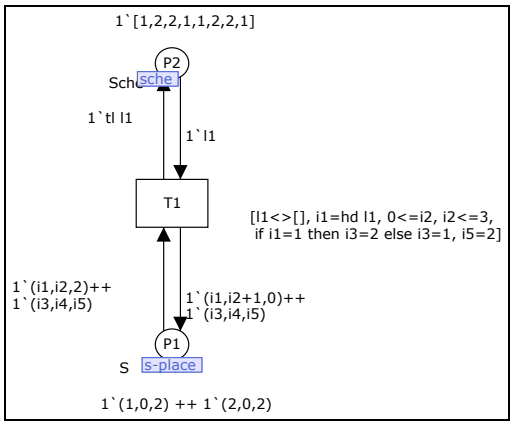


Figure 3: Entity A

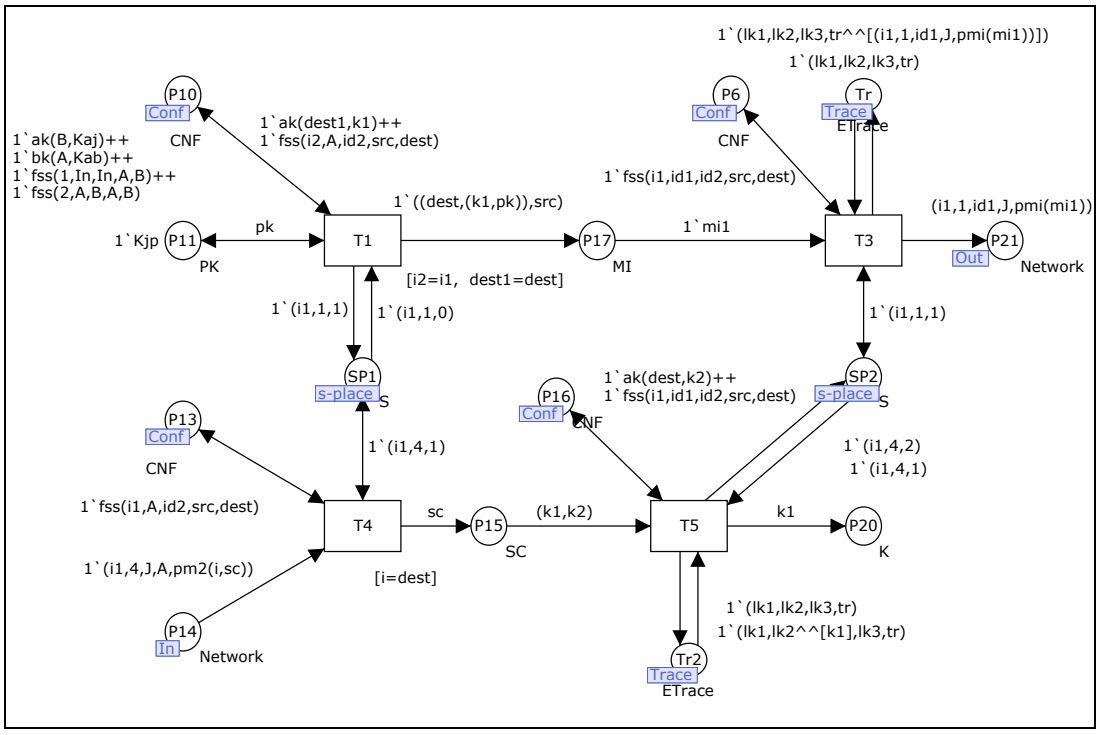


Figure 4: Entity A

Figure 2 shows the top level. There are 4 entities in our model which are A, B, J and attacker *In*. All messages that are exchanged between all users (A, B and J) pass through the attacker *In*. Figure 3 shows the control level. The input schedule *[1,2,2,1,1,2,2,1]* is given at the *Sche* place. The place *s-place* keeps session states where a session state consists of (*sid, sp, st*) where *sid* means the session identity, *sp* means the counter of protocol steps and *st* means states. There are three states which are 0 (ready), 1 (executing) and 2 (inactive or finished). For example, state (1,1,1) means that the session 1 is during the execution at the first protocol step. Initially, we have two session states: (1,0,2) and (2,0,2). Transition T1 activates the execution of a session at a time according to the schedule, and also increments the counter of protocol steps to be executed. As a result of the activation, the protocol step at the counter is executed.

Figure 4 shows the entity level for A. Transitions T1 and T3 are for creating the message sent at the first step of the protocol, whereas transitions T4 and T5 are for processing the message received at the last step. The place *Conf* stores a session configuration which consists of *fss, ak* and *bk* tokens where *fss(sid, i1, i2, i3, i4)* means that in session identity *sid, i1* and *i2* are identities of the actual initiator and actual responder, respectively, and *i3* and *i4* are identities that *i1* and *i2* use, respectively, in the message. Token *ak(id,k)* means that *k* is a shared key between A and *id*, and *bk(id,k)* describes a shared key *k* between B and *id* similarly. The place *trace* stores an attack trace when A sends a message to J. Transition T1 is enabled if session state is *(i1,1,0)* and A is the actual initiator in session *i1*. After the session *i1* is executed, its session state becomes *(i1,1,1)*. Transition T4 is to validate a received message, and T5 is to decrypt the received ciphertext. After the step 4 is terminated, its session state becomes *(i1,4,2)*.

4.2.3. Queries

After a state space is computed for each configuration, we search for attack states in the state space. As discussed previously, attack states are characterized by vulnerability events. It is straightforward to construct queries to detect the combined vulnerability events. However, our method is that queries are applied to only terminal states or nodes in the state space in order to verify if certain conditions are met. The terminal states are states which do not have any further possible computation, and they mean states at the completion of the protocol execution. Note that while we consider terminal states, Al-Azzoni et. al. in [14] consider all states which are unnecessary and inefficient.

The following shows a query for the first basic vulnerability event.

```
val LeafNodes=ListDeadMarkings();
fun SecrecyViolation1(k:K) :
Node list
= PredNodes (LeafNodes,
            fn n => (cf(cK(k), Mark.SymDec'P3 1 n) > 0),
             NoLimit);
```

The *SecrecyViolation1*(K_{aj}) produces a set of all terminal nodes in the state space where the key K_{aj} is present at the attacker's database which is represented by the place number P3.

The following shows a query for the second case of the second basic vulnerability events.

```
fun SecrecyViolation2(k:K) :
Node list
= PredNodes (LeafNodes,
            fn n => (cf(k, Mark.EntityA'P20 1 n) > 0),
            NoLimit);
```

The *SecrecyViolation2*(K_{aj}) produces a set of all terminal nodes in the state space where the key K_{aj} is present at A's database after the completion of the protocol. Note that A's database is represented by the place number P20.

So, a query to detect the combined vulnerability events can be created by applying the intersection on the queries with appropriate keys. For example, the combined event 1 is obtained by the intersection on the queries *SecrecyViolation1*(K_{ab}) and *SecrecyViolation2*(K_{ab}). Note that we do

not need to query for B's commitment on a session key since B always commits on key K_{ab} due to the protocol specification.

4.2.4. Obtaining attack traces

After the attack states are found by using the queries, we extract attack traces which describe how the attacker carries out those attacks successfully step by step. Our method to extract attack traces is very efficient. Since such an attack trace is recorded into a global fusion place while the protocol execution proceeds step by step. Thus, an attack trace can be obtained from the global fusion place in an attack state immediately. Note that in other CPN approaches for security protocols, an expensive computation is required to find a path from an initial state to an attack state, and then to extract relevant information from the path to create an attack trace.

4.2.5. Attack Classification

Usually, there can be a huge number of attack states found in a state space. In fact, we found 360 different attack states in a state space obtained from in the first configuration as shown in table 1.

After we obtain a large amount of the found attack traces, we analyze them manually first. We found that many of the attack traces share a similar pattern. In general, ciphertexts that are sent in all messages are essential to the attacks, but the identities of initiator and responder in the messages that are modified by the attacker are not important. However, the identities of initiator and responder in the messages that are sent from A in step 1, sent to B in step 2 and sent to A in step 4 are important. These identities are used by J for the key exchange function and by A and B for input validation. If we ignore unimportant parts of protocol messages in protocol traces, then we obtain a trace pattern. By considering trace patterns, the number of attacks is decreased tremendously as shown in table 1. Indeed, the process to find trace patterns is manual and protocol-dependent.

Then, we construct an attack pattern from similar attack traces. An attack pattern consists of a vulnerability event and a trace pattern for the event. The vulnerability event is the result of the attack, and the trace pattern is a list of minimal protocol traces which lead to the attack. An attack pattern is a general form of many attacks of the same kind. We argue that the attack pattern is more suitable for the protocol analysis than a detailed attack trace.

We develop an automated attack classification method to classify a huge amount of attack traces found by using attack patterns. First, given a current set of attack traces to be classified, we create a new attack pattern by taking the first attack trace in the set, and add the new pattern into a current set of known patterns. Then, by using a current set of known attack patterns, we filter out the known attack patterns from a current set of attack traces to be classified. And the process repeats again until there is no output set of attack traces to be classified.

4.2.6. New Attacks

The table 1 shows the number of attacks states (and attack traces) and attack patterns found in each configuration. In table 1, *(A,B) (In,In)* means the first configuration of two concurrent session where A and B are in the first session, and *In(A)* and *In(B)* are in the second session. Also, *Tr* and *Pat* means the number of attack traces and attack patterns, respectively. Note that our method does not find attack states for the events 1 and 3. Also, all attack traces found from the second, third and fourth configurations are just some variant forms of those attack traces from the first configuration. Therefore, in the following we discuss only the attacks from the first configuration.

Configurations	**Event 2**		**Event 4**		**Event 5**	
	Tr	**Pat**	**Tr**	**Pat**	**Tr**	**Pat**
1. (A,B) (In,In)	360	10	360	10	360	10
2. (A,In) (In,B)	144	4	144	4	144	4
3. (In,B) (A,In)	72	2	72	2	72	2
4. (A,B) (In,In)	36	1	36	1	0	0

Table 1: Number of Attack Traces and Patterns

By using our new CPN method, we found two new attacks of the TMN protocol for multiple sessions of protocol execution. The first and the second new attack is in the category of the combined vulnerability event 4 and 5, respectively. Also, we found many interesting variant forms of Lowe and Roscoe's attack.

The first new attack is the combined vulnerability event 4 where the attacker learns the session key and A's secret, and fools A to commit to a fake key which is the attacker's secret K_i. The event 4 represented by $[K_{ab},K_{aj}][K_i][K_{ab}]$ leads to a kind of the man-in-the-middle attacks in that the attacker uses K_i to learn the communication from A and use K_{ab} to create a fake communication to B. As a result, the attacker learns the communication between A and B, but both A and B think that they communicate to each other. In the event 4, we found 10 attack patterns. Each attack pattern represents a group of similar attacks with a slight and unimportant difference. Due to the space limit, we discuss only some attack patterns only. A full detail of the attacks in TMN is described in [21]. The following shows the first attack pattern.

1) $A \rightarrow In(J) : (B, \{K_{aj}\}PK\text{-}J), A$
   $In(J) \rightarrow J : (X2, \{K_i\}PK\text{-}J), X1$
1') $In(A) \rightarrow J : (X4, \{K_i\}PK\text{-}J), X3$
2') $J \rightarrow In(B) : X3$
2) $J \rightarrow In(B) : X1$
   $In(B) \rightarrow B : A$
3) $B \rightarrow J : (X1, \{K_{ab}\}PK\text{-}J), X2$
3') $In(B) \rightarrow J : (X3, \{K_{aj}\}PK\text{-}J), X4$
4') $J \rightarrow In(A) : X4, E_{Ki}(K_{aj})$
4) $J \rightarrow In(A) : X2, E_{Ki}(K_{ab})$
   $In(A) \rightarrow A : B, E_{Kaj}(K_i)$

where K_i is attacker's secret keys.

X1, X2, X3 and X4 stand for arbitrary identities that the attacker creates and uses in the messages during the attack. In step 1), the message that A sends to J is modified by the attacker. The original message is indicated by $A \rightarrow In(J)$, but the modified message by the attacker is indicated by $In(J) \rightarrow J$. Also, the messages at steps 2) and 4) are modified by the attacker.

In this attack pattern, the first message at the first session is modified to $\{K_i\}PK\text{-}J$, but it is replayed at the third step in the second session. Therefore, the attacker learns both K_{ab} and K_{aj}, and the attacker sends $E_{Kaj}(K_i)$ at the fourth step in the first session.

The goal of the first attack is to learn keys K_{aj} and K_{ab} before the message at the last step is sent to A. And at the last step, $E_{Kaj}(K_i)$ is sent instead to fool A to commit to K_i. The attack can be understood by considering the keys that are encrypted by J's public keys at steps 1 and 3 in both sessions. Based on those keys, the message at step 4 contains the key at step 3 which is encrypted by key at step 1. And if the key at step 1 is known, then so is the key at step 3.

In the first session, K_i and K_{ab} are used in steps 1 and 3, respectively, and in the second session, K_i and K_{aj} are used in steps 1 and 3, respectively. These can be represented by the notation $(<K_i,K_{ab}>,<K_i,K_{aj}>)$ where (P_1,P_2) means that P_1 and P_2 are information for the first and second sessions, respectively. Thus, the ciphertexts obtained at the step 4 in the first and second sessions are $E_{Ki}(K_{ab})$ and $E_{Ki}(K_{aj})$, respectively. So, the attacker obtains the target keys easily.

The second attack pattern which can be represented by $(<K_i,K_{aj}>,<K_i,K_{ab}>)$ is similar to the first attack pattern. The difference is on steps 3), 4), 3') and 4'). The different steps in the second attack pattern are shown as follows.

3) $B \rightarrow In(J) : (X1, \{K_{ab}\}PK\text{-}J), X2$
   $In(J) \rightarrow J : (X1, \{K_{aj}\}PK\text{-}J), X2$
3') $In(B) \rightarrow J : (X3, \{K_{ab}\}PK\text{-}J), X4$
4') $J \rightarrow In(A) : X4, E_{Ki}(K_{ab})$
4) $J \rightarrow In(A) : X2, E_{Ki}(K_{aj})$
   $In(A) \rightarrow A : B, E_{Kaj}(K_i)$

The third attack pattern which can be represented by $(<K_i,K_{ab}>,<K_{aj},K_i>)$ is shown as follows.

1) $A \rightarrow In(J) : (B, \{K_{aj}\}PK\text{-}J), A$
   $In(J) \rightarrow J : (X2, \{K_i\}PK\text{-}J), X1$
1') $In(A) \rightarrow J : (X4, \{K_{aj}\}PK\text{-}J), X3$
2') $J \rightarrow In(B) : X3$
2) $J \rightarrow In(B) : X1$
   $In(B) \rightarrow B : A$
3) $B \rightarrow J : (X1, \{K_{ab}\}PK\text{-}J), X2$
3') $In(B) \rightarrow J : (X3, \{K_i\}PK\text{-}J), X4$
4') $J \rightarrow In(A) : X4, E_{Kaj}(K_i)$
4) $J \rightarrow In(A) : X2, E_{Ki}(K_{ab})$
   $In(A) \rightarrow A : B, E_{Kaj}(K_i)$

The seven remaining attack patterns are $(<K_i,K_{ab}>,<K_{aj},K_{ab}>)$, $(<K_i,K_{aj}>,<K_{aj},K_{ab}>)$, $(<K_{aj},K_i>,<K_i,K_{ab}>)$, $(<K_{aj},K_{ab}>,<K_i,K_{ab}>)$, $(<K_{aj},K_{ab}>,<K_i,K_{aj}>)$, $(<K_{aj},K_{ab}>,<K_{aj},K_i>)$ and $(<K_{aj},K_i>,<K_{aj},K_{ab}>)$.

The second new attack is the combined vulnerability event 5 represented by $[K_{ab},K_{aj}][K_{aj}][K_{ab}]$. The event 5 leads to the similar kind of the man-in-the-middle attacks to the event 4, but here the attacker uses K_{aj} to learn the communication from A. As a result, the attacker learns the communication between A and B, but both A and B think that they communicate to each other. In the event 5, we found 10 attack patterns. These attack patterns are similar to those patterns for the event 4 except that the last step of the first session is change to the following.

4) $J \rightarrow In(A) : X2,\ldots$
   $In(A) \rightarrow A : B, E_{Kaj}(K_{aj})$

Conceptually, the difference between the first new attack and the second new attack is that in the former, $E_{Kaj}(K_i)$ is sent to A in the last step, but in the latter, $E_{Kaj}(K_{aj})$ is sent to A in the last step. So, user A is fooled to commit on his own secret.

Finally, we also found many interesting variant forms of Lowe and Roscoe's attack where the server J cannot detect the replay attack in the two concurrent sessions. Note that in [29], Zhang and Liu detected only one variant form of this kind, and so they can detect less number of attacks than our approach. Due to space limit, the details of these variants are omitted here.

4.2.7. Performance

The table 2 shows the computation time and the size of a state space from each configuration. In our experiment, the computation of the state space is executed on a PC with Intel Core2 Duo 2.33 Ghz with 2 GB of RAM.

Configurations	**Nodes**	***Arcs***	***Time (sec.)***
1. (A,B) (In,In)	104,346	109,476	976
2. (A,In) (In,B)	73,806	77,568	523
3. (In,B) (A,In)	51,212	52,639	282
4. (A,B) (In,In)	34,160	35,095	120

Table 2. Size and Time of the generated state spaces

At most, it takes 16 minutes to compute a state space to analyze the attack.

4.2.8. Discussion

We verify the completeness of our method by hands and found that the attack patterns discovered for the events 4 and 5 are complete with respect to the assumptions stated in section 4.2.1. Our argument is as follows. Consider the attack patterns in event 4 only. Let P be $(<\{K_i,K_{aj}\}, \{K_{ab},K_{aj},K_i\}>, <\{K_i,K_{aj}\}, \{K_{ab},K_{aj},K_i\}>)$ which uses a similar notation to the attack pattern representation. P means that in the first session, only the ciphertexts of K_i and K_{aj} and the ciphertexts of K_{ab}, K_{aj} and K_i can be sent or replayed at step 1 and at step 3, respectively, and similarly in the second session. Clearly, these are true. And it is easy to verify that all of our attack patterns are just all enumerations from P which lead to the attacks. Note that each enumeration is constructed by selecting each possible key to produce a ciphertext in each session.

Exploring all possible alternating executions within a state space is expensive and causes a huge state space. In our experiment, it takes more than 6 days to compute such a state space. Moreover, some alternating executions may be unnecessary but redundant. For example, schedule *[2,2,2,1,1,1,2,1]* is similar to schedule *[2,2,2,2,1,1,1,1]* regarding to attacks found since in the former, the information obtained from the three steps in session 1 is not useful for any replay attack on session 2. Note that step 4 contains a shared-key ciphertext, but steps 1 and 3 contain public-key ciphertexts. Similarly, schedules *[1,1,2,2,1,2,1,2]* is similar to *[1,2,1,2,1,2,1,2]* since step 2 in session 1 contains only plaintext which is already known by an attacker, and thus no information at step 2 in session 1 is useful for any replay attack on session 2.

4.3. Our CPN Analysis for Micali's ECS1 Protocol

In this section, we discuss the analysis of ECS1 by using our CPN method. In fact, our CPN framework for the analysis of ECS1 is similar to the framework for TMN discussed previously. So, we discuss only the difference between them. Also, some new attacks on ECS1 will be discussed in section 4.3.2.

4.3.1. Our CPN framework for Micali's ECS1 Protocol

The assumptions of the protocol execution for ECS1 are similar to those assumptions in definition 1 except for assumption 5. For ECS1, the same initiator and responder may participate in more than one session.

We assume two kinds of attackers which are *I* and *Ar*. The attacker *I* is exactly the same as the attacker *In* discussed in section 4.2.1. However, *Ar* is a different kind of attackers, and it is considered as a conspired user with attacker *I*. In fact, *Ar* is a participating user in the system who has all the abilities discussed in definition 2, except for the assumption 1. An attacker who has the ability in the assumption 1 can behave as an external observer who tries to manipulate messages between users in a session. Note that the external observer is not a user in the session. But, *Ar* is a malicious user who participates in a session and conspires with an attacker by sharing some information. These two attackers collaborate to cheat a user. Note that one attack found by Bao et. al. [28] involves these two kinds of attackers.

Attack states in ECS1 are characterized by vulnerability events. There is one vulnerability event in ECS1 protocol which is unfair states. An unfair state means that one party, who is either initiator or responder, gets another party commitment, but the latter does not get the former commitment. In fact, there are two kinds of unfair states where the initiator has an advantage and the responder has an advantage, respectively.

Definition 9 : The vulnerability events in ECS1

There are two unfair states.

1) The initiator has the responder's commitment, but the responder does not have the initiator's commitment. *(AgainAdv)*
2) The responder has the initiator's commitment, but the initiator does not have the responder's commitment. (*BgainAdv*)

First, we compute two basic vulnerability events where the responder has initiator's commitment (*ACommit*) and the initiator has responder's commitment (*BCommit*). Then, fair states (*BothCommit*) are computed by ($ACommit \cap BCommit$). Finally, the two main vulnerability events

can be computed by *AgainAdv = (BCommit – BothCommit)* and *BgainAdv = (ACommit – BothCommit)*.

The queries to compute *ACommit* and *BCommit* are straightforward. For example, to obtain *ACommit*, we write a query to find a set of all terminal nodes in the state space where initiator's commitment, which are $SIG_A(C,Z)$ and M, are present at responder's database. Also other queries can be created easily.

4.3.2. New attacks

We found one new single-session attack of Micali's ECS1, two new multi-session attacks in Micali's ECS1 and three new attacks of Bao's modified version of ECS1. Due to the space limit, we discuss only some new attacks in multiple sessions.

In the following, (1), (2), (3), … describe protocol steps in the first session but (1'), (2'), (3'), … describe steps in the second session.

We describe two new attacks in the original ECS1 protocol. In the first attack, a malicious responder gains an advantage over a well-behaved initiator if random *M* is reused in the two sessions. In the second attack, an initiator gains an advantage over a well-behaved responder if the same contract is signed in the two sessions.

In the first attack, a well-behaved initiator A communicates with the same malicious responder *I* in two sessions. The same random *M* is used in both sessions but different contracts are used in the two sessions. The attacker (*I*) simply ignores to participate in one session. Since the random M from another session can be used to construct the same Z, attacker *I* has enough information to show that A has committed to *I* in both sessions. The attack is shown as follows.

1) $A \rightarrow I : SIG_A(C1,Z1)$ where $Z1=ENC_{TTP}(A,I,ma1)$
1') $A \rightarrow I : SIG_A(C2,Z1)$ where $Z2=ENC_{TTP}(A,I,ma2)$
2') $I \rightarrow A$: Nothing
2) $I \rightarrow A : SIG_I(C1,Z1), SIG_I(Z1)$
3) $A \rightarrow I$: ma1

Note that step 2') means that I does not send any message to A, and thus the second session is aborted.

In the second attack, a well-behaved initiator communicates with a malicious responder *I* in one session but communicates with a well-behaved responder B in another session. The same contract is used in the two sessions. The malicious responder *I* replays A's signature for session with *I* to the session with B. Due to the duplicate contract, A has B's commitment. However, B does not have A's commitment since A's signature is replayed, and thus Z is not correct. Note that TTP cannot resolve the dispute due to the incorrect Z. The attack is shown as follows.

1) *$A \rightarrow I : SIG_A(C1,Z1)$ where $Z1=ENC_{TTP}(A,I,ma1)$*
1') $A \rightarrow I(B) : SIG_A(C1,Z2)$ where $Z2=ENC_{TTP}(A,B,ma2)$
$I(A) \rightarrow B : SIG_A(C1,Z1)$
2') $B \rightarrow A : SIG_B(C1,Z1), SIG_B(Z1)$
2) $I \rightarrow A$: Nothing
3') $A \rightarrow B$: ma2
4') $B \rightarrow TTP$: A, B, $SIG_B(C1,Z1), SIG_B(Z1)$
5a') $TTP \rightarrow A$: Error
5b') $TTP \rightarrow B$: Error

Since *I* does not send any message at step 2) in the first session, the session is ignored. In steps 5a') and 5b'), an error occurs due to the incorrect Z. Thus, no message is sent from TTP according to Micali's ECS1 protocol.

We discuss one new attack in Bao's modified ECS1 protocol only. In the attack, a well-behaved initiator communicates with malicious responder *I* in one session, but the malicious responder conspires with *Ar* in another session. As a result of the attack, a malicious responder gets a well-behaved initiator's commitment, but the initiator obtains *Ar*'s commitment instead. After *I*

receives A's signature in step 1, *I* forward A's signature to *Ar* and *Ar* requests TTP to resolve a dispute by replaying *Ar*'s signature. Thus, A will obtain *Ar*'s commitment instead, but *I* will obtain A's commitment. *I* gets A's commitment because TTP is fooled to issue random M to *I* by the help of *Ar*. The attack is shown as follows.

1) $A \rightarrow I : SIG_A(C1,Z1)$ where $Z1 = ENC_{TTP}(A,I,ma1)$
In the second session, I sends $SIG_A(C1,Z1)$ to Ar secretly.
2) $I \rightarrow A$: Nothing
4') $Ar \rightarrow TTP : A, Ar, SIG_{Ar}(C1,Z1), SIG_{Ar}(Z1)$
5') $TTP \rightarrow A : SIG_{Ar}(C1,Z1), SIG_{Ar}(Z1)$
6') $TTP \rightarrow Ar$: ma1
In the second session, Ar sends ma1 to I secretly.

4.3.3. Performance

The table 3 shows some examples of configurations and their computation results. Each row of the table shows a simplified form of a configuration of a session. For example, *(A,I,c1,ma1)* means that in the session *A* and *I* are initiator and responder, respectively, and *c1* is a contract and *ma1* is the random M. Note that "_" means any value. Also, "All States" means the number of all states in a state space and "Attack states" means the number of attack states found in the state space. "Times" means the amount of times in seconds that are used to generate each state space.

Configurations of two sessions	All States	Times (in seconds)	Attack States
(Ar,I,_,mi1)&(I,B,c1,mi1)	146,318	3,163	1,900
(Ar,I,_,mi1)&(A,B,c1,ma1)	78,594	1,806	2,048
(I,Ar,_,mi1)&(A,I,c1,ma1)	91,538	2,093	330
(I,Ar,_,mi1)&(I,B,c1,mi1)	339,918	13,200	2,952
(I,Ar,_,mi1)&(I,B,c1,mi2)	647,918	34,370	7,032
(I,Ar,_,mi1)&(A,B,c1,m1)	51,434	1,188	1,104
(A,I,c1,ma1)&(A,I,c2,ma1)	2,021	25	28
(A,I,c1,ma1)&(A,I,c2,ma2)	5,689	77	0
(A,I,c1,ma1)&(I,B,c1,mi1)	7,004	85	108

Table 3: Some results of the state space

Note that at some configuration, it requires 9 hours to compute the state space.

4.4. Discussion

Even though our CPN method offers many advantages, there is some disadvantage also. The size of each state in the computed state space is large since an attack trace is embedded into each state. As a future work, we aim to improve our method to overcome this disadvantage. Also, we aim to apply our new method to analyze other kinds of cryptographic protocols and to analyze other kinds of attacks.

5. Conclusion

In this paper, we propose a new CPN methodology for the security analysis of cryptographic protocols. Our approach offers a simple but effective way to analyze multiple sessions of protocol execution. We argue that our new CPN methodology improves on all existing CPN and Petri net methods for security protocols on several issues. In particular, our CPN method is the first CPN method which offers a security analysis methodology of multiple concurrent sessions of protocol execution. Furthermore, it offers a systematic method to analyze attacks in protocols. In particular, our method offers the decomposition and multi-session scheduling for the computation of state spaces, the intuitive approach to characterize attack states, the efficient way to extract attack traces and the systematic way to classify a large amount of attack traces. Also, it can detect more attacks with a better efficiency than all existing CPN methods for security protocols.

To demonstrate the practical uses of our approach, we apply our methodology to two case studies which are Micali's contract signing protocol ECS1 and TMN authenticated key exchange protocol. Surprisingly, we found many attacks in the two protocols. For ECS1, we found two new attacks in multiple sessions of protocol execution of the original ECS1, and three new attacks in Bao's modified ECS1. For TMN protocol, we found two new attacks in multiple sessions of protocol execution. In fact, the new attacks that we found in TMN protocol are quite surprisingly since TMN have been analyzed quite extensively.

6. Acknowledgement

We would like to thanks anonymous reviewers for their helpful and constructive comments. The first author would like to acknowledge a financial support from National Research Council of Thailand.

7. References

[1] J. Clark and J. Jacob, A survey on Authentication Protocols, *Research report*, University of York, 1997. (http://www.cs.york.ac.uk/~jac/papers/drareview.ps.gz)

[2] C. Meadows, Formal Verification of Cryptographic Protocols: A Survey, Advances in Cryptology - Asiacrypt '94, LNSC 917, Springer-Verlag, 1995

[3] C. Meadows, Formal Methods for Cryptographic Protocol Analysis: Emerging Issues and Trends, *IEEE Journal on Selected Areas in Communications*, 21(1), pp. 44-54, 2003.

[4] Ulrike Meyer, Susanne Wetzel, A man-in-the-middle attack on UMTS, In Proceedings of the 3rd ACM workshop on Wireless security, 2004, pp. 90-97

[5] Iliano Cervesato, Aaron D. Jaggard, Andre Scedrov, Joe-Kai Tsay, Christopher Walstad: Breaking and fixing public-key Kerberos. *Information and Computation*, 206(2-4): 402-424, 2008

[6] Paul F. Syverson, A Taxonomy of Replay Attacks, In proceedings of the 7th IEEE Computer Security Foundations Workshop (CSFW), pp. 187-191, 1994.

[7] T. Murata, Petri nets: properties, analysis and applications, Proceedings of the IEE, 77(4):541-580, 1989.

[8] C. Morton, L. Robart and S. Tavares, Decomposition Techniques for Cryptographic Protocol Analysis, *Proceedings of the Canadian Conference on Electrical and Computer Engineering*, Canada, 1994.

[9] A. Basyouni and S. Tavares, New Approach to Cryptographic Protocol Analysis using Coloured Petri Nets, *Proceedings of the Canadian Conference on Electrical and Computer Engineering*, Canada, 1997.

[10] G. Lee and J.Lee, Petri Net Based Models for Specification and Analysis of Cryptographic Protocols, *The Journal of Systems and Software*, 37:141-159, 1997.

[11] S. Lim, J. Ko, E. Jun and G. Lee, Specification and analysis of n-way key recovery system by Extended Cryptographic Timed Petri Net, *The Journal of Systems and Software*, 58:93-106, 2001.

[12] Suratose Tritilanunt, Colin Boyd, Ernest Foo, Juan Manuel González Nieto: Using Coloured Petri Nets to Simulate DoS-resistant Protocols. In *proceedings of Seventh Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools*, pages 261-280. University of Aarhus, Denmark, October 2006..

[13] W. Dresp, Security Analysis of the Secure Authentication Protocol by Means of Coloured Petri Nets, *Proceeding of 9th IFIP Communications and Multimedia Security*, 2005.

[14] Al-Azzoni, I., Down, D.G., and Khedri, R., "Modeling and Verification of Cryptographic Protocols Using Coloured Petri Nets and Design/CPN", *Nordic Journal of Computing*, 12(3): 201-228, 2005.

[15] F. Crazzolara and G. Winskel, Events in Security Protocols, *Proceedings of the 8th ACM Conference on Computer and Communication Security*, USA, 2001.

[16] K. Jensen, Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use, Vol.1. Monographs in Theoretical Computer Science, Springer-Verlag, 1997

[17] K. Jensen, L.M. Kristensen, and L. Wells, "Coloured Petri Nets and CPN Tools for Modelling and Validation of Concurrent Systems", *International Journal on Software Tools for Technology Transfer*, 9(3): 213-254, 2007.

[18] Y. Permpoontanalarp and W. Rujiwattanaphong, "An Improved Method to Analyze Cryptographic Protocols by using Coloured Petri Nets", *Proceedings of 1st Joint International Conference on Information Communication Technology*, Laos, 2007.

[19] P. Sornkhom and Y. Permpoontanalarp, Security Analysis of Micali's Fair Contract Signing Protocol by Using Coloured Petri Nets, *Proceedings of the 9th ACIS International Conference on Software*

Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, Thailand, IEEE press, 2008.

[20] P. Sornkhom and Y. Permpoontanalarp, Security Analysis of Micali's Fair Contract Signing Protocol by Using Coloured Petri Nets : Multi-session case, *In Proceedings of the 5th International Workshop on Security in Systems and Networks*, Italy, IEEE press, 2009.

[21] Y. Permpoontanalarp, Security Analysis of the TMN protocol by using Coloured Petri Nets, Technical Report, King Mongkut's University of Technology Thonburi, Bangkok, Thailand, 2009.

[22] S. Micali, "Simple and Fast Optimistic Protocols for Fair Electronics Exchange", *Proceedings of 21st Symposium on Principles of Distributed Computing.* USA, pp. 12-19, 2003.

[23] M. Tatebayashi, N. Matsuzaki, and D. Newman, Key Distribution Protocol for Digital Mobile Communication Systems, *In CRYPTO-89*, Springer-Verlag, 1990.

[24] R. Kemmerer, C. Meadows and J. Millen, Three systems for cryptographic protocol analysis. *Journal of Cryptology*, 7(2), 1994.

[25] J. Mitchell, M. Mitchell and U. Stern, Automated analysis of cryptographic protocols using Murϕ, *In IEEE Symposium on Security and privacy*, 1997.

[26] G. Lowe and B. Roscoe, Using CSP to Detect Errors in the TMN Protocol, *IEEE Transactions on Software Engineering*, 23(10), 1997.

[27] Y. Zhang and X. Liu, An approach to the formal analysis of TMN protocol, *Progress on Cryptography: 25 years of Cryptography in China*, Springer-Verlag, 2004.

[28] F. Bao, G. Wang, J. Zhou, and Z. Zhu, "Analysis and Improvement of Micali's Fair Contract Signing Protocol", *Proceedings of The 9th Australasian Conference on Information Security and Privacy*. Australia, 2004: 176-187.

[29] Yuqing Zhang, Zhiling Wang, Bo Yang, The Running-Mode Analysis of Two-Party Optimistic Fair Exchange Protocols, *International Conference on Computational Intelligence and Security*, Springer Verlag, 2005.

[30] Gavin Lowe, An Attack on the Needham-Schroeder Public-Key Authentication Protocol, *Information Processing Letters*, 56(3): 131-133 (1995)

[31] Thomas Y. C. Woo, Simon S. Lam: A Lesson on Authentication Protocol Design. *Operating Systems Review* 28(3): 24-37 (1994)

[32] R. Chadha, M. Konovich and A. Scedrov, Inductive Methods and Contract-Signing Protocols, *Proceedings of 8th ACM Conference on Computer and Communications Security*, 2001.

[33] V. Shmatikov and J.C. Mitchell, Finite-State Analysis of Two Contract Signing Protocols, *Theoretical Computer Science*, 283:419-450, June 2002.

[34] S. Gürgens and C. Rudolph, Security analysis of efficient (Un-)fair non-repudiation protocols, *Formal Aspect of Computing*, 17(3): 260-276, 2005.

[35] D. Dolev, A. Yao, On the security of public key protocols, *IEEE Transactions on Information Theory*, 29(2): 198-207, 1983.

A primer on the Petri Net Markup Language and ISO/IEC 15909-2

L.M. Hillah, UPMC & CNRS UMR 7606, Paris
E. Kindler, Technical University of Denmark (DTU), Lyngby
F. Kordon, UPMC & CNRS UMR 7606, Paris
L. Petrucci, University Paris 13 & CNRS UMR 7030, Villetaneuse
N. Trèves, CNAM & Cedric, Paris

1 Introduction

In 2000, there was a workshop [1] that should foster the definition of a standard transfer format for *Petri nets* as a satellite event of the annual 'Petri Net Conference' in Aarhus. As a result of this first workshop, after many other discussions and meetings, the *Petri Net Markup Language* (*PNML*) is about to be finally adopted as ISO/IEC 15909-2. Over the years, PNML has evolved and, unfortunately, there are many different intermediate versions and variants, that are still in use. With this paper, we would like to report on the final result and on PNML as it is defined in ISO/IEC 15909-2. This way, we hope to unify the different lines of PNML and advertise the use of ISO/IEC 15909-2.

Note that this paper is not a copy or exact reproduction of ISO/IEC 15909-2 (which, including all Annexes, has more than 100 pages). Rather it is a restructured excerpt that focuses on the most important issues and abstracts from some technical details, which can be found in ISO/IEC 15909-2. Most of the technical details can be derived from the RELAX NG grammars provided at the PNML web pages [17]. Together, this should provide a fair account of the standard, its ideas and concepts, and its practical use. For a in-depth discussion of the rationales and design decisions behind PNML, we refer to the bunch of earlier publications [2, 15, 4, 18, 19]

Though not an exact copy of ISO/IEC 15909-2, this paper reuses material of ISO/IEC 15909-2 with some modifications and simplifications with the kind permission of ISO/IEC, Geneva.

Originally, PNML was introduced as an interchange format for all kinds of Petri nets [2, 3, 4]. Some major concepts of PNML were driven by this objective. Technically, ISO/IEC 15909-2 defines a transfer syntax for *High-level Petri Net Graphs* and those subclasses of *Petri nets* only that have been conceptually and mathematically defined in the International Standard ISO/IEC 15909-1 [5], for capturing the essence of all kinds of coloured and high-level Petri nets [6, 7, 8, 9, 10, 11, 12, 13]. In this paper, the focus is on PNML for high-level nets in order not to mix up concepts that are part of ISO/IEC 15909-2 and some extensions, which are currently under consideration for ISO/IEC 15909-3. In the conclusion (Sect. 5), we will briefly discuss some of these perspectives, which make PNML applicable for all kinds of *Petri nets*.

2 Concepts

In this section, we discuss the main concepts of PNML, were the main idea is that any kind of *Petri net* can be considered to be a labelled graph. In particular, all information that is specific to a particular kind of *Petri net* can be captured in *labels*.

This will be discussed in more detail in Sect. 2.1. The concepts of PNML will be defined by a meta-model for PNML in terms of UML class diagrams. These meta-models, however, do not define the XML syntax for the transfer format. Therefore, there is a mapping from the concepts of the PNML meta-model to XML, which will be discussed in Sect. 4. Curious readers, who are interested in looking at the actual XML right away, might have a look at this and, in particular, at the example in Listing 1.

As mentioned above, ISO/IEC 15909-2 defines transfer formats for different versions of *Petri nets*: *Place/Transition Nets*, *High-level Petri Net Graphs* (*HLPNG*), and *Symmetric Nets* as defined in ISO/IEC

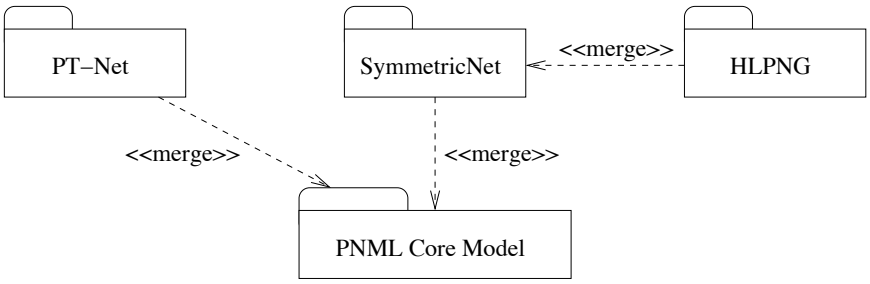


Figure 1: Overview of the UML packages of PNML

15909-1, where *Symmetric Nets* are a subclass of high-level nets, which originally were introduced under the name *well-formed nets* [14]. The main differences between them are the available data types, which will be discussed in Sect. 3.

2.1 General Principles

In order to deal with different kinds of *Petri nets*, the transfer format needs to be flexible and extensible. In order to obtain this flexibility, a *Petri net* is considered to be a labelled directed graph, where all type specific information of the net is represented in *labels*. A *label* may be associated with a *node*, an *arc*, or the *net* itself. This basic structure of a *PNML Document* is defined in the *PNML Core Model* using a UML class diagram. This will be discussed in Sect. 2.2.

The *PNML Core Model* imposes no restrictions on *labels*. Therefore, the *PNML Core Model* can represent any kind of *Petri net*. Due to this generality of the *PNML Core Model*, there can be even *PNML Documents* that do not correspond to a *Petri net* at all. For example, there could be *labels* from two different and even incompatible versions of *Petri nets* within the same *PNML Document*. For a concrete version of *Petri nets*, the legal *labels* will be defined by extending the *PNML Core Model* with another meta-model that exactly defines the legal *labels* of this particular type.

Technically, the *PNML Core Model* is a UML package, and there are additional UML packages for the different *Petri net types* that extend the *PNML Core Model* package. ISO/IEC 15909-2 defines a package for *Place/Transition Nets*, a package for *Symmetric Nets*, and a package for *High-level Petri Net Graphs*, where the package for *High-level Petri Net Graphs* extends the package for *Symmetric Nets*. Therefore, every *Symmetric Net* is also a *High-level Petri Net Graph*.

Figure 1 gives an overview of the different packages defined and on their dependencies. The package *PNML Core Model* defines the basic structure of *Petri nets*; this structure is extended by the package for each type. The *PNML Core Model* is discussed in Sect. 2.2 and the package PT-Net is discussed in Sect. 2.3.1. The general concepts of *High-level Petri Net Graphs* will be discussed in Sect. 2.3.2. The different data types used in the different versions of *high-level Petri nets* are discussed in Sect. 3.1. Based on these data types, the package SymmetricNet is discussed in Sect. 3.3.2 and the package HLPNG for general *High-level Petri Net Graph* is defined in Sect. 3.3.3.

2.2 The PNML Core Model

Figures 2 and 3 show the meta-model of the *PNML Core Model* as a UML class diagram. The diagram of Fig. 2 focusses on the conceptual parts, whereas the diagram of Fig. 3 focusses on the parts concerning the graphical representation (i. e. graphics). Note that the data type *String* is imported from a separate package *XMLSchemaDataTypes*, which is discussed in Sect. 2.2.6; since this is a technicality only, the references to this package are shown in tiny fonts in the diagram. The concepts of the *PNML Core Model* are discussed below.

2.2.1 Petri Net Documents, Petri Nets, and Objects

A document that meets the requirements of the *PNML* Core Model is called a *Petri Net Document* (PetriNetDoc) or a *PNML Document*. It contains one or more *Petri Nets* (PetriNet). Each *Petri Net* has a unique identifier and a *type*. The *type* is a name uniquely identifying a *Petri net type definition*;

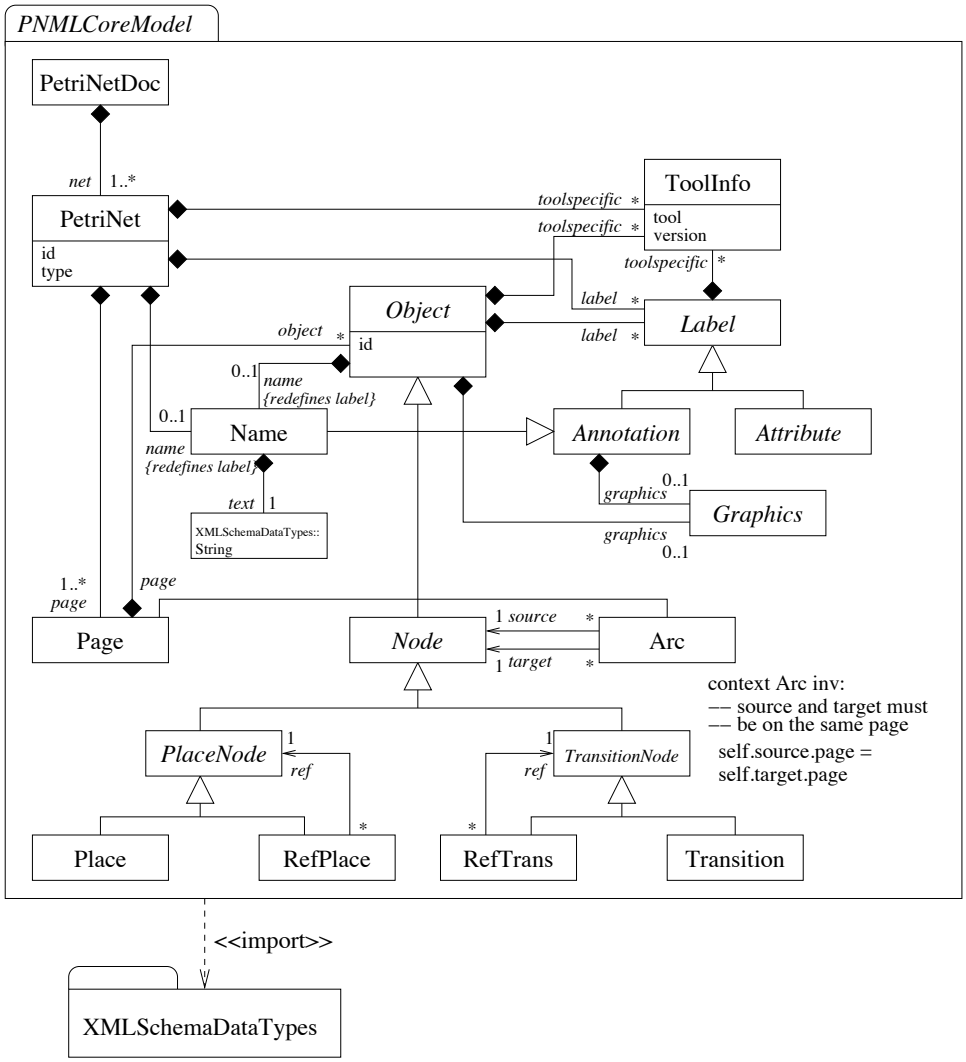


Figure 2: The *PNML Core Model* package: concepts

an example for such a name is `http://www.pnml.org/version-2009/grammar/ptnet` for the definition of *Place/Transition Nets*.

A *Petri net* consists of one or more *pages* that in turn consist of several *objects*. These *objects*, basically, represent the graph structure of the *Petri net*. Each *object* within a *Petri net document* has a unique *identifier*, which can be used for referring to this *object*. Moreover, each *object* may be equipped with graphical information defining its position, size, colour, shape and other attributes on its graphical appearance (*graphics*). The precise graphical information that can be provided for an *object* depends on the particular type of the *object* (see Sect. 2.2.4 for more details).

The most important *objects* of a *Petri net* are *places*, *transitions*, and *arcs*. For extensibility reasons and principles of good design, *places* and *transitions* are generalised to *nodes*. For reasons explained in Sect. 2.2.2, this generalisation is via *place nodes* and via *transition nodes*. *Nodes* of a *Petri net* can be connected by *arcs*.

Note that it is legal to have an *arc* from a *place* to a *place* or from a *transition* to a *transition* according to the *PNML Core Model*. The reason is that there are versions of *Petri nets* that support such *arcs*. If a *Petri net type* does not support such *arcs*, this restriction will be defined in the particular package defining this type.

2.2.2 Pages and Reference Nodes

Three other kinds of *objects* are used for structuring a *Petri net*: *pages*, *reference places*, and *reference transitions*. As mentioned above, a *page* may contain other *objects*; since a *page* is an *object* itself, a *page* may even contain other *pages*, thus defining a hierarchy of subpages.

Note that PNML requires that an *arc* must connect *nodes* on the same *page* only. The reason for this requirement is that *arcs* connecting *nodes* on different *pages* cannot be drawn graphically. In the *PNML Core Model* of Fig. 2, this requirement is captured by the OCL expression next to the class for *arcs*.

In order to connect *nodes* on different *pages* by an *arc*, a representative of one of the two *nodes* must be drawn on the same *page* as the other *node*. Then, this representative may be connected with the other *node* by an *arc*. This representative is called a *reference node*, because it has a reference to the *node* it represents. Note that a *reference place* must refer to a *place* or a *reference place*, and a *reference transition* must refer to a *transition* or a *reference transition*. Moreover, cyclic references among *reference nodes* are not allowed.

2.2.3 Labels

In order to assign further meaning to an *object*, each *object* may have *labels*. Typically, there are *labels* representing the name of a *node*, the initial marking of a *place*, the *transition condition*, or some *arc annotation*. In addition, the *Petri net* itself or its *pages* may have some *labels*, which are called *global labels*. For example, the package HLPNG defines *declarations* as *global labels* of a *High-level Petri Net*, which are used for defining *variables*, and user-defined *sorts* and *operators*.

In the *PNML Core Model*, we distinguish two kinds of *labels*: *annotations* and *attributes*. An *annotation* comprises information that is typically displayed as text next to the corresponding *object*. Examples of *annotations* are names, initial markings, arc annotations, transition conditions, and timing or stochastic information. In contrast, an *attribute* is, typically, not displayed as text next to the corresponding *object*. Rather, an *attribute* has an effect on the shape or colour of the corresponding *object*. For example, an *attribute* such as arc type could have domain {`normal`, `read`, `inhibitor`, `reset`}. ISO/IEC 15909-2, however, does not mandate the effect on the graphical appearance of an *attribute*.

Note that the classes for *label*, *annotation* and *attribute* are abstract in the *PNML Core Model*, which means that the *PNML Core Model* does not define concrete *labels*, *annotations*, and *attributes*. The only concrete *label* defined in the *PNML Core Model* is the *name*, which is a *label* that can be used for any *object* within any *Petri net type*. This way, any *object* such as *nodes*, *pages*, the *net* itself, and even *arcs* can have a *name*. The value of a *name* is a *String*, which is imported from the separate package *XMLSchemaDataTypes* (see Sect. 2.2.6 for more information). All other concrete *labels* are defined in the packages for the concrete *Petri net types* (see Sect. 2.3).

In order to support the exchange of information among tools that have different textual representation for the same concepts (i. e. when they have different concrete syntax), there are two ways for representing the information within an annotation: *textually* in some concrete syntax and *structurally* as an abstract syntax tree (see Sect. 2.3.2 and 4.1.2 for details).

Note that *reference nodes* may have *labels*, but these *labels* do not have any meaning. This choice was made in order to obtain a semantically equivalent *Petri net* without *pages* by merging every *reference node* to the *node* it directly or indirectly refers to. This is called *flattening* of the *Petri net* (see [15] for details). Still, the labels of a *reference node* can have an effect on the graphical appearance or can give some additional information to the user.

2.2.4 Graphical Information

In addition to the *Petri net* concepts, information concerning the graphical appearance can be associated with each *object* and each *annotation*. For a *node*, this information includes its position; for an *arc*, it includes a list of positions that define intermediate points of the *arc*; for an *object's annotation*, it includes its relative position with respect to the corresponding *object*; for an *annotation* of a *page*, the position is absolute. There can be further information concerning the size, colour and shape of nodes or arcs, or concerning the colour, font and font size of labels. Note that this information can be used for automatically transforming a *Petri Net* into *Scalable Vector Graphics* (SVG) by XSLT transformations (see [16] for more details). This transformation, however, is not part of ISO/IEC 15909-2.

Figure 3 shows the different graphical information that can be attached to the different types of *objects* and the different attributes. Note that these concepts still belong to the *PNML Core Model*; it is shown in

a different figure only in order to avoid clutter. Table 1 gives an overview of the meaning and the domain of the attributes of the different graphical features.

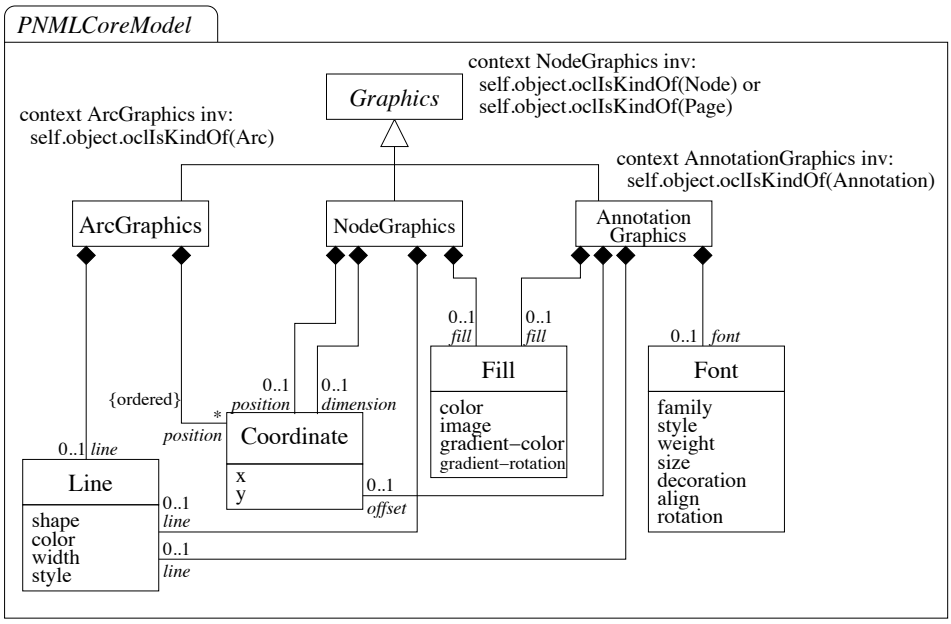


Figure 3: The *PNML Core Model*: graphical information

Table 1: PNML attributes of graphical information

Class	Attribute	Domain
Coordinate	x	decimal
	y	decimal
Fill	color	CSS2-color
	image	anyURI
	gradient-color	CSS2-color
	gradient-rotation	{vertical, horizontal, diagonal}
Line	shape	{line, curve}
	color	CSS2-color
	width	nonNegativeDecimal
	style	{solid, dash, dot}
Font	family	CSS2-font-family
	style	CSS2-font-style
	weight	CSS2-font-weight
	size	CSS2-font-size
	decoration	{underline, overline, line-through}
	align	{left, center, right}
	rotation	decimal

The *position* defines the absolute position for *nodes* and *pages*. For an *annotation* the *offset* defines its relative position to the *object* it is attached to; for a *global annotation*, the *offset* defines the absolute position on that *page*. Each absolute or relative position consists of a pair of Cartesian coordinates (x, y), where the units are points (pt). As for many graphical tools, the x-axis runs from left to right and the y-axis from top to bottom. And the reference point for the position of an *object* is its centre.

For an *arc*, the (possibly empty) sequence of *positions* defines its intermediate points (bend points). Note that the positions of the *start point* and the *end point* of an *arc* are not given explicitly for the arc. These positions are determined from the position of the source and the target *nodes* of the *arc* and the direction of the first resp. last segment of the *arc*, by the intersection of this segment with the nodes border. Altogether, the *arc* is displayed as a path from the *start point* on the border of the *source node* to the *end point* on the border of the *target node* via the intermediate points. Note that, even though there

is no way to define the *start point* and the *end point* of an arc explicitly, the above concepts allow a user making sure that an arc starts and ends exactly at the point, where he wants it to start and end: If the first, (or last) intermediate point of an arc lies exactly on the border of the respective node, this will be the *start point* (or the *end point* resp.) of that arc.

For *arcs*, there are also some *line* attributes, which define the *style*, the *colour*, and the *width* in which they are displayed.

Depending on the value of the attribute *shape* of element *line*, the path is displayed as a broken *line* (polyline) or as a quadratic Bezier *curve*. In the case of a Bezier curve the intermediate positions alternately are the line connectors or Bezier control points. The reference point of an arc is the middle of the arc, which is the middle of the middle segment of the arc, if there are an odd number of segments; it is the middle point, if there are an even number of segments.

The *dimension* of a *node* or *page* gives its size, again as a pair referring to its width (x) and height (y). Depending on the ratio of height and width, a *place* is displayed as an ellipse rather than a circle. A *transition* is displayed as a rectangle of the corresponding size. If the dimension of an element is missing, each tool is free to use its own default value for the dimensions.

The two elements *fill* and *line* define the interior and outline colours of the corresponding element. The value assigned to a *color* attribute must be a RGB value or a predefined colour as defined by CSS2 (Cascading Style Sheets 2). When the attribute *gradient-color* is defined, the fill colour continuously varies from color to gradient-color. The additional attribute *gradient-rotation* defines the orientation of the gradient. When the attribute *image* is defined, the node is displayed as the image which is provided at the specified URI, which must be a graphics file in JPEG or PNG format. In this case, all other attributes of *fill* and *line* are ignored.

For an *annotation*, the *font* element defines the font used to display the text of the *label*. The attributes *family*, *style*, *weight*, and *size* are CSS2 attributes for defining the appearance of the text. The detailed description of the possible values of these attributes and their effect can be found in the CSS2 specification. The *decoration* attribute defines additional properties of the appearance of the text such as underlining, overlining, or striking-through the text. Additionally, the *align* attribute defines the alignment of the text to the *left*, *right* or *center*. The *rotation* attribute defines a clockwise rotation of the text.

The reference point of an annotation is always its centre.

2.2.5 Tool Specific Information

For some tools, it might be necessary to store *tool specific information* (ToolInfo), which is not meant to be used by other tools. In order to store this information, *tool specific information* may be associated with each *object* and each *label*. The internal structure of the *tool specific information* depends on the tool and is not specified by PNML. PNML provides a mechanism for clearly marking *tool specific information along* with the name and the version of the tool adding this information. Therefore, other tools can easily ignore it, and adding *tool specific information* will never compromise a *Petri Net Document*.

The same *object* may be tagged with *tool specific information* from different tools. This way, the same document can be used and changed by different tools at the same time. The intention is that a tool should never change or delete the information added by another tool as long as the corresponding *object* is not deleted. Moreover, *tool specific information* should be self-contained and not refer to other *objects* of the net because the deletion of other *objects* by a different tool might make this reference invalid and leave the *tool specific information* inconsistent. This use of the *tool specific information* is strongly recommended; however, it is not normative.

2.2.6 Data Types

In this section, we discuss six XML data types, which are taken from XMLSchema. These can be used for defining labels for some *Petri net type* with numerical and textual information. Note that these data types are not related to the data types that may be used within high-level nets! Rather, they define the types of the attributes and their XML syntax that may be used in the meta-models of the *PNML Core Model* and of the *Petri net types*.

Figure 4 gives an overview of these XML data types and their relation.

String refers to any printable sequence of characters, which is mapped to XML PCDATA. *Decimals*, *NonNegativeDecimals*, *Integer*, *NonNegativeInteger*, and *PositiveInteger* refer to any character sequence which denotes a number of the corresponding kind.

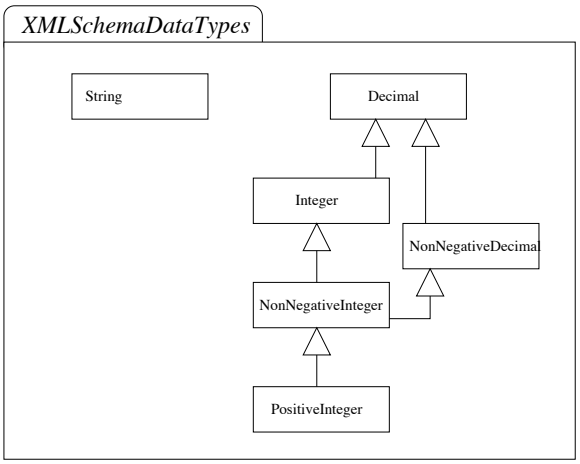


Figure 4: Standard data types

2.3 Petri Net Type Meta Models

Next, we discuss three versions of *Petri nets*: *Place/Transition Nets*, *Symmetric Nets*, and *High-Level Petri Net Graphs* (HLPNGs) as defined in ISO/IEC 15909-1, where *Symmetric Nets*[1] are a restricted version of *High-Level Petri Net Graphs* currently defined as an Amendment to ISO/IEC 15909-1. The general relation between the different types has already been shown in Fig. 1. Again, the concepts of these different versions are defined in terms of meta-models. These meta-models define the labels of the respective *Petri net type*.

ISO/IEC 15909-1 defines a mapping from the concepts of *Place/Transition Nets* to the concepts of *High-Level Petri Net Graphs* and shows how the usual mathematical representation of *Place/Transition Nets* can be represented as a restricted version of *High-level Petri Nets*. Though conceptually the same model, the syntax is different. Therefore, ISO/IEC 15909-2 introduces yet another Petri net version: *Place/Transiton Nets in High-level Notation*, which will be discussed in Sect. 3.3.1.

ISO/IEC 15909-2, however, introduces an explicit transfer format for *Place/Transition Nets* in order not to force tools for *Place/Transition Nets* to use the syntax of *High-Level Petri Net Graphs*. This format reflects the usual mathematical definition of *Place/Transition Nets*.

2.3.1 Place/Transition Nets

Since *Place/Transition Nets* are the simplest version, we start with this version: the concepts are defined in terms of a meta-model in UML notation: the package *PT-Net*.

A *Place/Transition Net* is a *net graph*, where each *place* can be labelled with a natural number representing the *initial marking* and each *arc* can be labelled with a non-zero natural number representing the *arc annotation*, with the usual meaning: the *label* of a *place* p denotes the initial marking $M(p)$, the *label* of an *arc* f denotes the arc weight $W(f)$.

Figure 5 shows the package PT-Net. Note that the only classes defined here are *PTMarking* and *PTArcAnnotation*. The classes *Place*, *Arc*, and *Annotation* come from the package *PNML Core Model*. They are imported (actually, they are merged) here in order to define the possible *labels* for the particular *nodes* of *Place/Transition Nets*. Likewise, the classes prefixed with *XMLSchemaDataTypes* are imported from the standard data type package (see Sect. 2.2.6).

The *initial marking* of a *place* is represented by the *annotation PTMarking*, the contents of which must be a non-negative integer. Technically, the representation of the contents of this *label* is defined by referring to the data type *NonNegativeInteger*.

The *arc annotation* is represented by the *annotation PTArcAnnotation*, the contents of which must be a non-zero natural number, which is defined by referring to the data type *PositiveInteger*.

Note that, according to this definition, it is legal that a *place* does not have an *annotation* for the *initial marking*. In that case, the *initial marking* is assumed to be empty, i. e. 0. Likewise, there may be no *annotation* for an *arc*. In that case, the *arc annotation* is assumed to be 1.

[1] Remember that *Symmetric Nets* were originally introduced under the name *well-formed nets* [14].

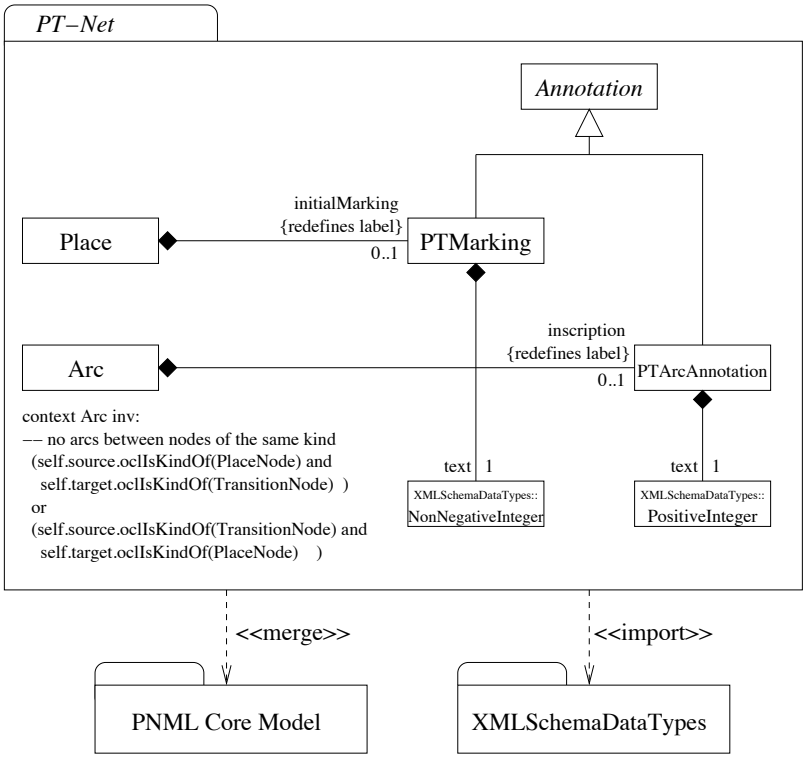


Figure 5: The package PT-Net

In addition to the definition of the new *labels*, this package also defines the structural restriction that, in a *Place/Transition Net*, an *arc* must not connect a *place* to a *place* or a *transition* to a *transition*. This is captured by the OCL expression below class *arc*.

Sometimes, one wants to store the position of the individual tokens within a place. In order not to mandate all tools to support this feature, ISO/IEC 15909-2 suggests a *tool specific information* for this purpose, which would refer to the tool *org.pnml.tool*. Since no tool is required to support tool specific features, every tool is free to use and support this feature or not.

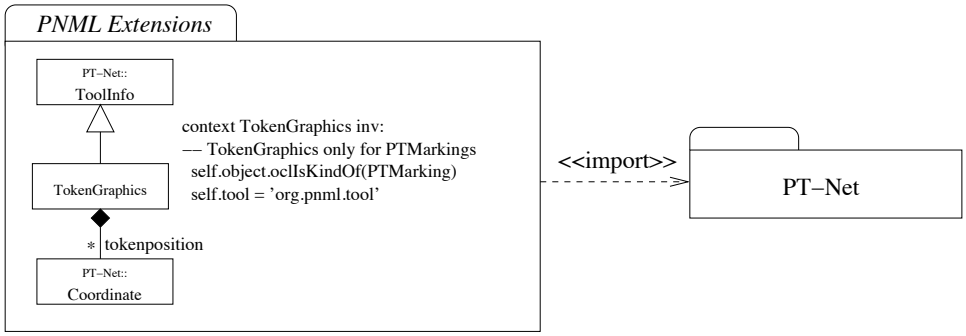


Figure 6: Tool specific extension for token positions

This extension is shown in Fig. 6. For a *PTMarking*, this label can be equipped with the *tool specific information* attached to *AnnotationGraphics*. This consists of information on a list of *tokenpositions*. Each of these elements represents the position of a token relative to the centre of the *place*; represented as a *Coordinate*, which is imported from package *PT-Net*, where it was obtained by a merge with the *PNML Core Model*. If this graphical information is present at all, the number of *tokenpositions* should be the same as indicated by the *PTMarking* (see example in Sect. 4.1.5).

2.3.2 High-Level Core Structure

ISO/IEC 15909-1 defines *High-level Petri Net Graphs*. *Symmetric Nets* and *Place/Transition Nets* are introduced as restricted versions of *High-level Petri Net Graphs*. The difference is in the *types* and *functions* that may be used in the different versions. To cater for this structure, we distill the common structure of all *High-level Petri Net Graphs* first: This is called the *High-Level Core Structure*. The built-in data types of the specific Petri net types are discussed in Sect. 3.1.

Syntactically, the basic features of a *High-Level Petri Net* are the *annotations* of *places*, *transitions*, and *arcs*. For each *place*, a *sort* defines the *type* of the tokens on this *place*. A *term* associated with a *place* denotes the *initial marking* and must have the respective *sort*. The *term* associated with an *arc* to or from a *place*, defines which tokens are added or removed, when the corresponding *transition* fires. These *terms* must also be of the respective *sort*.

For constructing such *terms*, one can use built-in *operators* and *sorts*, and user-defined *variables*, which are defined in a *variable declaration*. The *variable declarations* are *annotations* of the *net* or a *page*. Moreover, a *transition* can have a *condition* which is a *term* of *sort* boolean and imposes additional conditions on the situations in which a transition can fire.

The meta-model for *terms*, which defines all these concepts, is shown in Fig. 7: It defines the concepts of *sorts*, *operators*, *declarations*, and *terms*, and how *terms* are constructed from *variables* and *operators*.

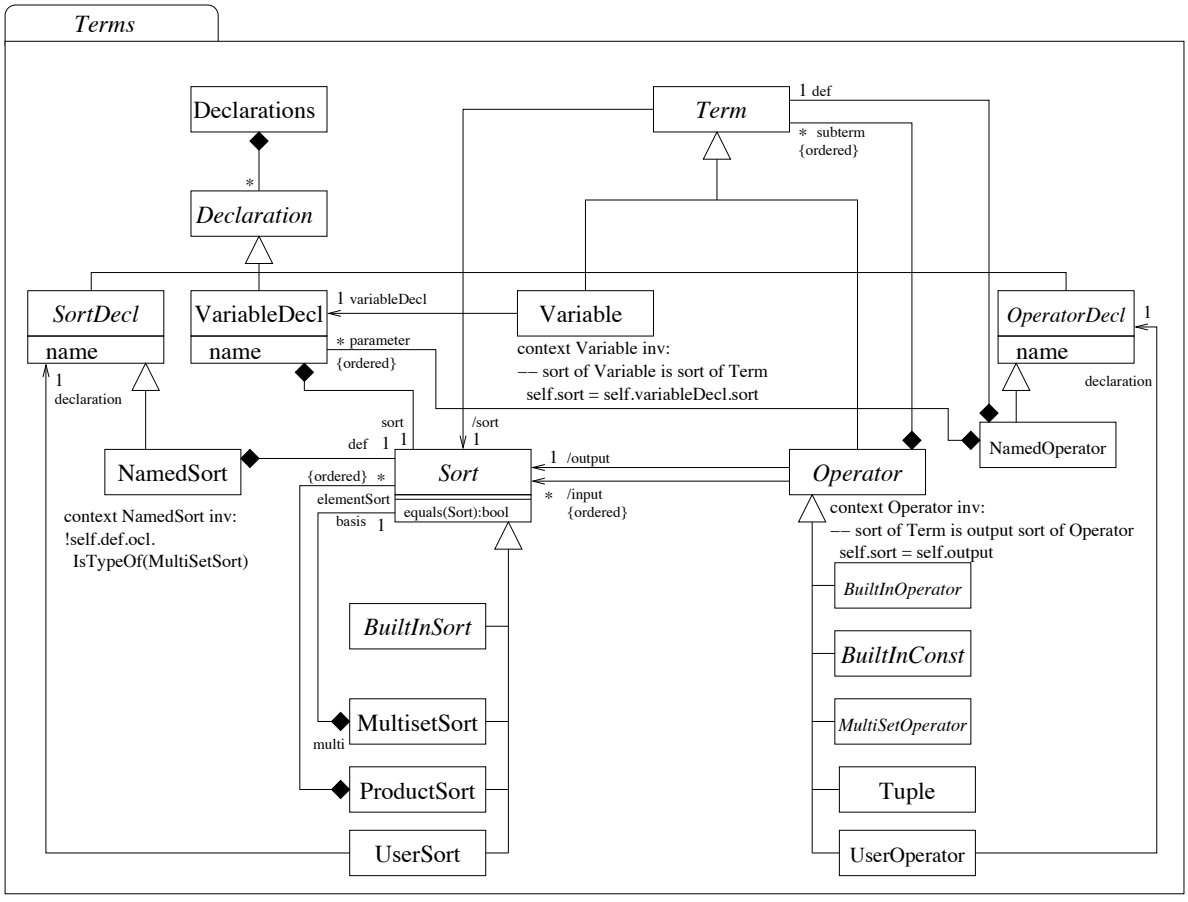


Figure 7: The meta-model for *Terms*

For each *variable declaration*, there is a corresponding *sort*. A *sort* can be a *built-in sort*, a *multiset sort* over some *basis sort*, a *product sort* over some *sorts*, or a *sort* which is given in a *user declaration*. In the core structure, the only possible *sort declaration* of a user is by constructing a new *sort* from existing ones and by giving them a new name. From these, the user can define new ones. Note that cyclic references in user-defined *sorts* are not allowed. In addition to these user-defined sorts, called *named sorts*, there will be *arbitrary sorts*. Since these are not allowed in *Symmetric Nets*, these are not defined in the core. They are discussed in Sect. 3.3.3 where the concepts of general high-level nets are explained.

An *operator* can be a *built-in constant* or a *built-in operator*, a *multiset operator* which among others can construct a *multiset* from an enumeration of its elements, or a *tuple* operator. Each *operator* has a sequence of *sorts* as its *input sorts*, and exactly one *output sort*, which defines its signature. As for *sorts*, the user can define his own *operators*. Here, it is only possible to define an abbreviation, which is called a *named operator*: It can use a *term*, which is built from existing *operators* and parameter variables, for defining a new *operator*. As for *sorts*, cycles (recursion) in these definitions are not allowed. As for sorts, there will be arbitrary operator declarations for *High-level Petri Net Graphs*, but not for *Symmetric Nets*. Therefore, these concepts will be discussed in Sect. 3.3.3.

From the built-in *operators* and the user-defined *operators* and *variables*, *terms* can be constructed in the usual way. The *sort* of a *term* is the *sort* of the *variable* or the *output sort* of the *operator*. Therefore, *sort* is indicated as a derived association; its definition is expressed by an OCL expression in the UML metamodel. Note that the *input* and *output sorts* of the *operator* are also represented as derived associations because, in some situations, they need not be given explicitly since they can be derived from the type of the *operator*.

Figure 8 shows the package *High-Level Core Structure*, which defines all the *annotations* for both *Symmetric Nets* and *High-Level Petri Net Graphs*. Note that, since the classes for built-in *sorts* and *operators* are abstract, we do not have any built-in *sorts* and *operators* yet. These are discussed in Sect. 3.1.

In addition to the *annotations* defined above, the package *High-Level Core Structure* requires that *arcs* must not connect two *places* and must not connect two *transitions*.

Note that this model defines an abstract syntax (composition *structure*) for all these concepts only. In order to allow tools to store the concrete text, all *annotations* of *High-level Nets* may also consist of text, which should be the same expression in the concrete syntax of some tool. This concrete syntax, however, is not mandated by ISO/IEC 15909-2; therefore, all the meaning is in the *structure* of the labels!

3 Details of High-level nets and their data types

As mentioned earlier, Sect. 2.3.2 discussed the core structure of *High-level Petri Net Graphs* only. In this section, we briefly discuss the details of the built-in *sorts* and *operators*. For each built-in *sort*, there is a UML package. Sections 3.1 gives an overview of these packages. Section 3.2 briefly discusses the package for user-defined declarations. Based on that, Sect. 3.3 defines the different types of high-level nets.

3.1 Data types

ISO/IEC 15909-2 defines the following data types for being used in the different versions of high-level Petri nets:

- *Dots* defines the sort which represents a type with exactly one element (token): $\{\bullet\}$. This is used to represent *Place/Transition Nets* as *High-level Net Graphs*.
- *Multiset* defines the multiset over any other sort, which are needed for defining the terms labelling arcs and for initial markings.
- *Booleans* defines the boolean type and associated operations.
- *Finite Enumerations*, *Cyclic Enumerations*, and *Finite Integer Ranges* allow the definition of finite ranges by explicitly enumerating them or by giving an integer range. Depending on the type, different structures are imposed on them [14], such as an order relation, or a successor and predecessor operation.
- *Partitions* allow the definition of finite enumerations that are partitioned into sub-ranges, where a partition function defines for each element to which partition it belongs. These partitions define a separate sort, where each partition is an element of the respective type.
- *Integer* defines the integers and the usual operations on them; it also defines the subtypes positive numbers and the natural numbers.
- *Strings* defines the strings and the usual operations on them.

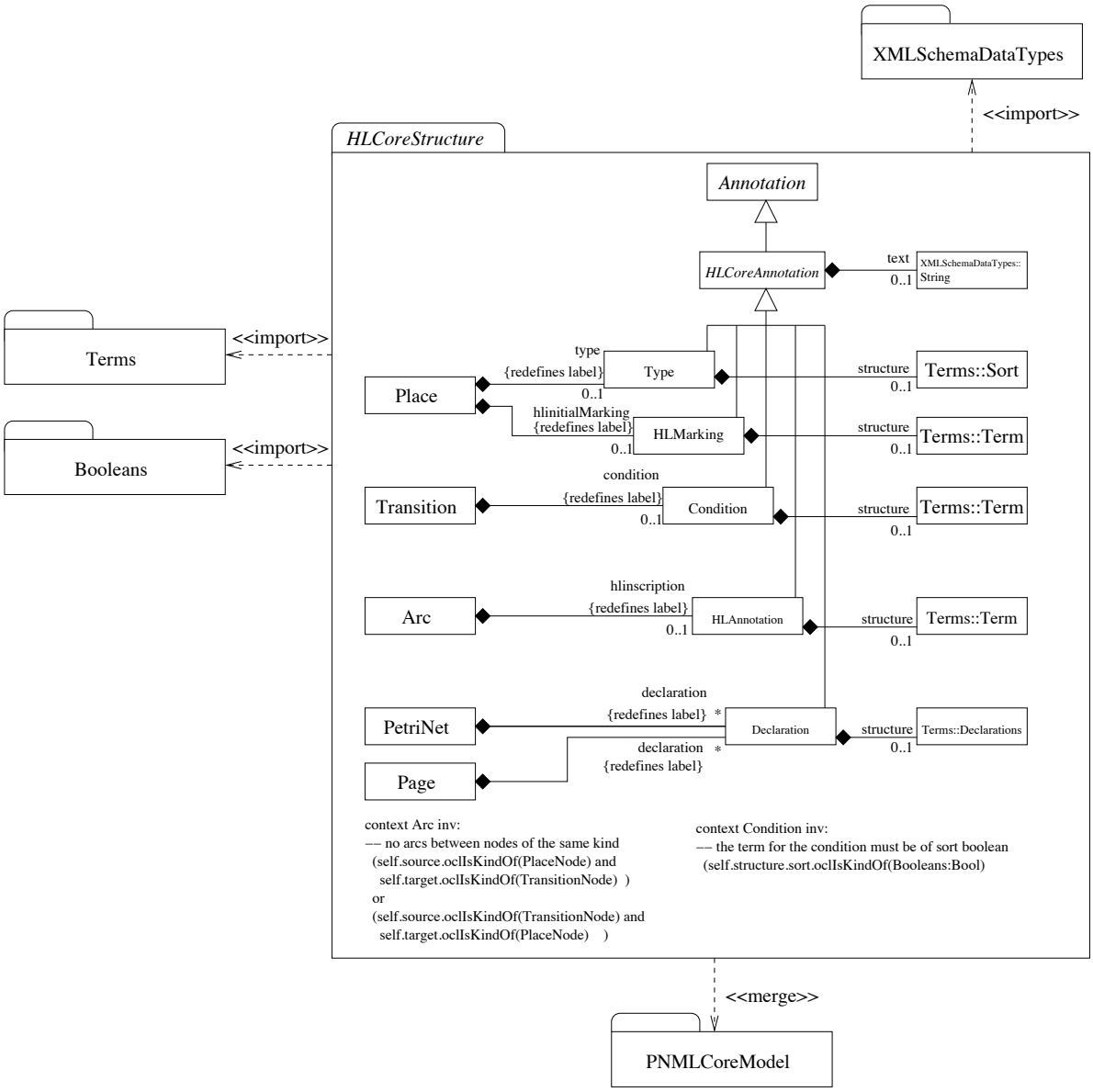


Figure 8: The package *High-Level Core Structure*

- *Lists* defines the Lists over the other data types and the usual operations on them.

For details, we refer to ISO/IEC 15909-2 or to the RELAX NG grammars which can be found at [17].

3.2 User declarations

In general *High-Level Petri Net Graphs* (HLPNGs), the user is allowed to define arbitrary *sorts* and *operators*. This package *ArbitraryDeclarations* is shown in Fig. 9. In contrast to *named sorts* and *named operators*, arbitrary *sorts* and *operators* do not come with a definition of the *sort* or *operation*; they just introduce a new symbol without giving a definition for it. So, these symbols do not have a meaning, but can be used for constructing *terms*.

The additional concept *Unparsed* in *terms* provides a means to include any text, which will not be parsed and interpreted by the tools. This is helpful for exchanging the general structure of a term, but not all its details.

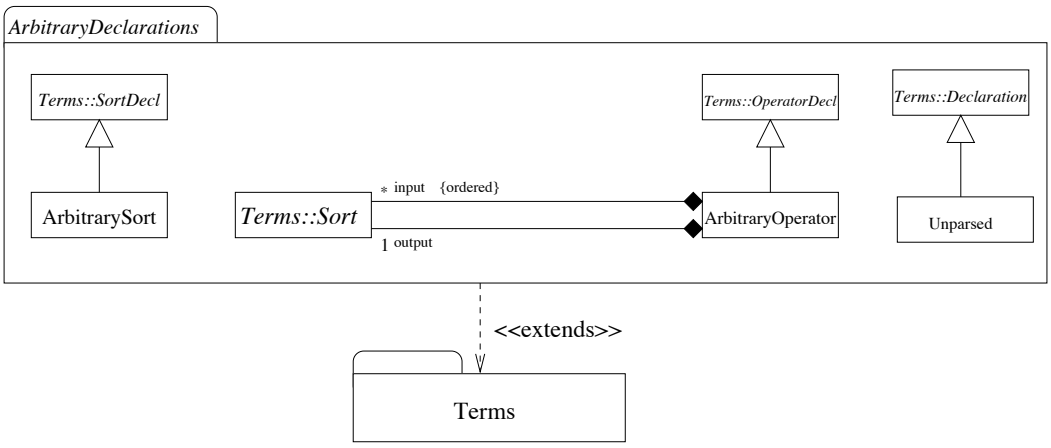


Figure 9: The package *ArbitraryDeclarations*

3.3 Net types

Based on the data types introduced in the previous section, we can now discuss the different versions of high-level nets. Note that, conceptually, high-level nets extend *Place/Transition-Systems*. But, syntactically they do not. That is why all these types are not extending the package *PT-Net*, which is discussed in Sect. 2.3.1.

Instead, the different versions of high-level nets form a separate hierarchy (see also Fig. 1). The lowest level in this hierarchy are *Place/Transition Nets in High-level Notation*.

3.3.1 Place/Transition Nets as High-level Net Graphs

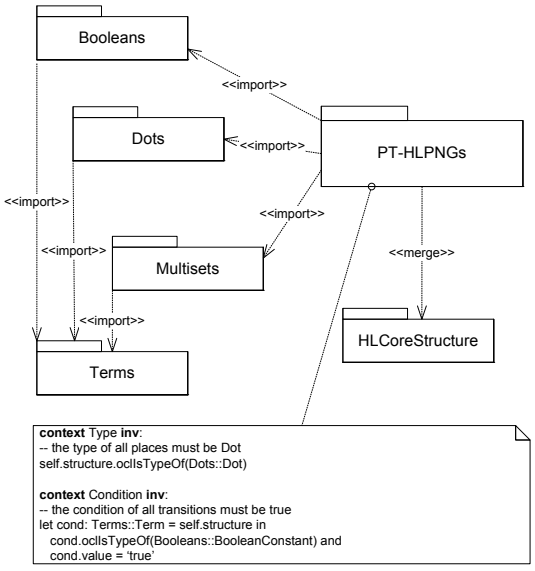


Figure 10: The package of *P/T Nets* defined as restricted *HLPNGs*

Annex B of ISO/IEC 15909-1 defines *Place/Transition Nets* as a restricted form of *High-level Net Graphs*. Fig. 10 shows the corresponding UML definition. This class allows for the use of the built-in *sorts Bool* and *Dot* only. It does neither allow any user *declarations*, nor *variables*, nor *sorts*, nor *operators*.

The *type* of each *place* must refer to *sort Dot*. All transition *conditions* need to be the constant *true*, if this label is present. And the *arc annotations* and the *initial markings* are *ground terms* of the *mulitset sort* over *Dot*.

3.3.2 Symmetric Nets

Symmetric Nets as currently defined in an Amendment (Annex B.2) of Part 1 of ISO/IEC 15909, are *High-Level Petri Net Graphs* with some restrictions. Basically, the carrier sets of all basic *sorts* are finite, and only a fixed set of *operations*, as defined below, are allowed. Therefore, the *sort* of a *place* must not be a *multiset sort*. The available built-in *sorts* are defined below.

Moreover, for *Symmetric Nets*, it is required that every HLPNG *annotation* has the *structural* information.

The built-in *sorts* of *Symmetric Nets* are the following: *Booleans*, *range of integers*, *finite enumerations*, *cyclic enumerations* and *dots*. Moreover, for every *sort*, there is the operator *all*, which is a multiset that contains exactly one element of its basis *sort*. This is often called the broadcast function.

Altogether, the *Symmetric Nets* can be defined by importing the packages as shown in Fig. 11.

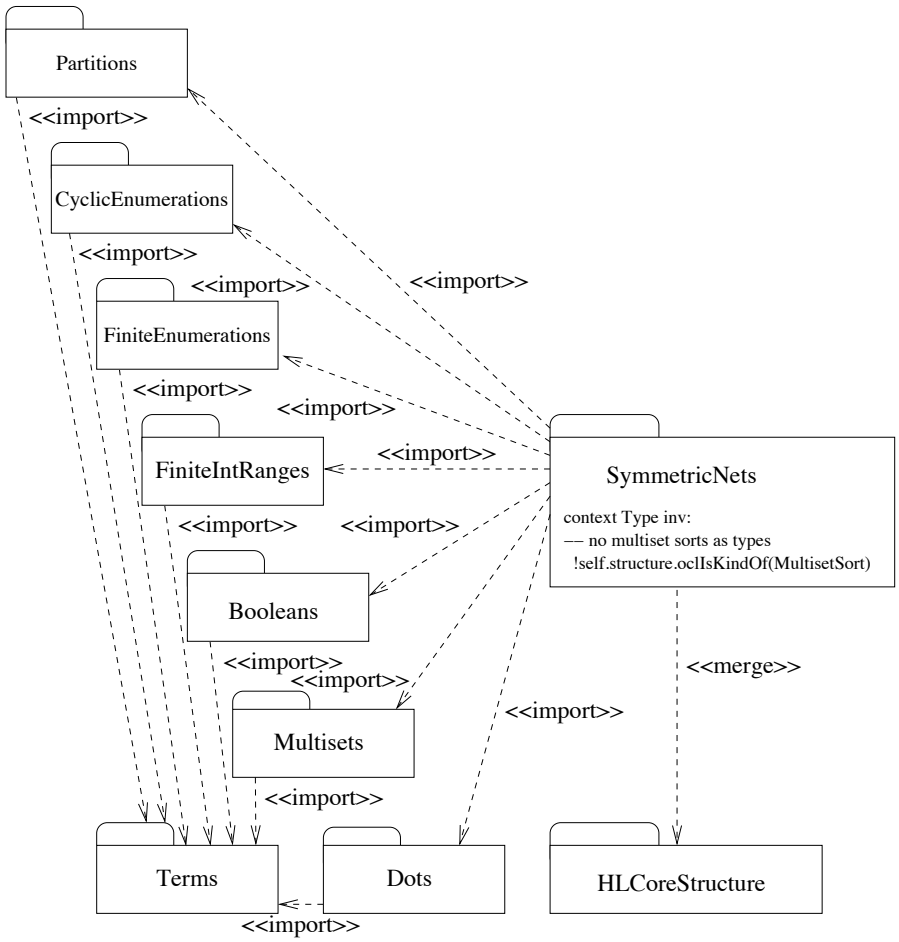


Figure 11: The package *Symmetric Nets* and its built-in *sorts* and *functions*

3.3.3 High-level Petri Net Graphs

The complete definition for *High-Level Petri Net Graphs* is shown in Fig. 12. It extends *Symmetric Nets* by declarations for *sorts* and *functions* and the additional built-in *sorts* for *Integer*, *String*, and *List*.

4 XML Syntax

Section 2 and 3 discussed the concepts of the *PNML Core Model* and the concepts of *Place/Transition Nets*, *High-level Petri Net Graphs*, and *Symmetric Nets* in terms of UML meta-models (and some additional constraints). This, however, does not define the concrete XML syntax for representing these concepts.

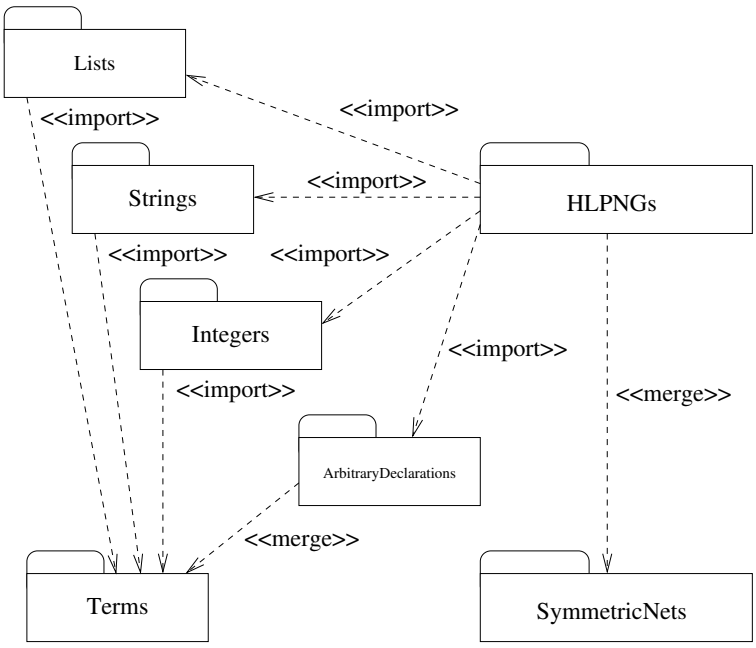


Figure 12: The package *HLPNGs*

The XML syntax of the *PNML Core Model* and the three *Petri net types* is discussed in this section. A RELAX NG grammar defining the exact XML syntax can also be found at the PNML web site [17].

In Sect. 4.1, we discuss the general format of *PNML Documents*; i. e. the XML syntax for the *PNML Core Model*. In Sect. 4.2, we discuss the format for *Place/Transition Nets*. In fact, for simple *Petri net types* such as *Place/Transition Nets* there are some general rules, as to how the concepts of a package defining some *Petri net type* are mapped to XML syntax. This idea is discussed here by the help of the example of *Place/Transition Nets*. For *High-level Petri Nets*, however, there is a dedicated mapping for the *labels* of these types to XML. This is discussed in Sect. 4.3.

Note that there is no need for a separate mapping for *Symmetric Nets* and *Place/Transition Nets in High-level Notation* to XML since, as a restricted version of *High-level Petri Nets*, these mappings are fully covered by the mapping for *High-level Petri Nets*.

4.1 XML syntax of the PNML Core Model

The mapping of the *PNML Core Model* concepts to XML syntax is defined for each class of the *PNML Core Model diagram* separately.

4.1.1 PNML Elements

Each concrete class[2] of the *PNML Core Model* is mapped to an XML element. The mapping of these classes along with the attributes and their data types is given in Table 2. These XML elements are the *keywords* of PNML and are called *PNML elements* for short. For each *PNML element*, the compositions of the *PNML Core Model* define in which elements it may occur as a child element.

The data type ID in Table 2 refers to a set of unique identifiers within the *PNML Document*. The data type IDRef refers to the set of all identifiers occurring in the document, i. e. they are meant as references to identifiers. A reference at some particular position, however, is restricted to objects of a particular type — as defined by the resp. associations in the *PNML Core Model*. For instance, the attribute `ref` of a *reference place* must refer to a *place* or a *reference place* of the same *net*. The set to which a reference is restricted is indicated in the table (e. g. for a *reference place*, the attribute `ref` should refer to the `id` of a *Place* or a *RefPlace*). Note that these requirements are defined in the UML meta-model already; here, these requirements are repeated just for better readability.

[2]A class in a UML diagram is concrete if its name is not displayed in italics.

Table 2: Translation of the PNML Core Model into PNML elements

Class	XML element	XML Attributes
PetriNetDoc	<pnml>	xmlns: anyURI (http://www.pnml.org/version-2009/grammar/pnml)
PetriNet	<net>	id: ID type: anyURI
Place	<place>	id: ID
Transition	<transition>	id: ID
Arc	<arc>	id: ID source: IDRef (Node) target: IDRef (Node)
Page	<page>	id: ID
RefPlace	<referencePlace>	id: ID ref: IDRef (Place or RefPlace)
RefTrans	<referenceTransition>	id: ID ref: IDRef (Transition or RefTrans)
ToolInfo	<toolspecific>	tool: string version: string
Graphics	<graphics>	
Name	<name>	

Note that the <pnml> element must have a namespace attribute xmlns. For the current version of PNML, this namespace is fixed: http://www.pnml.org/version-2009/grammar/pnml

4.1.2 Labels

Except for *names*, there are no explicit definitions of *PNML elements* for *labels* because the *PNML Core Model* does not define other *labels*. For concrete *Petri net types*, such as *Place/Transition Nets*, *Symmetric Nets*, and *High-level Petri Nets* the corresponding packages define these *labels*.

In general *PNML Documents*, any XML element that is not defined in the *PNML Core Model* (i.e. not occurring in Table 2) is considered as a *label* of the *PNML element* in which it occurs. For example, an <initialMarking> element could be a *label* of a *place*, which represents its initial marking, and <inscription>, which represents an arc annotation.

A legal element for a *label* must contain at least one of the two following elements, which represents the actual value of the *label*: a <text> element represents the value of the *label* as a simple string; the <structure> element can be used for representing the value as an abstract syntax tree in XML.

An optional PNML <graphics> element defines its graphical appearance; and optional PNML <toolspecific> elements may add tool specific information to the label. Note that ISO/IEC 15909-2 does not mandate the inner structure of <toolspecific> elements; every tool is free to structure its information inside that element at its discretion; as long as it produces well-formed XML.

4.1.3 Graphics

All *PNML elements* and all *labels* may include graphical information. The internal structure of the PNML <graphics> element, i.e. the legal XML children, depends on the element in which the graphics element occurs. Table 3 shows the XML elements which may occur within the <graphics> element (as defined by the UML model in Fig. 3).

Table 4 explicitly list the attributes for each graphical element defined in Table 3 (cf. Fig. 3 and Table 1). The domain of the attributes refers to the data types of either XML Schema, or Cascading Stylesheets 2 (CSS2), or is given by an explicit enumeration of the legal values.

4.1.4 Mapping of XMLSchemaDataTypes concepts

The concepts from the package *XMLSchemaDataTypes* are mapped to XML syntax in the following way: The *String* objects are mapped to XML PCDATA, i.e. there will be a PCDATA section within the element which contains the *String*. This, basically, corresponds to any printable text.

Table 3: Possible child elements of the <graphics> element

Parent element class	Sub-elements of <graphics>
Node, Page	<position>
	<dimension>
	<fill>
	<line>
Arc	<position> (zero or more)
	<line>
Annotation	<offset>
	<fill>
	<line>
	<font>

Table 4: PNML graphical elements

XML element	Attribute	Domain
<position>	x	decimal
	y	decimal
<offset>	x	decimal
	y	decimal
<dimension>	x	nonNegativeDecimal
	y	nonNegativeDecimal
<fill>	color	CSS2-color
	image	anyURI
	gradient-color	CSS2-color
	gradient-rotation	{vertical, horizontal, diagonal}
<line>	shape	{line, curve}
	color	CSS2-color
	width	nonNegativeDecimal
	style	{solid, dash, dot}
<font>	family	CSS2-font-family
	style	CSS2-font-style
	weight	CSS2-font-weight
	size	CSS2-font-size
	decoration	{underline, overline, line-through}
	align	{left, center, right}
	rotation	decimal

Likewise, *Integers*, *NonNegativeIntegers* and *PositiveIntegers* are mapped to the XMLSchema syntax constructs *integer*, *nonNegativeInteger*, and *positiveInteger*, respectively.

4.1.5 Example

In order to illustrate the structure of a *PNML Document*, there is a simple example *PNML Document* representing the *Petri net* shown in Fig. 13, which is a *Place/Transition Net*. Listing 1 shows the corresponding *PNML Document* in XML syntax. It is a straightforward translation, where there are *labels* for the *names* of objects, for the *initial markings*, and for *arc annotations*.

Note that, in this figure, the *initial marking* is not displayed as a textual *label* – it is shown graphically. This is due to the fact that the *initial marking* comes with a *tool specific* information on the positions of the *tokens*, tokens are shown at the individual positions as given in the elements <tokenposition>.

Since there is no information on the dimensions in this example (in order to fit the listing to a single page), the tool has chosen its default dimensions for the place and the transition.

4.2 XML syntax of the Petri net types

Based on the *PNML Core Model* of Sect. 2.2, Sect. 2.3.1 and 3.3 discussed the definition of some *Petri net types*, which restrict *PNML Documents* to the particular *labels* defined in the corresponding packages.

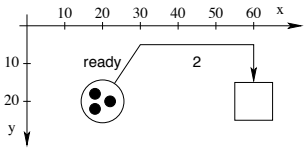


Figure 13: A simple *Place/Transition Net*

Listing 1: PNML code of the example net in Fig. 13

```
<pnml xmlns="http://www.pnml.org/version-2009/grammar/pnml">
  <net id="n1" type="http://www.pnml.org/version-2009/grammar/ptnet">
   <page id="top-level">
    <name>
      <text>An example P/T-net</text>
    </name>
    <place id="p1">
      <graphics>
        <position x="20" y="20"/>
      </graphics>
      <name>
        <text>ready</text>
        <graphics>
          <offset x="0" y="-10"/>
        </graphics>
      </name>
      <initialMarking>
        <text>3</text>
        <toolspecific tool="org.pnml.tool" version="1.0">
          <tokengraphics>
            <tokenposition x="-2" y="-2" />
            <tokenposition x="2" y="0" />
            <tokenposition x="-2" y="2" />
          </tokengraphics>
        </toolspecific>
      </initialMarking>
    </place>
    <transition id="t1">
      <graphics>
        <position x="60" y="20"/>
      </graphics>
    </transition>
    <arc id="a1" source="p1" target="t1">
      <graphics>
        <position x="30" y="5"/>
        <position x="60" y="5"/>
      </graphics>
      <inscription>
        <text>2</text>
        <graphics>
          <offset x="0" y="5"/>
        </graphics>
      </inscription>
    </arc>
   </page>
  </net>
</pnml>
```

Table 5: Tool specific information for token positions

XML element	Attribute	Domain
`<tokengraphics>`		
`<tokenposition>`	x	decimal
	y	decimal

These packages define the *labels* that are used in the particular *Petri net*. Here, it is shown how to map such a package to the corresponding XML syntax. This mapping is the same for all type definitions, unless the type definition explicitly defines a dedicated mapping for this type. This general mapping will be explained by the help of the example of *Place/Transition Nets* (see Fig. 5 and the *PNML Document* in Listing 1).

The PT-Net package defines two kinds of *labels* that can be used in a *Place/Transition Net*: PTMarkings and PTAnnotations. Each *place* can have one *annotation* PTMarking, and each *arc* can have one *annotation* PTAnnotation. This is indicated by the compositions in the UML diagram in Fig. 5.

The XML syntax for these *labels* is derived from the role names of these compositions. Every *annotation* PTMarking is mapped to an XML element `<initialMarking>`, and every *annotation* PTAnnotation is mapped to an element `<inscription>`. Listing 1 shows an example for the XML syntax of a *Place/Transition Net*.

Since all *labels* in this package are *annotations*, all graphical elements defined for *annotations* may occur as children in these elements.

In the PT-Net package each *label* is defined to have a `<text>` element, which defines the actual content of this *annotation*. For *Place/Transition Nets* there are no structured elements.

The corresponding classes from package DataTypes define the XML content of the `<text>` element. These were discussed in Sect. 4.1.4 already.

Note that for *Place/Transition Nets*, there is a predefined tool specific extension for the label `<initialMarking>` (see Sect. 2.3.1), which represents the positions of tokens within a place. The class *TokenGraphics* is mapped to the XML element `<tokengraphics>` with no attributes. The token positions contained by this element are represented by the elements `<tokenposition>` as shown in Table 5. Within the element `<tokengraphics>` there can be any number of `<tokenposition>` elements with two attributes x and y.

4.3 Mapping for *High-Level Nets*

Table 6 defines the mappings between the meta-model elements and their XML representation. In these tables, only meta-model elements that have corresponding PNML elements or attributes are displayed. Thus, most abstract elements are not shown in the tables, except if they have attributes. All attribute types are mapped to *XMLSchema-datatypes* library data types. For details, we refer to the RELAX NG grammars [17].

5 Conclusion

In this paper, we have outlined the main concepts and the syntax of PNML as defined in ISO/IEC-15909-2, which passed its final ballot and will hopefully be published soon. For lack of space, we could not discuss all the details. In particular, we could not discuss the rationale behind the design of PNML and we could not discuss all built-in data types and their XML syntax. For the design rationale, we refer to earlier publications [2, 15, 4] and for some more information on the XML syntax we refer to the PNML web pages [17], which has the full RELAX NG grammar for all currently standardized Petri net types. The ultimate source of information will be the International Standard ISO/IEC 15909-2.

Another great help for implementing the PNML might be the PNML Framework [18], which provides an API for reading and writing Petri nets in PNML. This API is automatically generated from the PNML meta-models in the *Eclipse Modeling Framework* (EMF). More details can be found on the PNML web pages [17].

These web pages are also used for discussing future extensions, some of which might be covered in Part 3 of ISO/IEC 15909. This concerns in particular the precise definition of how to define a *Petri*

Table 6: *High-level meta-model elements and their PNML constructs*

Model element	**PNML element**	**PNML attributes**
Booleans::Bool	bool	
Booleans::And	and	
Booleans::Or	or	
Booleans::Not	not	
Booleans::Imply	imply	
Booleans::Equality	equality	
Booleans::Inequality	inequality	
Booleans::BooleanConstant	booleanconstant	value: boolean
HLCoreStructure::Declaration	declaration	
HLCoreStructure::Type	type	
HLCoreStructure::HLMarking	hlinitialmarking	
HLCoreStructure::Condition	condition	
HLCoreStructure::HLAnnotation	hlinscription	
Terms::Declarations	declarations	
Terms::VariableDeclaration	variabledecl	id: ID; name: string
Terms::OperatorDeclaration	No element	id: ID; name: string
Terms::SortDeclaration	No element	id: ID; name: string
Terms::Variable	variable	variabledecl: IDREF
Terms::NamedSort	namedsort	
Terms::NamedOperator	namedoperator	
Terms::Term	No element	
Terms::Sort	No element	
Terms::MultisetSort	multisetsort	
Terms::ProductSort	productsort	
Terms::UserSort	usersort	declaration: IDREF
Terms::Tuple	tuple	
Terms::Operator	No element	
Terms::UserOperator	useroperator	declaration: IDREF
ArbitraryDeclarations::ArbitrarySort	arbitrarysort	
ArbitraryDeclarations::Unparsed	unparsed	
ArbitraryDeclarations::ArbitraryOperator	arbitraryoperator	

net type (which where used informally only up to now) and its *features*, and the use this *Petri net type definition* mechanism for defining some more standard *Petri types* such as timed and stochastic Petri nets. Another major issue is the definition of a module concept for PNML [15, 19, 20]. This setting should be powerful enough to encompass most structuring mechanisms. Hence, it will be possible to exchange modular and hierarchical models, using different structuring paradigms, between tools so as to apply their specific analysis techniques.

If you are interested in some of these issues, you might also want to join the work of WG19 of ISO/IEC JTC1 SC7 or the discussion list available at the PNML web pages. All ideas are welcome.

Acknowledgements The work on PNML and ISO/IEC 15909-2 has been going on over 10 years now and many people have been involved in the discussion and development process in different stages and different intensity. Since, it is hard to weigh the individual contributions, we thank all of them in alphabetical order: Joao Paulo Barros, Jean Bérubé, Jonathan Billington, Didier Buchs, Søren Christensen, Jörg Desel, Erik Fischer, Giuliana Franceschinis, Guillaume Giffo, Jun Ginbayashi, Kees van Hee, Nisse Husberg, Kurt Jensen, Matthias Jüngel, Albert Koelmans, Olaf Kummer, Kjeld Høyer Mortensen, Reinier Post, Wolfgang Reisig, Stefan Roch, Karsten Wolf, Christian Stehno, Kimmo Varpaaniemi, Michael Weber, Lisa Wells, Jan Martijn van der Werf, and Lukasz Zogolowek.

We would also like to thank ISO/IEC for their kind permission to publish this paper based on material of the upcoming standard ISO/IEC 15909.

References

[1] Bastide, R., Billington, J., Kindler, E., Kordon, F., Mortensen, K.H., eds.: Meeting on XML/SGML based Interchange Formats for Petri Nets, University of Aarhus, Dept. of Computer Science (2000)

[2] Jüngel, M., Kindler, E., Weber, M.: The Petri Net Markup Language. Petri Net Newsletter **59** (2000) 24–29

[3] Weber, M., Kindler, E.: The Petri Net Kernel. In Ehrig, H., Reisig, W., Rozenberg, G., Weber, H., eds.: Petri Net Technologies for Modeling Communication Based Systems. Volume 2472 of LNCS. Springer (2003) 109–123

[4] Billington, J., Christensen, S., van Hee, K., Kindler, E., Kummer, O., Petrucci, L., Post, R., Stehno, C., Weber, M.: The Petri Net Markup Language: Concepts, technology, and tools. In van der Aalst, W., Best, E., eds.: Application and Theory of Petri Nets 2003, 24^{th} International Conference. Volume 2679 of LNCS., Springer (2003) 483–505

[5] ISO/IEC: Software and Systems Engineering – High-level Petri Nets, Part 1: Concepts, Definitions and Graphical Notation, International Standard ISO/IEC 15909 (2004)

[6] Genrich, H.J., Lautenbach, K.: System modelling with high-level Petri nets. Theoretical Computer Science **13** (1981) 109–136

[7] Jensen, K.: Coloured Petri nets and invariant methods. Theoretical Computer Science **14** (1981) 317–336

[8] Berthomieu, B., Choquet, N., Colin, C., Loyer, B., Martin, J., Mauboussin, A.: Abstract Data Nets combining Petri nets and abstract data types for high level specification of distributed systems. In: Proceedings of VII European Workshop on Application and Theory of Petri Nets. (1986)

[9] Vautherin, J.: Parallel systems specifications with coloured Petri nets and algebraic specifications. In Rozenberg, G., ed.: Advances in Petri Nets. Volume 266 of LNCS. Springer-Verlag (1987) 293–308

[10] Billington, J.: Many-sorted high-level nets. In: Proceedings of the 3rd International Workshop on Petri Nets and Performance Models, IEEE Computer Society Press (1989) 166–179

[11] Reisig, W.: Petri nets and algebraic specifications. Theoretical Computer Science **80** (1991) 1–34

[12] Jensen, K., (Eds.), G.R.: High-level Petri Nets, Theory and Application. Springer-Verlag (1991)

[13] Jensen, K.: Coloured Petri Nets, Volume 1: Basic Concepts. EATCS Monographs on Theoretical Computer Science. Springer-Verlag (1992)

[14] Chiola, G., Dutheillet, C., Franceschinis, G., Haddad, S.: On well-formed coloured nets and their symbolic reachability graph. In Jensen, K., Rozenberg, G., eds.: Petri Nets: Theory and Application. (1991) 373–396

[15] Weber, M., Kindler, E.: The Petri Net Markup Language. In Ehrig, H., Reisig, W., Rozenberg, G., Weber, H., eds.: Petri Net Technologies for Modeling Communication Based Systems. Volume 2472 of LNCS. Springer (2003) 124–144

[16] Stehno, C.: Petri Net Markup Language: Implementation and Application. In Desel, J., Weske, M., eds.: Promise 2002. Lecture Notes in Informatics P-21., Gesellschaft für Informatik (2002) 18–30

[17] PNML team: PNML.org: The Petri Net Markup Language home page. (URL `http://www.pnml.org/`) 2009/06/8.

[18] Hillah, L., Kordon, F., Petrucci, L., Trèves, N.: Model engineering on Petri nets for ISO/IEC 15909-2: API framework for Petri net types metamodels. Petri Net Newsletter **69** (2005) 22–40

[19] Kindler, E.: Modular PNML revisited: Some ideas for strict typing. In: Proc. AWPN 2007, Koblenz, Germany. (2007)

[20] Kindler, E., Petrucci, L.: Towards a standard for modular Petri nets: A formalisation. In Franceschinis, G., Wolf, K., eds.: Application and Theory of Petri Nets 2009, Internat. Conference, Proceedings. Volume 5606 of LNCS., Springer-Verlag (2009) 43–62

A framework for the definition of variants of high-level Petri nets

E. Kindler[1] and L. Petrucci[2]

[1] Informatics and Mathematical Modelling
Technical University of Denmark (DTU)
DK-2800 Lyngby, DENMARK
eki@imm.dtu.dk
[2] LIPN, CNRS UMR 7030, Université Paris XIII
99, avenue Jean-Baptiste Clément
F-93430 Villetaneuse, FRANCE
laure.petrucci@lipn.univ-paris13.fr

Abstract. There exist many different variants of high-level Petri nets. Many differences between these variants, however, do not concern the features of the particular versions of Petri nets, but they concern the data types that can be defined and used in the different variants of high-level nets. One famous example of a restricted version of high-level nets are well-formed nets (which are currently standardised as symmetric nets in ISO/IEC 15909-1), which basically restrict the data types to finite sets with a very limited set of operations on them. Due to these restrictions, there exist some more powerful analysis algorithms for symmetric nets.
During the standardisation of high-level nets and some of their variations, it turned out that defining the legal data types and the operations on them is the most difficult part. In particular, these definitions become lengthy and mix Petri net specific issues with data-type specific issues, which often blocks the view for the really relevant parts. Even worse, supposedly simpler versions of high-level nets often are more difficult to define than high-level nets in general.
This paper introduces the concepts and the mathematical tools to ease the definition of new variants and versions of high-level Petri nets: a *framework* for defining variants of high-level nets. The main ingredient of this framework is the concept of *generators*, which we recently introduced for formalising modular PNML, and the newly introduced concept of *constructs*.

Keywords: High-level Petri nets, variations, well-formed nets, symmetric nets.

1 Introduction

There exist many different variants of high-level Petri nets. Some of them have special concepts or are different in the way they are formalised. However, in many cases, the differences between these versions are not with the actual Petri net features, but they concern the data types that can be defined or are built-in to the specific version of Petri nets. One example are well-formed nets [1], which are currently standardised under the name *symmetric nets* in an addendum to ISO/IEC 15909-1. The main idea of symmetric nets is to restrict the data types to finite sets with a very limited set of operations on them; in turn, these restrictions results in some powerful analysis techniques.

The problem with the many variants of high-level Petri nets is that the formalisations are very different, and the actual differences are blurred by mixing the conceptual differences with the standard definitions on a very low level of abstraction. In this paper, we introduce a mathematical framework for the definition of different versions of high-level Petri nets, which separates these issues and helps defining the supported constructs on an adequate level of abstraction. In the end, it is possible to define a specific version of high-level nets by three parameters, which can be defined independently from the actual definition of high-level nets. The main ingredients of this framework are generators, which have proven to be useful — if not necessary — for defining and using high-level Petri nets from modules [2], and the new concept of constructs, which will be used for characterising the legal constructs in the algebras.

In order to validate this framework, we give the definition of several versions of high-level Petri nets known from the literature.

2 An example

In order to illustrate the concepts of high-level nets and to discuss some of the features in which the various versions of high-level nets differ, we start with an example. The example is taken from [3, 4], which models a distributed algorithm for computing the minimal distance of every node (agent) to some distinguished root nodes in some communication network.

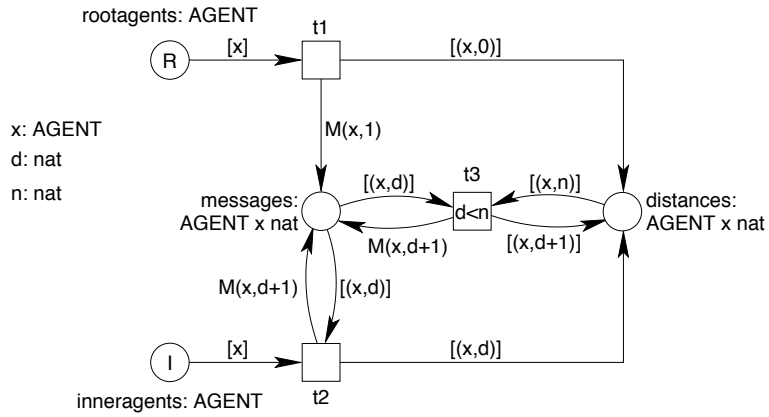


Fig. 1. Example: Minimal distance algorithm

Figure 1 show the algebraic Petri net modelling this algorithm. We assume that there is a *network* of *agents*; the agents could be represented by a set A and the network by a symmetric binary relation $N \subseteq A \times A$. Moreover, there is a distinguished set of *root agents* $R \subseteq A$. Initially, the root agents are represented on place *rootagents*, whereas the other agents, which are called *inner agents*, are represented on place *inneragents*. The algorithm is quite simple: Initially, every root agent sends a message to all its neighbours and stores distance 0 as its own distance; for each neighbour, the message says that it has a distance of at most 1 to some root node. The distance of an agent x is represented by a pair (x, d) on place distances, where x represents the agent, and d its distance. A message to an agent y is represented on the place *messages* by a pairs (y, d), where y represents the agent to which this message is addressed, and d the tentative shortest distance. Sending these initial messages is represented by transition t_1, where $M(x, 1)$ represents a set (resp. a multiset) of all the messages with value 1 to every neighbour of agent x.

Initially, the inner agents cannot do anything. They just wait for a message from one of their neighbours. Once a message arrives for an inner agent x with distance d, it takes that distance d and stores it for itself (represented as pair (x, d) on place *distances*). Moreover, it sends a message to all its neighbours with distance $d+1$, which is represented by $M(x, d+1)$. This behaviour is modelled by transition t_2. But, the inner agents are not finished yet. It might be that later an inner agent receives a distance d that is even shorter than its current distance n. In that case, it takes that shorter distance d as its new distance, and again informs all its neighbours about that. This behaviour is modelled by transition t_3.

Here, we will not discuss the algorithm further. But, we would like to point out some of the notations and concepts used in the algebraic Petri net in Fig. 1. First and foremost, we would like to point out that this is actually not an algebraic net; it is what we call an *algebraic net scheme* [3]. The reason is that the set of agents A, and the way they are connected N, may vary. A can be any set, and N can be any symmetric binary relation over A. This is why the set A is not directly used in the Petri net.

Instead, we use a sort symbol AGENT, which could be interpreted by different sets. Likewise the set of root agents is represented by a constant symbol R and the set of inner agents is represented by the symbol I, and the interpretation of these symbols might be different, depending on which are the root nodes and which are not. Likewise, the structure of the network is represented by the interpretation of the operation symbol M, which produces the messages for all the neighbours.

But, there is another interesting point here. The sort AGENT, the constants R and I, as well as the operation M are specific to this model (actually, they are specific to a class of algorithms, which we call network algorithms). Therefore, their syntax as well as as their legal interpretations need to be defined by the modeller explicitly. In contrast to that, the sort nat does not need to be defined, because it is a standard sort, and also the constants and operations on that sort are standard and have a standard interpretation, which the modeller would not need to define. Actually, the classical approach of algebraic Petri nets [5] forces a modeller to define these sorts and operators even though they are standard. And a modeller would also be forced to explicitly define the pairs for sorts and the boolean operations. In this paper, we will introduce a mechanism that helps avoiding this: Generators help to define the standard sorts and operations of a specific class of high-level nets in a simple and flexible way.

A third observation is that, for the sorts and operations that must be defined by the user, not all possible interpretations are legal. Sometimes, we would like to restrict the interpretations to sorts with a finite set, and also restrict the operations on them. In our example, AGENTS should be finite sets, and the operation M should only be those functions that represent a network (resp. the sending of messages in a network). To this end, this paper introduces the concept of *constructs*. Using this mechanism, we can for example restrict the user defined sorts to the ones that are legal in symmetric nets.

3 Basic Definitions

In this section, we formalise algebraic Petri nets and all the pre-requisites. We introduce the standard concepts of algebraic specifications [6] and of algebraic Petri nets [7–9, 5, 10]. The notation, however, is slightly adjusted for easing the readability of the concepts; the presentation follows the lines of [2] — streamlined a bit for the settings in this paper.

3.1 Basic notations

As usual, $\mathbb{N}$ stands for the set of *natural numbers* (including 0), and $\mathbb{B}$ stands for the set of *booleans*, i.e., $\mathbb{B} = \{\mathit{false}, \mathit{true}\}$. For some set A, A^+ denotes the set of all non-empty finite sequences over A. For some function $f : A \to B$ and some set C, the restriction of f to C is defined as the function $f|_C : A \cap C \to B$ with $f|_C(a) = f(a)$ for all $a \in A \cap C$. For two functions $f : A \to B$ and $g : C \to D$ with disjoint domains A and C, we define $f \cup g$ as the function $(f \cup g) : A \cup C \to B \cup D$ with $(f \cup g)(a) = f(a)$ for all $a \in A$ and $(f \cup g)(c) = g(c)$ for all $c \in C$.

For some set I, a set A together with a mapping $i : A \to I$ is an *I-indexed set* (A, i). The I-indexed set (A, i) is *finite* if A is finite. When i is understood from the context, we often use A for denoting the I-indexed set. For every $j \in I$, we define the set of all elements indexed by j: $A_j = \{a \in A \mid i(a) = j\}$. By definition, all A_j are disjoint. For an I-indexed set (A, i) and some set B, we define $(A, i) \cap B = (A \cap B, i|_B)$.

For some set A, a mapping $m : A \to \mathbb{N}$ is called a *multiset over A* if $\sum_{a \in A} m(a)$ is finite. The set of all multisets over A is denoted by $MS(A)$. For two multisets $m_1, m_2 \in MS(A)$, the operation $+$ is defined pointwise: $m = m_1 + m_2$ is defined by $m(a) = m_1(a) + m_2(a)$ for every $a \in A$. This way, the *addition operation* $+$ is lifted from the natural numbers to multisets. The *empty multiset* is denoted by $[]$ and defined by $[](a) = 0$ for all $a \in A$. This is in line with a special case of our notation that denotes a multiset by enumerating all its elements: $[a_1, a_2, \ldots, a_n]$ (where the number of occurrences of the same element is relevant).

3.2 Signatures and algebras

The idea of high-level nets is that there are different kinds of tokens, which are often called colours. Mathematically, the tokens can come from some set which is associated with a place. Different functions allow for manipulating them. In order to represent these sets and functions, some syntax must be introduced. Here, we use the approach of algebraic nets, where we use signatures for the syntax, and the associated algebras for the meaning.

Definition 1 (Signature). *A* signature $SIG = (S, O)$ *consists of a set of* sort symbols S *(often called sorts for short) and an* S^+*-indexed set of* operation symbols O *such that* S *and* O *are disjoint. The set* $S \cup O$ *is called the set of* symbols *of* SIG*. For some signature* SIG*, we denote the set of its sorts by* S^{SIG} *and the set of its operations by* O^{SIG}*.*

Sometimes, we want to restrict some signature $SIG = (S, (O, i))$ to a subset of symbols A. This is denoted by $SIG|_A$. In the definition, we take care that all the operation symbols operating on an eliminated sort are also eliminated: We define $SIG|_A = (S \cap A, (O', i'))$ where $O' = \{x \in O \mid i(x) \in (S \cap A)^+\}$ and $i' = i|_{O'}$.

Definition 2 (Signature extension). *A signature* SIG' extends a signature SIG *if, for some set* A*,* $SIG'|_A = SIG$*. This is denoted by* $SIG \subseteq SIG'$*. Let* $SIG = (S, O)$ *and* $SIG' = (S', O')$ *be two signatures with a disjoint set of symbols, then we define the union* $SIG \cup SIG' = (S \cup S', O \cup O')$*.*

By definition, $SIG \cup SIG'$ is a signature, which extends both SIG and SIG'.

Definition 3 (Signature homomorphism). *For two signatures* $SIG = (S, O)$ *and* $SIG' = (S', O')$*, a mapping* $\sigma : S \cup O \to S' \cup O'$ *is called a* signature homomorphism*, if for every* $s \in S$ *we have* $\sigma(s) \in S'$ *and for every* $o \in O_{s_1 \ldots s_n}$ *we have* $\sigma(o) \in O'_{\sigma(s_1)\ldots\sigma(s_n)}$*.*

Definition 4 (Algebra). *A* SIG-algebra $\mathcal{A}$ *assigns a* carrier set *to every sort of* SIG *and a* function *to every operation of* SIG*.*

Technically, an algebra $\mathcal{A}$ is a mapping such that, for every $s \in S$, $\mathcal{A}(s)$ is a set and, for every $o \in O_{s_1 \ldots s_n s_{n+1}}$, $\mathcal{A}(o)$ is a function with $\mathcal{A}(o) : \mathcal{A}(s_1) \times \ldots \times \mathcal{A}(s_n) \to \mathcal{A}(s_{n+1})$.

Definition 5 (Algebra extension). *Let* SIG *and* SIG' *be two signatures with* $SIG \subseteq SIG'$*, and let* $\mathcal{A}$ *be a* SIG*-algebra and* $\mathcal{A}'$ *be a* SIG'*-algebra. Algebra* $\mathcal{A}'$ extends *algebra* $\mathcal{A}$*, if* $\mathcal{A}'|_{S^{SIG} \cup O^{SIG}} = \mathcal{A}$*. If* $\mathcal{A}'$ extends $\mathcal{A}$*, we write* $\mathcal{A} \subseteq \mathcal{A}'$*.*

3.3 Variables and terms

The operations of a signature can be used to construct terms, which will be discussed in this section. We start with the definition of variables.

Definition 6 (Variables). *Let* $SIG = (S, O)$ *be a signature. An* S*-indexed set* X *is a set of* SIG-variables*, if* X *is disjoint from* O*.*
$\mathbb{V}^{SIG}$ *denotes the class of all* SIG*-variable sets.*

From the set of operations O of the signature and a set of SIG-variables X, we can construct *terms* of some sort s inductively.

Definition 7 (Terms). *Let* $SIG = (S, O)$ *be a signature and* X *be a set of* SIG-variables*. The set of all* SIG-terms *of sort* s *over a set of variables* X *is denoted by* $\mathbb{T}_s^{SIG}(X)$*. It is inductively defined as follows:*

- $X_s \subseteq \mathbb{T}_s^{SIG}(X)$.
- *For every operation symbol* $o \in O_{s_1 \ldots s_n s_{n+1}}$*, and, for every* k *with* $1 \leq k \leq n$*,* $t_k \in \mathbb{T}_{s_k}^{SIG}(X)$*, we have* $(o, t_1, \ldots, t_n) \in \mathbb{T}_{s_{n+1}}^{SIG}(X)$*.*

When SIG is clear from the context, we also write $\mathbb{T}_s(X)$ instead of $\mathbb{T}_s^{SIG}(X)$. The set of all terms is $\mathbb{T}^{SIG}(X) = \bigcup_{s \in S} \mathbb{T}_s^{SIG}(X)$. Terms without variables are called *ground terms* and are defined by $\mathbb{T}^{SIG} = \mathbb{T}(\emptyset)$ and by $\mathbb{T}_s^{SIG} = \mathbb{T}_s^{SIG}(\emptyset)$.

Sometimes, we need to refer to some terms with variables, but without specifically mentioning the set of variables. The set of such terms is denoted by $\mathbb{T}_s^{SIG}(\mathbb{V}^{SIG})$.

Note that, in practice, terms are often written $o(t_1, \ldots, t_n)$ to make clear that the operation is applied to the arguments. In order to emphasise the syntactical nature of terms, we use the tuple notation $(o, t_1, \ldots, t_n)$ in all our formal definitions.

3.4 Assignment and evaluation

Terms are a purely syntactical construct. In order to give them a meaning, they are evaluated in a given algebra. In order to evaluate terms with variables, we need to bind their variables to some value, which is called a binding or an *assignment*.

Definition 8 (Assignment and evaluation). *Let $SIG = (S, O)$ be a signature, $\mathcal{A}$ a SIG-algebra, and X a set of SIG-variables. An* assignment β of X in $\mathcal{A}$ *is a mapping such that, for every $s \in S$ and every $x \in X_s$, we have $\beta(x) \in \mathcal{A}(s)$.*

An assignment β of variables X in $\mathcal{A}$ can be inductively extended to a mapping $\overline{\beta}$ that applies to all terms $\mathbb{T}^{SIG}(X)$, which is called evaluation of terms in $\mathcal{A}$*:*

- *For every $x \in X$, we define $\overline{\beta}(x) = \beta(x)$.*
- *For every $o \in O_{s_1 \ldots s_{n+1}}$, and every $i \in \{1, \ldots, n\}$ and term $t_i \in \mathbb{T}_{s_i}^{SIG}(X)$, we define $\overline{\beta}((o, t_1, \ldots, t_n)) = \mathcal{A}(o)(\overline{\beta}(t_1), \ldots, \overline{\beta}(t_n))$.*

For an empty set of variables $\emptyset$, there is a unique assignment of $\emptyset$ to $\mathcal{A}$, which we denote with ϵ. The extension $\overline{\epsilon}$ can be used to evaluate ground terms, and is called ground evaluation.

3.5 Generators

In high-level nets and high-level net modules [2] in particular, we often have some sorts provided, and we need to construct other sorts from them in a standard way. For example, we would like to use the product over some existing sorts (see the example in Sect. 2); and, for every sort s, we also need a sort that represents the multiset sort over that sort, $ms(s)$. Moreover, the sets associated with these new sorts are defined based on the sets associated with the underlying sorts. For example, the set associated with $ms(s)$ is the set of all multisets over $\mathcal{A}(s)$, i. e. $MS(\mathcal{A}(s))$.

For that purpose, we need a mechanism for constructing new sorts and operations from some signature and a way to define their meaning. To this end, we introduce *generators*. A generator defines which new sorts and operators can be constructed out of existing sorts, and once the associated sets are known for every sort, what the meaning of the corresponding constructed sorts and operators should be. Since generators are needed anyway, we can also use them for introducing the standard sorts along with their operations (e. g. `nat` or *bool* in our example).

Definition 9 (Generator). *A* generator $G = (GS, GA)$ *consists of*

- *a* signature generator function *GS that, for any given signature $SIG = (S, O)$, returns a signature $GS(SIG)$ such that $SIG \subseteq GS(SIG)$; the signature $GS(SIG)$ is called the signature generated from SIG by the generator G;*
- *an* algebra generator function *GA that, for any SIG-algebra $\mathcal{A}$, returns a $GS(SIG)$-algebra such that the algebra $GA(\mathcal{A})$ extends algebra $\mathcal{A}$.*

In [2], we needed a single generator only, because we were dealing with a single version of Petri nets only. Here, we need different generators for the different versions of high-level nets. Therefore, we will define several generators and operators for constructing new generators out of the existing ones later in the paper. In order to give a feeling for the purpose of a generator, we use the one from [2]

as a first example here; we will introduce other ones later. The basic idea of this example generator $G = (GS, GA)$, is to include, in addition to the existing sorts of some algebra also the booleans, the associated multiset sort $ms(s)$ for every sort s, and all the product sorts. In order to emphasise the syntactical nature, and to distinguish the newly constructed sorts from already existing ones, we use the notation $(bool)$, (ms, s) and $(\times, s_1, \ldots, s_n)$ for these generated sorts. Likewise, the generator will generate the boolean constants $(true)$ and $(false)$ and the standard operations on booleans, the operation $([], s, n)$, which makes a multiset out of n elements, the tupling operation $((), s_1, \ldots, s_n)$, and the projection operation $(pr, i, s_1, \ldots, s_n)$ on the i-th element of a tuple.

Definition 10 (Sort generator). *Let $SIG = (S, O)$ be an arbitrary signature, then $GS(SIG) = (S', O')$ is defined as follows:*

- *S' is the least set for which the following conditions hold:*
  1. *$S \subseteq S'$,*
  2. *$(bool) \in S'$,*
  3. *$(ms, s) \in S'$ for every $s \in S'$, and*
  4. *$(\times, s_1, \ldots, s_n) \in S'$ for all sorts $s_1, \ldots, s_n \in S'$.*
- *O' is the least S'-indexed set for which the following conditions hold:*
  1. *$O \subseteq O'$,*
  2. *$(true), (false) \in O'_{(bool)}$,*
  3. *$(not) \in O'_{(bool)(bool)}$,*
  4. *$(and), (or) \in O'_{(bool)(bool)(bool)}$,*
  5. *$([], s, n) \in O'_{s \ldots s(ms,s)}$ for every sort $s \in S'$ and $n \in \mathbb{N}$, where the number of s elements in the index of O' is n,*
  6. *$(+, s) \in O'_{(ms,s)(ms,s)(ms,s)}$ for every $s \in S'$,*
  7. *$((), s_1, \ldots, s_n) \in O'_{s_1 \ldots s_n(\times, s_1, \ldots, s_n)}$ for all $s_1, \ldots, s_n \in S'$, and*
  8. *for every $0 \leq i \leq n$, $(pr, i, s_1, \ldots, s_n) \in O'_{(\times, s_1, \ldots, s_n)s_i}$.*

Definition 11 (Algebra generator). *Let $\mathcal{A}$ be a SIG-algebra with $SIG = (S, O)$ and let $GS(SIG) = (S', O')$. Then we define $GA(\mathcal{A})$ by:*

- *The mapping of the sorts of $GA(\mathcal{A})$ is defined as follows:*
  1. *$GA(\mathcal{A})|_S = \mathcal{A}|_S$,*
  2. *$GA(\mathcal{A})((bool)) = \mathbb{B}$,*
  3. *$GA(\mathcal{A})((ms, s)) = MS(GA(\mathcal{A})(s))$ for every sort $s \in S'$, and*
  4. *$GA(\mathcal{A})((\times, s_1, \ldots, s_n)) = GA(\mathcal{A})(s_1) \times \ldots \times GA(\mathcal{A})(s_n)$ for all sorts $s_1, \ldots, s_n \in S'$.*
- *The mapping of the operations of $GA(\mathcal{A})$ is defined as follows:*
  1. *$GA(\mathcal{A})|_O = \mathcal{A}|_O$,*
  2. *$GA(\mathcal{A})((true)) = true$ and $GA(\mathcal{A})((false)) = false$,*
  3. *$GA(\mathcal{A})((not)) = \neg$, where $\neg$ is the boolean negation function,*
  4. *$GA(\mathcal{A})((and)) = \wedge$ and $GA(\mathcal{A})((or)) = \vee$, where $\wedge$ and $\vee$ are the boolean conjunction and disjunction functions,*
  5. *$GA(\mathcal{A})(([], s, n))(a_1, \ldots, a_n) = [a_1, \ldots, a_n]$, for every $n \in \mathbb{N}$ and every sort $s \in S'$ and all $a_1, \ldots, a_n \in GA(\mathcal{A})(s)$; i. e. the multiset over s containing exactly the elements $a_1, \ldots, a_n$,*
  6. *$GA(\mathcal{A})((+, s)) = +$ for every sort $s \in S'$, where $+$ denotes the addition of two multisets over $GA(\mathcal{A})(s)$,*
  7. *$GA(\mathcal{A})(((), s_1, \ldots, s_n))(a_1, \ldots, a_n) = (a_1, \ldots, a_n)$ for all $s_1, \ldots, s_n \in S'$ and $a_1 \in GA(\mathcal{A})(s_1), \ldots, a_n \in GA(\mathcal{A})(s_n)$, i. e., the usual tupling, and*
  8. *$GA(\mathcal{A})((pr, i, s_1, \ldots, s_n))((a_1, \ldots, a_n)) = a_i$ for every $0 \leq i \leq n$ and $a_1 \in GA(\mathcal{A})(s_1), \ldots, a_n \in GA(\mathcal{A})(s_n)$; i. e., the usual projection function on the i-th component.*

Note that we need to make sure that all the symbols used in a basic signature *SIG* and introduced by the generators $GS(SIG)$ are interpreted in the same way. In some cases, this might restrict the legal signatures and algebras to which a generator can be applied. In order to avoid overly complex mathematics, we do not introduce an explicit mechanism for that; we rather construct and use generators in a systematic way. For example, in many cases the symbols used in *SIG* are flat and unstructured, whereas the symbols introduced in $GS(SIG)$ are tuples — some of them, like $(bool)$, are 1-tuples. Since this is needed only for making the mathematics work, our examples will use *bool* for $(bool)$ and $ms(s)$ for (ms, s). However, we stick to the technical notations $(bool)$ and (ms, s) in all formal definitions.

Definition 12 (Generator homomorphism). *A signature homomorphism σ from some signature SIG to some signature SIG' carries over to a signature homomorphism σ^G from $GS(SIG)$ to $GS(SIG')$ in a canonical way for any given generator G. In the case of our example, it is defined as follows:*

- *1.* $\sigma^G(s) = \sigma(s)$ *for every* $s \in S$,
  2. $\sigma^G((bool)) = (bool)$,
  3. $\sigma^G((ms, s)) = (ms, \sigma^G(s))$ *for every* $s \in S$, *and*
  4. $\sigma^G((\times, s_1, \ldots, s_n)) = (\times, \sigma^G(s_1), \ldots, \sigma^G(s_n))$ *for all* $s_1, \ldots, s_n \in S$.
- *1.* $\sigma^G(o) = \sigma(o)$ *for every operation* $o \in O$,
  2. $\sigma^G((true)) = (true)$ *and* $\sigma^G((false)) = (false)$,
  3. $\sigma^G((not)) = (not)$,
  4. $\sigma^G((and)) = (and)$, *and*
  $\sigma^G((or)) = (or)$,
  5. $\sigma^G(([], s, n)) = ([], \sigma^G(s), n)$ *for every sort* $s \in S$ *and every* $n \in \mathbb{N}$,
  6. $\sigma^G((+, s)) = (+, \sigma^G(s))$ *for every sort* $s \in S$,
  7. $\sigma^G((()), s_1, \ldots, s_n)) = ((), \sigma^G(s_1), \ldots, \sigma^G(s_n))$ *for all* $s_1, \ldots, s_n \in S$, *and*
  8. $\sigma^G((pr, i, s_1, \ldots, s_n)) = (pr, i, \sigma^G(s_1), \ldots, \sigma^G(s_n))$ *for every* $0 \leq i \leq n$ *and* $s_1, \ldots, s_n \in S$.

3.6 Nets

At last, we introduce the basic notion of *Petri nets*.

Definition 13 (Net). *A* net $N = (P, T, F)$ *consists of two disjoint sets P and T and a set of arcs $F \subseteq (P \times T) \cup (T \times P)$.*

4 Algebraic nets and their behaviour

Now we are prepared to give a first definition of algebraic nets. This definition will be refined later, in order to make it more flexible for defining an algebraic net of a particular kind. In Sect. 4.1, we define algebraic nets; in Sect. 4.2, we define their behaviour. Note that the focus of this paper is not on behaviour; but for completeness sake, we do not want to introduce a formal definition of algebraic nets without a definition of their behaviour.

4.1 Algebraic nets

For a clear separation between syntax and semantics, we distinguish between *algebraic net schemes* and *algebraic nets* [3]. Later we will define different versions of high-level nets, where the generators are one of the main defining factors of a version. For now, we just use the fixed generator G as defined in Sect. 3.5.

By contrast to most classical definitions of algebraic Petri nets and by contrast to our example, we formalise a version of high-level nets, where the scope of a variable is a transition (inspired by [11]). Note that is not a fundamental change; we even have the impression that many people think of variables in Petri nets in this way even when variables are declared globally. But since this more local scope of variables is slightly more complicated to formalise and to use, most formal definitions do not take that view. However, foreseeing some future extensions in the work of ISO/IEC 15909, we deam it necessary to be prepared for this in our formal definition.

Definition 14 (Algebraic net scheme).
An algebraic net scheme *is a tuple* $\Sigma = (N, SIG, sort, vars, l, c, i)$ *consisting of:*

1. *a* net $N = (P, T, F)$,
2. *a* signature SIG,
3. *a* place sort *mapping* $sort : P \to S^{GS(SIG)}$,
4. *a* transition variable *mapping* $vars : T \to \mathbb{V}^{GS(SIG)}$
5. *an* arc label *mapping* $l : F \to \mathbb{T}^{GS(SIG)}(\mathbb{V}^{GS(SIG)})$ *such that:*
   - *for all* $(p,t) \in F \cap (P \times T) : l((p,t)) \in \mathbb{T}^{GS(SIG)}_{(ms,sort(p))}(vars(t))$
   - *for all* $(t,p) \in F \cap (T \times P) : l((t,p)) \in \mathbb{T}^{GS(SIG)}_{(ms,sort(p))}(vars(t))$,
6. *a* transition condition *mapping* $c : T \to \mathbb{T}^{GS(SIG)}_{(bool)}(\mathbb{V}^{GS(SIG)})$ *such that* $c(t) \in \mathbb{T}^{GS(SIG)}_{(bool)}(vars(t))$ *for every* $t \in T$,
7. *an* initial marking $i : P \to \mathbb{T}^{GS(SIG)}$ *such that,* $i(p) \in \mathbb{T}^{GS(SIG)}_{(ms,sort(p))}$ *for every place* $p \in P$.

The mapping *sort* assigns a sort to each place, which defines the type of its tokens. The mapping *vars* defines for each transition the set of its variables. The annotations of all arcs attached to a transition may use only these variables; the same applies for the transition condition. Condition 5 formulates this restriction of the variables of the arc annotations as well as the restriction that the arc annotation must denote a multiset over the attached place type. Condition 7 guarantees that the term for the inititial marking denotes a multiset of the respective type.

Definition 15 (Algebraic net). *An* algebraic net *is pair* $(\Sigma, \mathcal{A})$*, where* Σ *is an algebraic net scheme equipped with a* SIG*-algebra* $\mathcal{A}$.

4.2 Behaviour of algebraic nets

For defining the behaviour of an algebraic net, we first need to define markings.

Definition 16 (Marking).
Let $(\Sigma, \mathcal{A})$ *be an algebraic net with* $\Sigma = (N, SIG, sort, vars, l, c, i)$ *and* $N = (P, T, F)$*. A* marking m *of* $(\Sigma, \mathcal{A})$ *is a mapping such that for every place* $p \in P$*, we have* $m(p) \in MS(\mathcal{A}(sort(p)))$.

The operation $+$ *on multisets can be lifted to markings by defining* $m = m_1 + m_2$ *by* $m(p) = m_1(p) + m_2(p)$ *for every place* $p \in P$.

The firing mode of a transition is the set of assignments to its variables such that the transition condition evaluates to true.

Definition 17 (Firing mode).
Let $(\Sigma, \mathcal{A})$ *be an algebraic net with* $\Sigma = (N, SIG, sort, vars, l, c, i)$ *and* $N = (P, T, F)$.

For some transition $t \in T$*, an assignment* β *of the variables* $vars(t)$ *in* $\mathcal{A}$ *is called a* transition mode *of transition* t*, if* $\overline{\beta}(c(t)) = true$.

For any transition t *and any firing mode* β *of* t*, we define the markings* ${}^{-}t_\beta$ *and* t^{+}_β *of* $(\Sigma, \mathcal{A})$ *as follows: For every place* $p \in P$

$$ {}^{-}t_\beta(p) = \begin{cases} \overline{\beta}(l(p,t)) & \text{if } (p,t) \in F \\ [] & \text{otherwise} \end{cases} $$

and

$$ t^{+}_\beta(p) = \begin{cases} \overline{\beta}(l(t,p)) & \text{if } (t,p) \in F \\ [] & \text{otherwise} \end{cases} $$

Next, we define when and how two markings are reachable from each other by a transition in a firing mode.

Definition 18 (Firing rule).
Let $(\Sigma, \mathcal{A})$ be an algebraic net with $\Sigma = (N, SIG, sort, vars, l, c, i)$ and $N = (P, T, F)$. Moreover, let $t \in T$ be a transition of N, β a firing mode of t, and let m_1 and m_2 be two markings. We say that m_2 is reachable from m_1 by firing t in mode β if there exists a marking m' such that $m_1 = {}^-t_\beta + m'$ and $m_2 = m' + t^+_\beta$. Then, we write $m_1 \xrightarrow{t,\beta} m_2$.

If there exists a sequence $m_1 \xrightarrow{t_1,\beta_1} m_2 \longrightarrow \ldots \longrightarrow m_n \xrightarrow{t_n,\beta_n} m_{n+1}$, we write $m_1 \xrightarrow{*} m_{n+1}$ and say that m_{n+1} is reachable from m_1 *in $(\Sigma, \mathcal{A})$.*

In many publications on Petri nets, $m_1 \xrightarrow{t,\beta} m_2$ would be formalised in a different way by $m_1 \geq {}^-t_\beta$ and $m_2 = (m_1 - {}^-t_\beta) + t^+_\beta$. But, this would require to define the comparision operator $\geq$ on markings and the subtraction operation $-$ first. With our definition, we could avoid that. The only operation necessary for defining the firing rule of Petri nets is the addition operation on markings, $+$.

From these concepts, we can now define the reachability graph as the semantics of an algebraic net.

Definition 19 (Reachability graph).
Let $(\Sigma, \mathcal{A})$ be an algebraic net with $\Sigma = (N, SIG, sort, vars, l, c, i)$ and $N = (P, T, F)$.

We define the initial marking m_0 *of $(\Sigma, \mathcal{A})$ by $m_0(p) = \overline{\epsilon}(i(p))$ for each $p \in P$. We define the set of* reachable markings *of $(\Sigma, \mathcal{A})$ by $\mathcal{M} = \{m \mid m_0 \xrightarrow{*} m\}$. The relation*

$$\mathcal{R} = \{(m_1, (t, \beta), m_2) \mid m_1, m_2 \in \mathcal{M}, t \in T, \text{and } \beta \text{ a mode of } t \text{ with } m_1 \xrightarrow{t,\beta} m_2\}$$

is called the reachability relation *of $(\Sigma, \mathcal{A})$.*

$\Gamma = (\mathcal{M}, m_0, \mathcal{R})$ is called the reachability graph of $(\Sigma, \mathcal{A})$.

5 Generators

As pointed out earlier, different generators can be used for defining different classes of high-level Petri nets. In order to define the respective generators, we define some basic generators and some operations on generators for defining new generators out of existing ones.

5.1 Basic generators

We start with defining some of the basic generators. In these definitions, we assume that $SIG = (S, O)$ is some signature and $\mathcal{A}$ some SIG-algebra.

Identity For technical reasons, we start with introducing the simplest generator of all: the identity generator ID, which does not add any new sorts or operators. This generator is defined by $ID = (ID_{SIG}, ID_{ALG})$, where $ID_{SIG}(SIG) = SIG$ for all signatures SIG. By definition (see Def. 9), ID_{ALG} is then uniquely defined by: $ID_{ALG}(\mathcal{A})(x) = \mathcal{A}(x)$ for all SIG-algebras $\mathcal{A}$ and all symbols x of SIG.

Booleans The generator $BOOL = (BOOL_{SIG}, BOOL_{ALG})$ adds the booleans and their operations to an existing signature: We define $BOOL_{SIG}(SIG) = (S', O')$ by $S' = S \cup \widehat{S}$ with $\widehat{S} = \{(bool)\}$ and $O' = O \cup \widehat{O}$ with $\widehat{O} = \{(true), (false), (not), (and), (or)\}$ with $(true), (false) \in O'_{(bool)}$, $(not) \in O'_{(bool)(bool)}$, and $(and), (or) \in O'_{(bool)(bool)(bool)}$.

We define $BOOL_{ALG}(\mathcal{A})$ by $BOOL_{ALG}(\mathcal{A})(x) = \mathcal{A}(x)$ for every symbol x of $SIG \setminus (\widehat{S} \cup \widehat{O})$. For the symbols $x \in \widehat{S} \cup \widehat{O}$, we define: $BOOL_{ALG}(\mathcal{A})((bool)) = \mathbb{B}$, $BOOL_{ALG}(\mathcal{A})((true)) = true$, $BOOL_{ALG}(\mathcal{A})((false)) = false$, $BOOL_{ALG}(\mathcal{A})((not)) = \neg$, where $\neg$ is the boolean negation function, $BOOL_{ALG}(\mathcal{A})((and)) = \wedge$ and $BOOL_{ALG}(\mathcal{A})((or)) = \vee$, where $\wedge$ and $\vee$ are the boolean conjunction and disjunction functions.

Note that, for some signature SIG, a SIG-algebra $\mathcal{A}$, and some symbol x of SIG, it could happen that $GA(\mathcal{A})(x) \neq \mathcal{A}(x)$. For example, if $(bool)$ occured in SIG but with a completely different meaning. This way, this generator would rule out this algebra as illegal; and we will use this mechanism for enforcing that symbols defined by the generators will be used in this interpretation only.

Restricted booleans Sometimes, we do not even want the full power of booleans. This is for example the case for *Place/Transition nets in high-level net notation* as defined in Part 2 of ISO/IEC 15909. The syntactical definition, however, requires that the trivial transition conditions exist. Therefore, we introduce a generator that introduces a restricted version of the booleans with sort *bool* and only the constant *true*, which can be used in this setting. We thus call this generator $TRUE$. The generator $TRUE = (TRUE_{SIG}, TRUE_{ALG})$ is defined by $TRUE_{SIG}(SIG) = (S', O')$ where $S' = S \cup \widehat{S}$ with $\widehat{S} = \{(bool)\}$ and $O' = O \cup \widehat{O}$ with $\widehat{O} = \{(true)\}$ with $(true) \in O'_{(bool)}$. We define $TRUE_{ALG}(\mathcal{A})(x) = \mathcal{A}(x)$ for every symbol x of $SIG \setminus (\widehat{S} \cup \widehat{O})$. For the other symbols $x \in \widehat{S} \cup \widehat{O}$, we define: $BOOL_{ALG}(\mathcal{A})((bool)) = \mathbb{B}$ and $BOOL_{ALG}(\mathcal{A})((true)) = true$.

Black tokens For representing Place/Transition nets, we also need a sort that represents the black tokens, which is often called *dots*. The generator DOT adds this sort and a single constant to a signature and the respective algebra. We define the generator $DOT = (DOT_{SIG}, DOT_{ALG})$ by $DOT_{SIG}(SIG) = (S', O')$ where $S' = S \cup \widehat{S}$ with $\widehat{S} = \{(dots)\}$ and $O' = O \cup \widehat{O}$ with $\widehat{O} = \{(dot)\}$ with $(dot) \in O'_{(dots)}$. We define $DOT_{ALG}(\mathcal{A})(x) = \mathcal{A}(x)$ for every symbol x of $SIG \setminus (\widehat{S} \cup \widehat{O})$. For the other symbols, we define: $BOOL_{ALG}(\mathcal{A})((dots)) = \{\bullet\}$ and $BOOL_{ALG}(\mathcal{A})((dot)) = \bullet$.

Natural numbers The generator $NAT = (NAT_{SIG}, NAT_{ALG})$ defines the natural numbers. $NAT_{SIG}(SIG) = (S', O')$ is defined by $S' = S \cup \widehat{S}$ with $\widehat{S} = \{(nat)\}$ and $O' = O \cup \widehat{O}$ with $\widehat{O} = \{(0), (succ)\}$ with $(0) \in O'_{(nat)}$, and $(succ) \in O'_{(nat)(nat)}$. We define $NAT_{ALG}(\mathcal{A})$ by $NAT_{ALG}(\mathcal{A})(x) = \mathcal{A}(x)$ for every symbol x of $SIG \setminus (\widehat{S} \cup \widehat{O})$. For the other symbols, we define: $NAT_{ALG}(\mathcal{A})((nat)) = \mathbb{N}$, $NAT_{ALG}(\mathcal{A})((0)) = 0$ and $NAT_{ALG}(\mathcal{A})((succ)) = ++$ where $++$ is the successor operation.

Multisets All versions of high-level Petri nets need some way of denoting multisets. To this end, we define a generator that adds the multisets over the given sorts to the signature and algebra: The generator $MULT = (MULT_{SIG}, MULT_{ALG})$ adds the constructs that are needed for constructing the multisets over the sorts of an algebra. Here, we confine ourselves to a minimal version. Later, we introduce a generator with some more operations on multisets: We define $MULT_{SIG}(SIG) = (S', O')$; where $S' = S \cup \widehat{S}$ with $\widehat{S} = \{(ms, s) \mid s \in S\}$ and $O' = O \cup \widehat{O}$ with $\widehat{O} = \{([], s, n) \mid s \in S, n \in \mathbb{N}\} \cup \{(+, s) \mid s \in S\}$, where $([], s, n) \in O'_{s \ldots s(ms,s)}$ and $(+, s) \in O'_{(ms,s)(ms,s)}$.

We define $MULT_{ALG}(\mathcal{A})$ by $MULT_{ALG}(\mathcal{A})(x) = \mathcal{A}(x)$ for every symbol x of $SIG \setminus (\widehat{S} \cup \widehat{O})$. For the other symbols, we define: $MULT_{ALG}(\mathcal{A})((ms, s)) = MS(\mathcal{A}(s))$, $MULT_{ALG}(\mathcal{A})(([], s, n))(a_1, \ldots, a_n) = [a_1, \ldots, a_n]$, and $MULT_{ALG}(\mathcal{A})((+, s)) = +$, where $+$ is the addition operation on multisets $MS(\mathcal{A}(s))$.

Extended multisets Some versions of high-level Petri nets have more powerful operations on multisets and for constructing multisets. To this end, we could use a generator that adds these operations. We denote this generator for extended multisets by $MULTX = (MULTX_{SIG}, MULTX_{ALG})$, but we leave the definition open, since it is yet to be decided which operations should be in there. We just assume that $MULT_{SIG}(SIG) \subseteq MULTX_{SIG}(SIG)$ and $MULT_{ALG}(\mathcal{A}) \subseteq MULTX_{ALG}(\mathcal{A})$.

One special extension of the multisets, which we need later for defining symmetric nets with bags is $MULTB$. This generator $MULTB = (MULTB_{SIG}, MULTB_{ALG})$ is defined by $MULTB_{SIG}(SIG) = MULT_{SIG}(SIG) \cup (\emptyset, \{(card), (unique)\})$ where $(card) \in O_{(ms,s)(nat)}$, and $(unique) \in O_{(ms,s)(bool)}$. We define $MULTB_{ALG}(\mathcal{A})$ by $MULTB_{ALG}(\mathcal{A})(x) = \mathcal{A}(x)$ for every symbol x of $MULT_{SIG}$. For the other symbols, we define: $MULTB_{ALG}(\mathcal{A})((card))([a_1, \ldots, a_n]) = n$, i. e. the number of elements in the multiset. $MULTB_{ALG}(\mathcal{A})((unique))([a_1, \ldots, a_n])$ is the operator checking that the multiset is actually a set (i. e. there is at most one occurrence of each element).

Products The generator $PROD = (PROD_{SIG}, PROD_{ALG})$ adds all products of the existing sorts and some operations on these products: We define $PROD_{SIG}(SIG) = (S', O')$; where $S' = S \cup \widehat{S}$ with $\widehat{S} = \{(\times, s_1, \ldots, s_n) \mid n \in \mathbb{N}, s_1, \ldots, s_n \in S\}$ and $O' = O \cup \widehat{O}$ with $\widehat{O} = \{((), s_1, \ldots, s_n) \mid n \in \mathbb{N}, s_1, \ldots, s_n \in$

$S\} \cup \{(pr, i, s_1, \ldots, s_n) \mid n \in \mathbb{N}, i \in \mathbb{N}, 1 \leq i \leq n, s_1, \ldots, s_n \in S\}$ with $((), s_1, \ldots, s_n) \in O'_{s_1 \ldots s_n (\times, s_1, \ldots, s_n)}$ and $(pr, i, s_1, \ldots, s_n) \in O'_{(\times, s_1, \ldots, s_n) s_i}$.

We define $PROD_{ALG}(\mathcal{A})$ by $PROD_{ALG}(\mathcal{A})(x) = \mathcal{A}(x)$ for every symbol x of $SIG \setminus (\widehat{S} \cup \widehat{O})$. For the other symbols, we define: $PROD_{ALG}(\mathcal{A})((\times, s_1, \ldots, s_n)) = \mathcal{A}(s_1) \times \ldots \times \mathcal{A}(s_n)$, $PROD_{ALG}(\mathcal{A})(((), s_1, \ldots, s_n))(a_1, \ldots, a_n) = (a_1, \ldots, a_n)$ for all $s_1, \ldots, s_n \in S$ and $a_1 \in \mathcal{A}(s_1), \ldots, a_n \in \mathcal{A}(s_n)$, and $PROD_{ALG}(\mathcal{A})((pr, i, s_1, \ldots, s_n))(a_1, \ldots, a_n) = a_i$.

Note that *PROD* defines only products of the sorts that exist in the input signature already. It does not define products of products recursively. This recursive construction of products is not necessary here, since we can achieve this by the closure construction on this generator, which will be defined later.

Other sorts and operations Note that there could be many other generators that add some of the commonly used sorts and operators, such as the integers, strings, or even more generic sorts like queues or lists over existing sorts. Since, this paper is more on the framework then on actually defining and classifying all the existing versions of high-level nets, we do not need them here, and therefore do not formally define them here.

5.2 Constructions on generators

From existing basic generators, we can now built more complex generators. The operations needed in this paper are the composition $\circ$, the union $\cup$ and the closure construct *.

Sequential composition The sequential composition of two generators $G_1 = (GS_1, GA_1)$ and $G_2 = (GS_2, GA_2)$ is defined as the function composition, i. e. $G_2 \circ G_1 = (GS_2 \circ GS_1, GA_2 \circ GA_1)$. For some signature *SIG* and some *SIG*-algebra $\mathcal{A}$, this means $(GS_2 \circ GS_1)(SIG) = GS_2(GS_1(SIG))$ and $(GA_2 \circ GA_1)(\mathcal{A}) = GA_2(GA_1(\mathcal{A}))$.

Union Likewise, the union $\cup$ is the union of the two mappings $G_1 \cup G_2 = (GS_1 \cup GS_2, GA_1 \cup GA_2)$. Note that we do not require the two generators to add disjoint symbols; actually, the way generators are defined, both generators will add the original symbols, which is why they are not disjoint. But, we assume (resp. we use and combine generators only in such a way) that they both agree on the meaning of these symbols, i. e. the generated algebras will assign the same meaning to these symbols.

Closure The closure of a generator applies the generator over and over again. Let G, be some generator. We define $G^0 = ID$ and, for each $i \in \mathbb{N}$, we define $G^{i+1} = G \circ G^i$. Then we define $G^* = \bigcup_{i \in \mathbb{N}} G^i$.

For example, we can now define the generator G from Sect. 3.5 from the basic generators above using the following constructions: $G = (MULT \cup PROD)^* \circ BOOL$.

6 Constructs

Some variants of high-level Petri nets differ in the data structures that may be used in the underlying algebras. For example, *symmetric nets* allow for data types with finite domains and a very limited set of operations only. It would be possible to use the concept of generators with some sophisticated and parameterised sort definitions, for explicitly introducing these data types. This however, would introduce much syntactical overhead. Due to this overhead, it would also be very tedious to check whether the generators really do what they are supposed to do, and designing and validating such generators would need much care and experience.

This can be alleviated, if we allow to directly provide the respective algebras in the definition of the algebraic net. This however requires a systematic way to characterise the algebras that are legal for a specific version of high-level Petri nets and those which are not. In this section, we will introduce the concepts for doing so: We slightly extend the concept of signatures to define its syntactical part, along with a set of legal algebras. We call them *constructs*. Then a variant of high-level Petri nets can be defined by a set of such constructs.

6.1 Formal definition of constructs

As mentioned above, a construct is, basically, a signature defining the sorts and operators of the construct. In addition to that, we need to identify the *role* of the sorts and operators, or in general the role of symbols. One role could, for example, be that it is a standard symbol (which is defined via some generator); for example the sort booleans or some of the standard operations on it, which have a fixed meaning in all the algebras and may even occur in different constructs with that same meaning. Another role can be that the symbol must not be used by or occur in other constructs, with the same role, which we call *disjoint*; i. e. these symbols of different constructs do not overlap with other constructs. The precise meaning of these roles will be made clear later in Def. 21. For now, we allow to make these roles explicit.

One of the tricky parts of this definition is that we need to take care that the symbols with a fixed interpretation actually have a fixed interpretation. To this end, we exploit the generators again which define the symbols with a fixed meaning. In this definition of constructs, we need to make sure that a construct is compatible with the used generator.

Definition 20 (Construct).
Let $G = (GS, GA)$ be some generator. A construct *$CON = (SIG, F, D, \mathbb{A})$ consists of a signature SIG, two subsets F and D of symbols of SIG and a class of SIG-algebras $\mathbb{A}$, such that*

1. *$GS(SIG|_{\overline{F}}) \supseteq SIG|_F$ and*
2. *for every $\mathcal{A} \in \mathbb{A}$ we have $GA(\mathcal{A}|_{\overline{F}})|_F = \mathcal{A}|_F$,*

where $\overline{F}$ is the set of all symbols of SIG except the symbols of F. The symbols of F are called the fixed symbols *of CON and the symbols of D are called the* disjoint *symbols of CON.*

The class of all constructs with respect to a generator G is denoted by $\mathcal{C}_G$.

Basically, a construct consists of a signature, and some algebras which give all the legal interpretations. The disjoint symbols D define which symbols may not overlap with other constructs, which will be defined in Def. 21. The fixed symbols F, are the symbols with a fixed meaning. Technically, we require that the fixed symbols in the algebra are the ones which would be added by the generator (condition 1) and also have the same meaning in all algebras as they have in the generated parts (condition 2). Note that we use $\supseteq$, because the generator could introduce many more symbols than the ones occuring in the signature (typically, there are infinitely many symbols in the generator but only finitely many in a construct).

Now, a set of such constructs can be used for characterising some specific algebras as defined below.

Definition 21 (Construct mapping and legal algebras).
Let $G = (GS, GA)$ be some generator and $\mathcal{C}$ be a set of constructs, SIG' be a signature, and $\mathcal{A}'$ be a SIG'-algebra.

A construct mapping *$\mathcal{H}$ for $\mathcal{C}$ and SIG is a set of pairs (CON, h), where $CON \in \mathcal{C}$ is a construct with $CON = (SIG, F, D, \mathbb{A})$ and $h : SIG \to SIG'$ is a signature homormorphism such that the following conditions are met:*

1. *$(h|_{\overline{F}})^G|_F = h|_F$ (i. e. the homomorphism respects the interpretation of the fixed symbols as defined by the generator G).*
2. *For all signatures $SIG'' \subseteq SIG'$ with $SIG' \subseteq GS(SIG'')$, for all $((SIG, F, D, \mathbb{A}), h) \in \mathcal{H}$, and for all symbols x of SIG with $h(x)$ in SIG' but not in SIG'', we have $x \in F$ (i. e. symbols with an interpretation that comes from a generator are a fixed symbol of the construct).*
3. *For all $(CON_1, h_1), (CON_2, h_2) \in \mathcal{H}$ and all symbols $x \in D_1$ and $y \in D_2$ with $h_1(x) = h_2(y)$, we have $CON_1 = CON_2$, $h_1 = h_2$ and $x = y$ (i. e. disjoint symbols of two different constructs do not overlap in SIG').*
4. *For all symbols x' of SIG', there exists a $(CON, h) \in \mathcal{H}$ and a symbol $x \in SIG$ such that $h(x) = x'$ (i. e. all symbols in the signature SIG' are part of at least one construct).*
5. *For every $(CON, h) \in \mathcal{H}$, we have $\mathcal{A}' \circ h \in \mathbb{A}$ (i. e. the part of the algebra corresponding to this construct is one allowed by this construct).*

Then, the algebra $\mathcal{A}'$ is said to be a legally constructed algebra *with respect to $\mathcal{C}$ and $\mathcal{H}$. We also say that $\mathcal{H}$ is a construct mapping from $\mathcal{C}$ to $\mathcal{A}'$.*

The main idea of this definition is that in a legal algebra $\mathcal{A}'$, all parts come from some construct. Condition 1 says that the interpretation of the fixed symbols of a construct is the one defined by the generator if these symbols were not there. Condition 2 makes sure that symbols that have a fixed interpretation according to a generator, have been defined as such in the construct. Condition 3 says that the disjoint symbols of all constructs are mapped to different symbols in SIG', so the constructs do not overlap on these symbols. Condition 4 says that all symbols of SIG' result from one of the constructs, i. e. there are no symbols of SIG' that are not injected by any construct mapping to SIG'. At last, condition 5 says that the semantics of the symbols comes from the construct. Note that in condition 5, h is a signature homomorphism from SIG to SIG'; therefore, $\mathcal{A}' \circ h$ is a SIG-algebra, and exactly this 'part' of SIG must be an algebra allowed by the construct (i. e. an algebra from $\mathbb{A}$).

6.2 Examples of constructs

In this section, we will define some examples of constructs, which will be used for defining symmetric nets and some other versions of high-level nets in Sect. 7.2.

Unordered sets The construct of *unordered sets* UO is defined by $UO = (UOSIG, \emptyset, \{u\}, \mathbb{A}_u)$, where $UOSIG = (\{u\}, \emptyset)$ and $\mathbb{A}_u$ contains every $UOSIG$-algebra $\mathcal{A}$ such that $\mathcal{A}(u)$ is a finite set.

This is basically defining a sort symbol u, which in all legal interpretations must be a finite set. Since u is in the set of disjoint symbols of UO, this sort symbol may not overlap with disjoint symbols of the other constructs.

Linearly ordered sets The construct of *linearly ordered sets* LO is defined by $LO = (LOSIG, \{(bool)\}, \{o, lt\}, \mathbb{A}_o)$, where $LOSIG = (\{o, (bool)\}, (\{lt\}, i))$ with $i(lt) = o\, o\, (bool)$ and where $\mathbb{A}_o$ contains every $LOSIG$-algebra $\mathcal{A}$ such that $\mathcal{A}(o)$ is a finite set and $\mathcal{A}(lt)$ defines an irreflexive total order on $\mathcal{A}(o)$.

Similarly to unordered sets, linearly order sets define a sort o which is associated with a finite set. In addition, linearly ordered sets define an operation lt which defines a total (irreflexive) order on o. Technically, lt is an operation into the booleans. This is where the fixed symbols come in: $(bool)$ is a fixed symbol of this construct, and therefore must obtain the interpretaion as defined in the generator. Note that this sort $(bool)$ might be used by different constructs. The symbols o and lt are disjoint symbols.

Cyclic sets The construct of *cyclic sets* CS is defined by $CS = (CSSIG, \emptyset, \{c, pred, succ\}, \mathbb{A}_c)$, where $CSSIG = (\{c\}, (\{pred, succ\}, i))$ with $i(pred) = i(succ) = c\, c$ and where $\mathbb{A}_c$ contains every $CSSIG$-algebra $\mathcal{A}$ such that $\mathcal{A}(c)$ is a finite set and $\mathcal{A}(succ)$ is a function defining a cycle on all elements of $\mathcal{A}(c)$, and $\mathcal{A}(pred)$ is the inverse of $\mathcal{A}(succ)$.

The construct of cyclic sets is very similar to linearly order sets. The elements are arranged in a cycle, which is expressed by the operators *succ* and *pred*.

Partitions The construct of *partitions* PAR is defined by $PAR = (PARSIG, \emptyset, \{p, f\}, \mathbb{A}_p)$, where $PARSIG = (\{s, p\}, (\{f\}, i))$ with $i(f) = s\, p$ and where $\mathbb{A}_p$ contains every $PARSIG$-algebra $\mathcal{A}$ such that $\mathcal{A}(p)$ is a finite set and $\mathcal{A}(f)$ is a surjective function.

Equality The construct of *equality* EQ is defined by $EQ = (EQSIG, \{(bool)\}, \emptyset, \mathbb{A}_e)$, where $EQSIG = (\{e, (bool)\}, (\{eq\}, i))$ with $i(eq) = e\, e\, (bool)$ and where $\mathbb{A}_e$ contains every $EQSIG$-algebra $\mathcal{A}$ such that $\mathcal{A}(eq)$ is the equality function on $\mathcal{A}(e)$ (i. e. it evaluates to true if both arguments are the same).

Note that, in this construct, the sort symbol e as well as the operation symbol eq expressing the equality are not in the set of the construct's disjoint symbols. This way, the equality operation may be added to any sort (even to the ones coming from other constructs).

All Likewise, the following construct allows to introduce a constant for a sort s that denotes a multiset in which each element of the set associated with this sort occurs exactly once, which is often denoted by all_s; we use, the notation (all, s) here.

The construct ALL is defined by $ALL = (ALLSIG, \{(ms, s)\}, \emptyset, \mathbb{A}_a)$, where $ALLSIG = (\{s, (ms, s)\}, (\{all\}, i))$ with $i(all) = (ms, s)$ and where $\mathbb{A}_a$ contains every $ALLSIG$-algebra $\mathcal{A}$ with $\mathcal{A}(s) = \{a_1, \ldots, a_n\}$ such that $\mathcal{A}((ms, s)) = MS(\mathcal{A}(s))$ and $\mathcal{A}(all) = [a_1, \ldots, a_n]$.

Simple multisets The construct $MULTSNB$ introduces the construction of a multiset element, the complement of a multiset w.r.t. its carrier set, the difference between two multisets, and for a sort s a constant singleton multiset with (all, s) as its only element. This construct is defined by $MULTSNB = (MULTSNBSIG, \{s, (ms, s)\}, \emptyset, \mathbb{A}_m)$, where $MULTSNBSIG = (\{s, (ms, s), (ms, (ms, s))\}, (\{\mathbf{\{\}}, \tilde{}, \setminus, whole\}, i))$ with $i(\mathbf{\{\}}) = (ms, s)(ms, (ms, s))$, $i(\tilde{}) = (ms, s)(ms, s)$, $i(\setminus) = (ms, s)(ms, s)(ms, s)$, $i(whole) = (ms, (ms, s))$, and where $\mathbb{A}_m$ contains every $MULTSNBSIG$-algebra $\mathcal{A}$ with $\mathcal{A}(\mathbf{\{\}})([b_1, \ldots, b_n]) = [[b_1, \ldots, b_n]]$, $\mathcal{A}(\tilde{})([b_1, \ldots, b_k]) = [c_1, \ldots, c_j]$ such that $[c_1, \ldots, c_j]$ contains exactly one occurrence of all the elements of $\mathcal{A}(s)$ not in $[b_1, \ldots, b_k]$, $\mathcal{A}(\setminus)([b_1, \ldots, b_k], [c_1, \ldots, c_j]) = [b_1, \ldots, b_k] \setminus [c_1, \ldots, c_j]$, i.e. the difference between the two multisets, and finally $\mathcal{A}(whole) = [(all, s)]$.

7 The Framework

Now, we can extend the definition of algebraic nets with respect to the used generator and with respect to some constructs, which will then allow us to define different kinds of algebraic nets. Actually, we use two generators: the first one defines the basic sorts and operations for the tokens on the places. The second generator introduces the necessary multiset structure on top of these basic sorts, which are used to construct the arc annotations and the transition conditions. Note that, in the most general case, the first generator has the full power so that tokens could be multisets and even multisets of multisets, etc. The more interesting versions, however, are the ones with a more restrictive first generator.

7.1 Formal definition

Definition 22 (High-level net version definition). *Let G_1, G_2 be two generators, and let $\mathcal{C}$ be a set of constructs with respect to $G_2 \circ G_1$. Then $K = (G_1, G_2, \mathcal{C})$ is a* version definition of high-level nets.

The main idea of a definition of a version $K = (G_1, G_2, \mathcal{C})$ is that the constructs $\mathcal{C}$ define the signatures and algebras that may be explicitly defined by the user, and the generator G_1 defines which other sorts and operations may be constructed from them. These together define the *basic sorts* of this version of algebraic nets. The generator G_2 defines the multiset sorts and the operations that may be used for the annotations of the net (markings, arc labels and transition conditions).

Definition 23 (Algebraic net of kind K). *Let $K = (G_1, G_2, \mathcal{C})$ be a version definition of high-level nets with $G_1 = (GS_1, GA_1)$ and $G_2 = (GS_2, GA_2)$.*

An algebraic net scheme of kind K is a tuple $\Sigma = (N, SIG, sort, vars, l, c, i)$ with

1. *a* net $N = (P, T, F)$,
2. *a* signature SIG,
3. *a* place sort *mapping* $sort : P \to S^{GS_1(SIG)}$,
4. *a* transition variable *mapping* $vars : T \to \mathbb{V}^{GS_1(SIG)}$,
5. *an* arc label *mapping* $l : F \to \mathbb{T}^{GS_2(GS_1(SIG))}(\mathbb{V}^{GS_1(SIG)})$ *such that:*
   - *for all* $(p, t) \in F \cap (P \times T) : l((p, t)) \in \mathbb{T}^{GS_2(GS_1(SIG))}_{(ms, sort(p))}(vars(t))$
   - *for all* $(t, p) \in F \cap (T \times P) : l((t, p)) \in \mathbb{T}^{GS_2(GS_1(SIG))}_{(ms, sort(p))}(vars(t))$,

6. *a* transition condition *mapping* $c : T \to \mathbb{T}_{(bool)}^{GS_2(GS_1(SIG))}(\mathbb{V}^{GS_1(SIG)})$ *with* $c(t) \in \mathbb{T}_{(bool)}^{GS_2(GS_1(SIG))}(vars(t))$ *for every* $t \in T$, *and*
7. *an* initial marking $i : P \to \mathbb{T}^{GS_2(GS_1(SIG))}$ *such that,* $i(p) \in \mathbb{T}_{(ms,sort(p))}^{GS_2(GS_1(SIG))}$ *for every place* $p \in P$.

An algebraic net scheme Σ *of kind* K *together with a SIG-algebra* $\mathcal{A}$ *and a construct mapping* $\mathcal{H}$ *from* $\mathcal{C}$ *to* $\mathcal{A}$ *form an algebraic net* $(\Sigma, \mathcal{A}, \mathcal{H})$ *of kind* K.

Note that the semantics and the reachability graph for algebraic nets of any kind is the same as the one defined before in Sect. 4.2. The exact details of the generators do not play any role, as long as the booleans and the multiset addition are included.

7.2 Examples of definitions of net versions

At last, we apply this framework for defining some versions of high-level Petri nets. Note that this is mainly meant for demonstrating the use of the framework; a more complete, more systematic and careful definition and classification of most of the relevant high-level Petri net versions is out of the scope of this paper.

P/T-systems We start with the most primitive version of high-level nets, which is Place/Transition systems, just in the setting of high-level nets. In ISO/IEC 15909-2 this version is called *Place/Transition systems in high-level net notation*. The basic idea is that all places are of sort *dots*, which represents the black tokens.

This version can be defined by $PT = (DOT, TRUE \circ MULT, \emptyset)$. Since the set of legal constructs is empty, the signature and algebra provided for an algebraic net of this kind must be empty. Since the first generator DOT only creates the sort *dots* from an empty signature, this is the only legal sort for places. For constructing the legal arc inscriptions, also the sorts and operations generated by $TRUE \circ MULT$ on top of that can be used. As discussed earlier, $TRUE$ is a simple version of the booleans with *true* as the only possible value; this is necessary for technical reasons, since in high-level nets as defined above a transition must have condition.

Symmetric nets Next, we define symmetric nets by the help of a specific generator (products and multisets cannot be built recursively) and the symmetric net constructs. We define the generators for symmetric nets by $SN_1 = PROD \circ BOOL$, $SN_2 = MULT$.

We define the constructs for symmetric nets by $\mathcal{C}_{SN} = \{UO, LO, CS, PAR, EQ, ALL\}$.

Then the definition of the symmetric net version of an algebraic net is $K_{SN} = (SN_1, SN_2, \mathcal{C}_{SN})$.

Symmetric nets with bags Symmetric nets where extended in [12] to allow for manipulating multiset elements in places, on arcs and in transitions conditions. This feature provides additional flexibility for modelling with symmetric nets, without losing the analysis techniques such as the symbolic reachability graph.

The definition of the symmetric net with bags version of algebraic net is $K_{SNB} = (MULT \circ PROD \circ BOOL, MULTB \cup NAT, \mathcal{C}_{SN} \cup \{MULTSNB\})$.

Algebraic Petri nets with fixed arc weight Next, we characterise the version of algebraic nets as defined by Reisig [5]. The major characteristics of this version is that the number of tokens flowing through an arc is always the same, which is why they are sometimes called nets with fixed arc weight (see [13]).

The main point is that the multiset structure of the signature is constructed in a fixed way on top of an arbitrary algebra. This is reflected in the formal definition $AN_1 = (BOOL, MULT, \mathcal{C}_{BOOL})$.

The constructor $BOOL$ is used for using $BOOL$ as a pre-defined sort. All other sorts can be defined by arbitrary constructs over that generator. The multisets only come in via the second constructor $MULT$ (which does not provide any operation for flexible arc weights).

Algebraic Petri nets with flexible arc weight In contrast to that, Kindler and Völzer [13] introduce a version of algebraic nets in which the signature and algebra can define arbitrary operations on multisets. And multiset sorts may even be used as the sort of some places, and multisets of multisets are possible. Therefore, the first generator allows to use this, and the constructs for the algebra may even define some of these operations.

This version can be defined as $AN_2 = (MULT^* \circ BOOL, ID, \mathcal{C}_{MULT^* \circ BOOL})$.

Note that the second generator does not need to contribute any further operations, since all the needed multiset operations are already added by the first generator. Therefore, the second generator is just the identity ID.

General version In principle, algebraic nets with flexible arc weights have all the necessary power. But, to obtain this power, they require that all more powerful operations are defined by the user by providing a signature and an algebra with all the desired operations. It would be much easier, if some of these operations were built-in and could be used without explicitly defining them everytime a new algebraic net is used. This is in particular true for products. Therefore, a more general version would have the products and some more operations on multisets available. Only very special sorts or operations would then come from the signature and algebra defined by the user.

This could look like $AN_3 = ((MULTX \cup PROD)^* \circ (BOOL \cup DOT), ID, \mathcal{C}_{(MULTX \cup PROD)^* \circ (BOOL \cup DOT)})$. Actually, even some other sorts like strings, integers, and list could be included here. But, we leave the exact set of operations and sorts that should be built-in open for a discussion here, as we left the exact definition of $MULTX$ open.

8 Conclusion

In this paper, we have introduced a mathematical framework that allows us to define different versions of high-level Petri nets, with different built-in sorts and operators, and different legal constructs in the underlying algebra. The examples at the end of this paper show that a wide variety of different kinds of high-level net can be defined this way. The main advantage is that the legal constructs and the built-in sorts and operations can be defined independently from the actual definition of algebraic Petri nets. This way, it is much easier to define, to compare and classify different versions of high-level Petri nets and make the built-in sorts and operations and the legal constructs explicit.

Note that for making this possible, we needed to introduce some mathematics, which might be hard to understand for non-experts. But, once this framework is there (and validated by some experts), it can be used in an intuitive way, by selecting and combining the respective constructs and generators. Defining a new version can be done without understanding the details of the underlying framework; one just needs to select the basic building blocks, and combine them with each other, which is demonstrated by the examples in Sect. 7.2.

References

1. Chiola, G., Dutheillet, C., Franceschinis, G., Haddad, S.: On well-formed coloured nets and their symbolic reachability graph. In Jensen, K., Rozenberg, G., eds.: Petri Nets: Theory and Application. Springer-Verlag (1991) 373–396
2. Kindler, E., Petrucci, L.: Towards a standard for modular Petri nets: A formalisation. In Franceschinis, G., Wolf, K., eds.: Application and Theory of Petri Nets 2009, Internat. Conference, Proceedings. Volume 5606 of LNCS., Springer-Verlag (2009) 43–62
3. Kindler, E., Reisig, W.: Algebraic system nets for modelling distributed algorithms. Petri Net Newsletter **51** (1996) 16–31
4. Kindler, E., Völzer, H.: Algebraic nets with flexible arcs. Theoretical Computer Science (2001)
5. Reisig, W.: Petri nets and algebraic specifications. Theoretical Computer Science **80** (1991) 1–34
6. Ehrig, H., Mahr, B.: Fundamentals of Algebraic Specifications 1, Equations and Initial Semantics. Volume 6 of EATCS Monographs on Theoretical Computer Science. Springer-Verlag (1985)

7. Berthomieu, B., Choquet, N., Colin, C., Loyer, B., Martin, J., Mauboussin, A.: Abstract Data Nets combining Petri nets and abstract data types for high level specification of distributed systems. In: Proceedings of VII European Workshop on Application and Theory of Petri Nets. (1986)
8. Vautherin, J.: Parallel systems specifications with coloured Petri nets and algebraic specifications. In Rozenberg, G., ed.: Advances in Petri Nets. Volume 266 of LNCS. Springer-Verlag (1987) 293–308
9. Billington, J.: Many-sorted high-level nets. In: Proceedings of the 3rd International Workshop on Petri Nets and Performance Models, IEEE Computer Society Press (1989) 166–179
10. ISO/IEC: Software and Systems Engineering – High-level Petri Nets, Part 1: Concepts, Definitions and Graphical Notation, International Standard ISO/IEC 15909 (2004)
11. Schmidt, K.: Verification of siphons and traps for algebraic Petri nets. In Azéma, P., Balbo, G., eds.: Application and Theory of Petri Nets 1997, Internat. Conference, Proceedings. Volume 1248 of LNCS., Springer-Verlag (1997) 427–446
12. Haddad, S., Kordon, F., Petrucci, L., Pradat-Peyre, J., Trèves, N.: Efficient state-based analysis by introducing bags in Petri nets colour domains. In: Proc. 28th American Control Conference (ACC2009), St Louis, Missouri, USA. (2009) 5018–5025
13. Kindler, E., Völzer, H.: Flexibility in algebraic nets. In Desel, J., Silva, M., eds.: Application and Theory of Petri Nets 1998, 19^{th} International Conference. Volume 1420 of LNCS., Springer-Verlag (1998) 345–364

Verification of Railway Interlocking Tables using Coloured Petri Nets*

Somsak Vanit-Anunchai

School of Telecommunication Engineering
Institute of Engineering, Suranaree University of Technology
Muang, Nakhon Ratchasima 30000, Thailand
Email: somsav@sut.ac.th

Abstract. A functional specification for railway signalling systems called "control table" plays a vital role in the signalling design and installation processes. Control tables are the tabular representation specifying the routes, on which the passage of the train is allowed. Associated with the route, the states and actions of all related signalling equipment are also specified. Although various software tools are available for generating, editing and checking the control tables, there are still some drawbacks. Firstly, those tools are usually bound up with the a specific railway company. Secondly, each railway company has its own operating rules and regulations that control tables need to comply with. The control tables that are automatically generated and verified still require manual inspection by the railway signal engineer. This checking process is very labor intensive and prone to errors. To detect and eliminate errors, we propose to formally model and analyse the interlocking tables using CPN Tools. Our CPN model comprises two parts: *Signalling Layout* model and *Interlocking* model. We use ML functions on arc inscription in the *Interlocking* model. These ML functions can be generated directly from the content of the control table using Extensible Stylesheet Language Transformations (XSLT). Thus our CPN model can be easily adapted and reused and CPN models of other control tables can be rapidly built. Finally some experimental results are discussed to convince us of the correctness of our CPN model and the control tables.

Keywords: Control Tables, Interlocking Tables, Railway Signalling Systems, Coloured Petri Nets, State space Analysis

1 Introduction

Currently the State Railway of Thailand (SRT) has been undertaking several railway signalling projects involving either improvement of the existing signalling systems or expansion of the existing railway lines. During the whole process of designing, installing and testing the signalling system, "Interlocking Tables" or "Control Tables" play a vital role in every stage. The control table is a tabular representation specifying how the trains move together with the required states and actions of all related equipment. This important document also acts as an agreement between the railway administrators and the contractors. Many signalling contractors have software tools for editing, generating and verifying the control tables. Usually the control table generated by a software tool is bound up with a specific railway company. But SRT has its own operating regulations, requirements and signalling principles that control tables need to comply with. Thus after the control tables are designed and checked by the contractors, they need to be rechecked by SRT's signal engineers. Now SRT signal engineers manually inspect the submitted control tables without any software tools. Thus the checking process is very slow, labour intensive and prone to errors. In order to assist their inspection, detect and rectify errors rapidly, we propose to formally model and analyze the control tables using CPN Tools [12]. Because SRT's signalling projects involve hundreds of Interlocking

* Supported by National Research Council of Thailand Grant no. PorKor/2551-153

systems, we wish to study how to rapidly re-build the CPN model of the control tables for other Interlocking systems. Our counter part, SRT's signal engineers, believe that the signal engineers should build, maintain and modify the CPN models of control tables themselves due to the details and complication of the problem. Thus CPN Tools is a good candidate because its graphical language and the user interface are easy to use.

The contribution of this paper is two folds. Firstly, the Phanthong's control table is modelled and verified against its desired property. Secondly, we discuss design decision on how to create a CPN model of a control table that can be easily adapted and reused for other control tables. In particular, we propose to standardize the format of control tables using XML and using XSLT to transform the content of the control table to ML functions used in the CPN model. Thus the signal engineers who follow our methodology do not need to be a programming expert in C, Java or ML.

The rest of this paper is organised as follows. Section 2 briefly explains the concept of railway signalling system and control tables. Section 3 discusses related work. Section 4 defines the scope of work by discussing assumptions, modelling approach and model structure. The CPN model of Panthong control table is illustrated in Section 5 and 6. Section 7 discusses our analysis results. Conclusion and Future work are presented in Section 8.

2 Railway Signalling Systems and Control Tables

2.1 Signalling Systems

In general the railway lines are basically divided into *sections*. To avoid collision, only one train is allowed in one *section* at a time. The train can enter or leave the *section* when the driver receives authorization from a signalman via a signal indicator. Before the signalman issues the authorization, he needs to ensure that no object blocks the passage of the train. SRT's regulation divides the *section* into two categories: between two stations and within the station area. The *section* between two railway stations, which involves two signalmen, is called "*block section*". Usually railway companies have a strict procedure how to admit trains into a *block section*. To prevent human error, which often leads to collisions, the strict operation on a *block section* is controlled by an equipment called "Block Instrument". The Block Instrument has 3 possible states. It is *Normal* when there is no train in the block section and no one requests block possession. It is in *Going* state when the permission for the block possession is given to the outgoing train. It is in *Coming* state when the permission for the block possession is given to the incoming train.

Figure 1 shows the signalling layout of a small station named "Panthong". It comprises a collection of railway tracks and signalling equipment such as track circuits, points and signals. Main signals are classified into three types:, warner, home and starter signals. SRT defines that the station area is between two home signals (signal no.3 and signal no.4). Each piece of signalling equipment has an identification number and holds a certain state as follows.

Track Circuits A track Circuit is an electrical devices used to detect the presence of a train. A track circuit (e.g. 42T, 2T) is either *cleared* indicating no train on the track or *occupied* indicating the possible[1] presence of a train.

[1] When the track circuit fails, its state is occupied even if there is no train.

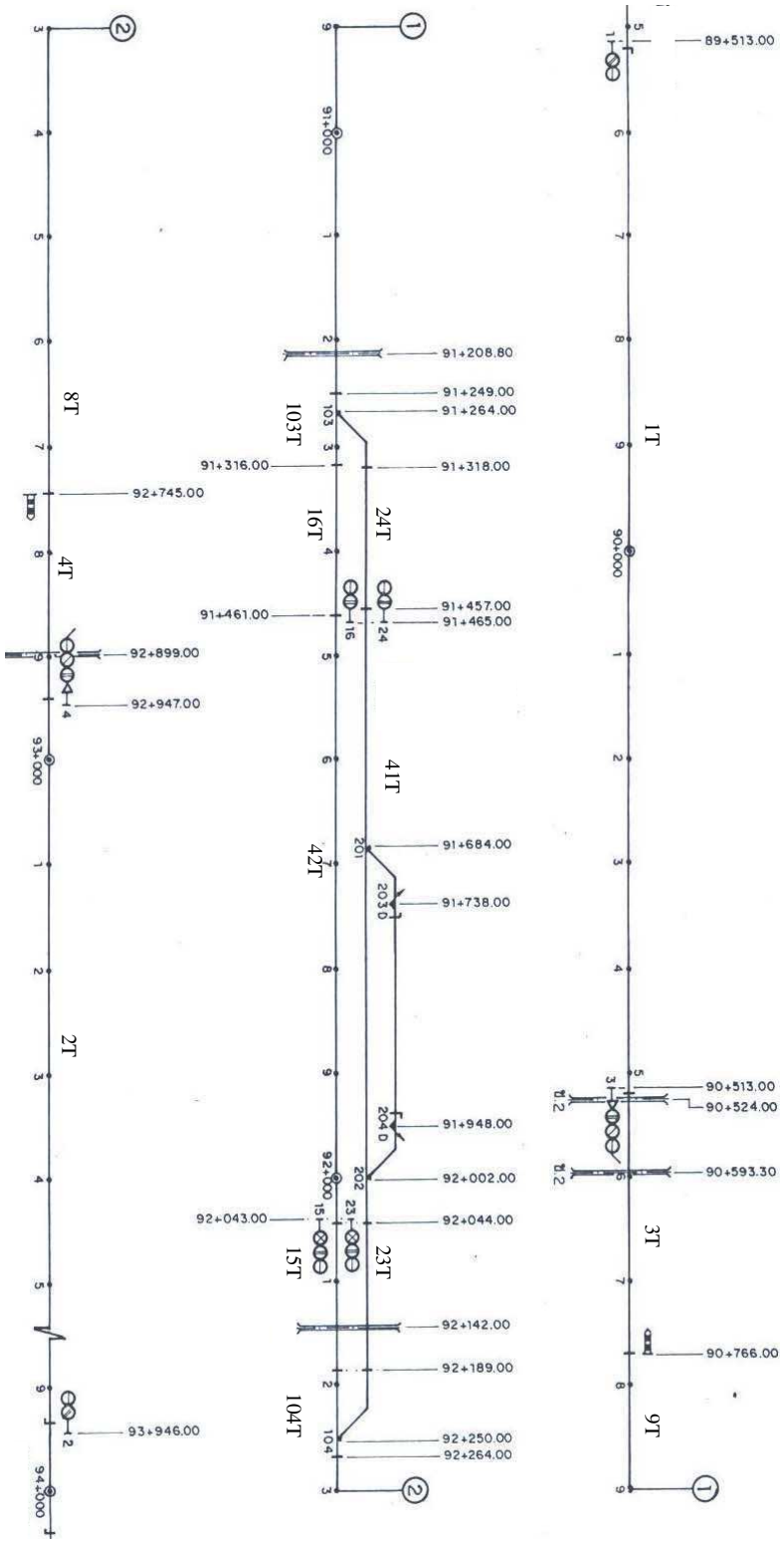


Fig. 1. Signalling layout of the Panthong Station

Warner signals A warner signal (e.g. 1, 2) has two aspects: *yellow* or *green*. It informs drivers about the status of the next signal.

Home signals A home signal (e.g. 3, 4) has three aspects: *red* or *yellow* or *green*.

It displays *red* when forbidding the train enter the *station area*.

It displays *yellow* giving the driver authorities to move the train into the *station area* and prepare to stop at the next signal.

It displays *green* giving the driver authorities to move the train passing the *station* and enter the next *block section*.

Starter signals A starter (e.g. 16, 24, 15, 23) has two aspects: *red* or *green*.

It displays *red* when forbidding the train to enter the *block section*.

It displays *green* when giving the driver authorities to move the train into the *block section*.

Point A point (e.g. 103, 104) or railway switch or turnout is a mechanical installation used to guide a train from one track to another. A point usually has a straight through track called *main line* and a diverging track called *loop line*. A point is *right-hand* when a moving train from a joint track diverges to the right of the straight track. Similarly a *left-hand* point has the diverging track on the left of the straight line. When a point diverges the train, it is in *reverse* position. When a point lets the train move straight through, it is in *normal* position. When the interlocking prevents a point from changing position, the point is *locked*.

2.2 Control Tables

A collection of track circuits along the reserved *section* is called "*route*". An entry signal shall be cleared to let the train enter the route. Although the request to clear the entry signal is issued by the signalman, the route entry permission is decided by the interlocking system using safety rules and control methods specified in the agreed control tables. Table 1 and 2 are the control tables for Panthong station of which signalling layout is shown in Fig. 1. Data in the first column, "From", is the route identifications which are labeled by the entry signal: 3(1); 3(2); 4(1); 4(2); 15; 16; 23 and 24. Each row in the tables represents the requirement how to set and release each route. For example, route 3(2) comprises the track circuits 3T, 9T, 103T, 16T, 42T, 15T and requires that the points 103 and 104 are in normal position. Routes 3(1) and 3(2) distinguish that behind signal 3 two routes are possible. Similar rule applies to routes 4(1) and 4(2).

Different Interlocking systems from different manufacturers may have different control methods. However there are four basic control methods widely accepted and used among railway companies.

Route locking Route setting involves a collection of adjacent track circuits, points and signals. To assure the safety, firstly, the interlocking system verifies that the route does not conflict with other routes previously set. The column "Requires Route Normal" shows conflict routes. A route cannot be set if any conflict routes have been set and not yet released. For route 3(2) the conflict routes are 16, 23, 24, 4(1), 4(2) and 3(1). Secondly, the points along the route are locked in the correct positions. If the related points are not in the correct positions, the controller will attempt to set and lock them in the correct positions. Thirdly, the track

Table 1. A control table for Panthong station (part 1:Route locking)

ROUTE		INTERLOCKING				CONTROL			
		REQUIRES	SET & LOCKS POINTS		REQUIRES KEYLOCK	ASPECT	SIGNAL AHEAD	REQUIRES TC	
								CLEAR	AT TIME OF CLEARING ONLY
From	To	ROUTE NORMAL	NORMAL	REVERSE	NORMAL				TC CLEAR OR OCC FOR TIME
1	3					Y	3 AT R#		
						G	3 AT Y# OR G#		
2	4					Y	4 AT R#		
						G	4 AT Y# OR G#		
3(1)	23	16,24,4(1),4(2),3(2)		103,104	201,202, 203,204	Y+JI	23 AT R#	3T,9T,103T,24T,41T, 23T,104T,8T,4T	42 / 42 FOR 60 sec
3(2)	15	16,24,4(1),4(2),3(1)	103,104			Y G	15 AT R# 15 AT G#	3T,9T,103T,16T,42T, 15T,104T,8T,4T	41 / 41 FOR 60 sec
4(1)	16	15,23,3(1),3(2),4(2)	104,103			Y G	16 AT R# 16 AT G#	4T,8T,104T,15T,42T, 16T,103T,9T,3T	41 / 41 FOR 60 sec
4(2)	24	15,23,3(1),3(2),4(1)		104,103	201,202, 203,204	Y+JI	24 AT R#	4T,8T,104T,23T,41T, 24T,103T,9T,3T	42 / 42 FOR 60 sec
15	UP BLOCK SECTION	23,4(1),4(2)	104,103			G		15T,104T,8T,4T,2T,TOL	
23	UP BLOCK SECTION	15,4(1),4(2)		104,103		G		23T,104T,8T,4T,2T,TOL	
16	DOWN BLOCK SECTION	24,3(1),3(2)	103,104			G		16T,103T,9T,3T,1T,TOL	
24	DOWN BLOCK SECTION	16,3(1),3(2)		103,104		G		24T,103T,9T,3T,1T,TOL	

Table 2. A control table for Panthong station (part 2:Approach locking)

ROUTE		CONTROL						
		APPROACH LOCKED WHEN SIGNAL CLEARED &		ROUTE RELEASED BY				Notes
		AND		TC	TC OCC	TC OCC	OR	AND / OR
From	TO	TC OCC	OR TIME	CLEAR	& CLEAR		Emergency RELEASE AFTER	REMARKS
1	3							WARNER REPEAT 3
2	4							WARNER REPEAT 4
3(1)	23	1	120s	3T,9T	103T	41T	240s	DOWN BLOCK NOTSET DOWN XING BOOM DOWN
3(2)	15	1	120s	3T,9T	103T	42T	240s	DOWN BLOCK NOTSET DOWN XING BOOM DOWN
4(1)	16	2	120s	4T,8T	104T	42T	240s	UP BLOCK NOTSET UP XING BOOM DOWN
4(2)	24	2	120s	4T,8T	104T	41T	240s	UP BLOCK NOTSET UP XING BOOM DOWN
15	UP BLOCK SECTION	42T	120s	15T,104T, 8T	4	2	240s	UP BLOCK SET UP XING BOOM DOWN
23	UP BLOCK SECTION	41T	120s	23T,104T, 8T	4	2	240s	UP BLOCK SET UP XING BOOM DOWN
16	DOWN BLOCK SECTION	42T	120s	16T,103T, 9T	3	1	240s	DOWN BLOCK SET DOWN XING BOOM DOWN
24	DOWN BLOCK SECTION	41T	120s	24T,103T, 9T	3	1	240s	DOWN BLOCK SET DOWN XING BOOM DOWN

circuits along the required route are all cleared or unoccupied so that nothing obstructs the passage of the train. Then the entry signal can be cleared (showing yellow or green). The home signal will be green if the exit signal of the route shows green too. For example the exit (starter) signal of the route 3(2) is 15. If starter signal 15 shows green and route 3(2) is set, the home signal 3 will show green.

Approach locking After a route is set; the points are locked; and the entry signal is cleared, if the track circuit in front of (approaching) the entry signal is occupied, then the signalman cannot cancel the route and the entry signal by the normal procedure. Approach locking prevents the train driver from the sudden change of signal aspect from green or yellow to red.

Column 3 in Table 2, "APPROACH LOCKED WHEN SIGNAL CLEARED & TC OCC", presents locking when the approach track circuit is occupied. For example, route 3(2) will be approach locked if the route is set and track 1T is occupied. The approach locking also happens after the signal is cleared longer than 2 minutes.

Route released By the passage of the train, the reserved route is automatically released. Column "Route Released by" in Table 2 presents route released mechanism for the signalling layout in Fig. 1. Route 3(2) will be released when track 3T, 9T is cleared; track 103T is occupied and then cleared; and track 42T is occupied. The reserved route can be emergency released but the release action will be delayed for 4 minute after the signalman issues "emergency route released" command.

Flank protection The equipments within the surrounding area of the reserved route that may cause an accident shall be protected even if no train is expected to pass such a signal or such points. For example points should be in such positions that they do not give immediate access to the route. The last two columns of Table 2 presents an example of flank protection. For route 3(2), the track 41T, which is not in the route 3(2), shall be unoccupied. If it is occupied, the object on the track 41T should stand still. This condition is implied if the track 41T is occupied longer than 1 minute.

3 Related Work

Fokkink and Hollingshead [7] provided a perspective that can classify the research work regarding verification of railway signaling systems. According to [7] the railway signalling system is divided into three layers: infrastructure, interlocking and logistic layers. All layers must provide safety for railway operation. The infrastructure layer involves objects or equipment used in the yard. The work in this category, for instance [1,3,6,13], ties closely with manufacturer's products. The logistic layer involves human operation and train scheduling which aims at efficiency and deadlock free. It involves the operation of whole railway network (e.g. [9,11]) thus the state explosion problem is often encountered. The interlocking layer provides the interface between the logistic and infrastructure layers. It prevents us from accidents caused by human errors or equipment failures. The work in this category such as [10,15,16] models the control tables and verify it against the safety regulations and signalling principles.

Hansen [10] presented a VDM (Vienna Development Method) model of a railway interlocking system, and validated it through simulation using ML. The work focuses on the principles and concepts of Danish systems rather than a particular interlocking system. He also pointed out that Interlocking systems from other countries may be different from the Interlocking described in [10]. Borälv [1] and Peterson [13] constructed interlocking programs using a special language called STERNOL, which was developed by ADTranz in Sweden, and verified the interlocking programs using NP-Tools.

Because relay interlocking and computer interlocking [3] are designed based on ladder logic diagrams, Fokkink and Hollingshead [7] proposed to convert ladder logic diagrams to Boolean formulae. Then they applied a theorem prover to verify these Boolean formulae.

Winter et al [14] proposed to create two formal models during the design process of interlockings. One is the formal model of the Signalling Principles called Principle model. The

other is the formal model of the functional specification for a specific track-layout called Interlocking model. The Control Tables are translated into a Interlocking model and then checked against the Principle model. At first she used CSP (Communicating Sequential Processes) as a modelling language but later found that the CSP models of the interlocking system and the signalling principle are difficult to understand and validate. Thus [16] used Abstract State Machine (ASM) [8] notation to model the semantic of control tables. The ASM model was then automatically converted to NuSMV code [4] while the safety properties were modeled in CTL (Computation Tree Logic). Finally [15] they modelled the safety properties in ASM and then translated both ASM models into the NP Prover tool [1] in order to compare the performance between NuSMV and NP-tool. They discovered that if the track layout was divided into smaller segments for verification, the NuSMV outperformed the NP-Tool. Our work shares the same goal to [1, 7, 13–16] but our tools and signalling principles differ from their.

Hagalisletto et al [9] demonstrated how to construct and refine models of railway using a component-oriented approach. They modelled atomic nets, such as track circuits and turnouts, using Coloured Petri Nets. Although atomic nets were created using Design/CPN, the models were simulated [2] using Maude [5] due to the state explosion problem. Even though our CPN models of each piece of equipment, such as track circuits, was inspired by [9], our work aims at the interlocking table of one station while [9] involved the logistic layer and the whole railway network.

4 CPN Model of the Panthong's Control Table - Overview

Currently SRT has been undertaking track doubling projects which need to verify hundreds of interlocking systems. Thus it is necessary to seek out a modelling approach to rapidly build and verify these control tables. An existing control table of a single track station named "Panthong" was selected as a modelling exercise because its new control table with double track is being designed by a contractor. We wish to upgrade our CPN model for the double track station to verify the new control table in the near future.

4.1 Modelling Scope and Assumptions

To reduce the complexity of the model as well as the state explosion problem which has been reported by a number of researchers [9,16] who investigated the similar problems, we need to make the following assumptions regarding train movement and signalling operations:

1. We assume that a train has no length and it occupies one track at a time. The train moves in only one direction. Train shunting is not considered.

2. We assume the trains running at the same speed.

3. Our model does not include the auxiliary signals such as Call-on, Shunting and Junction indicators.

4. Our model does not include timers. However we use time stamps when modelling the trains moving along the track. This implies moving a train takes longer than other actions in the system. For example the train must not move through a track circuit so fast that the interlocking cannot detect the presence of the train.

5. Our model does not include emergency route release, emergency point operation.

6. Normally the signalling system provides a safety mechanism when equipment fails. Our model does not consider when equipment fails.

7. Our model does not include level crossings.

8. Our model includes high level abstraction of block systems but we do not model their operations in detail.

9. Our model does not include flank protections.

10. The train drivers strictly obey the signals.

4.2 Modelling Approach and Model Structure

Our CPN model comprises 2 parts: *Signalling Layout* and *Interlocking*. The *Signalling Layout* part shown in Fig. 3 and Fig. 4 represents the system we wish to control. The *Interlocking* part represents the interlocking controller. CPN models in Fig. 3 and Fig.4 mimic the signalling plan of Panthong station (Fig.1). It comprises three kinds of CPN modules modelling wayside equipment of which functions and states are described in Section 2.1. The CPN model in Fig. 3 and Fig. 4 provides not only geographic information how each way side equipment is connected to each other but also the ability to simulate trains moving along the tracks. Basically our CPN model includes three kinds of the train movements:

a) Train movement between two consecutive track circuits.

b) Trains passing a signal.

c) Trains passing a point.

Comparing with the signalling layout in Fig.1 our model does not include the third loop line because no track circuit is installed there. But we model the key lock of manual operating points and de-railers.

The *Interlocking* part comprises three CPN pages: `UserCommand`, `RouteSetting` (Fig. 6) and `RouteReleased`. They model point setting, route locking, signal clearing and route release functions as specified in the control table and described in Section 2.2. Unlike [15] that does not include the functionality of approach locking (to avoid the state explosion problem), our CPN model does include the approach locking function. Even though the control table of each railway station has different contents, the functionalities: route locking; approach locking; route release; and flank protection are essentially the same. Attempting to create a generic interlocking model, we extract the content of the control table and code them into ML functions which are used in arc inscriptions. To model control tables of other railway stations we simply change the content of the ML functions while using the same CPN models of the *Interlocking* part.

Next we attempt to create these ML functions automatically as illustrated in Fig. 2. In previous projects contractors submitted the control tables in Microsoft-Excel to SRT. Instead of Excel, we encourage SRT to maintain the control table in XML format. As shown in Fig. 2 the control table in Microsoft-Excel is transformed to XML. Then it is transformed to ML functions using Extensible Stylesheet Language Transformations (XSLT). All operations are done using Microsoft-Excel and Microsoft-Word version 7.

5 The CPN Model of the Signalling Layout

The top layer of our CPN model, named `SouthSTA` (Fig. 3), mimics the signalling layout of Panthong's southern section. It comprises a sequence of places, `T1`, `T3`, `T9`, `T103`, `T16`, `T24`, `T41` and `T42`. Each place, typed by *`TCCT`*, represents a track circuit storing the track circuit number and its state. *`TCCT`* is defined in line 4 of Listing 1.1 as a product comprising Track

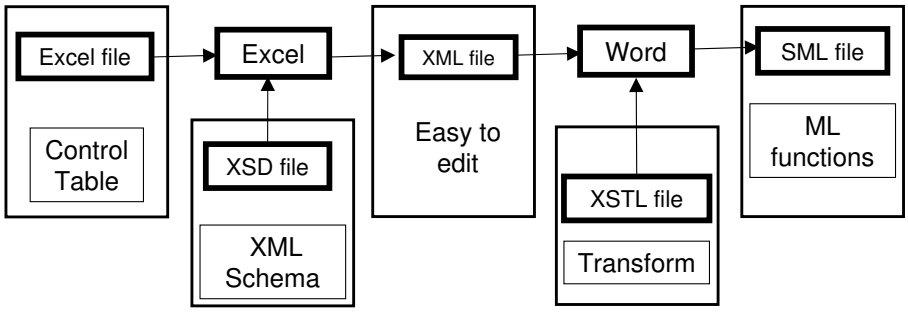


Fig. 2. Transformation of the control table to ML functions using XSLT

number and Train Description (`TD`). Line 2 of Listing 1.1 defines `TD` as a colour set representing states of a track circuit: unoccupied (`noTrain`); occupied by a train moving away from Bangkok (`TrainUP`); and occupied by a train moving toward Bangkok (`TrainDOWN`). Place `TOL_SOUTH` (Train-on-line) models the track section[2] between Panthong and the adjacent station in the south[3]. The `SouthSTA` is linked to `NorthSTA` page (Fig. 4) which models the northern section of the station. Besides signalling layout, `SouthSTA` page also includes substitution transitions `UserRequestBlock` and `UserRequestRoute` modelling the actions of *block setting* and *route setting* by the signalman. When approach locking does not occur, the signalman can cancel the route setting. This *route cancel* command is modelled by a token in the fusion place `RouteCancel`. Substitution transitions `S1`, `LOS` and `PT9` linked to `TrackCCT` model the passage of the train between two adjacent track circuits. Substitution transition `S3` linked to `HomeSignalUP` page (Fig. 5) models the passage of the train when passing the home signal no.3. Similarly, substitution transitions `S16` and `S24`, both linked to `StarterDOWN` page model the passage of the train when passing the starter signals no. 16 and 24. We do not need to model the train passing a warner signal. Because the warner has no red signal signal and it cannot stop the train. It acts as a repeater of the home signal. Its aspect depends on the aspect of the home signal. Thus we model warner signal no. 1 together with home signal no. 3 in substitution transition `S3`. Substitution transition `P103`, linked to `PointSouthLeft` page models the passage of the train when passing the point no. 103. Due to space limit, we choose to explains a CPN subpage named `HomeSignalUP` because `PointSouthLeft` page is too complex and `HomeSignalUP` is similar to `StarterDOWN` and `TrackCCT` pages.

HomeSignalUP page When a train passes the home signal, two signals' aspects return to the normal states. The warner signal returns to yellow and the home signal returns to red. `HomeSignalUP` page (Fig. 5) models these aspect restorations. The `HomeSignalUP` page is reused at other locations thus it needs an identification place to identify the signal numbers. For example, place `SIG1_3` in Fig. 3 links to place `WarnerHome` in Fig. 5. Place `SIG1_3` and `WarnerHome` are typed by `WARNERxHOME` which is defined in line 6 of listing 1.1 as a product of two strings that are the numbers of the warner and home signals. Place `ApproachLock` in Fig. 5 is used to ensure that the train will not pass the signal post before the approach locking takes place. We defer the explanation about places `SIGNAL_POOL` and `TrackPool` to section 6.1.

[2] No track circuit is installed at this location.
[3] South or down means toward Bangkok, north or up means away from Bangkok.

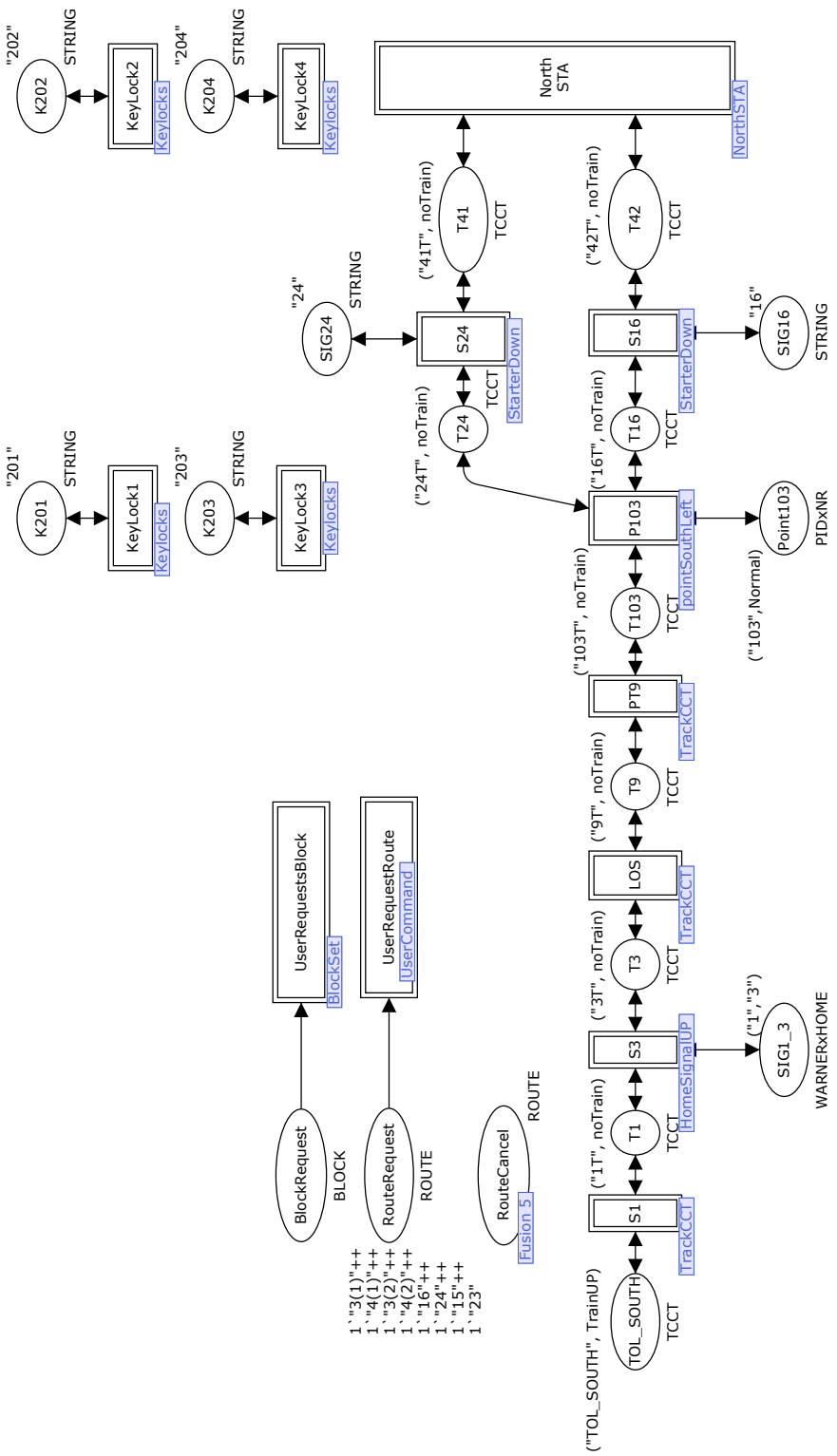


Fig. 3. CPN model: the SouthSTA page

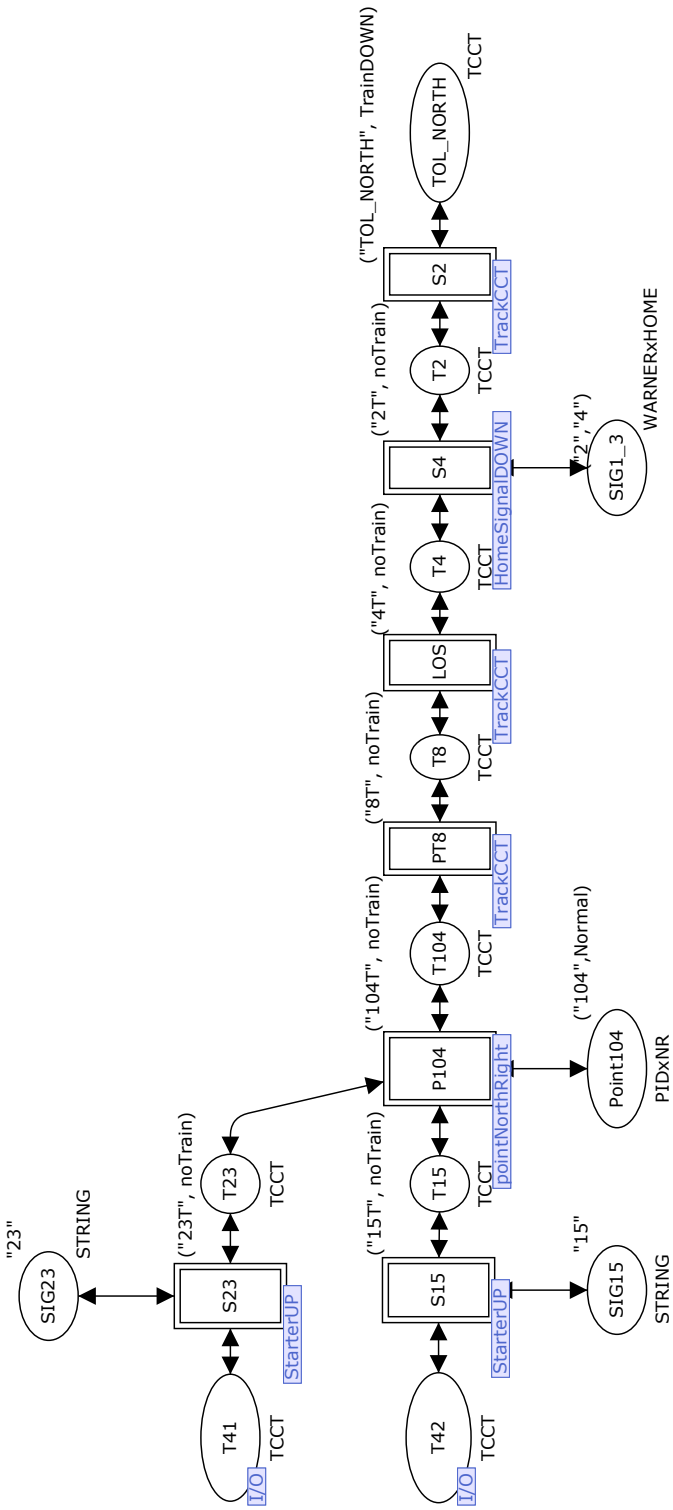


Fig. 4. CPN model: the NorthSTA page

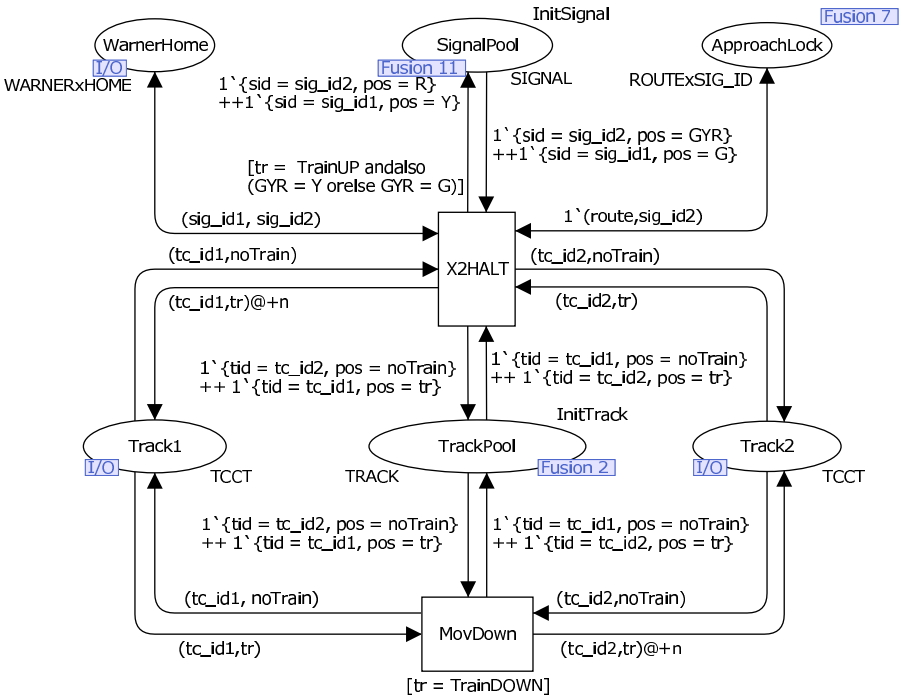


Fig. 5. CPN model: the HomeSignalUP Page

Listing 1.1. Declarations

```
1  (* Global Declarations *)
2  colset TD = with noTrain | TrainUP | TrainDOWN;
3  var tr,train_direction:TD;
4  colset TCCT = product STRING * TD timed;
5  colset TRACK = record tid:STRING * pos:TD;
6  colset WARNERxHOME = product STRING * STRING;
7  colset E = with e;
8  colset NR = with Normal | Reverse;
9  var pos1:NR;
10 colset PIDxNR = product STRING * NR;
11 colset SIG = with G | Y | R;
12 var w,h,s,GYR:SIG;
13 colset SIGNAL = record sid:STRING * pos:SIG;
14 var p_id,tc_id1,tc_id2,sig_id1,sig_id2:STRING;
15 var entry,dst,warner,home,starter,block_no:STRING;
16 colset ROUTE = STRING;
17 var route,setting_route,in_route2:ROUTE;
18 colset ROUTExSIG_ID = product ROUTE * STRING;
19 colset POINT = record pid:STRING * pos:NR * lock:BOOL;
20 var point:POINT;
21 colset BLOCK_POS = with COMING | NORMAL | GOING;
22 var CNG:BLOCK_POS;
23 colset BLOCK = record bid:STRING * pos:BLOCK_POS;
24 val n =10;
25 var x:BOOL;
26 use "C:/InitMarkings.sml";
27 use "C:/FromXSL.sml";
```

The CPN diagrams of HomeSignalDOWN, StarterUP and StarterDOWN pages are very similar to the HomeSignalUP. We do not fold these pages together because, we think, at the early stage of the model development, folding will cause confusion. We consider that it is faster to select and plug in one of the four CPN modules.

6 The CPN Model of the Interlocking

6.1 Fusion Places - Communication channels between the way side equipment and the interlocking center

This section attempts to explain the CPN model of the interlocking controller according to the functions specified in the control table. To perform its functions, the controller needs to know the states of equipment in the system. Five fusion places are used to store the states of equipment: TrackPool (typed by TRACK - line 5 of of listing 1.1); PointPool (typed by POINT - line 19); SignalPool (typed by SIGNAL - line 13); BlockPool (typed by BLOCK - line 23); and RouteNormal (typed by ROUTE - line 16). TRACK is defined as a record of track identification and train description. POINT is defined as record of point identification, its position (Normal or Reverse) and locking status. SIGNAL is defined as a record of signal identification and its aspect (green, yellow or red). BLOCK is defined as a record of block identification and its state (train coming, train going or block normal). Fusion place RouteNormal stores the route identifications that have not been set.

For example, when a train moves from "TOL_SOUTH" to "1T", substitution transition S1 updates not only place T1 with a token ("1T", TrainUP) but also place TrackPool with a token 1‘{tid = "1T", pos = TrainUP}. It is obvious that the same information has been stored twice which is bad due to the state explosion problem. A solution to the problem is that deleting the state of track circuits in each track place (e.g. T1, T3). But we will loose an ability to view the passage of train in Fig. 3 and 4 which is useful for simulating and debugging the model. On the other hand, storing a track circuit state twice mimics the real situation. In the real signalling system, the status of track circuit "1T" is actually kept in two locations: at a way side box and at the interlocking controller. Putting a token into place TrackPool can be viewed as the track circuit sending its state to the interlocking center via the fusion place TrackPool. A similar situation applies to fusion places PointPool.

Due to space limit we choose to explain only the RouteSetting page because this page plays a central role of the interlocking function in the model.

RouteSetting page Figure 6 shows the RouteSetting page which models route setting and cancelations. Transition SetRoute takes tokens from fusion places TrackPool, PointPool and RouteNormal, and checks if

1. No conflict route is being set. This models by function *require_route_normal*.
2. The tracks along the route are unoccupied. This models by function *require_track_clear*.
3. The related points is set and lock in the required positions. This models by function *require_point_normal* and *require_point_reverse*.

If all three vital conditions are met, the setting route is reserved. But the signals is not clear yet, because non-vital conditions such as block status and level crossing booms down have not yet been proved. At this stage the signalman can cancel the route by putting a route token into fusion place RouteCancel. The route cancel at this stage is modelled by transition RouteCancel1. It restores the route identification into fusion place RouteNormal and unlocks

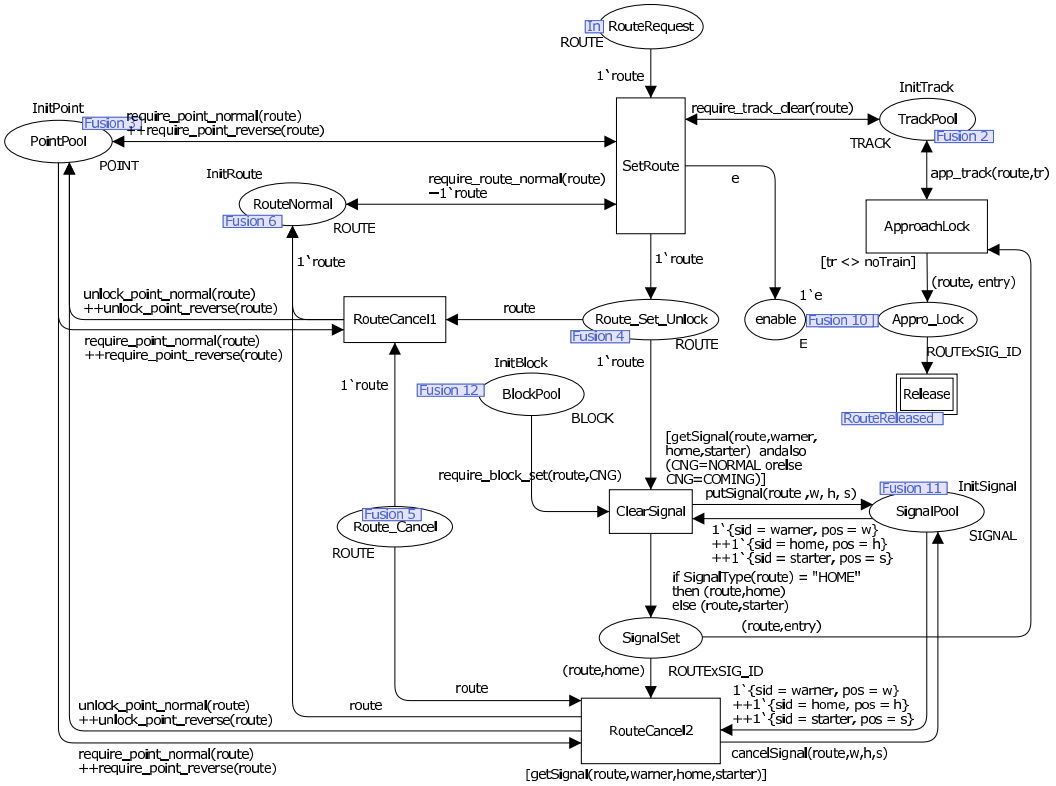


Fig. 6. CPN model: the RouteSetting page

related points. Place `enable` is used to ensure that the interlocking controller will service the next route request command only after the previous valid command have its route reserved.

After the non-vital conditions are complied, transition `ClearSignal` clears the related signals. After route reserved and signal cleared but approach locking has not taken place yet, the signalman can still cancel the route which is modelled by transition `RouteCancel2`. This route cancelation restores the route identification into fusion place `RouteNormal`, unlocks related points and returns the state of all related signals to normal. Without route cancellation, at this stage the interlocking waits for approach locking taking place. This action is modelled by transition `ApproachLock`.

6.2 Mapping the Contents in Control Table to ML Functions

The complexity of our CPN model in the *Interlocking* part does not depend on the number of possible routes partly because the details are hidden in the ML functions on the arc inscriptions. The content in each column of the control table can be directly mapped to an ML function. Due to space limit we can explain only one example of ML functions. Listing 1.2 shows an ML functions representing the column "REQUIRES ROUTE NORMAL"in the control table (Table 1). When a route is being set, the route identification token is taken from fusion place `RouteNormal` in the `UserCommand` page. When a route is released, the route identification is restored into fusion place `RouteNormal` in the *RouteReleased* page. To set a route, all conflict routes must be normal (not set) and the route tokens are stored in fusion place `RouteNormal`. This condition is checked by transition `SetRoute` (Fig. 6) using the ML function, require_route_normal(route) shown in Listing 1.2.

Listing 1.2. Function require_route_normal(route)

```
1 (* Function require_route_normal(route) *)
2 fun rroute(route) = case (route) of
3 ("3(1)") => [ "3(1)" , "16" , "24" , "4(1)" , "4(2)" , "15", "3(2)" ]
4 | ("3(2)") => [ "3(2)" , "16" , "24" , "4(1)" , "4(2)" , "23",  "3(1)"]
5 | ("4(1)") => [ "4(1)" , "15" , "23" , "3(1)" , "3(2)" , "24", "4(2)" ]
6 | ("4(2)") => [ "4(2)" , "15" , "23" , "3(1)" , "3(2)" , "16", "4(1)" ]
7 | ("15") => [ "15" , "23" , "4(1)" , "4(2)" , "24" ]
8 | ("23") => [ "23" , "15" , "4(1)" , "4(2)" , "16" ]
9 | ("16") => [ "16" , "24" , "3(1)" , "3(2)" , "23" ]
10 | ("24") => [ "24" , "16" , "3(1)" , "3(2)" , "15" ]
11 | _ => [];
12 fun require_route_normal(route) = list_to_ms (rroute(route));
```

6.3 Generating the ML Functions using XSLT

Our CPN model uses twelve ML functions that are mapped from the contents of the control table. In the future hundreds of control tables need to be verify thus manual mapping the control table to the ML functions is tedious, inappropriate and prone to errors. This paper proposes a method to automatically map the content of the control tables to the ML functions using XSLT. Listing 1.3 shows the content of the control table for route 3(1) in XML which is created from the control table in Excel. We use the XSLT script file in Listing 1.4 transforming Listing 1.3 to Listing 1.2.

7 Experimental Results

7.1 Desired Properties

Two basic safety properties that railway signalling must provide are no collision and no derailment. Refer to `HomeSignalUP` page, notice that moving a train requires a token with `noTrain` in the designation place. Thus each track circuit place can contain only one token. Our modelling decision causes two effects. First, two train tokens in the same place are not allowed. Second, train tokens cannot move pass each other. We conclude that two trains have a chance of collision if they are on two consecutive tracks. After generating the state space, we use ML query functions, in Listing 1.5, searching the entire state space for the markings that have trains in two consecutive track places. To ensure that our CPN model and the query functions work as we expect. We create a wrong control table by deleting line 31 in Listing 1.3. We experiment by attempting to set the route 3(1) and bring the train from 1T into 41T which is already occupied by another train. After generating the state space and search for a collision, we found a terminal marking that contain a train at 24T and another train at 41T.

Usually derailment can happen when the point is moved (emergency move) while a train is passing that point. Our CPN model does not allow this situation. Another possibility of derailment is when the derailer is in the derail position and a train runs through it. Our model does not include derailers thus checking derailment action is not required in this paper.

7.2 Initial Configurations

We analyse our CPN model using CPN Tools verson 2.2.0 on a AMD9650 2.30 GHz with 3.5 GB of RAM. Due to space limit we select to discuss only three analysis cases of which the initial markings are shown in Table 3. The initial markings are:

Listing 1.3. The content in the control table for route 3(1) in XML

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<Route xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
        <Row>
                <SignalType>HOME</SignalType>
                <Entry>3(1)</Entry>
                <Range>1T:41T</Range>
                <End>23</End>
                <RequireRouteNormal>16</RequireRouteNormal>
                <RequireRouteNormal>24</RequireRouteNormal>
                 <RequireRouteNormal>4(1)</RequireRouteNormal>
                 <RequireRouteNormal>4(2)</RequireRouteNormal>
                <RequireRouteNormal>15</RequireRouteNormal>
                 <RequireRouteNormal>3(2)</RequireRouteNormal>
                 <RequirePointReverse>103,104</RequirePointReverse>
                 <RequireKeyLockNormal>201,202,203,204</RequireKeyLockNormal>
                <AspectEntrance1>Y+JI</AspectEntrance1>
                <AspectEnd1>23 AT R#</AspectEnd1>
                <AppLockbyTcOcc>1T</AppLockbyTcOcc>
                <AppLockWhenTime>120s</AppLockWhenTime>
                 <RouteReleaseByTcClr>3T</RouteReleaseByTcClr>
                 <RouteReleaseByTcClr>9T</RouteReleaseByTcClr>
                 <RouteReleaseByTcOccClr>103T</RouteReleaseByTcOccClr>
                 <RouteReleaseByTcOcc>41T</RouteReleaseByTcOcc>
                <EmergencyRelease>240s</EmergencyRelease>
                 <DetectsPointsNormal>201,202,203,204</DetectsPointsNormal>
                 <DetectsPointsReverse>103,104</DetectsPointsReverse>
                <RequireTcClear>3T</RequireTcClear>
                <RequireTcClear>9T</RequireTcClear>
                <RequireTcClear>103T</RequireTcClear>
                <RequireTcClear>24T</RequireTcClear>
                <RequireTcClear>41T</RequireTcClear>
                <RequireTcClear>23T</RequireTcClear>
                <RequireTcClear>104T</RequireTcClear>
                <RequireTcClear>8T</RequireTcClear>
                <RequireTcClear>4T</RequireTcClear>
                <ReqTcClr>42T</ReqTcClr>
                <ReqTcOcc>42T FOR 60 sec</ReqTcOcc>
        </Row>
```

Listing 1.4. An example of XSLT transforming the XML control table to the SML file

```
<p>fun rroute(route) = case (route) of </p>
<xsl:for-each select="Route/Row">
  ("<xsl:value-of select="Entry"/>") => [
    "<xsl:apply-templates select="Entry"/>"
                <xsl:for-each select="RequireRouteNormal">,
                        "<xsl:apply-templates/>"
                </xsl:for-each>]<p></p> |
</xsl:for-each>
_ => [];
<p>fun require_route_normal(route) = list_to_ms (rroute(route));</p>
```

Case A is a dead lock case when two platform track at the station are fully occupied and one train coming from the south and the other train from the north.

Case B is when two trains coming from the north and south. Another train prepares to depart from platform track no. 41T to Bangkok. The track no. 42T is unoccupied.

Case C is when two trains coming from the north and south and both platform tracks

Listing 1.5. Query functions for checking train collisions

```
fun getTrackList n =
ms_to_list(Mark.Interlocking'TrackPool 1 n);
fun TrainExist (hhh::[]:LTRACK):LSTRING =
      (if (#pos(hhh) = TrainUP orelse #pos(hhh) = TrainDOWN)
                then #tid(hhh)::[] else [])
| TrainExist (hhh::ttt:LTRACK):LSTRING =
      (if(#pos(hhh) = TrainUP orelse #pos(hhh) = TrainDOWN)
                then #tid(hhh)::TrainExist(ttt) else TrainExist(ttt));
val pattern_match:LLSTRING = [["TOL_SOUTH","1T"],["1T","3T"],
["3T","9T"],["9T","103T"],["103T","16T"],["16T","42T"],["24T","41T"],
["103T","24T"],["42T","15T"],["41T","23T"],["15T","104T"],
["23T","104T"],["104T","8T"],["8T","4T"],["4T","2T"],["2T","TOL_NORTH"]];
fun  matchCollide(h::[]:LLSTRING, n) =
let
     val xxxx = TrainExist(getTrackList n);
in
     if contains xxxx h then true else false
end
| matchCollide(h::t:LLSTRING, n) =
let
     val xxxx = TrainExist(getTrackList n);
in
     if contains xxxx h then true else matchCollide(t, n)
end;
fun eva n = matchCollide(pattern_match, n);
val result = PredNodes(EntireGraph, eva, NoLimit);
val _ = print("Satifies Collide:  ");
length(result);
```

are unoccupied.

In all cases other track circuits are *unoccupied*; all points are in *Normal* position and *unlocked*. The keylocks are always *locked*. All signals are in *normal* states. Blocks in both directions are in *Incoming* states. A block request command for the departure train toward Bangkok in case A and B are prepared in Place `BlockRequest` in `SouthSTA` page. This command cannot be executed unless the block state returns to *Normal*. Finally the signalman attempts to set all possible eight routes at once via place `RouteRequest` in `SouthSTA` page. Setting all eight routes sounds unrealistic but this will provide a complete test vector to the model.

Table 3. Initial configurations

	Initiial Markings			
Case	TOL_SOUTH	41T	42T	TOL_NORTH
A	TrainUP	TrainDOWN	TrainUP	TrainDOWN
B	TrainUP	TrainDOWN	noTrain	TrainDOWN
C	TrainUP	noTrain	noTrain	TrainDOWN

7.3 Analysis Results

The analysis results for the control table of Panthong station are shown in Table 4 and 5 We searched for the train collision condition using the ML query functions (Listing 1.5) and found

Table 4. Summary of state space results

Case	Nodes	Arcs	Number of Terminal Markings
A	4	4	1
B	212	524	3
C	960	3,140	8

Table 5. Terminal Markings

Case	Node No.	Route Request (used)	TOL_SOUTH	1T	41T	42T	2T	TOL_NORTH	Signal	Point 103	Point 104
A	4	none	noTrain	TrainUP	noTrain	noTrain	TrainDOWN	noTrain	Normal	Normal Unlock	Normal Unlock
B	78	4(1)	noTrain	TrainUP	TrainDOWN	TrainDOWN	noTrain	noTrain	Normal	Normal Unlock	Normal Unlock
	95	3(2),16	noTrain	noTrain	TrainDOWN	TrainUP	TrainDOWN	noTrain	(16,G)	Normal Lock	Normal Lock
	212	3(2),24,4(2)	TrainDOWN	noTrain	TrainDOWN	TrainUP	noTrain	noTrain	Normal	Reverse Unlock	Reverse Unlock
C	958	3(1),4(1)	noTrain	noTrain	TrainUP	TrainDOWN	noTrain	noTrain	Normal	Normal Unlock	Normal Unlock
	957	4(2),3(2)	noTrain	noTrain	TrainDOWN	TrainUP	noTrain	noTrain	Normal	Normal Unlock	Normal Unlock
	959	4(1),3(1)	noTrain	noTrain	TrainUP	TrainDOWN	noTrain	noTrain	Normal	Reverse Unlock	Reverse Unlock
	960	3(2),4(2)	noTrain	noTrain	TrainDOWN	TrainUP	noTrain	noTrain	Normal	Reverse Unlock	Reverse Unlock
	891	3(1),3(2)	noTrain	noTrain	TrainUP	noTrain	TrainDOWN	noTrain	(3,Y)	Normal Lock	Normal Lock
	934	3(2),3(1)	noTrain	noTrain	noTrain	TrainUP	TrainDOWN	noTrain	(3,Y)	Reverse lock	Reverse lock
	931	4(1),4(2)	noTrain	TrainUP	noTrain	TrainDOWN	noTrain	noTrain	(4,Y)	Reverse lock	Reverse lock
	890	4(2),4(1)	noTrain	TrainUP	TrainDOWN	noTrain	noTrain	noTrain	(4,Y)	Normal Lock	Normal Lock

that there was no collision in all cases. Besides the safety properties, the signalling system should handle the train movement as we expect. Table 5 shows the terminal markings of all cases.

1. Terminal markings no. 4 in Case A is when route request cannot be executed because either tracks are not clear or block cannot be set to Going state. The only two possible actions are when `TrainUP` moves from `TOL_SOUTH` to 1T and `TrainDOWN` moves from `TOL_NORTH` to 2T.

2. Terminal marking no. 78 in Case B is when `TrainDOWN` from `TOL_NORTH` is brought to platform track 42T by route no. 4(1). But both trains on platform tracks 41T and 42T cannot leave and route 16 or 24 cannot be set because `TrainUP` on track 1T obstructs the traffic. After route released, points 103 and 104 are *Normal* and *unlocked*.

3. Terminal marking no. 95 in Case B is when `TrainUP` from `TOL_SOUTH` is brought to platform track 42T by setting route no. 3(2). Then the signalman sets request block down and route no.16 permitting `TrainUP` on track 42T going south but the `TrainUP` plans going north instead. Because route 16 is set, points 103 and 104 are *Normal* and *locked*.

4. Terminal marking no. 212 in Case B is when `TrainUP` from `TOL_SOUTH` is brought to platform track 42T by setting route no. 3(2). Then, the signalman requests block down set to `Going` state and brings `TrainDOWN` on 41T to `TOL_SOUTH` by setting route no. 24. After that, `TrainDOWN` from `TOL_NORTH` is brought to 41T by route no.4(2). After route released, points 103 and 104 are *Reverse* and *unlocked*.

5. There are eight terminal markings in Case C.

5.1 Terminal marking no. 958 is when route 3(1) is set and `TrainUP` from `TOL_SOUTH` is brought to platform track 41T. Then `TrainDOWN` from `TOL_NORTH` is brought to platform track 42T by route no. 4(1). After route released, points 103 and 104 are *Normal* and *unlocked*.

5.2 Terminal marking no. 957 is when route 4(2) is set and `TrainDOWN` from `TOL_NORTH` is brought to platform track 41T. Then `TrainUP` from `TOL_SOUTH` is brought to platform track 42T by route no. 3(2). After route released, points 103 and 104 are *Normal* and *unlocked*.

5.3 Terminal marking no. 959 is when route 4(1) is set and `TrainDOWN` from `TOL_NORTH` is brought to platform track 42T. Then `TrainUP` from `TOL_SOUTH` is brought to platform track 41T by route no. 3(1). After route released, points 103 and 104 are *Reverse* and *unlocked*.

5.4 Terminal marking no. 960 is when route 3(2) is set and `TrainUP` from `TOL_SOUTH` is brought to platform track 42T. Then `TrainDOWN` from `TOL_NORTH` is brought to platform track 41T by route no. 4(2). After route released, points 103 and 104 are *Reverse* and *unlocked*.

5.5 Terminal marking no. 891 is when route 3(1) is set and `TrainUP` from `TOL_SOUTH` is brought to platform track 41T. Then the signalman sets route no. 3(2).

5.6 Terminal marking no. 934 is when route 3(2) is set and `TrainUP` from `TOL_SOUTH` is brought to platform track 42T. Then the signalman sets route no. 3(1).

5.7 Terminal marking no. 931 is when route 4(1) is set and `TrainDOWN` from `TOL_NORTH` is brought to platform track 42T. Then the signalman sets route no. 4(2).

5.8 Terminal marking no. 890 is when route 4(2) is set and `TrainDOWN` from `TOL_NORTH` is brought to platform track 41T. Then the signalman sets route no. 4(1).

According to Panthong's control table, SRT allows only one passage of the train within the station area. If the station is so crowded, traffic is obstructed and the number of possible train movements is small. On the other hand if the station is empty, the number of possible movements will be large. This behaviour is witnessed by the state space size in Table 4. Besides no train collision, all terminal markings in Table 5 demonstrate that our CPN model works as we expect. We do not discover any errors in the Panthong's control table.

8 Conclusions and Future Work

This paper has outlined an approach for developing a CPN model of SRT's railway signalling system. We mainly focus on modelling the control tables and propose an approach such that the model can be adapted and reused to create other control table models rapidly. We separate the CPN model into two parts: *Signalling Layout* and *Interlocking*. The *Signalling Layout* part comprises net structures which depend on signalling plans. This CPN model mimics the signalling layout and stores information about how each piece of equipment connects to each other. Contrast to the *Signalling Layout* part, the *Interlocking* part does not depend on the signalling plan and has the contents of the control tables coded in twelve ML functions. To assist model development for other CPN models of railway interlocking we propose to use XSLT transforming the control table in XML to the ML functions.

This paper also demonstrates the analysis of three scenarios. We trace terminal markings in each scenario and find that our CPN model works as we expect. The generated state spaces are verified against the desired property that there is no train on two consecutive track circuits. We search the entire state space in all cases and do not find any marking that contains trains on two consecutive tracks. These results convince us of the correctness of the CPN model and the control table.

Although we start with modelling and analysis of a small station, the model development begins with a lot of assumptions. We would like to relax these assumptions and refine the

model in the future. In particular, the assumption that the train occupies one track at a time is unrealistic and sometimes we need two consecutive tracks occupied in order to trigger an event.

In the future we wish to directly generate our CPN model from the signalling layouts and control tables. However nowadays SRT has not yet standardized the file format for the layouts and control tables. The automatic model generator requires a library of CPN patterns for signalling equipment and interlocking which need further investigation.

Acknowledgements : The authors are grateful to the reviewers for their comments that have helped to improve the quality of this paper.

References

1. A. Borälv. Case study: Formal Verification of a Computerized Railway Interlocking. *Formal Aspect of Computing*, 10(4):338–360, 1998.
2. J. Bjørk, A. M. Hagalisletto, and P. Enger. Large Scale simulations of Railroad Nets. In *Proceedings of the Fourth International Workshop on Modelling of Objects, Components and Agents, MOCA'06,Bericht 272, FBI-HH-B-272/06*, pages 45–101, June 2006.
3. C. Chevilat, D. Carrington, P. Strooper, J. G. Süß, and L. Wildman. Model-Based Generation of Interlocking Controller Software from Control Tables. In *Proceeding of ECMDA-FA 2008*, volume 5095 of *Lecture Notes in Computer Science*, pages 349–360. Springer, Heidelberg, 2008.
4. A. Cimatti, E. E. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: A new symbolic model verifier. In *Proceedings of International Conference on Computer Aided Verification, CAV'99*, volume 1633 of *Lecture Notes in Computer Science*, pages 495–499. Springer Verlag, 1999.
5. M. Clavel, F. Duran, S. Eker, P. Lincoln, N. Mart-Oliet, J. Meseguer, and C. Talcott. Maude Manual. http://maude.cs.uiuc.edu/.
6. K. Czarnecki et al. The Future of Train Signaling. In *Proceedings of MoDELS 2008*, volume 5301 of *Lecture Notes in Computer Science*, pages 128–142. Springer Verlag, 2008.
7. W.J. Fokkink and P.R. Hollingshead. Verification of Interlockings: from Control Tables to Ladder Logic Diagrams. In *Proceedings of 3rd Workshop on Formal Methods for Industrial Critical Systems (FMICS'98)*, pages 171–185, Amsterdam, May 1998. Stichting Mathematisch Centrum.
8. Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1993.
9. A. M. Hagalisletto, J. Bjørk, I. C. Yu, and P. Enger. Constructing and Refining Large-Scale Railway Models Represented by Petri Nets. *IEEE Transactions on Systems, Man, and Cybernetics, Part C*, 37(4):444–460, 2007.
10. K. M. Hansen. Formalizing Railway Interlocking Systems. In *Nordic Seminar on Dependable Computing Systems*, pages 83–94. Department of Computer Science, Technical University of Denmark, 1994.
11. C. W. Janczura. *Modelling and Analysis of Railway Network Control Logic using Coloured Petri Nets.* PhD thesis, School of Mathematics and Institute for Telecommunications Research, University of South Australia, Adelaide, Australia, August 1998.
12. K. Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use. Vol. 1, Basic Concepts.* Monographs in Theoretical Computer Science. Springer, Heidelberg, 2nd edition, 1997.
13. J. L. Petersen. Automatic Verification of Railway Interlocking Systems: a Case Study. In *Proceedings of 2nd workshop on Formal methods in software practice (FMSP'98)*, pages 1–6, New York, 1998. ACM.
14. K. Winter. Model Checking Railway Interlocking Systems. In *Proceeding of the 25th Australian Computer Science Conference (ACSC 2002)*, volume 24, pages 303–310. Australian Computer Science Communications, 2002.
15. K. Winter, W. Johnston, P. Robinson, P. Strooper, and L. van den Berg. Tool Support for Checking Railway Interlocking Designs. In *Proceeding of the 10th Australian Workshop on Safety Related Programmable Systems (SCS'05)*, pages 101–107. Australian Computer Science Communications, 2005.
16. K. Winter and N. Robinson. Modelling Large Railway Interlockings and Model Checking Small Ones. In *Proceeding of the Australian Cumputer Science Conference (ACSC 2003)*, volume 25, pages 309–316. AAustralian Computer Science Communications, 2003.

Towards Formal Modeling and Analysis of BitTorrent using Coloured Petri Nets

Jing LIU[1, 3], Xinming YE[2], Tao SUN[2]

[1] Institute of Computing Technology, Chinese Academy of Sciences, China
[2] College of Computer Science, Inner Mongolia University, Hohhot, China
[3] Graduate University of Chinese Academy of Sciences, China
liujing@ict.ac.cn, {xmy, cssunt}@imu.edu.cn

Abstract. BitTorrent is widely adopted in P2P applications, such as file sharing and video streaming. As intricate communication and concurrency are characteristics of BitTorrent, it is difficult to formally and effectively model its functional behaviors in peer-level. In this paper, a coloured Petri Nets based hierarchical modeling architecture and detailed model instances of BitTorrent are proposed. Then simulation, state spaces analysis and model checking technologies are utilized combinatively to validate the formal models and verify the functional properties of BitTorrent. The proposed formal model could not only be served as an unambiguous and visual formal specification, but also facilitate the behaviors simulation and properties verification where it relieves the notorious state space explosion problem.

Keywords: BitTorrent, coloured Petri Nets, simulation, state space analysis

1 Introduction

Various peer-to-peer (P2P) applications have become prodigiously popular on the Internet. BitTorrent [1] accounts for more than a half traffic of all P2P traffic in most countries, so it is widely recognized as a more popular P2P content distribution protocol. The advantages of BitTorrent are high scalability, stable distribution performance and easy deployment, all of which result from the basic idea of BitTorrent, that is, individual peers could effectively utilize their incoming access link bandwidth. Peers participate in an application level overlay network and behave as both client and server. They download and upload file pieces mutually at the same time. This kind of cooperative behavior makes the file sharing more effective and more efficient. However, it also introduces more intricate communication and concurrence, making the functional behaviors modeling and analysis of BitTorrent more difficult.

In recent years, there are many notable studies concerning about different aspects of BitTorrent, such as system design, traffic measurement, performance analysis and key algorithms optimization. As for the modeling of BitTorrent, most researches focus on the performance modeling and analysis. [2] utilizes a simple fluid model to describe and analyze the dynamic behaviors of the BitTorrent systems. [3] utilizes a Markov model to analyze the freerider phenomenon, where a peer just downloads from others without uploading its contents as expected. [4] extends the fluid model

mentioned in [2] to perform extensive measurement and trace analysis of a single-torrent system. [5] adopts a simple mathematical model to characterize the group-level properties of BitTorrent system execution and perform a complex observed performance analysis. [6] models the whole downloading process as several consequent phases using a three-dimension Markov chain, which is suitable for capturing the peer downloading behaviors compared with real world traces. It also launches notable analysis of the stability of BitTorrent. [7] uses a stochastic fluid model to characterize the behavior of on-demand stored media content delivery based on BitTorrent. It provides insight into transient and steady-state system behavior for its performance evaluation.

To summarize, most of the studies adopt various mathematical models to analyze the performance of BitTorrent, and they usually focus on the aggregate properties, such as average downloading or uploading rates, network utilization and cost, etc. Few studies focus on the functional behavior modeling in peer level, which aims to construct a formal function model of BitTorrent and validate its soundness.

Therefore in this paper, a coloured Petri Nets based function model of BitTorrent system is proposed and effectively validated. Coloured Petri Nets are quite suitable for modeling and validating the system in which concurrence, communication and synchronization play a major role. Using coloured Petri Nets to model BitTorrent system is absolutely a good choice, because these models not only specify the functional details hierarchically and unambiguously, but also present visible execution of the concurrent behaviors of BitTorrent. Our contributions are listed as follows:

+ *A modeling architecture of BitTorrent is proposed.* It presents significant guidance about model hierarchy, data abstraction and model refinement. It also applies to modeling other protocols with intricate communication and concurrence as their behavior characteristic like BitTorrent.
+ *A coloured Petri Nets based hierarchical model of BitTorrent is constructed.* To the best of our knowledge, it is the first time to present a function model of BitTorrent in peer level. It could not only be served as an unambiguous and visual formal specification for different system implementations, but also facilitate the behaviors simulation and properties verification of BitTorrent.
+ *An effective model validation and analysis method is presented.* Taking full advantage of CPN Tools, an integrated validation and analysis method is performed, combining simulation, state space analysis and model checking technologies. They are used towards different abstract levels of above models to validate the model soundness and effectiveness, and check whether those models satisfy the key requirement properties of BitTorrent system, such as no out-of-orders executions, or random downloading behaviors, etc.

The rest of this paper is organized as follows. As the background of our studies, section 2 presents an overview of BitTorrent system and coloured Petri Nets with CPN Tools. Then, section 3 describes the modeling architecture as a general guidance, and specific model instances are constructed and explained in section 4, together with some modeling assumptions and data modeling. Section 5 focuses on model validation and analysis, which is used to confirm the validity of formal models proposed in section 4. Finally, section 6 concludes the paper and sketches our future research issues.

2 Background

2.1 BitTorrent Overview

BitTorrent aims to facilitate fast downloading of popular files. According to the informal specification of BitTorrent [1], a sharing file is divided into several pieces of fixed size (e.g. 256 KB each), which are basic sharing units during file distribution. A special torrent file, as the uniform identification of the source file, is made to record the information and data hash of these pieces. All participating peers that download or upload the same file will form a random and temporary mesh network, and they download different pieces from different peers. If there are numerous simultaneous peers sharing the same file, downloading an entire copy will become faster.

In a BitTorrent distribution network, there are two kinds of peers: *leecher*, who downloads its lack and uploads its having at the same time, and *seed*, who has entire file and just uploads for other leechers. Besides, there is a *tracker* to keep tracking information (e.g. peer positions and their downloading progress) of all the participating peers dealing with the same file. It stores and maintains such information, and uses them to organize the peer list, which helps a new leecher find peer candidates to connect. As illustrated in figure 1, a leecher firstly asks the tracker for the peer list. Then it establishes connections to those peers in the list which are sharing the same file, and records which pieces they have through bitmap messages. Finally, this leecher requests pieces from other connecting peers, leechers or seeds, and continues the file sharing until it becomes a seed. Many protocols and algorithms play a significant part in above procedures.

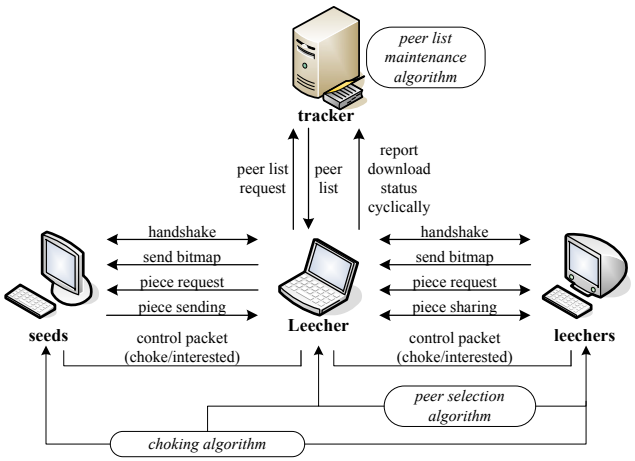


Fig. 1: The major components of a BitTorrent system.

There are two specific protocols in BitTorrent. One protocol specifies the communication between a peer and a tracker, called *tracker protocol*; the other specifies the interaction behaviors between peers, called *peer protocol*. The tracker protocol describes the functionalities including peer list request and response, and peer downloading status cyclical report. It is much simpler compared with the peer proto-

col. The peer protocol describes various interactions during file sharing between a leecher and a seed, or between leechers. It consists of handshake procedure, bitmap exchange, piece request and sending, piece having announcement, and other control communication used to achieve incentive mechanism [1] in BitTorrent.

Besides, there are several key algorithms to promote the efficiency of file sharing, including peer list maintenance algorithm implemented in trackers, and piece selection algorithm and choking algorithm implemented in peers. As described in [1], there are four default piece selection algorithms, i.e. strict priority, rarest first, random first piece and endgame mode. They work collaboratively to improve the whole distribution performance of a BitTorrent system. Choking algorithm is designed to guarantee reasonable and fair downloading speed for every leecher, and put some positive feedback to freeriders.

In practice, a BitTorrent system is very complicated. A peer usually participates in different file downloading and uploading procedures. Learning from the good modeling exercise, we tradeoff between the sufficiency and validity of formal models and simplicity and feasibility of the model analysis. We make full use of hierarchical modeling capability of coloured Petri Nets to model the BitTorrent protocol in several abstract levels, which does not lost the necessary functional details of BitTorrent, together with a modest size of the model for practical analysis.

2.2 CPN and CPN Tools

Coloured Petri Nets (CPN) [8] is always adopted to model and validate systems with high concurrence and complex communication. In recent years, there are many successful projects [11, 12, 13, 14]. CPN has many strong capabilities used to facilitate modeling complex systems, i.e. token colors with abundant data types and operations; hierarchical modeling using substitution transitions, page instances and fusion sets; flexible functional programming language to specify control and data constrains. Such techniques facilitate modeling the BitTorrent into several abstract layers, and expressing data type and functional behaviors accurately and flexibly.

These are several CPN based modeling tools for researcher to do experiments. As far as well known, CPN Tools [9, 10] is the most powerful one. Regarded as industrial-strength software for modeling and analysis CPN models, CPN Tools gains more than 8000 users from nearly 140 countries, and widely adopted in many significant projects [10]. In the paper, we take full advantage of editing, simulation, and state space analysis capabilities of CPN Tools to construct and validate BitTorrent models.

3 Modeling Architecture

The informal specification of BitTorrent [1] just covers the fundamental and necessary parts, such as systems deployment, key data structures, core algorithms, and main flow of contents publishing and files distribution. From the point of view of functional modeling, there are two major hurdles in constructing an accurate and appropriate model based on such specification. On the one hand, some sections in the specification do not need to be modeled, such as system deployment procedure or

some data collection behaviors for user layer displaying. Modeling these behaviors contributes few to system functional analysis, and to make matters worse, introduces incogitable but unnecessary state space explosion. How to distinguish such kind of functionalities from other indispensable parts is a really challenge. On the other hand, some detailed algorithms or message interactions in the specification are not explained clearly or even not mentioned at all. Take the choking and interesting messages for example, the trigger time and orders of such messages interaction are not mentioned clearly. It needs further consideration and complementarities from the perspective of design or implementation phases.

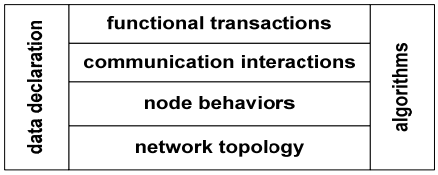


Fig. 2: The modeling architecture of BitTorrent.

In this paper, we propose a non-trivial modeling architecture to cope with above two challenges. As illustrated in figure 2, the modeling architecture contains four layers, which represent different modeling levels, together with data declaration and algorithms layers throughout. We could firstly mode the network topology, and then refine the node behaviors and other upper layer issues, as this paper demonstrated. With the equivalent effects, we could also model detail algorithms and specific functionalities of peer entities firstly, and then compose them to form communication interactions and construct node and network layers. Each layer focuses on certain aspects of system functional behaviors, and the tradeoff between the model size in one layer and the function sufficiency in that layer should be paid more attentions.

Network topology layer focuses on the modeling of entire network environments, including the participating entities and their relationship from the network topology point of view. Especially, the number of different types of entities and their position in the network environments should be considered carefully. Redundant entities or incorrect relations not only introduce a potential huge state space, but also could not give prominence to the key properties of the system.

Node behaviors layer focuses on the execution states and their transfer relation in a specific entity, such as a tracker or a peer node. As for network protocols, sending requests and receiving responses, together with some connectivity control actions are usually modeled in this layer.

Communication interactions layer focuses on messages interactions between protocol entities. As for network protocols, collecting property data, generating requests, parsing response and switching to subsequent processing are major modeling issues in this layer. The logical relationship of such behaviors needs careful consideration.

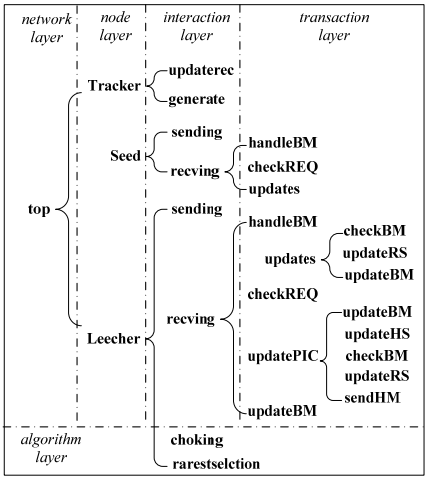
Transactions and algorithms layer focuses on the detailed functionalities, for example, the control flows, maintenance of key data structures, sampling the required data, and core algorithms. The redundant or inaccurate modeling in this layer will lead to notorious state space explosion, especially when the high concurrence and complex communications exist. Therefore, we should iteratively refine the models to construct an optimum model with modest sizes and functional descriptions in different layers.

The model size of this layer is often bigger than that of other layers, so we could divide transaction layer into several sub-layers for legible modeling.

To sum up, taking full advantage of hierarchical abstraction methodology, above modeling architecture facilitates modeling system functionalities into several abstract layers, and expressing behavior details accurately and flexibly. It is quite suitable and feasible for guiding complex system modeling. According to different modeling and analysis purposes, we could adjust the modeling scale inter-layer and inner-layer, and perform efficient analysis in suitable layers. CPN is considered to be an effective actualization of above modeling architecture, and the following sections demonstrate the validity of such actualization.

4 CPN Modeling of BitTorrent

Guided by the modeling architecture in above section, we construct an entire BitTorrent CPN model with 44 page instances (24 if replicated page instances are not counted), as shown in figure 3. This model assumes the absence of exceptions, that is, communication infrastructure is reliable, and there are no vulnerabilities during the protocol execution. Section 4.1 discusses some function related assumptions in modeling. Key date types are modeled as different color sets in section 4.2. From section 4.3 to 4.7, we present BitTorrent CPN models in different layers respectively. Six specific page instances are presented as a representative. They cover all modeling layers, and the most significant functionalities of BitTorrent.



pages	descriptions
top	describe the entire network topology
Tracker	describe the genaral behavior of a tracker
updaterec	update tracking information
generate	generate and send response packet with required peer list
Seed	describe the genaral behavior of a seed peer
sending	send communication packets of peer protocols
recving	receive and phrase communication packets
handleBM	describe behaviors as receiving bitmap message
checkREQ	generate a new piece packet as receiving request packet
updates	describe behaviors as receiving choking related messages
Leecher	describe the genaral behavior of a leecher peer
sending	send communication packets of tracker or peer protocols
recving	receive and phrase communication packets
handleBM	describe behaviors as receiving bitmap message
updates	describe behaviors as receiving choking related messages
checkBM	look up the required peer bitmap information
updateRS	generate a new request packet
updateBM	update bitmap information
checkREQ	generate a new piece packet as receiving request packet
updatePIC	describe consequent behaviors as receiving piece packet
updateHS	record the required piece have been stored
sendHM	generate and send have packets
choking	implement choking algorithm
rarestselection	implement the rarest piece selection algorithm

Fig. 3: The entire CPN model of BitTorrent.

4.1 Modeling Assumptions

As discussed in section 3, modeling all aspects specified in original BitTorrent specification [1] will definitely fell into the state space explosion embarrassment. There-

fore, some function related modeling assumptions should be made at first, that is, some complex behaviors should be simplified and some inconsiderable parts should be omitted directly. According to the modeling architecture, we discard the dross and select the essential, so detailed modeling assumptions are listed as follows.

+ *Some inconsiderable functionality is omitted.* Firstly, web server related processing is beyond the core BitTorrent functionalities, and not worth while to analysis. Then, some over-detailed methods, such as Bencoding and Hash checking [1], are modeled just as a transition under the reasonable assumption that they go wrong in a very small probability. At last, the hash value of torrent file is used to indentify each sharing file instead of using the whole torrent file. The generating and parsing of torrent file are omitted together.
+ *Only single file sharing is considered.* The single file sharing scenario covers all functionalities of BitTorrent, and is more feasible for analysis, so without losing the generality, we carry out simulation and state space analysis based on single file sharing configuration in BitTorrent executing.
+ *File piece is the basic sharing unit.* According to the original specification, the basic request unit of a file is slice [1], and a piece is composed of several slices. Taking into account the similar processing behaviors, we adopt piece as basic sharing unit for simpler analysis.
+ *Some less important mechanisms are simplified.* Endgame mode [1] is just a piece selection optimized method in the end phase of file downloading. Not modeling its behaviors does not affect the main functionalities of BitTorrent, and avoid introducing huge concurrent state space. Besides, as for the choking algorithm, we just model the fundamental part without optimistic unchoking and anti-snubbing [1], which are used for improving the performance and fairness of BitTorrent.

To sum up, above modeling assumptions are necessary to focus our modeling issues on the most significant parts of BitTorrent, and control the size of CPN models for effective and efficient behaviors analysis.

4.2 Data Modeling

The major data types used in BitTorrent are modeled as different color sets, shown in figure 4. We utilize simple color sets (unit, integer, and string) and compound color sets (product, record, union, and list) together to model property fields (e.g. infohash, peerid, and some flags), communication packets (e.g. handshake packets, choking packets, bitmap packets, request packets, piece packets, have packets) and key data structures (e.g. peer bitmap, piece request set, piece having set) of BitTorrent system.

It is well-known that complex color sets will possibly result in more difficult analysis work. We hold the following principle in data modeling: capturing the indispensable data elements and organizing them using suitable color sets to achieve both clear representation and easy operation. For example, the fields in communication packets used in tracker protocol and peer protocol are less than the original BitTorrent specification, because only necessary fields are picked up, and organized with suitable color sets as simple as possible.

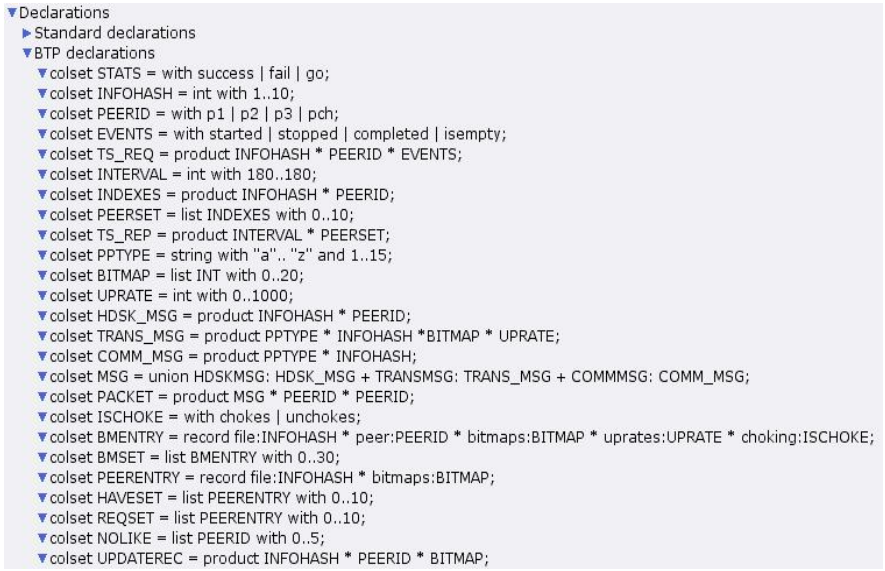


Fig. 4: The color sets in CPN model of BitTorrent.

HSDK_MSG stands for main data fields of handshake messages. TRANS_MSG stands for main data fields of bitmap, request, piece and have messages. COMM_ MSG stands for choke, unchoke, interested and uninterested messages. Union type is used to uniformly model such messages, and combined with source peerid and destination peerid to compose the entire communication packets. Besides, there are three significant data structure: BMSET, representing the piece distribution information of other peers, HAVESET, representing the index of pieces that have been downloaded, REQSET, representing the index of pieces that have been requested but not downloaded. They all modeled as list type for easy data retrieval and update. Furthermore, corresponding variables are named almost the same with its host color sets except for spelling with lowercase, so we do not list them for limited space.

4.3 Network-layer Modeling

Figure 5 indicates the top page of BitTorrent CPN models. It describes the network topology of the BitTorrent system for our analysis. There are one tracker (*Tracker*), one seed (*Seed*), and two leechers (*Leecher1 and Leecher2*). *Leecher1* acts as a new joining peer, and *Leecher2* acts as an existing leecher with part file. Because they have same behaviors, the subpages derived from the substitution transitions Leecher1 and Leecher2 in the top page are the same. We only assign different initial markings of BMSET to identify that *Leecher1* has empty file and *Leecher2* has part file, and *Leecher1* could download the file from both *Seed* and *Leecher2*. The underlying transmission network is model as several places with color set PACKET. These four entities compose a least topology set which could cover the whole desired functionalities of BitTorrent, and the protocol executions among these entities are already very complicated for feasible and effective model analysis processing.

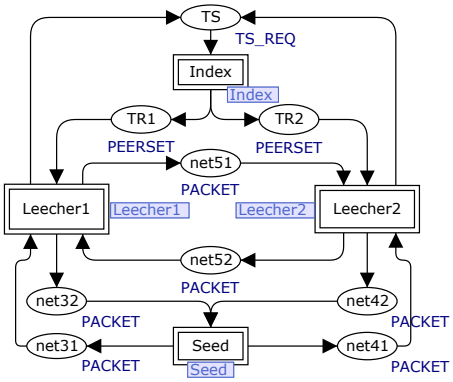


Fig. 5: The network topology model.

4.4 Node-layer Modeling

We choose the leecher model as a representative of node layer models because the tracker model is simpler and the seed model is just a bridged edition of leecher model without piece requesting related functionalities. Figure 6 indicates the specific behaviors of a leecher. The key aspect of modeling in this layer is to outline the protocol execution flows as a whole. For example, the main body of leecher behaviors is various communications involved in the tracker protocol and the peer protocol, together with their temporal orders. According to this principle, we model four kinds of leecher behaviors: peer list request, peer list parsing, packets generation and sending of the peer protocol, packets receiving and parsing of the peer protocol. These behaviors interact causally and cooperatively.

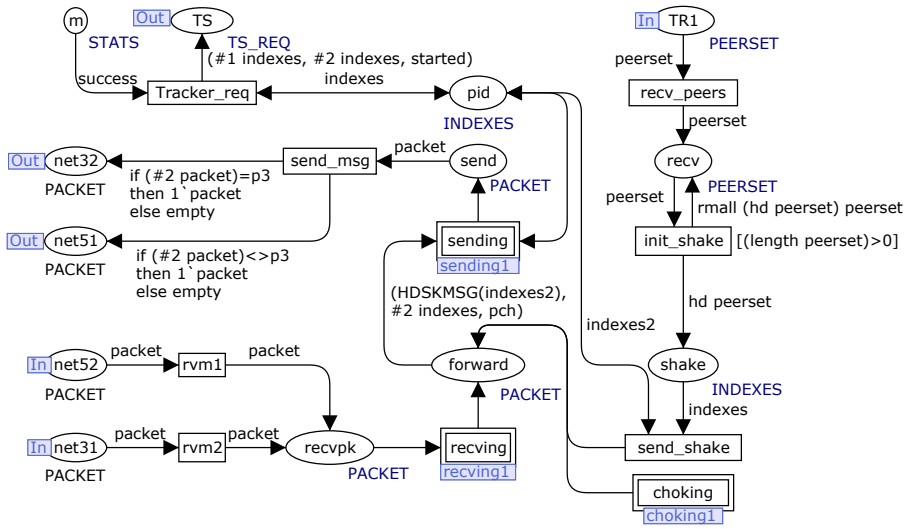


Fig. 6: The model of a leecher.

4.5 Interaction-layer Modeling

Modeling in the interaction layer focuses on the processing of protocol packets. As for the tracker protocol and peer protocol in BitTorrent, generating requests from data fields, parsing responses and switching to subsequent processing respectively are major modeling issues. As a typical example, figure 7 presents the behaviors when a leecher receives different packets from other connected peers. It is actually a subpage of substitution transitions *receive* in figure 6. On receiving handshake packets, the leecher will generate the corresponding bitmap message after successful handshake verification. On receiving (un)choke or (un)interested packets, the leecher will continue subsequent steps, which are modeled as another substitution transitions *updates* in detail. On receiving bitmap, request, piece or have packets, the leecher will carry out respective processing actions modeled as different substitution transitions.

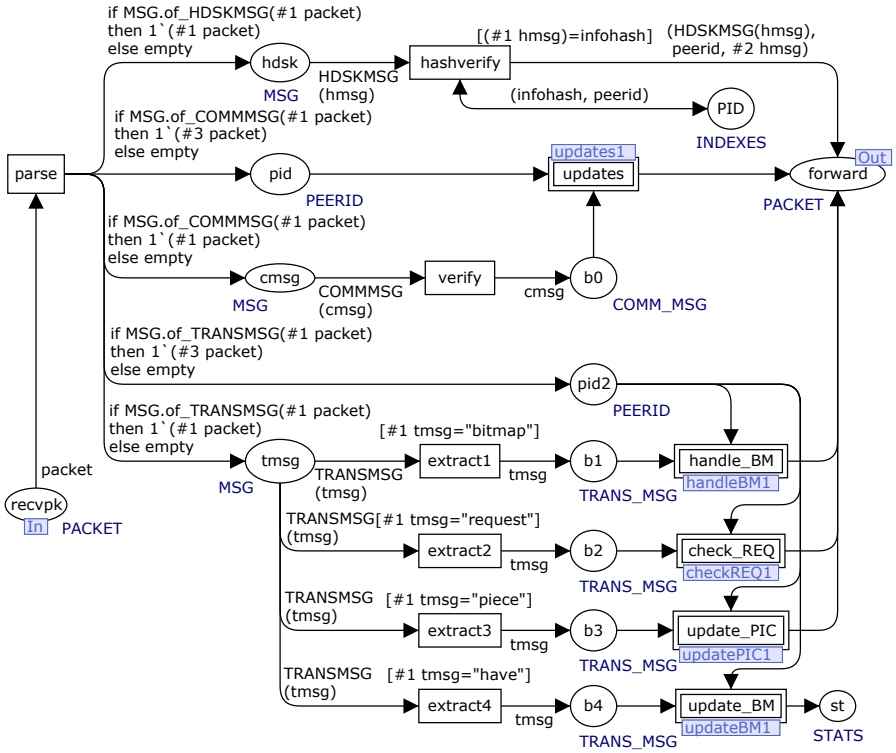


Fig. 7: The receiving behavior model of a leecher.

Detailed processing of choking and interesting related messages (COMM_MSG) is modeled with an identical substitution transition, because processing methods to these messages are similar and simple. But detailed processing of bitmap, request, piece and have messages (TRANS_MSG) is modeled with respective substitution transitions, because processing methods to these messages are more complex and different with each other. So adjusting the model scale inter-layer and inner-layer is quite helpful to obtain the modest model size for feasible analysis.

4.6 Transaction-layer Modeling

Transaction layer are fundamental page instances to model specific functionalities of BitTorrent. Modeling in this layer requires many tradeoffs to pursue the golden section of modest model size, so iterative model refinement is indispensable and significant. Two kinds of transaction pages are exemplified as follows. Figure 8 presents the behaviors when a leecher receives choking or interesting packets. It contains some substitution transitions to describe more detailed behaviors, while figure 9 is an absolute leaf page instance with no substitution transitions. It indicates the behaviors that a leecher firstly checks and calculates which piece should be requested and then generates request message with available data.

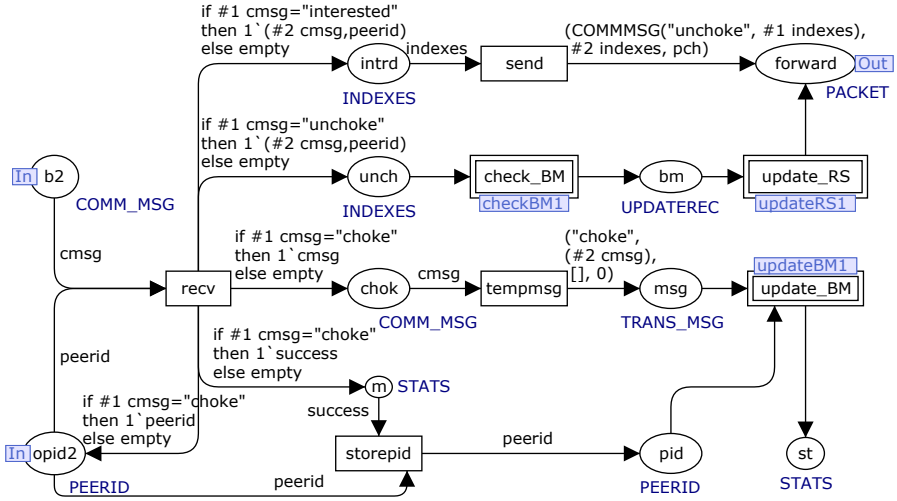


Fig. 8: The transaction behavior model of a leecher.

The size of page instances in this layer tends to be large, and any redundant or inaccurate behaviors modeling will lead to serious state space explosion. Based on our experience, two kinds of problems are worth notice. One is about concurrent operations semantics. There often exist some seeming concurrent actions, which could be modeled sequentially without any harm to protocol functionalities. For example, when a leecher has received a piece, it should update HAVESET structure and request a new piece. These two behaviors are independent, and could execute concurrently or sequentially. If model them as concurrently execution, many unnecessary concurrent states will be introduced, so we coercively arrange the execution order of these actions, that is, a control place is added to make corresponding transitions fired sequentially. The other problem is about the balance between the complexity of the net structure and the data inscriptions. It is wheezy but vital. Considering CPN Tools provides powerful ML programming language for describing constrains, we prefer specifying ML inscriptions to introducing new places or transitions when modeling some exception behaviors. Taking retrieving list data as an example, transition guard inscription is used to model null-list checking instead of making a new transition. The former does no harm to protocol functionalities, and reduces much redundant states.

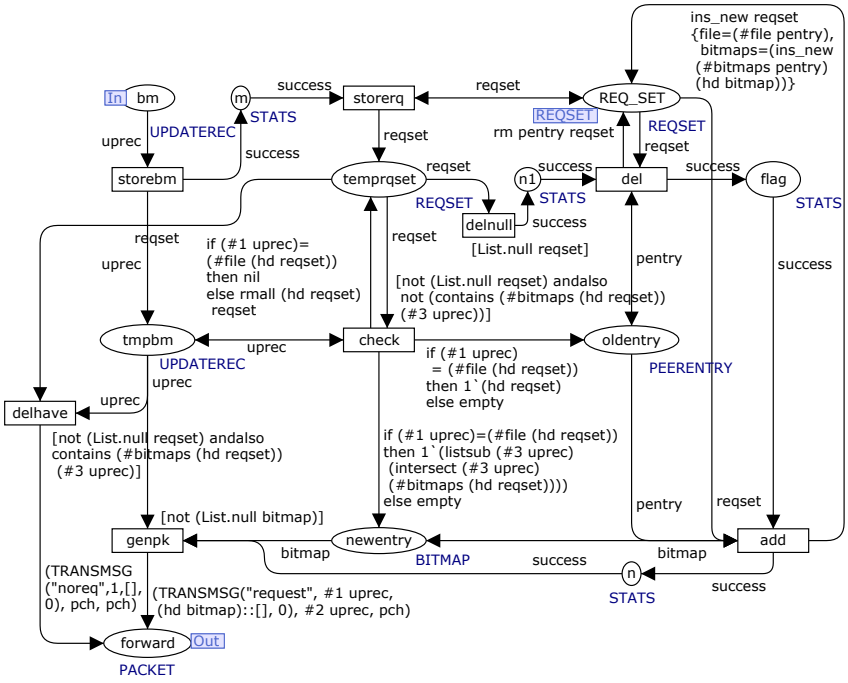


Fig. 9: The transaction behavior model of a leecher.

4.7 Algorithm-layer Modeling

There are several algorithms implemented in BitTorrent. Choking algorithm and piece selection algorithms are most important.

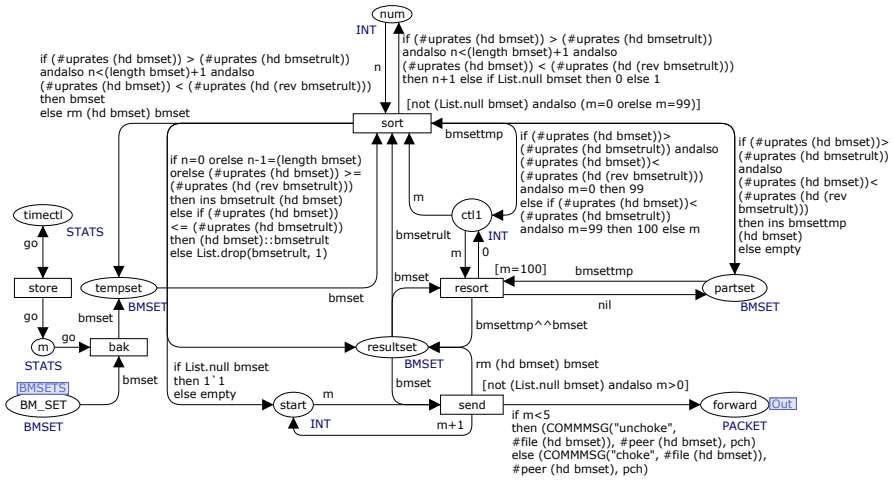


Fig. 10: The choking algorithm model of a peer.

BitTorrent has no central resource allocation, and each peer tries to maximize its own download rate based on local conditions. A peer uploads pieces to certain leechers according to the download rate they obtain from such leechers. It is a variant of tit-for-tat. In order to achieve better performance, a peer usually chokes some non-active peers temporarily, and readjusts the choking peer list periodically. The choking algorithm specified in BitTorrent is composed of three parts: the basic choking algorithm, the optimistic unchoking algorithm and anti-snubbing algorithm. In our CPN models, only the basic choking algorithms are modeled. As shown in figure 10, the main behavior in the algorithm is to order the entries in BMSET according to the download rates. The first four are considered as unchoking peers and others as choking peers, and corresponding choke or unchoke packets are sent respectively.

Piece selection algorithms focus on selecting pieces to download with modest orders for better performance. There are four piece selection algorithms. *Strict Priority* is concerned about slice downloading, that is, once a slice has requested from one peer, the remaining slices in that piece are also requested from the same peer. We do not model this algorithm because we just model the piece level behaviors as explained in section 4.1. *Rarest First* is the core algorithm in piece selection. Considering all connecting peers with a certain peer, if a piece has least copies storing among these peers, this piece should be downloaded firstly. This algorithm guarantees that the rarest pieces could be distributed as quickly and early as possible. Similar to choking algorithm, the main behavior of rarest first algorithm is to order the entries in BMSET according to the number of each piece storing in other connecting peers, and the least pieces are put forward. Because of behavior similarity to choking algorithm, models of this algorithm are not presented for space limitation. *Random First Piece* is used when downloading starts for obtaining a complete piece as soon as possible. We utilize a random initial marking to model this algorithm. At last, *Endgame Mode* is used to conquer the problem where the last piece is usually hard to get. As mentioned in section 4.1, this algorithm is not modeled because of serious concurrent behaviors.

In our constructed CPN models, we trigger the choking algorithm at the time that piece request starts. Also, we consider the executing of algorithm as atomic events, that is, the piece request procedure will never start unless the choking algorithm is over. Without losing the generality, this simplification could reduce huge concurrent behaviors and make protocol analysis more practicable. But in fact, the choking algorithm are essentially time-driven, that is, it works periodically and independently, so we will try to utilize time modeling capability provided by CPN Tools to refine algorithm models in further research issues.

5 Analysis of BitTorrent CPN Models

Having constructed the CPN models of BitTorrent, we should make further analysis to validate and revise the model, that is, to validate the effectiveness of models, and check whether those models satisfy the key requirement properties of BitTorrent system, such as no out-of-orders executions, or random downloading behaviors, etc. Unfortunately, as concurrence and intricate communication are essential characteristics of BitTorrent systems, the constructed models are so large that the direct state spaces

analysis becomes infeasible because of the notorious state space explosion problem. In order to launch practical analysis, and make it as complete and reliable as possible, we introduce an integrated method which combines CPN Tools supported simulation, state space analysis and model checking technologies, and uses them towards different profiles of the models.

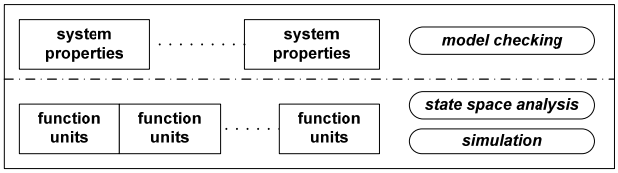


Fig. 11: The analysis framework of BitTorrent models.

As shown in figure 11, our analysis work is composed of two aspects. In the bottom, function unit is a basic functional flow of the protocol execution, for example, a leecher asks the peer list of the sharing the file, or a leecher downloads a piece from another peer. Based on the same models, different specific initial marking assignments can form different function units, that is, different initial marking will result in different executions of the protocol. Several function units could execute sequentially or concurrently to form a more complex functionality. We adopt simulation to validate the function unit, together with some state spaces analysis to check basic properties of them, such as boundedness, liveness, or deadlock checking. Just like the relationship between unit testing and system testing, based on the correct function unit, some higher properties of the protocol system should be verified, such as no out-of-orders executions, or a peer downloads pieces randomly, etc. Such properties are usually described as some temporal logics and verified using model checking technologies. In this paper, we use ASKCTL [18], provided by the CPN Tools, to describe such properties, and exemplify some key properties verification processes based on abstract models. From the point of view of model validation, function units simulation and analysis help validate the effectiveness of protocol detailed behaviors, and higher properties checking help verify the satisfiability to protocol requirements.

5.1 Function Units Validation

During the process of model construction, simulation is frequently performed to check whether the model behaves as expected. Because the simulation has immediate visual feedbacks, it is quite useful in finding modeling errors. Especially, the single-stepping through the simulation is very helpful to understand the details of original protocol specification, and make necessary refinement to CPN models. It is a good way to modify the model immediately when such simulation is performing.

According to sufficient simulations of BitTorrent CPN models, we find that most of concurrent behaviors existed in the model could be serialized. For example, when a piece packet is received, the BMSET should be updated and some new piece requests should be sent. These two behaviors execute independently, and if we model them as two independent substitution transitions, they will introduce concurrence in protocol running, together with large state space. In fact, those concurrent behaviors are not

intrinsically concurrent, and they could be serialized by assigning an execution order in the model manually. As discussed in section 4.6, we coercively arrange the execution order of these behaviors to effectively reduce much unnecessary state space.

Unfortunately, there are still some true concurrent behaviors happened in BitTorrent CPN models. For example, when a leecher receives a peer list from the tracker with at least two peer candidates, it will connect to both concurrently for different piece requests. As shown in figure 12, the left trace (nodes 11->12->15->19->24->30 ->37->45…) and the right trace (11->13->16->20->25->31->38->46…) respectively indicate that a leecher request pieces from two different peers, and the other traces all present the interleaving executions between these two behaviors. In fact, most of such traces are meaningless for analysis because they are too detailed. Some conflict access of significant data is worthy of consideration, and simulation related capabilities in CPN Tools are strong enough to validate such behaviors because of the visual feedbacks to check whether conflicts really happen. According to such observations and inferences, we generate several function units based on both functionality of protocol and true concurrent behaviors, that is, each function unit represents a relatively independent functional flow of protocol executions with no or controllable true concurrent behaviors. These function units should cover all paths of the model, and their sequentially or concurrently executions form all feasible functionalities of original specification. Towards each function unit, we perform both simulation and state spaces based static analysis to validate the reliable execution of such function unit.

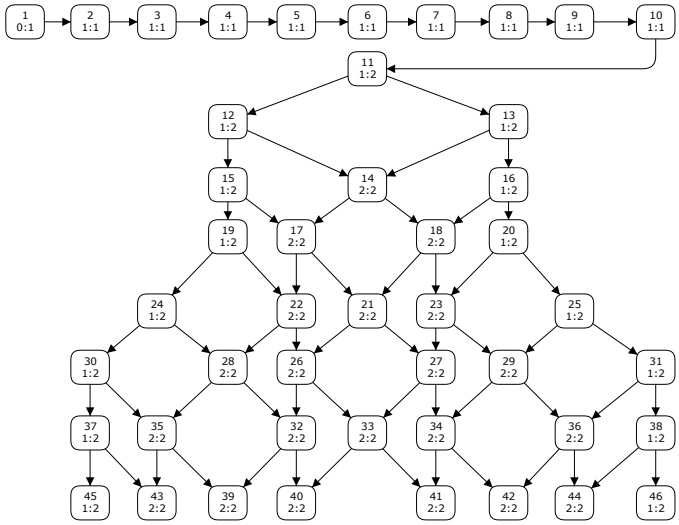


Fig. 12: The example state spaces of BitTorrent CPN models.

In our analysis, four function units are designed:

(1) *Leecher1* firstly asks *Tracker* for peer list of the sharing file (supposed to composed of two pieces), then *Tracker* replies with list containing *Leecher2* and *Seed*, where *Leecher2* has one piece and *Seed* has entire file.
(2) *Leecher1* connects to *Leecher2*, and download one piece without further pieces requests.

(3) *Leecher1* connects to *Seed*, downloads two pieces, and announces *Leecher2* that it has the entire file using piece having packet.
(4) *Leecher1* executes the rarest first piece selection algorithm, and *Seed* executes choking algorithm when receiving piece request from *Leecher1*.

Function unit (2) and (3) can execute concurrently after (1) has finished, and (4) can execute sequentially before (3). These four function units cover all paths of CPN models and major functionalities of BitTorrent systems according to the original specifications. After such function units division, it is simple but representative for model analysis. Firstly, we perform sufficient simulation towards these function units to remove unnecessary concurrence and refine models. The refinement process includes two aspects, correcting the inaccurate behaviors modeling and abstracting the redundant behaviors without meaningless details. Then, based on these function units, automatic state spaces analysis provided by CPN Tools is performed to check basic properties of such function units. The analysis results of above four function units are overviewed in table 1.

Table 1: State space analysis result of four function units.

function units	*state space status*	*nodes*	*arcs*	*boundness*	*home markings*	*dead markings*	*live transitions*
1	full	10	9	normal*	last marking+	last marking	none
2	full	100	99	normal	last marking	last marking	none
3	full	4910	8978	normal	none	last marking	none
4	full	18	17	normal	none	none	all
1+2+3	partial#	60358	111126	—	—	—	—

* *"normal" indicates no exceptions existing in boundness checking.*
\+ *"last marking" indicates the state in models where function unit executes successfully and terminates.*
\# *"partial" indicates it can not generate full state spaces under the time limitation of 1000 seconds.*

5.2 Model Checking of System Properties

After above validation procedures, function units are verified thoroughly. Each function unit execution starts from a specific initial marking and with no or controllable concurrent behaviors, therefore, the state space generated for this function unit only contains the states that could be reached from that initial marking, and the size of such state space is usually not too large to analysis. However, if we focus on checking higher system properties, such as mutual relationship among function units or system level requirements, we need full state space to enumerate every possible execution of protocol systems. Unfortunately, based on BitTorrent CPN models constructed in section 4, the state space explosion happens that we can not utilize advanced state spaces queries [17] or model checking technologies [18] to verify such properties.

In this paper, instead of considering the concrete full state space generated from original CPN models constructed in section 4, we check higher properties over a finite abstraction. According to modeling architecture mentioned in section 3, the abstract models only cover network, node and interaction layers. More specifically, the network and node layers in abstract models remain the same as the original models, and the interaction layers in abstract models are modeled as leaf page instances without

substitution transitions, that is, replacing the substitution transitions in original models with ordinary transitions. Besides, the abstract models contain some new places representing key data structures, the same as that appeared in original models. Such abstraction takes effect just because on one hand the functionalities of original transaction layer or algorithm layer model have been validated, the ordinary transitions could represent equal and valid functionalities as original substitution transitions, and on the other hand, original transaction layer models are always independent in functionalities with each other except for accessing the common data structures, so we reserve these data structures in new abstract models to keep the interaction relationship between corresponding behaviors. The abstract models could effectively relieves the notorious state space explosion, and make model checking feasible. This kind of abstraction could be considered as a kind of over-approximation. On checking higher properties on the abstract models, if the property passes verification, it also holds in original detailed models. Otherwise, simulation is utilized to find out the reason of failed verification: modeling error, protocol defects, or inaccurate abstraction.

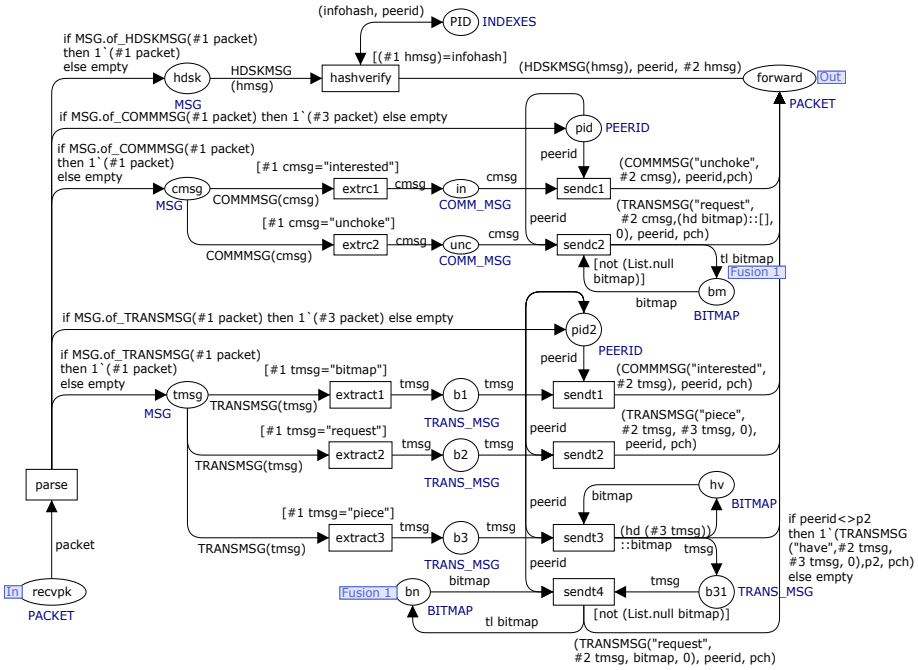


Fig. 13: The abstract receiving behavior model of a leecher.

As a representative, figure 13 presents packet receiving behaviors of a leecher in abstract models. Compared with figure 7, it has no substitution transitions, and only focuses on protocol communication behaviors. The newly constructed abstract model has 10 page instances in total. We assign different initial markings to conduct different execution flows. The rationality of such assignments has been validated in function unit analysis. As an example, we consider the concurrent execution of function units (2) and (3) defined in section 5.1. The full state space contains 9180 states and 22546 arcs. There are no home markings, no live transitions and 16 dead markings.

Using “State Space to Sim” tool in CPN Tools, we could clearly observe the color tokens in certain places, which proves that those dead markings exactly correspond to different concurrent execution results. Based on full state space, we also perform higher properties verification using ASKCTL based model checking. For example, considering a situation that a peer receives a piece without having received a unchoke message before, we specify *BTFormula* to check such situation never happens. Because ASKCTL has no definition of formula like “A→B”, we use its equivalent form as “OR(NOT A, B)” instead. Figure 14 presents the property description and checking result. This kind of properties usually referred as safety properties, and hard to simulate manually. Its successful verification (val it = true: bool) indicates that both abstract and detailed models behave accurately according to the property. Many other safety properties could be checked effectively and efficiently in the same way.

```
use (ogpath^"/ASKCTL/ASKCTLloader.sml")

fun IsUnchoke a =
(Bind.receives1'sendc2 (1, {peerid=p2, cmsg=("unchoke",1), bitmap=[1,2]}) = ArcToBE a);

fun IsRecvPiece a =
(Bind.receives1'sendt3 (1, {peerid=p2, tmsg=("piece", 1, [1], 0), bitmap=[]}) = ArcToBE a);

val BTFormula =
INV(OR(MODAL(AF("Unchoke", IsUnchoke)), NOT(MODAL(AF("ReceivePiece", IsRecvPiece)))));

eval_node BTFormula InitNode;
```

```
val IsUnchoke = fn : Arc -> bool
val IsRecvPiece = fn : Arc -> bool
val BTFormula =
 NOT
   (EXIST_UNTIL
     (TT,
      NOT
        (OR
          (MODAL (AF ("Unchoke",fn)),
           NOT (MODAL (AF ("ReceivePiece",fn)))))))) : A
val it = true : bool
```

Fig. 14: The ASKCTL based model checking of higher properties.

From above property verification process, it is clear that this abstraction guided checking method not only takes full advantage of sufficient validation to function units, but also makes higher properties checking practical and effective. According to our limited experience on network protocols modeling and analysis, this method is always regarded as a cost-efficient choice.

6 Conclusion and Future Research Issues

BitTorrent is one of the most popular protocols used in P2P applications providing fast file distribution and effective file sharing. It has complex communications and concurrent behaviors, which are major hurdles for formal functional modeling and validation. In this paper, towards such complex protocol system, a hierarchical modeling architecture is proposed to facilitate modeling system functionalities into several abstract layers, and expressing behavior details accurately and flexibly. Then, we utilize CPN as an effective actualization of above modeling architecture to construct BitTorrent CPN models, taking full advantage of the industrial-strength modeling capabilities of CPN. Several exemplified CPN pages are presented, and corresponding

modeling techniques are discussed at the same time. To the best of our knowledge, it is the first time to present a functional formal model of BitTorrent in peer level. It could not only be served as an unambiguous and visual formal specification for different system implementations, but also facilitate the behaviors simulation and properties verification of BitTorrent. At last, taking full advantage of strong analysis capabilities in CPN Tools, efficient and sufficient BitTorrent models validation is performed in both function unit level and system requirement level using simulation, state space analysis and model checking technologies together. They are used towards different abstract levels of above models to validate the effectiveness of models, and check whether these models satisfy the requirement properties of BitTorrent.

As for the future research, time factors will be introduces into current CPN models, because the choking algorithm and some peer selection algorithms are essentially time-driven. Besides, continuous improvements of making our BitTorrent CPN models more complete and more efficient need inevitably further studies.

Acknowledgments: This work was supported by the National Natural Science Foundation of China under Grant No. 60863015, the Key Program of Natural Science Foundation of Inner Mongolia of China under Grant No. 20080404ZD20, and the ChunHui Program of the Ministry of Education of China under Grand No. Z2007-1-01042.

References

1. Bram Cohen. Incentives Build Robustness in BitTorrent. In Proceedings of the 1st Workshop on Economics of Peer-to-Peer Systems, page 5, Jun. 2003
2. D. Qiu and R. Srikant. Modeling and Performance Analysis of BitTorrent-Like Peer-to-Peer Networks. In Proceedings of the ACM SIGCOMM'04, pages 367~378, Aug. 2004
3. M. Barbera, A. Lombardo, G. Schembra, etc. A Markov Model of a Freerider in a BitTorrent P2P Network. In Proceedings of the IEEE Globecom'05, pages 985~989, Nov. 2005
4. Lei Guo, Songqing Chen, Zhen Xiao, etc. A Performance Study of BitTorrent-like Peer-to-Peer Systems. IEEE Journal on Selected Areas in Communications, Vol. 25, No. 1:155~169, Jan. 2007
5. Amir H. Rasti and Reza Rejaie. Understanding Peer-level Performance in BitTorrent: A Measurement Study. In Proceedings of the 16th International Conference on Computer Communications and Networks (ICCCN 2007), pages 109~114, Aug. 2007
6. Vivek Rai, Swaminathan Sivasubramanian, Sandjai Bhulai, etc. A Multiphased Approach for Modeling and Analysis of the BitTorrent Protocol. In Proceedings of the 27th International Conference on Distributed Computing Systems (ICDCS 2007), page 10, Jun. 2007
7. K.N. Parvez, C. Williamson, Anirban Mahanti, etc. Analysis of BitTorrent-like Protocols for On-Demand Stored Media Streaming. In Proceedings of ACM SIGMETRICS'08, pages 301~312, Jun. 2008
8. Kurt Jensen. Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use. Springer-Verlag, 2nd edition, Vol. 1~3, 1997.
9. Kurt Jensen, Lars Michael Kristensen and Lisa Wells. Coloured Petri Nets and CPN Tools for Modeling and Validation of Concurrent Systems. International Journal on Software Tools for Technology Transfer, Vol. 9, No. 3-4: 213-254, Springer Verlag, Jun. 2007
10. CPN Tools. Online: http://wiki.daimi.au.dk/cpntools/cpntools.wiki.

11. Jonathan Billington and Amar Kumar Gupta. Effectiveness of Coloured Petri nets for Modeling and Analyzing the Contract Net Protocol. In Proceedings of the 8th CPN Workshop, pages 49~64, Oct. 2007
12. Marko Bago, Nedjeljko Peric and Sinisa Marijan. Modeling Bus Communication Protocols Using Timed Colored Petri Nets - The Controller Area Network Example. In Proceedings of the 9th CPN Workshop, pages 103~122, Oct. 2008
13. Jing Liu, Xinming Ye, Jun Zhang, etc. Security Verification of 802.11i 4-way Handshake Protocol. In Proceedings of the IEEE International Conference on Communications (ICC 2008), pages 1642~1647, May. 2008
14. Panagiotis Katsaros. A Roadmap to Electronic Payment Transaction Guarantees and a Colored Petri Net Model Checking Approach. Information and Software Technology archive. Vol. 51, No. 2: 235-257, Feb. 2009
15. Yanlan Ding and Guiping Su. A Reduction method for Verification of Security Protocol through CPN. In Proceedings of IEEE International Conference on Networking, Sensing and Control (ICNSC 2008), pages 73~77, Apr. 2008
16. Jinan Yi-xin, Lin Chuang, Qu Yang. Research on Model-Checking Based on Petri Nets. Journal of Software, Vol. 15, No. 9:1265-1276, Sep. 2004 (in Chinese)
17. Kurt Jensen, Soren Christensen and Lars M. Kristensen. CPN Tools State Space Manual. Jan. 2006
18. Allan Cheng, Soren Christensen and Kjeld H. Mortensen. Model Checking Coloured Petri Nets Exploiting Strongly Connected Components. Technical Report of Computer Science Department, Aarhus University, Denmark, Mar. 1997
19. Timo Latvala. Model Checking LTL Properties of High-Level Petri Nets with Fairness Constraints. In Proceedings of the 22nd International Conference on Application and Theory of Petri Nets (ICATPN 2001), pages 242~262, Jun. 2001
20. Lisa Wells. Performance Analysis using CPN Tools. In Proceedings of the 1st International Conference on Performance evaluation Methodologies and Tools (ValueTools 2006), page 10, Oct. 2006

A Full Automated Fault Diagnosis System based on Colored Petri Net

A. L.G. Colaço
Companhia Energética do Ceará
anacolaco@coelce.com.br

G.C. Barroso
Universidade Federal do Ceará
gcb@fisica.ufc.br

R.A.Azevedo
Universidade Federal do Ceará
robsonaazevedo@gmail.com

R.P.S. Leao
Universidade Federal do Ceará
rleao@dee.ufc.br

R.F. Sampaio
Universidade Federal do Ceará
rfurtado@dee.ufc.br

E.B de Medeiros
Companhia Energética do Ceará
eudes@coelce.com.br

Abstract

As the power systems increase in size and complexity, the real-time operation becomes an important and demanding task. When the system is under fault condition, a great deal of information reaches the operator monitor desk, making it hard and stressful to take the right decision to restore to normal the system operation. To cope with this situation, a fault diagnosis method based on Colored Petri Nets (CPN) is reported in this paper. The proposed method aims to ease the burden of the power system operators by presenting a fast and accurate diagnosis of faults located in both the substations and the transmission and distribution lines. A fault diagnosis system (FDS) based on CPN has been already developed which is independent of the power system topology. The method is suitable to any type of fault, and it does not require any voltage and current measurements. It can point out the fault type, the faulted phase(s), and whether relays and circuit breaker have failed.

1. INTRODUCTION

The continuous growth and increasing demand for reliability have required the deployment of automatic systems for supervision and control of the power systems. The current Supervisory Control and Data Acquisition (SCADA) systems have contributed to increase the security and reliability of the power systems, providing support to the system operators. However, when a fault occurs in a power system a great deal of information is reported by the SCADA system to the power system control center for evaluation by the system operators. Very often the huge amount of information makes the evaluation a difficult task and it may lead to erroneous interpretation and incorrect diagnosis of the occurrence. In order to sort this problem out fault diagnosis systems (FDS) have been developed to make possible a fast and accurate diagnosis and suitable decision making.

The fault diagnosis of electric power system is a process of discriminating faulted system elements (e.g., a transmission or distribution line, busbar, or transformer) which can be switched off from the system by the tripping of protective relays and circuit breakers. Several tools and different approaches based for example on expert systems [1], artificial neural networks [2], wavelet transforms [3], genetic algorithms

[4], fuzzy logic [5], and Petri nets [6]-[8] have been proposed for fault diagnosis systems development. The proposals may differ on the response speed, accuracy, complexity, maintainability effort, and adaptability to system topology changes.

The FDS is run based on the status changing of the power system relays and circuit breakers. In order to present the fault diagnosis to the system operators in the control center the CPN output is converted into an easy and usual operational language. An offline table is then built by an expert given the CPN output interpretation. This paper aims to automate the fault diagnosis making it independent from the expert interpretation of the CPN output.

This paper is organized as follows. Section 2 outlines some approaches applied for fault diagnosis based on Petri nets. Section 3 describes the fault diagnosis system which uses a lookup table built by an expert on power system operation to give the interpretation for the CPN output. In section 4 a full automated fault diagnosis system is explained where the lookup table is replaced making the diagnosis independent from an expert. Case studies are presented in Section 5, and finally the conclusion is given in section 6.

2. AN OVERVIEW ON PETRI NET APPROACH FOR FAULT DIAGNOSIS

From the power system point of view, an event can be caused by external and internal disturbances which may bring about changes in the energy flow, and as a result it may cause the state of the system to change having an effect on the system operation. Although the power system is considered a continuous-time system, the protection system can be seen as a discrete-time setup, since the state change of the power system devices is well defined. A fault causes the relay state to change from “ready-to-trip” to “tripped”. The relay trip causes the circuit breaker state to change from “closed” to “opened”, and as a result the system changes from “energized” to “de-energized” condition. Due to these characteristics Petri nets have been successfully applied for fault diagnosis.

A method for fault diagnosis applied to power substation based on Place/Transition Petri nets is presented in [6, 7]. The method is of easy implementation as it is based on simple operation of matrices. The FDS determines the fault location, it checks for incorrect/uncertain signals, and identifies the protection equipment that has tripped such as relays and circuit breakers. In [8] an hybrid method to the fault diagnosis in electric power systems is presented based on fuzzy Petri nets. Petri nets are

used for the ability of describing the relation of protective relays and circuit breakers and concurrent operating mechanism along with the fuzzy logic with its special ability to deal with uncertainty and incompleteness of tripping information.

In [11], [12] an extension to the method presented in [6], [7] is given. The method adds the relay function "breaker failure" normally found in the numeric relays, making it possible to consider the sensitivity of relay in the FDS.

A FDS based on Hierarchical Colored Petri Nets (HCPN) for power distribution substation is given in [12]. In this approach the FDS is located at the substation automation level, which reports the substation diagnosis to the control center. A HCPN model is deployed for each substation bay (high voltage bay, medium voltage bay, transformer bay, feeders bay, and compensation bay) encompassing various subnets to model the whole protection system of the substation.

In [13] a FDS based on CPN is proposed, which is able to diagnose faults in the power substations as well as in the network connecting them. A single CPN graph is designed for the whole power system and faults outside the power substation can be also assessed and diagnosed. The model is independent from the power system topology and size as it uses information only about the state of relays and circuit breakers. When the power system experiences a fault, the CP-net is run providing an output marking. An off-line table is built to match the CPN outputs with the assigned interpretation in a language usual to the operators. The off-line table represents the system topology and it is built by an expert in power system protection and updated whenever the power system goes through a permanent change in topology. In order to make the difference between this model and that one presented next, the model that makes use of the CPN output and a lookup table will be referred to as FDS-LookupTable.

A FDS based on CP-net and a set of IF-THEN rules is presented in [14]. The set of rules were used to interpret the codes of the CPN output, thus replacing the lookup table. This approach was found to give a fast and accurate fault diagnosis in a format and language usual to the system operators.

This paper presents an alternative to those approaches presented in [13], [14]. Instead of using a lookup table or a set of IF-THEN rules to interpret the CPN output marking, the proposed method uses the output marking from the FDS_Lookup table approach [13] as the initial marking of a new Hierarchical Colored Petri Net (HCPN) to full automate the fault diagnosis system. The proposed method is referred to as FDS-FullNet.

3. THE FDS WITH INTERPRETATION THROUGH A LOOKUP TABLE

The developed FDS-LookupTable application program has three layers as depicted in Figure 1. The fault diagnosis system collects data from the SCADA system. Since the SCADA often provides a great deal of data, the relevant information are filtered out at the FDS first layer, the Input Interface. This is performed by analyzing the SCADA tables in order to identify data considered relevant to an accurate diagnosis with less amount of information. The input interface layer is accountable for selecting the data from the supervisory system and converting them into an initial marking for the Colored Petri Net (CPN_1) in the Interpretation layer. When an event occurs in the power system, the CPN_1 is run leading to a final marking (dead marking). The event diagnosis is then obtained from the comparison between the final marking and a lookup table that contains the likely diagnosis. In the third layer, the diagnosis is reported to the operators (Output Interface), which gives the following information: the tripped relays and circuit breakers; the tripped protection function; phase(s) in fault; fault type; the equipment under fault; and failure report of relays and circuit breakers.

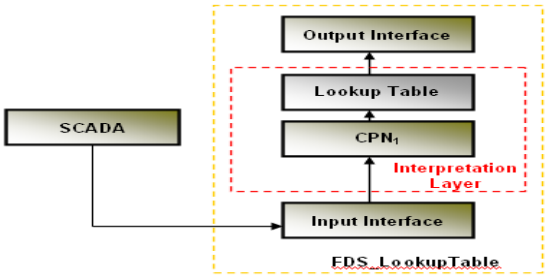


Figure 1. The framework of the Fault Diagnosis System based on a lookup table.

The CPN_1 graph is shown in Figure 2 and it is accountable for the fault diagnosis itself. The net is started up whenever a circuit breaker is tripped by the system protection. The CPN_1 has two *initial places (Relay and Circuit Breaker)* and seven *final places (Supervision, Tripped Function, Open CB for open circuit breaker, Reclosed CB, CB Out of Operation, CB Transferred, CB in Transfer)*. The initial places receive data from the input interface, converted into initial marking. When an event takes place, transitions are fired, and when there are no more enabled transitions, tokens have been placed in the final places. Final places are the ones with tokens in the dead marking (named here final marking). The information in the final places is the input data of a table that relates the final marking with the respective diagnosis.

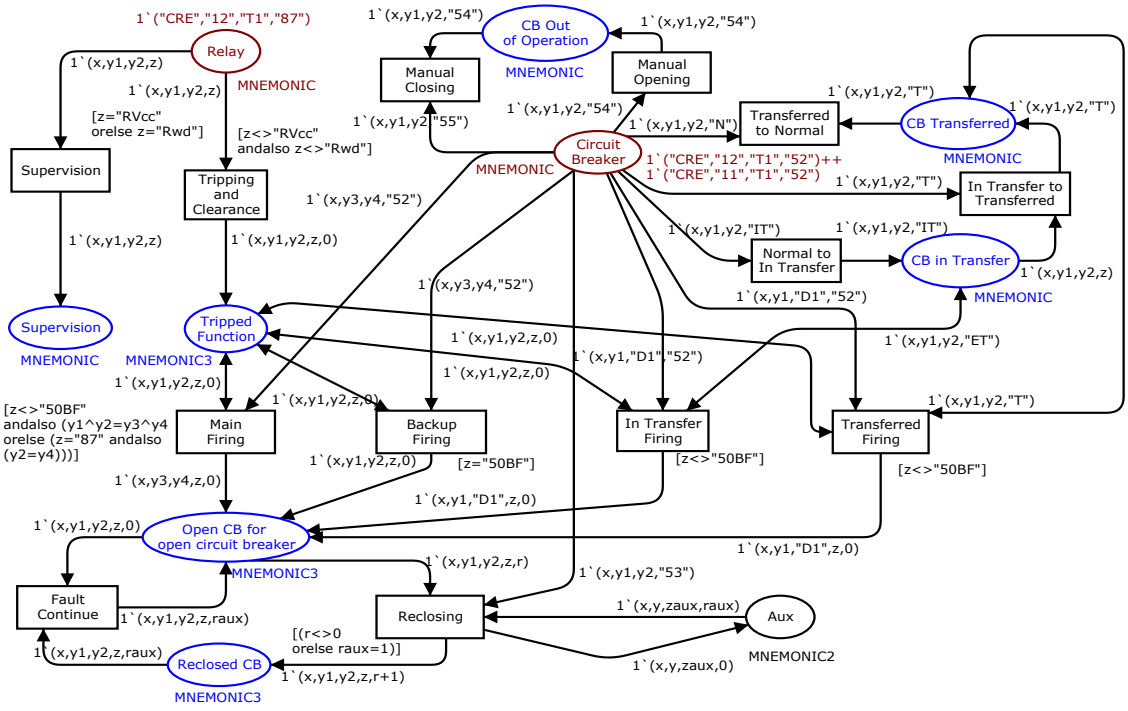


Figure 2. The CPN_1 model of the FDS-LookupTable (Figure 1) application program.

The CPN_1 tokens in the initial places are of the type DATA*DATA*DATA*DATA. Table 1 shows an example of mnemonic codes used to the tokens. The first field in the token indicates the substation code where the equipment is installed, the second field corresponds to the voltage level, the third field corresponds to the equipment code, and the fourth one gives the function related to the equipment operation.

Table 1. Example of mnemonics used for the tokens specification.

Field		**Mnemonics**	**Description**
1	**Substations**	ARU	Araras
		CRE	Cariré
		SBD	Sobral
2, 3	**Equipment**	12C2	Voltage level and Equipment code (relay or circuit breaker)
		12J6	Voltage level and Equipment code (relay or circuit breaker)
		12L1	Voltage level and Equipment code (relay or circuit breaker)
4	**Relay Functions**	50A	Instantaneous overcurrent Phase A
		51C	Instantaneous overcurrent Phase C
		87	Current Differential
		52	Circuit breaker open
		55	Circuit breaker manual closing
		Rwd	Relay malfunctioning

The relays and circuit breakers related to a given substation are identified by a similar code. They differ from each other by their code function. As an example, consider a token with the code (“ARU”, “12”, “C2”, “50A”). It means that the equipment is installed in the substation Araras (“ARU”), it corresponds to the protection relay “12C2”, and the device function code is the instantaneous overcurrent function corresponding to phase A (“50A”). One knows that the token indicates a relay tripping due to the associated function code “50A”. The corresponding token for a circuit breaker would have the code (“ARU”, “12”, “C2”, “52”), where it can be seen that the code “52” identifies a circuit breaker.

The opening of a circuit breaker by the system protection initiates the FDS-LookupTable algorithm. In the initial marking, the *Circuit Breaker* place has the tripped circuit breakers and the *Relay* place has the corresponding protection functions. From the initial marking the transitions are enabled and fired according to the CPN_1 net dynamics, and then the final marking is reached. As aforementioned the final marking is compared to the diagnosis given in a lookup table and then a concise diagnosis can be presented to the operator in the control center. In case an event not previously anticipated by the expertise takes place, although the CPN_1 net provides the diagnosis, the lack of interpretation in the table results in an absence of fault diagnosis to the operator, that is not desirable and should be circumvented.

The FDS-LookupTable program is in test, integrated in the supervisory system of the power distribution utility COELCE in Brazil. So far the application program has responded accordingly, presenting a prompt and accurate diagnosis to the experienced system disturbances. The application tool was initially modeled using the software CPN Tools and then codified using the Csharp program language.

This paper presents a new proposal for the FDS where the lookup table is replaced by a CP-net that models the system topology overcoming the shortages of the previous method as explained in the next sections. On this new approach the topology is inside the CPN model, while in the FDS-LookupTable the topology is outside the CPN model. As the power system topology has permanent changes, it is quite simple to update the topology on the new approach, because the inclusion of new tokens or the updating of the ones already in solves this problem. The update of the FDS-LookupTable requires a study by an expert of the system new configuration in order to update the table with the fault diagnosis.

4. THE FDS USING INTERPRETATION THROUGH COLORED PETRI NET

This is a new alternative to the fault diagnosis method presented in section 3 which aims to replace the expert for the interpretation of the CPN_1 output marking. This approach is denoted as FDS_FullNet.

The FDS-FullNet outlined in Figure 3 has a framework similar to that given in Figure 1, except that the sub layer Lookup Table is replaced by a new CPN_2 net whose initial marking is taken from the places *Tripped Function* and *Open CB for open circuit breaker* of the CPN_1 net developed in FDS-LookupTable method. The CPN_2 net provides the fault diagnosis in substitution to the offline table, making the FDS fully automated.

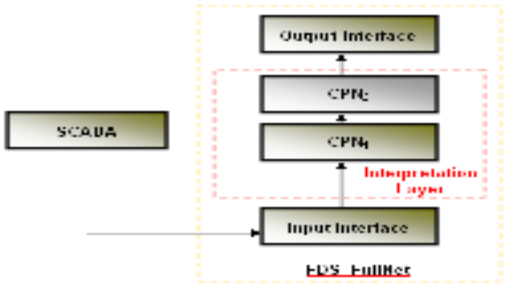


Figure 3 Layers of the FDS-FullNet.

The CPN_2 net has input places with information about the equipment monitored power system topology. The information related to the system topology consist of the protection devices at the lines terminals, transformers, busbars, feeders, capacitor bank as well as the number of parallel circuits and parallel transformers. For those equipment in the network, i.e., outside a substation, it is reported the upstream and downstream substation for each equipment of the network. On the other hand, for equipment inside a substation it is reported the input and output buses.

The CPN_2 is a hierarchical net, partitioned in pages as such: Main page, Diagnosis Analysis page, Fault Analysis page, Diagnostics page, HV Bus page, Lines page, Single Circuit Breaker page, Transformer page, MV Bus page, Feeders page and Capacitor Bank page.

The Main page is the one of major abstraction in the HCPN based net, and its graph is depicted in Figure 4. It includes the initial places *Tripped Function* and *Open Circuit Breaker*, and the final places *SC HV Bus* for short circuited high voltage bus, *SC TL* for short circuited transmission line, *SC TL or HV Bus*, *SC Bay Transformer*, *Damaged Transformer*, *SC Capacitor Bank*, *SC Feeder* and *SC MV Bus* for short

circuited medium voltage bus. The final places provide the fault diagnosis when the transition Diagnosis Analysis is fired. The CPN_2 initial places, *Tripped Function* and *Open Circuit Breaker* places are fusion places with the ones of the same name from CPN_1 net.

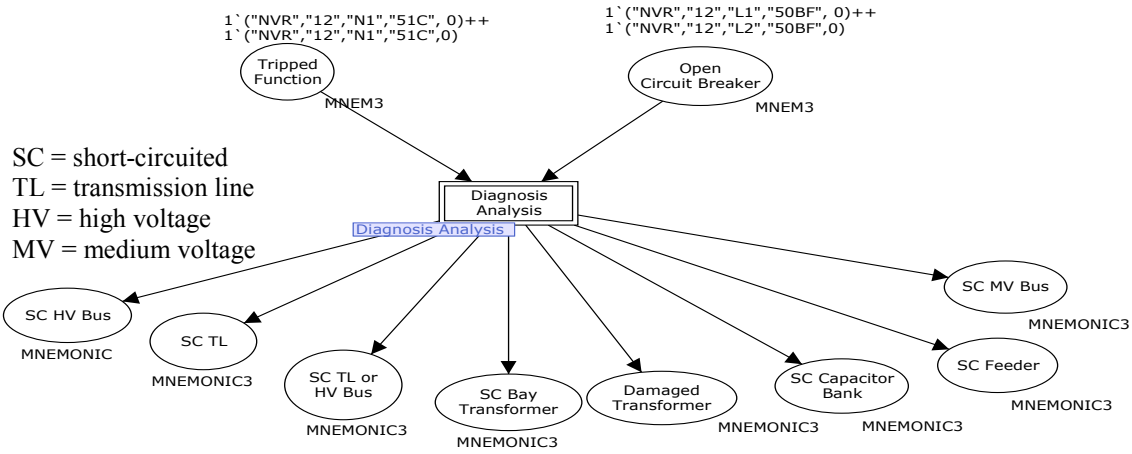


Figure 4. The Main Page of the FDS-FullNet approach.

Figure 5 shows the declarations of the CPN_2, which define the color sets, functions and the variables of each color set of the fault diagnosis system.

```
Declarations
  Declarations
    colset INT = int;
    colset DATA = string;
    colset MNEMONIC = product DATA * DATA * DATA*DATA*INT*DATA;
    colset MNEMONIC3 = product DATA * DATA * DATA*DATA*INT;
    colset MNEM3 = list MNEMONIC3;
    var x,x1,x2,x3,x4: DATA;
    var y,y1,y2,y3,y4: DATA;
    var z,z1: DATA;
    var m6,m1,m2,m4:MNEM3;
    var k p
    var m5,m8,m3,m7: MNEMONIC3;
  Functions
    (* Returns true if x is a member of list l, else returns false *)
    fun mem(x, l) = List.exists (fn y => x = y) l;
    (* Intersection of two lists. Note: for lists that are not sets, that is, have
     multiple occurrence of elements, this operation is non-commutative but it
     should not matter.*)
    fun intersect(l1, l2) = List.foldr (fn (x, ys) => if mem(x, l2) then x :: ys
      else ys) [] l1;
    (* Removes the first occurrence of x from l *)
    fun remElem(x, [] ) = []
      | remElem(x, y::ys) = if x = y then ys else y::remElem(x,ys);
    fun removeRandom1(l1, l2) =
        let
          val l1Andl2 = intersect(l1,l2);
          val len = List.length(l1Andl2) - 1;
          val randElem = List.nth(l1Andl2, discrete(0, len));
          val l1' = remElem(randElem, l1);
          val l2' = remElem(randElem, l2);
        in
           (randElem, randElem,l1', l2')
        end;
    fun removeRandom2(l1, l2) =
        let
          val len1 = List.length(l1)- 1;
          val len2 = List.length(l2) - 1;
          val randElem1 = List.nth(l1, discrete(0, len1));
          val randElem2 = List.nth(l2, discrete(0, len2));
          val l1' = remElem(randElem1, l1);
          val l2' = remElem(randElem2, l2);
        in
           (randElem1, randElem2, l1', l2')
        end;
```

Figure 5. Declarations for the FDS_FullNet based on CPN Tools.

The places in the CPN_2 net such as *Tripped Function* and *Circuit Breaker* present the respective color set as a list where each element has the format DATA*DATA*DATA*DATA*INT representing respectively the substation where the equipment is located, the equipment voltage level, the equipment code, the protection function and the number of reclosing.

The Diagnosis Analysis page, shown in Figure 6, comprises two substitution transitions that lead to the Fault Analysis page and the Diagnoses page, accountable for analyzing the correct operation of the equipment involved in the event and presenting the diagnostic to the operator in the control center, respectively. The other mentioned pages are in fact subpages of the Diagnoses page, which are in charge to describe the event. Thus, if a fault occurs in a transmission line only the transition related to the substitution Lines is enabled presenting a final marking with tokens placed at the SC TL indicating that a fault has taken place in a identified transmission line, the protection function that has tripped and the phases in fault.

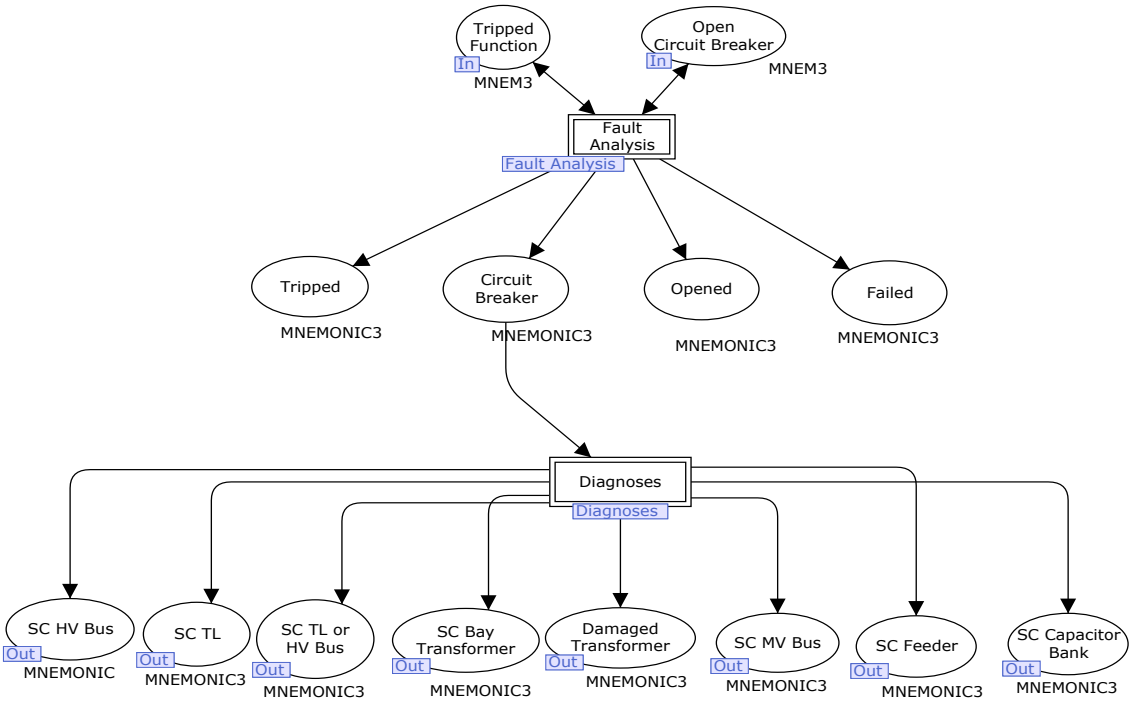


Figure 6. Graph of the page Diagnosis Analysis.

The Fault Analysis page is shown in figure 7. As aforementioned this page will analyze if the equipment involved in the event operated correctly. The transition *equipment not fail* will be enabled if the protection devices operated correctly. This transition will fire when places *Tripped Function* and *Open Circuit Breaker* will have

the same marking. If they will not have the same marking, the transition *equipment fail* will be enabled.

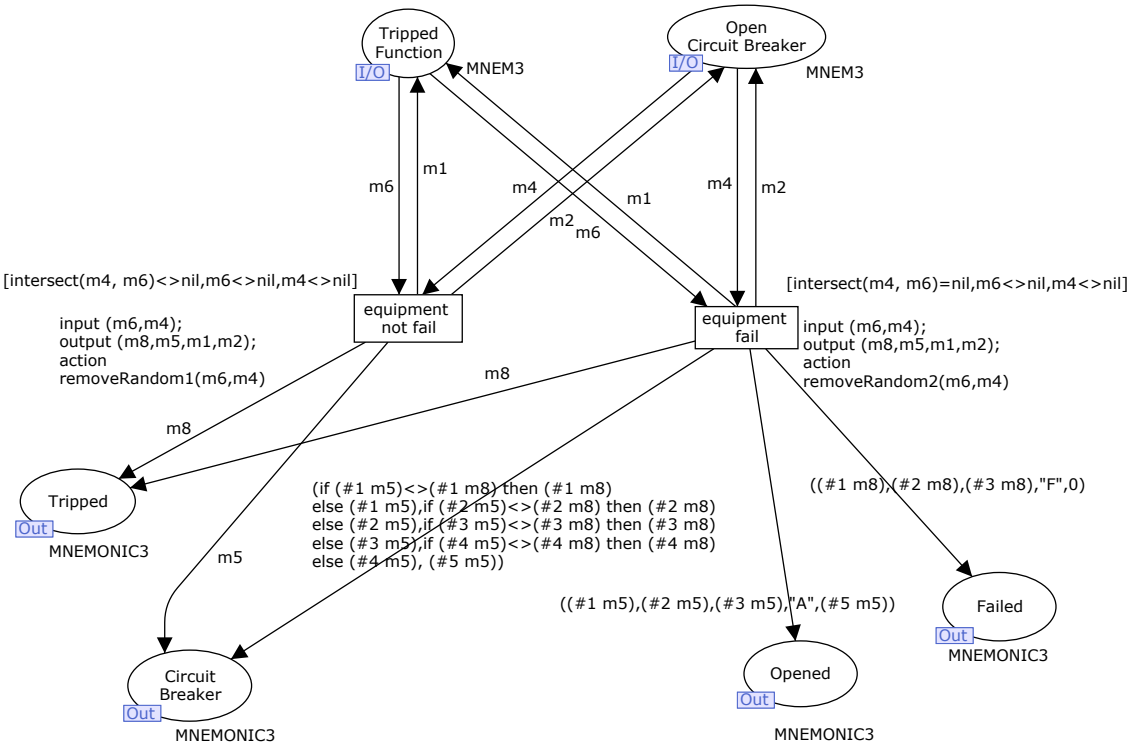


Figure 7 . Subpage Fault Analysis

The places that model the system topology present the format DATA*DATA*DATA*DATA*INT*DATA representing the upstream substation of the equipment, the voltage level, the equipment code, information whether there is equipment at the opposite side of the circuit, number of parallel circuits and the downstream substation of the equipment. When the equipment is in a substation one has the information about the upstream and downstream equipment buses. Figure 8 presents an illustration about the information related to a transmission line when describing the system topology. The token related to the circuit breaker CB1 is given as: 1`("CRE","12","CB1","Y",1,"IBP") where CRE and IBP are respectively the codes of the upstream and downstream substations to the circuit breaker CB1, 12 refers to the voltage level, 1 is the number of transmission lines between the substation buses CRE and IBP and Y indicates the existence of a protective equipment at the end terminal of the line.

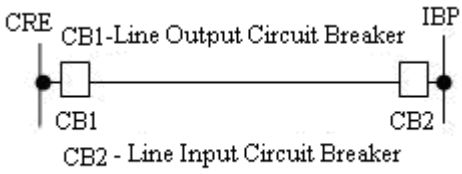


Figure 8. An example to describe the information related to network topology.

The subpage related to the transition Diagnoses is depicted in Figure 9. The places that model the system topology are *Line Input*, *Line Output*, *Transformer*, *MV Bus*, *Feeder* and *Capacitor Bank*. All equipment is modeled by a color set. For instance, in the place *Line Input* is recorded all the circuit breakers which are at the input lines of the substations.

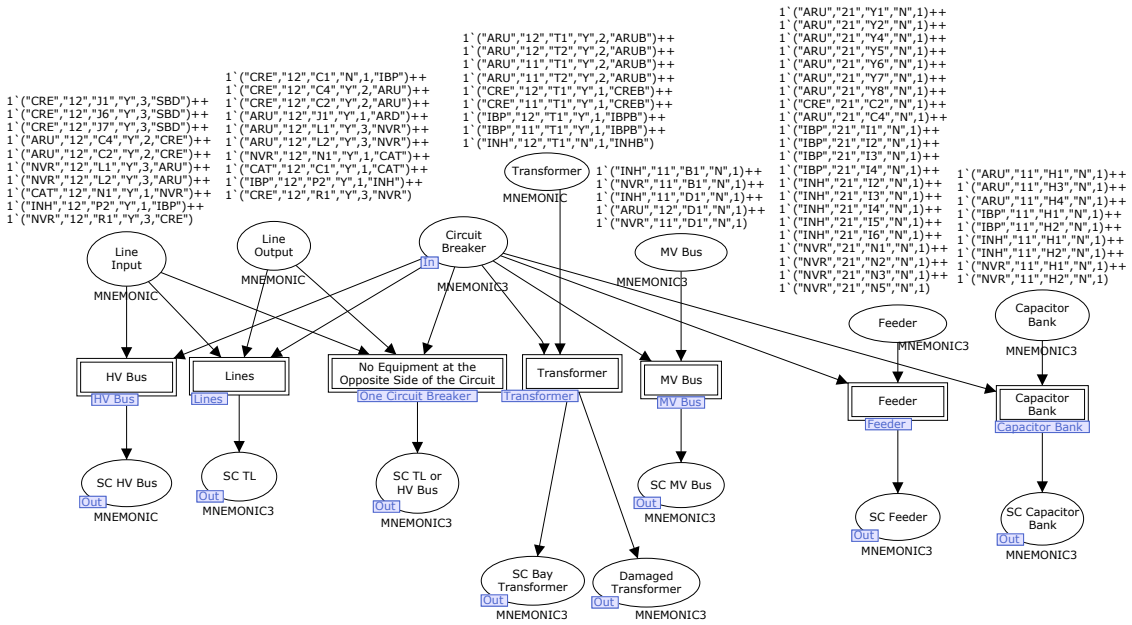


Figure 9. Subpage Diagnoses.

The tokens in this place carry information about the voltage level, equipment code, number of parallel circuits, the upstream substation and if there is a circuit breaker at the end of the lines in the upstream substation. Each final place can give out a likely diagnosis. The values of the tokens in the final places are passed to the Output Interface (Figure 3) that provides the fault diagnosis to the operators in the control center. The fault diagnosis brings information as the sort of fault (one-phase short circuit, two-phase short circuit, and so on), the phases in fault, the relays and circuit breakers that have successful tripped. The information about circuit breaker failure can be found in the

place *Failed*; in case that this place is empty it means that all circuit breakers tripped accordingly.

5. CASE STUDIES

In order to validate and show the performance of the FDS-FullNet approach two case studies are presented: a fault in a 69kV transmission line with a circuit breaker failure and a fault in a 69kV bus. The cases are based on real disturbances that have occurred on the test case system which is part of the distribution system of the utility Ceará Energy Company (COELCE) in Brazil. The system one-line diagram is shown in Figure 10 and it includes eight substations and eleven 69kV transmission lines.

The substation named Sobral (SBD) represents a supply bus such that the power flows from the left to the right side as represented in Figure 10.

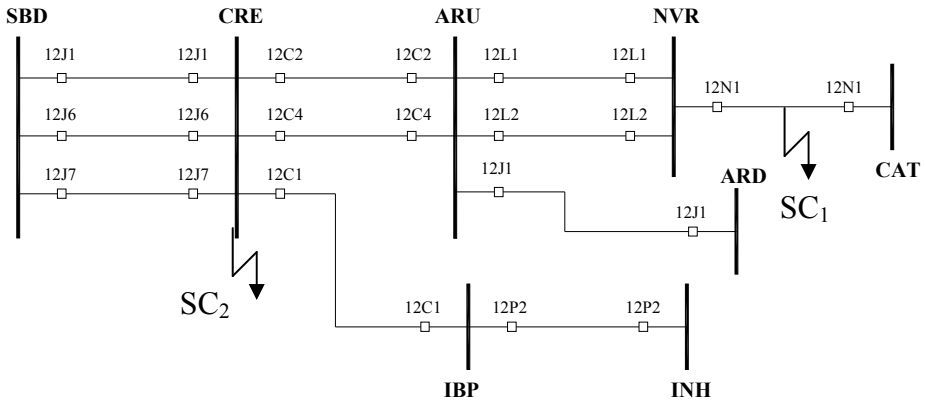


Figure 10. Distribution power system used as test system to validate the FDS-FullNet application program.

In Cariré substation (CRE) there are the circuit breakers 12J1, 12J6 and 12J7 at the substation input lines and the circuit breakers 12C1, 12C4 and 12C4 at the output lines. Connecting the substations CRE (Cariré) and ARU (Araras Um) there are two transmission lines in parallel (02C2 and 02C4 CRE-ARU). From this brief description the two occurrences are considered as follows.

5.1 Case Study 1: Short-circuit in the Transmission Line with circuit breaker failure.

Let it be a short circuit (SC_1) in the transmission line 12N1 that connects the substation NVR (Nova Russas) to CAT (Crateús), shown in Figure 10. To this disturbance, normally one would have a command from the relay associated with the circuit breaker 12N1 at the substation NVR to open the circuit breaker 12N1. Under the

consideration that the main circuit breaker 12N1 has failed to open, the protection function “circuit breaker failure” (50BF) is tripped and the relay associated to 12N1 sends a command to open the backup circuit breakers 12L1 and 12L2 from the substation NVR.

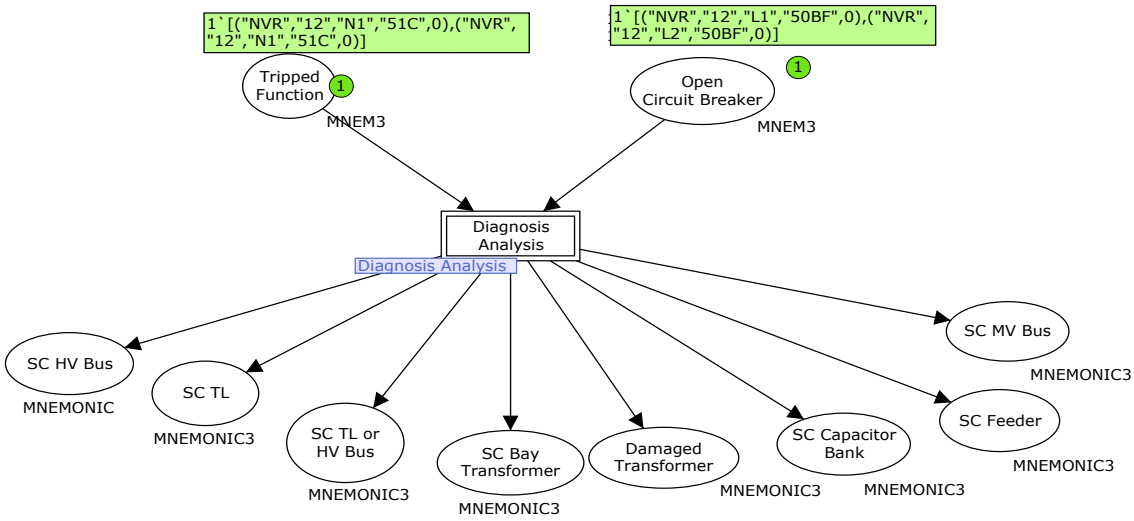


Figure 11. The initial marking for a disturbance in a transmission line.

Figure 11 presents the main page of the CPN_2 in which the data from CPN_1 comes into to be analyzed making it the FDS-FullNet approach. The data are about the circuit breakers (“NVR”,“12”,“L1”,“50BF”, 0) and (“NVR”,“12”,“L2”,“50BF”,0) that have been switched off placed in the *Open CB* place and about the relay (“NVR”,“12”,“N1”,“51C”, 0) that has tripped and placed in the *Tripped Function* place.

The substitution transition Diagnosis Analysis leads to the subpage of the same name, presented in Figure 12. Following the transition Fault Analysis firing, it can be noted that there was failure of operation in the related equipment because the place *Failed* has a token with the circuit breaker that has failed (2`(“NVR”, “12”, “N1”, “F”)).

The substitution transition Diagnoses leads to the subpage of the same name, presented in Figure 13. The system protection devices of the monitored power system are registered in this subpage with tokens in the places *Line Input*, *Line Output*, *Transformer*, *MV Bus*, *Feeder*, *Capacitor Bank*. In this page there is also the substitution transitions related to the likely diagnosis. In Figure 13 the substitution transition Diagnoses is enabled, which informs that a fault has taken place in a transmission line.

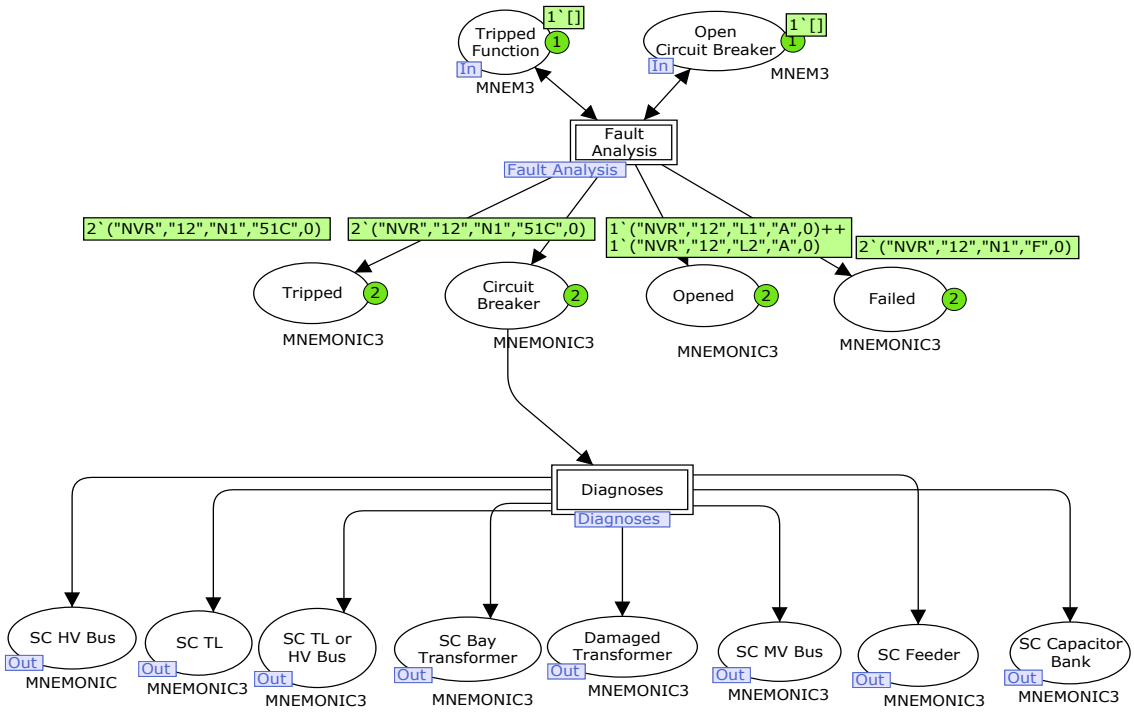


Figure 12. Marking in the subpage Diagnosis Analysis after the transition Fault Analysis is fired because of a fault in a transmission line.

The substitution transition Lines leads to the subpage of the same name, shown in Figure 14, and fires when a fault occurs in a transmission line, identifying the transmission line. In the situation in analysis the line in fault is a single circuit located between the substations NVR and CAT, as it can be seen in Figure 10, and for that the enabled transition is the Single Line. This transition compares the devices that have operated with those protection devices in the input line in substation CAT e the output line in substation NVR.

Figure 15 shows the subpage Lines after the transition Single Line is fired. The token (“NVR”,“CAT” “12N1”,“51C”,0) is in the place SC LT. This token informs that a fault has taken place in the transmission line 12N1 between the substation NVR and CAT.

The main page with the final marking of the proposed FDS-FullNet application program is shown in Figure 16. The final place with token points out the sort of problem and the token identify the equipment under fault. Thus, the final marking presented in Figure 16 indicates the following diagnosis: “Short-circuit in Transmission Line 12N1 NVR-CAT with the tripping of the protection function 51 phase C”.

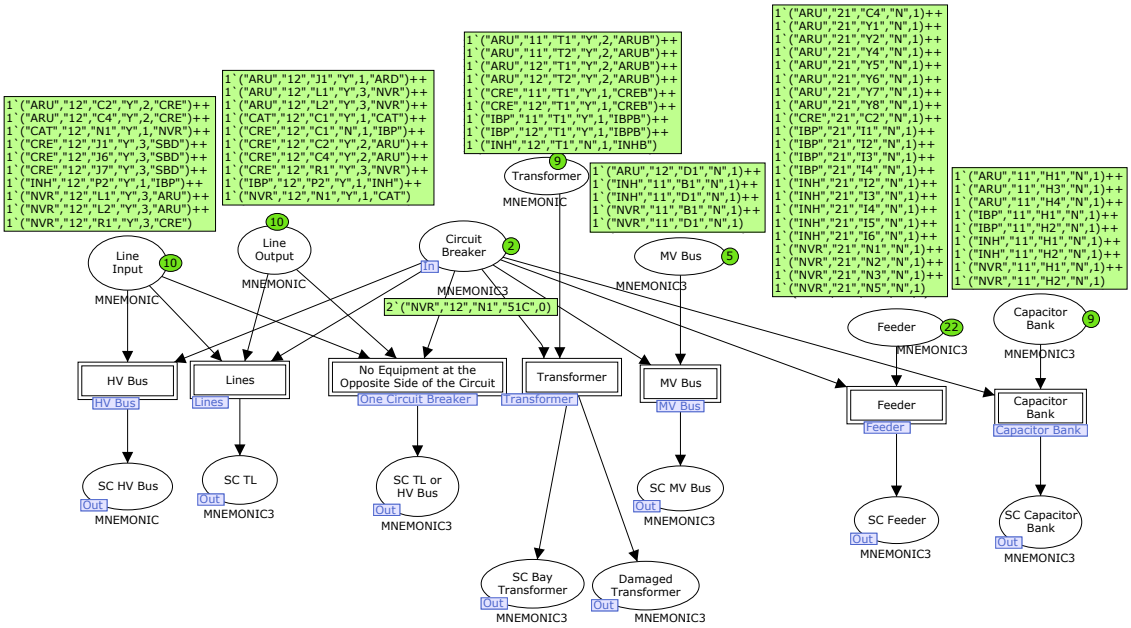


Figure 13. Marking in the subpage Diagnoses for the case study 1.

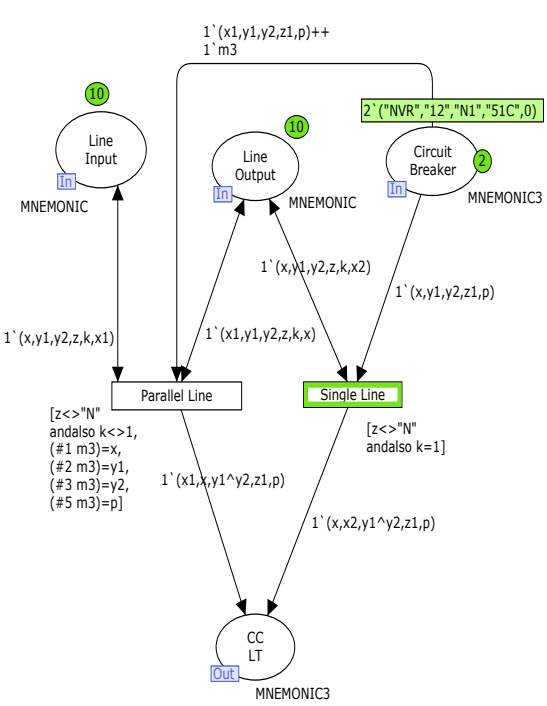


Figure 14. Initial marking in the subpage Lines

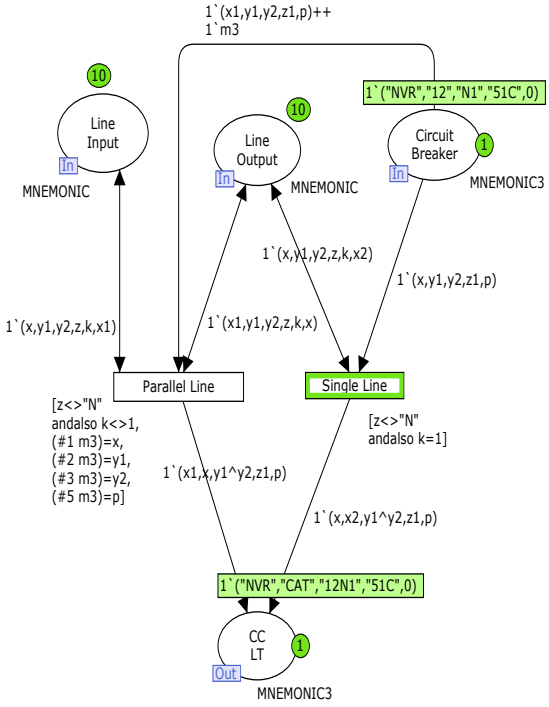


Figure 15. Final marking in the subpage Lines

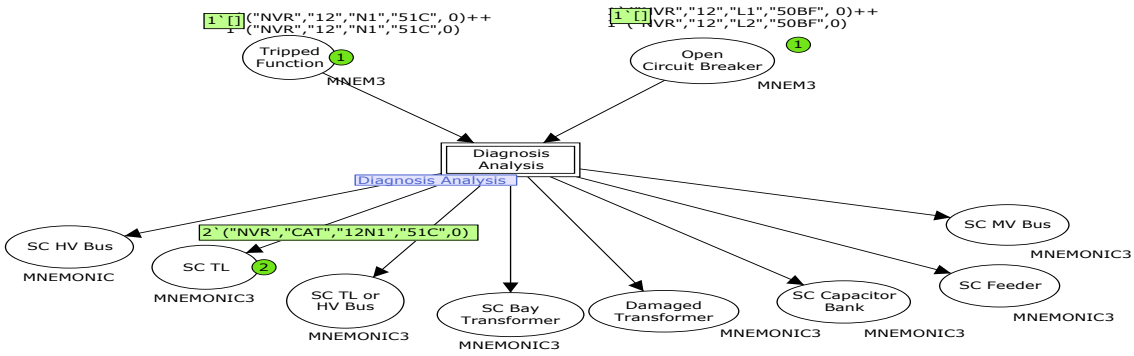


Figure 16. Final marking given by the proposed FDS-FullNet for a fault in a single transmission line.

5.2 Case Study 2: Short Circuit in a 69kV Bus

Consider a short-circuit (SC_2) in the 69kV bus of the substation CRE (Cariré), shown in Figure 10. Data are received from SCADA through the input interface and transformed in initial marking in the CPN_1 net. When the CPN_1 is run it provides the final marking about the circuit breakers that have switched off ("CRE", "12", "J1", "51C", 0), ("CRE", "12", "J6", "51C", 0) and ("CRE", "12", "J7", "51C", 0) and the relays ("CRE", "12", "J1", "51C", 0), ("CRE", "12", "J6", "51C", 0) and ("CRE", "12", "J7", "51C", 0) that have tripped. These tokens form the initial marking of the CPN_2 net disposed accordingly in the *Open CB* and *Tripped Function* input places. Figure 17 shows the main page of the CPN_2 net with the initial marking. As the commanded circuit breakers have not failed, it can be noted that the tokens in both places *Open CB* and *Tripped Function* are the same.

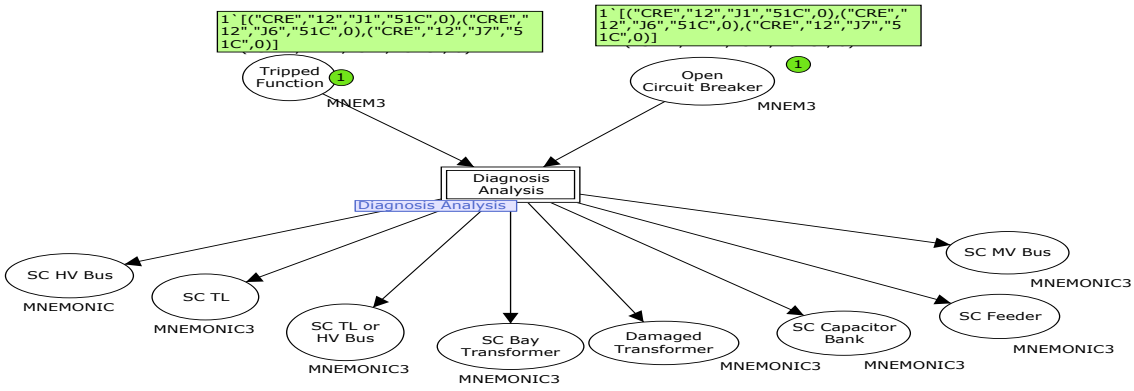


Figure 17. The initial marking in the main page of the CPN_2 net for a fault in a bus.

It can be observed in Figure 18 that there was no circuit breaker failure in this disturbance for the place *Failed* is void of tokens.

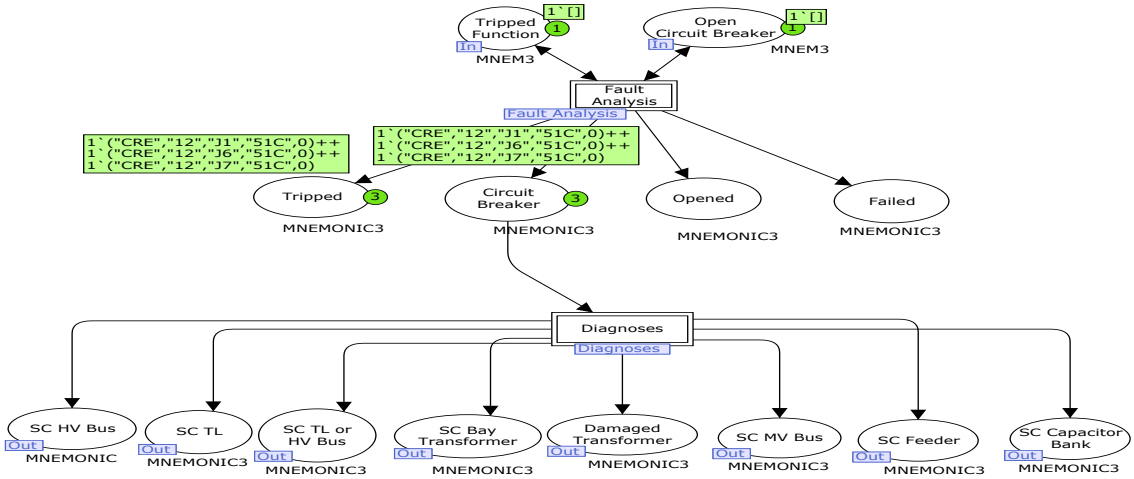


Figure 18. Marking in the subpage Diagnosis Analysis after the transition Fault Analysis is fired due to a fault in a substation bus.

In Figure 19 the substitution transition HV Bus is enabled, indicating that a short circuit has taken place in a high voltage bus.

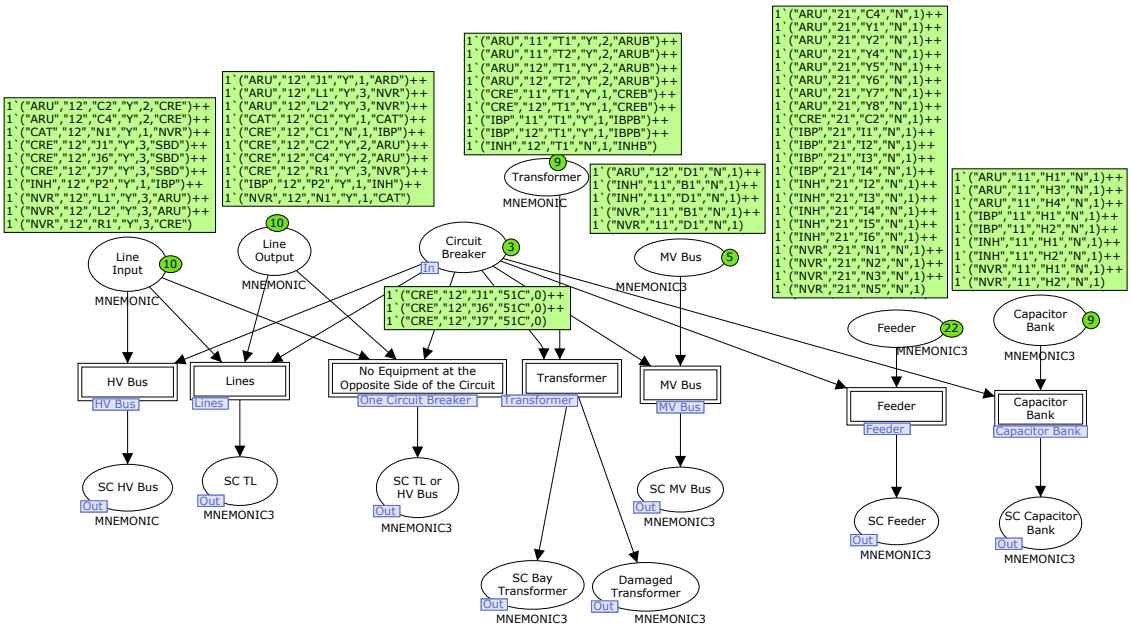


Figure 19. Marking in the subpage Diagnoses due to a fault in a substation bus.

The substitution transition HV Bus sub page is shown in Figure 20. This subpage points out that a short circuit has taken place in 69kV bus and identifies the bus. As there are three incoming transmission line circuits at the substation bus CRE, as it can be seen in Figure 10, the transition Bus3 is enabled, indicating that three input lines are connected to the bus. The transition Bus3 then compares the equipment that

has operated (tokens in the *Open CB* place) with the recorded input line protection devices (tokens in the *Line Input* place).

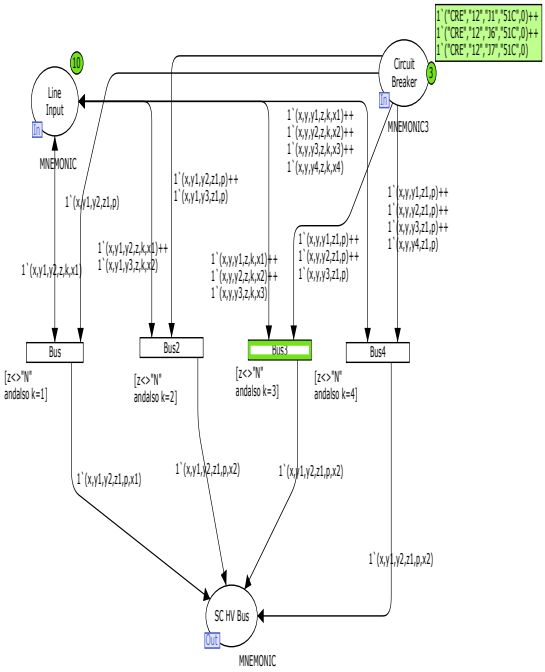


Figure 20. Initial marking in the subpage Bus

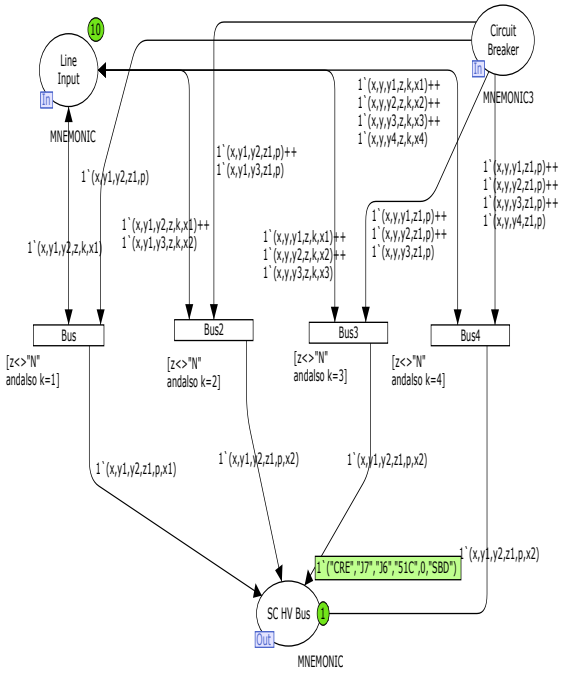


Figure 21. Final marking in the subpage Bus

In Figure 21 it can be observed that the subpage Bus after the transition Bus3 is fired. At the final place SC HV Bus is the token ("CRE","J7","J6","51C",0) which shows that a short circuit has taken place at the bus in the substation CRE. In can also be noticed in Figure 21 that the model considers up to four incoming transmission lines which covers the maximum number of incoming transmission lines within the Coelce system.

The main page with the final marking is presented in Figure 22. The final marking message displayed to the operator in the control Center says: "Short-circuit in 69kV Bus at the substation CRE with the tripping of the protection function 51 phase C".

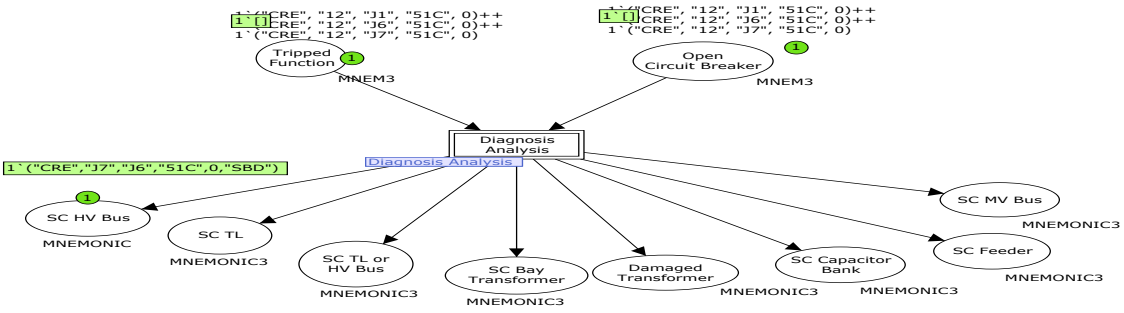


Figure 22. Final marking given by the proposed FDS-FullNet for a fault in a bus with incoming parallel transmission lines.

6. CONCLUSION

Considering the number of likely topologies and arrangements of the power systems, the use of Colored Petri Nets is an adequate tool for the development of Fault Diagnosis Systems. A CP-net based method has been shown that is independent of the power system size. because the power system topology is recorded as initial marking in the extended CP-net proposed. So, the maintenance of the FDS application program under permanent changes in the power system topology is quite simple, which can be carried out by the inclusion of new tokens or the updating of the ones already in. This represents an advance in relation to the FDS based on a lookup table, which requires an off-line study by an expert of the system new configuration in order to update the table with the fault diagnosis.

The paper has presented a centralized fault diagnosis system, located at the control center, which is able to diagnose faults not only in power substations but also in the transmission and distribution networks. The diagnosis reports can identify the fault location, the relay function and the faulted phase(s).

The application program has been tested for many disturbances and two real disturbance cases have been evaluated. The program performed accurately bringing about a fast and succinct diagnosis to the system operator. The application tool is very much valuable to the system operator and contributes to a reliable diagnosis and fast restoration of the power system.

7. REFERENCES

[1] H.-J. Lee, B.-S. Ahn and Y.-M. Park, **"**A fault diagnosis expert system for distribution substations**",** *IEEE Trans. on Power Delivery,* Vol. 15, No.1, Jan. 2000 pp.92-97.

[2] J. C. S. Souza, M. A. P. Rodrigues, M. Th. Schilling, and M. B. D. C. Filho, "Fault location in electrical power systems using intelligent systems techniques" *IEEE Trans. on Power Delivery*, Vol. 16, No.1, Jan.2001, pp. 59–67.

[3] K. M. Silva, B. A. Souza and N. S. D. Brito, "Fault detection and classification in transmission lines based on wavelet transform and ANN", *IEEE Transactions on Power Delivery*, Vol. 21, No. 4, Oct. 2006, pp.2058 – 2063.

[4] Wen, F., Han, Z. (1995) Fault Section Estimation in Power Systems Using A Genetic Algorithm. *Electric Power Systems Research 34*, pp 165-172.

[5] H.-C. Chin, “Fault section diagnosis of power system using fuzzy logic”**,** *IEEE Transactions on Power Systems*, Vol. 18, No. 1, Feb. 2003, pp.245–250.

[6] K. L Lo, H. S. Ng and J. Trecat. Power Systems Fault Diagnosis Using Petri Nets. In*: IEE Proceedings on Generation, Transmission and Distribution*, Vol. 144, No. 3, May. 1997, pp. 231-236.

[7] K. L. Lo, H. S. Ng, D. M. Grant and. J. Trecat, “Extended Petri Nets Models for Fault Diagnosis For Substation Automation”. *IEE Proc.-Gener. Transm. Distrib.* Vol. 146, No. 3, pp. 229-234, May. 1999.

[8] J. Sun, S.-Y. Qin and Y.-H. Song, “Fault Diagnosis of Electric Power Systems Based on Fuzzy Petri Nets”, *IEEE Transactions on Power Systems*, Vol. 19, No. 4, November 2004, pp.2053-2058.

[9] T. Murata, “Petri nets: Properties, analysis and applications. In: IEEE Proceedings, Vol.4, No.4, April 1989, pp.541-580.

[10] Jensen, K., “Coloured Petri Nets: Basic Concepts, Analysis Methods and Pratical Use”. Vol .1, 2nd Edition, Springer-Verlag, 1997.

[11] Bezerra, J. R., Caetano, M., Souza, J. R., Furtado, R., Barroso, G., Leão, R. P. S., “Uma Abordagem para Diagnóstico de Faltas em Subestações de Sistemas Elétricos de Potência Usando Redes de Petri”, VI Simpósio Brasileiro de Automação Inteligente. Bauru - SP, Setembro, 2003.

[12] Sampaio, R. F.; Barroso, G. C.; Sousa, J. R. B.; Leão, R. P. S. An Advanced Function for the Supervisory System of an Electrical Distribution Substation: An Application using Colored Petri Nets. In: Iasted International Conference - 2003, Palm Springs - USA, ISSN: 1021-8181. pp.192-197.

[13] Medeiros, E. B.; Barroso, G. C.; Colaço, A. L. G.; Santos Filho, F. G.; Sampaio, R. F. ; Leão, R. P. S. ; Vieira, J. M.; Lourenço, T. G. M; Augusto Junior, M. F. ;Fault Diagnosis System for Power Systems Based on Coloured petri Nets. In: XII Latin-American Congress on Automatic Control, 2006, Salvador-Bahia. Proceedings of the XII Latin-American Congress on Automatic Control, 2006. p. 172-177.

[14] Santos, F. G. Filho, “Diagnóstico de Faltas em Sistemas Elétricos Baseado em Redes de Petri Coloridas e Técnicas de Sistemas Especialistas”, Universidade Federal do Ceará – UFC, 2007, 96p.

Towards an Open and Extensible Business Process Simulation Engine

Luciano García-Bañuelos* and Marlon Dumas

Department of Computer Science
University of Tartu, Estonia
{luciano.garcia,marlon.dumas}@ut.ee

Abstract. This paper outlines the architecture and initial proof-of-concept implementation of an open and extensible business process model simulator based on CPN tools. The key component of the simulator is a transformation from BPMN process models to CPNs. This transformation is structured as a set of templates that can be extended and modified by developers in order to incorporate new functionality into the simulator. The paper illustrates how this templating mechanism can be used to capture different types of tasks and resource allocation policies.

1 Introduction

Business process simulation is a widely used technique for analyzing business process models with respect to performance metrics such as cycle time, cost and resource utilization. Many commercial business process modeling tools incorporate a simulation component, e.g. TIBCO Business Studio, IBM Websphere Business Modeler (WBM), ARIS, FileNet and Protos [5]. However, these process simulators have two architectural limitations:

1. Only models designed with the tools themselves can be simulated.
2. No extensibility mechanism is provided to add new features or change the pre-built simulation and reporting options.

The first limitation stems from two factors. Firstly, the simulation component is generally hidden inside the tool, meaning that it does not expose an Application Programming Interface (API). Secondly, there is a long-standing problem of lack of business process model interoperability. The Business Process Modeling Notation (BPMN) – particularly its upcoming version 2 – attempts to address this issue by providing a standard business process meta-model with an interoperable XML serialization. While BPMN does not define any simulation parameters (e.g. arrival times and branching probabilities), it includes extensibility mechanisms that allow one to add such details.

The second limitation is to a large extent connected to the first one: since the tools do not offer an explicit interface for their simulation component, it

* On leave from Autonomous University of Tlaxcala; work funded by the ERDF through the Estonian Centre of Excellence in Computer Science.

is not possible to plug-in additional functionality into it. This limitation has been raised in recent work [7]. Motivated by simulation requirements found in supply chain management, the authors introduce an extensibility mechanism into IBM WBM. The idea is to allow developers to attach scripts to processes and tasks. Six types of scripts are supported: pre-processing scripts (executed before a task/process is activated), post-processing scripts, delay scripts (to introduce waiting times), cost, revenue and duration scripts. However, this mechanism is limited – it does not allow one to extend the execution semantics that drives the simulation engine. For example, one cannot apply such extensibility mechanism in order to simulate advanced resource allocation patterns [10] nor to capture control-flow connectors beyond those offered by the modeling tool.

In this paper, we outline the architecture and current implementation status of OXProS - an Open and Extensible Process Simulator for BPMN. OXProS provides a RESTful service interface allowing third-party applications to submit BPMN process models for simulation. Simulation parameters are incorporated into the process model using BPMN's extensibility mechanism. OXProS uses CPN Tools as its underlying execution environment. In other words, BPMN models are translated into CPNs for simulation purposes. This translation is based on a templating mechanism that enables developers to extend OXProS by adding new templates or modifying existing ones.

In the rest of the paper we present the architecture of OXProS and its template-based extensibility mechanism. We present the current implementation status of OXProS and a roadmap for future work.

2 Architecture

Figure 1 depicts the architecture of OXProS. In this figure, the yellow boxes represent OXProS components, the light-blue boxes represent CPN Tools and associated components, and the white boxes correspond to third-party software.

At the architectural level, OXProS is structured as a set of XML over HTTP Web services, designed according to the principles of RESTful service architectures [3]. A simulation is treated as a resource that can be created/started through a POST operation. Subsequently, the status of the simulation and its output can be retrieved using GET operations. As a shortcut, it is possible to create a simulation model and retrieve its results through a single POST operation.

OXProS accepts either BPMN process models (enhanced with simulation attributes) or CPN models. If a BPMN model is submitted, it is transformed into a CPN model and passed on to the CPN simulation service. At present, there is no standard serialization of BPMN. It is expected that the upcoming BPMN 2.0 standard will have its own XML serialization format. In the meantime, the BPMN simulation service assumes that the input models are serialized using the XML Process Definition Language (XPDL) version 2.1 [1].

The services provided by OXProS are implemented using Java Servlets. The CPN simulation servlet is built on top of Access/CPN [13] – a tool that enables

programmatic access to the CPN Tools Simulator. Since Access/CPN is implemented as a set of OSGi bundles/components, its implementation relies on the HTTP Service provided by Equinox – the Eclipse OSGi implementation[1].

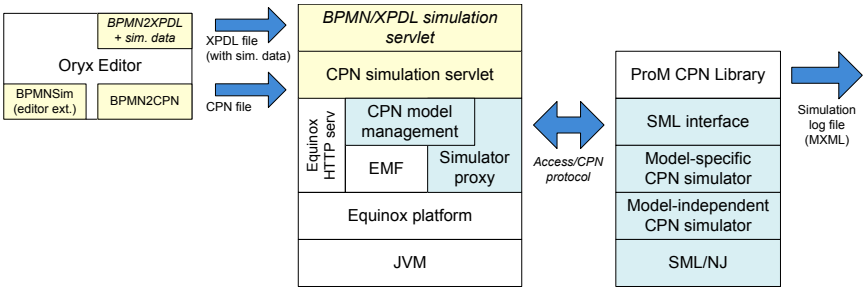


Fig. 1. OXProS architecture

When the CPN simulation servlet receives a simulation model, it invokes Access/CPN to check the submitted CPN and starts the simulation on the background simulator daemon. Upon completion of the simulation, an MXML log file is generated from the simulation output. The produced MXML log file can be analyzed off-line using the ProM framework[2] (e.g. to extract key performance indicators). In order to generate MXML log files, we make use of the ProM CPN Library [6].

In the current prototype, BPMN processes are modeled with the Oryx Editor[3]. We extended the Oryx front-end to allow users to add simulation parameters into a BPMN process model. This Oryx extension is called BPMNSim. Moreover, we extended the Oryx back-end with a module that generates CPNs from Oryx's internal representation of process models. In the future, it is expected that Oryx will support generation of standard BPMN models so that these models can be submitted to the BPMN simulation servlet.

3 Template-based Generation of CPNs

The key component of OxProS is the transformation from extended BPMN models to CPNs. This transformation is designed using a templating approach. To illustrate the transformation, we consider the simplified "teleclaims" process of an insurance company, presented in Figure 2. This process handles insurance claims made by phone. The process is supported by staff in a call center and in a claims handling department.

In the basic form, a BPMN process diagram consists of events (circles), activities (rounded rectangles) and gateways (diamonds). Events denote the start,

[1] http://www.eclipse.org/equinox/server
[2] http://www.processmining.org
[3] http://oryx-editor.org

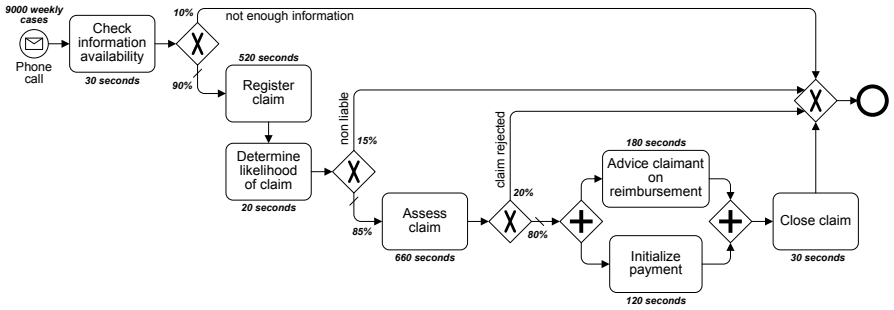


Fig. 2. Insurance claim handling process (adapted from [12])

the end of a process case or something that happens during the process execution. In the running example, we used a "message start event" to specify that a case starts with the reception of a "Phone call". Activities denote work that must be done, for example, an agent in the call center executes the task "Check information availability". Gateways are routing constructs. The two basic types of gateways are AND gateways (marked with a "+" symbol) and XOR gateways ("X" symbol). A gateway is either to be a split gateway if it has multiple outgoing flows or a join gateway if it has multiple incoming flows. An XOR-split is a decision point, meaning that one outgoing path is taken according to the result of the evaluation of a logical condition. For instance, in the running example, 10% of cases are rejected due to insufficient information. A XOR-join is a merging point. An AND-split starts two or more parallel threads. For instance, tasks "Advice claimant on reimbursement" and "Initialize payment" are executed in parallel. Finally an AND-join synchronizes parallel threads.

In order to simulate a BPMN process model, a number of simulation parameters need to be specified. These include: (i) the arrival rate of new cases and its associated distribution (e.g. exponential); (ii) the branching probabilities for each arc (flow) leaving an XOR-split, and a number of performance attributes attached to activities/events, such as time, cost and revenue. These attributes are generally represented using a mean value and a probability distribution (e.g. normal). A simulation model includes a number of *resource pools* denoting sets of resources. Each activity may be associated to a resource pool and at runtime, one resource from the pool is selected to perform the activity based on a resource allocation policy.

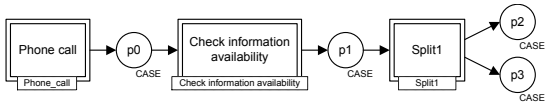
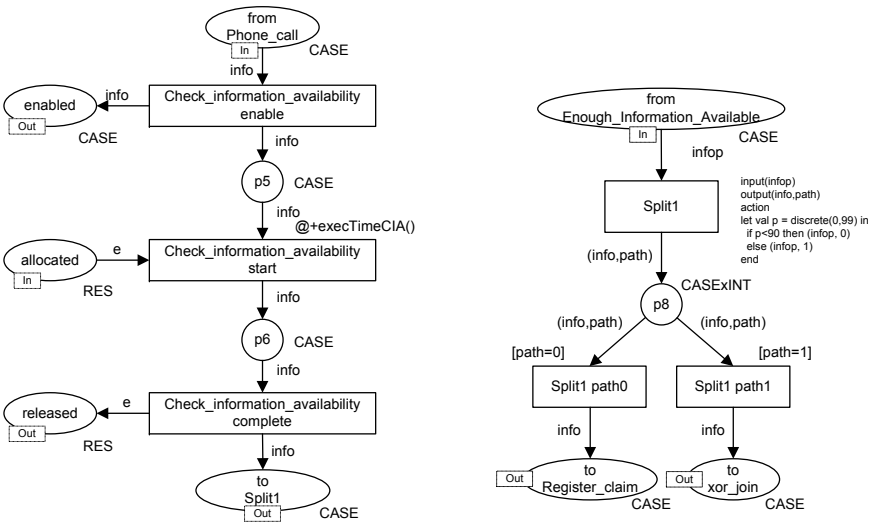


Fig. 3. Mapping of the root process

Given a BPMN model extended with simulation attributes, we generate a hierarchical CPN, with two top-level pages: one for the control flow perspective and the other for resource perspective. Figure 3 presents a partial view of the CPN model for the control-flow of the insurance claim process.



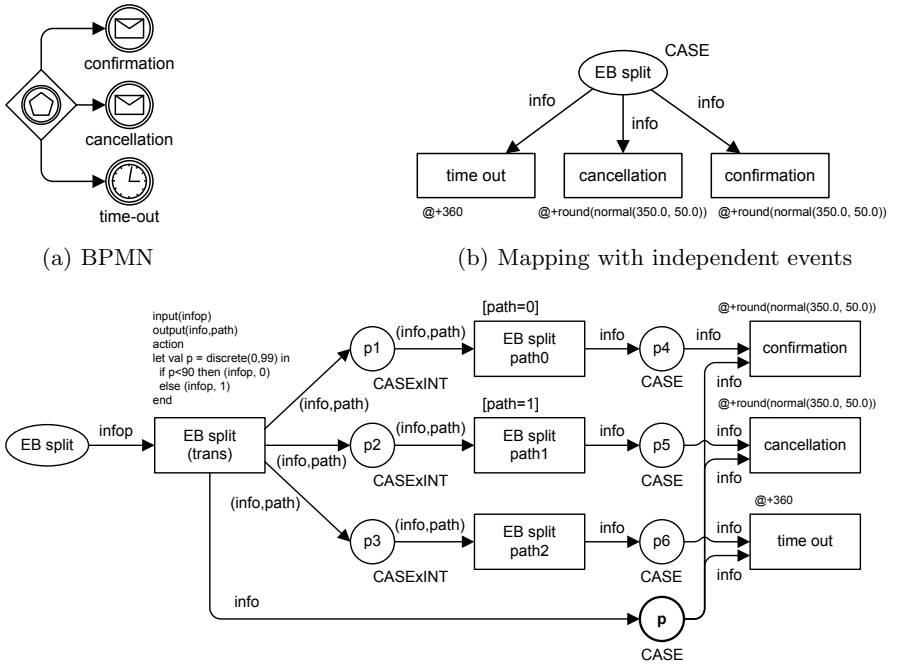
(a) Task (b) XOR (split) gateway

Fig. 4. Default mapping of tasks and XOR-split gateways

In order to make the mapping as modular as possible, every element in a process model is mapped to an individual page. For example, Figure 4(a) presents the page with the mapping of task "Check information availability" from the running example. It contains two places to synchronize the start and completion of the task, to model the control flow perspective. Additionally, the page includes three other places to handle the assignment of the task to a staff member for execution. A simulation would occur as follows. First, the task gets enabled and waits until a staff member takes it. The timing annotation on transition "Check_information_availability start" generates a random simulation time that is associated to the execution of the task. When the task is completed, the resource is released.

Let us now consider the case of data-based XOR split gateways. The corresponding CPN must handle the selection of a path according to a probability distribution. Figure 4(b) presents the page for the first XOR gateway in the running example. The transition "Split1" generates a random value between 0 and 99. For every path, there is a guarded transition which is activated according to the value of the variable "path". Thus, the transition "Split1" sets "path" to 0 in 90% of the cases and it sets "path" to 1 in the remaining 10% of the cases.

In addition to the XOR decision gateway discussed above (called *data-driven decision gateway*), BPMN features a so-called event-driven XOR decision gateway. The data-based and the event-based XOR gateways are similar in that they pass the flow of control along one of their outgoing paths only – thus, they represent a "choice". However, whilst in a data-based XOR gateway the choice is performed by the system based on boolean conditions, in the case of an event-based XOR gateway, the choice is made by the environment. Specifically, an



(a) BPMN

(b) Mapping with independent events

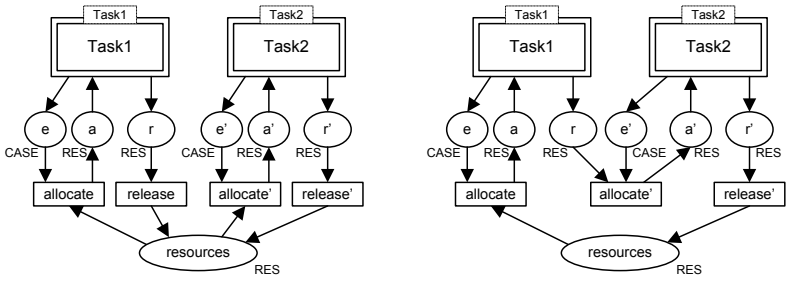
(c) Mapping with dependent events

Fig. 5. The BPMN Event-based XOR gateway

event-based XOR gateway is directly followed by two or more events. When the gateway is reached, the flow of control stops at that point until one of the events in question occurs. The first event to occur determines which path is taken. Figure 5(a) shows a typical scenario involving an event-based XOR gateway. In this example the process waits for either a confirmation or a cancellation message from the customer, or a time-out.

Multiple approaches can be adopted to simulate BPMN models with event-based XOR gateways. One approach is to capture it purely using delays: each event is annotated with a delay, that may be a fixed duration (in the case of time events with constant values), or a probability distribution (in the case of message events or time events with a dynamically-evaluated duration). Figure 5(b) illustrates this approach. Here, the delays of the two message events (cancellation and confirmation) follow a normal distribution with identical means and standard deviations. This mapping works under the assumption that the two events are independent.

However, in this example one would expect that these two events are not independent, but rather exclusive: if a confirmation message is received, no cancellation message will arrive and vice-versa. Figure 5(c) presents an alternative mapping that captures this exclusion dependency. In this case, the customer re-



(a) Role-based distribution (b) Chained execution

Fig. 6. Two templates for resource allocation

sponds with a confirmation message in 90% of the cases or with a cancellation message in 10% of the cases. The ML annotation on the first transition activates the path to the "time-out" event and to one of the other two paths (the confirmation path and the cancellation path). From that point on, there is a race between the time event and one of the two message events, reflecting the fact that only one of the two message events may eventually occur.

In order to implement this alternative mapping, one has to extend BPMN with the ability to express exclusion dependencies between events that are connected to a common event-based gateway. In the current BPMN standard, when two events are attached to an event-based gateway, it might be that either event may occur or that both events may occur (even though the second one to occur may be discarded). In the above example, the case where both messages are eventually received by the process is excluded. In other words, the events are not in a "race" but rather in an exclusion relation. While this exclusion relation between message events is not relevant for execution purposes, it is relevant for (stochastic) simulation since it means that their occurrence should be drawn from a common probability distribution. In OXProS, we rely on a non-standard extension of event-based decision gateways to capture this exclusion dependency.

In OXProS, CPN simulation models are generated using *templates*. A template takes as input a BPMN element of a given type, and produces a CPN page. Multiple templates can exist for a given type of BPMN element. Since each BPMN construct is mapped to a separate page, the mapping of a type of element (e.g. task, event-based XOR gateway) can be replaced by a different mapping without affecting other elements, so long as the inputs/output places of the page generated by the new mapping are compatible with those of the previous mapping.

Templates are also used for capturing resource management. For example, Figure 6 shows two templates for resource allocation. The first one – Figure 6(a) – corresponds to the "Role-based distribution" workflow resource pattern [10].[4] The idea behind this pattern is that at runtime the execution engine routes the

[4] For readability reasons, we omitted some details on the colored Petri nets in Figure 6.

task to the worklist of the resources that can perform a given task, e.g. staff members with a particular profile/role. Eventually, one of those resources will remove the workitem from worklist and perform it. This can be simulated with an ML code attached to the transition "allocate", and a timing annotation can be used to record the waiting time. When the task is completed, the resource is sent back to the resource pool by transition "release". Existing business process simulation tools generally implement this approach. However, there exist other resource allocation patterns. As an example, consider the CPN shown in Figure 6(b) that captures the so-called "Chained execution" pattern. In this case, the two subsequent tasks are assigned to the same resource. Both of these templates are provided in OXProS, and developers may introduce additional alternative templates. When a BPMN model is submitted to OXProS for simulation, the request may refer to a *simulation profile* that determines which template should be used for which BPMN construct. If no profile is specified, the default profile is used.

4 Related work

Many commercial business process modeling tools incorporate a simulation component, e.g. TIBCO Business Studio, IBM Websphere Business Modeler, ARIS, FileNet and Protos, among others. Jansen-Vullers and Netjes [5] survey a number of process simulation tools and evaluate their suitability with respect to a number of requirements. They conclude that no tool covers all the requirements, and that CPN tools is a suitable option in terms of expressiveness but that it may be too complex for use by business analysts.

A translation of Protos simulation models to CPN models is presented in [4]. This translation is rather straightforward because Protos models have many commonalities with Petri nets. The authors also describe an extension of their translation to handle configurable process models. A configurable process model provides an integrated view of multiple process model variants. The tool presented in [4] is geared towards comparing the performance of multiple process model variants captured in a given configurable process model.

Rozinat et al. [8] present an approach to mining simulation models from event logs. The idea is to automatically generate a process model, represented as a CPN, which will behave in a similar way as the process model that generated the original event log. Depending on the richness of the event log, the resulting CPN may cover not only the control-flow perspective, but also the data perspective (e.g. data attributes and branching conditions), the organizational perspective (e.g. roles, resources) and the performance perspective (e.g. distribution of execution times). The authors follow the approach of mapping each activity in the process to a separate sub-page, an idea which is also followed in OXProS.

CPN tools has been used for generating synthetic logs to be used in the design and testing of Process Mining algorithms [6]. In contrast to real-life data, the log generated by simulation is free of noise and will include the perspectives that

are important for tuning a given mining algorithm. In [6], the authors outline a set of ML functions to extend CPN tools in order to generate logs in the MXML format. This is the format used by several process mining tools. OXProS follows the idea of representing the simulation output in MXML and reuses the library of MXML generation functions defined in [6].

In [11], the authors argue that current approaches to model resource allocation for process simulation are not realistic. For instance, scenarios where the same resource is assigned to multiple processes (and thus needs to split its time between them) are not supported by existing tools. Also, existing tools fail to take into account that people work in batches and that they divide their time into discrete intervals ("chunks") that they allocate to different types of tasks. They present a novel approach to capturing simulation models in which each resource is captured as a separate CPN page. The CPN page of a resource captures the resource's lifecycle. Therefore, each resource (or each resource class) may have a different lifecycle. While this approach leads to more realistic resource models, one has to note that implementing such an approach requires significant input from the modeler, since the modeler has to provide additional information for each resource (class) and this information may not be available in some cases.

In line with the above body of research, we adopt CPNs as a basis for business process model simulation. Unlike this body of work however, we adopt BPMN as the notation for modeling business processes, since it is a widely-adopted standard that strikes a tradeoff between ease-of-use for business analysts and expressiveness. While some of the previous work can be easily adapted to support the simulation of BPMN models, BPMN brings in certain specificities that warrant further study. Specifically, BPMN has a rich set of event types and events can be used in different settings (e.g. event-based gateways, error events). We have shown in this paper that there are multiple possible approaches for simulating process models with event-based gateways and message events. Each of these approaches strikes a different tradeoff between the amount of input data required for simulation and the precision with which the simulation reproduces the real-world phenomenon. The multiplicity of possible approaches justifies the need for an extensible architecture which is one of the driving requirements of OXProS.

5 Outlook

The current implementation of OXProS supports a restricted subset of BPMN – XOR/AND gateways, tasks, plain events and start message events. Ongoing work aims at extending the coverage of BPMN by adding templates incrementally. The majority of these templates will be designed on the basis of the BPMN to plain Petri nets transformation outlined in [2].

In order to validate the suitability of the template-based extensibility mechanisms, we then plan to simulate process models from the logistics domain, where resources (e.g. trucks) have capacity and speed constraints that cannot be captured using commercial business process simulation tools.

The current OXProS architecture is only able to generate MXML simulation logs and leaves it up to the user to analyze these raw logs. In future, we plan to incorporate analytics services into the OXProS architecture in order to compute key performance indicators from the simulation logs. Another avenue for future work is to allow OXProS to take as input not only process models with simulation attributes, but also process execution logs, in order to simulate process models based on real past executions and starting from a given state [9].

References

1. Workflow Management Coallition. Process Definition Interface – XML Process Definition Language, October 2008.
2. R. Dijkman, M. Dumas, and C. Ouyang. Formal Semantics and Analysis of BPMN Process Models. *Information and Software Technology*, 50(12):1281–1294, 2008.
3. Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.
4. F. Gottschalk, W. M. P. van der Aalst, M. H. Jansen-Vullers, and H. M. W. Verbeek. Protos2CPN: using colored Petri nets for configuring and testing business processes. *International Journal on Software Tools for Technology Transfer*, 10(1):95–110, December 2007.
5. M.H. Jansen-Vullers and M. Netjes. Business Process Simulation – A tool survey. In *Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools*, 2006.
6. A. K. Alves De Medeiros and C. W. Günther. Using CPN Tools to Create Test Logs for Mining Algorithms. In *Proceedings of the Sixth Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools*, pages 177–190, 2005.
7. C. Ren, W. Wang, J. Dong, H. Ding, B. Shao, and Q. Wang. Towards a Flexible Business Process Modeling and Simulation Environment. In *WSC '08: Proceedings of the 40th Winter Simulation Conference*, pages 1694–1701, 2008.
8. A. Rozinat, R. S. Mans, M. Song, and W. M. P. van der Aalst. Discovering Colored Petri Nets from Event Logs. *International Journal on Software Tools for Technology Transfer*, 10(1):57–74, December 2007.
9. A. Rozinat, M.T. Wynn, W.M.P. van der Aalst, A.H.M. ter Hofstede, and C.J. Fidge. Workflow Simulation for Operational Decision Support. *Data & Knowledge Engineering*, 68(9):834–850, 2009.
10. N. Russell, W.M. P. van der Aalst, A.H. M. ter Hofstede, and D. Edmond. Workflow Resource Patterns: Identification, Representation and Tool Support. In *17th International Conference on Advanced Information Systems Engineering (CAiSE), Porto, Portugal, June 13-17, 2005*, pages 216–232. Springer, 2005.
11. W.M.P. van der Aalst, J. Nakatumba, A. Rozinat, and N. Russell. Business Process Simulation: How to get it right? Technical Report BPM-08-07, BPMcenter.org, 2008.
12. W.M.P. van der Aalst, M. Rosemann, and M. Dumas. Deadline-based Escalation in Process-Aware Information Systems. *Decision Support Systems*, 43(2):492–511, 2007.
13. Michael Westergaard and Lars Kristensen. The Access/CPN Framework: A Tool for Interacting with the CPN Tools Simulator. *Applications and Theory of Petri Nets*, pages 313–322, 2009.

Relaxed Timed Coloured Petri Nets - A Motivational Case Study *

Guy Edward Gallasch and Jonathan Billington

Computer Systems Engineering Centre
University of South Australia
Mawson Lakes Campus, SA, 5095, AUSTRALIA
Email: guy.gallasch@unisa.edu.au

Abstract. Transitions in a Timed Coloured Petri Net exhibit an *eagerness-to-execute* property due to the time semantics employed. Every transition will occur at the earliest model time that it becomes enabled, unless conflict prevents it from doing so. It is known that this property may preclude discovery of optimal schedules, i.e. those that achieve the goal in the shortest time. In this paper we propose a relaxation of the time semantics that eliminates the eagerness-to-execute property. We provide motivation for our proposal in the context of task scheduling and describe an attempt to model the effect of our proposal.
Keywords: Timed Coloured Petri Nets, Eagerness-to-Execute, Task Scheduling.

1 Introduction

The Computer Systems Engineering Centre has long been involved in collaborative projects that apply Coloured Petri Nets (CPNs) [11] to industrial-scale systems. Two significant projects have been with Australia's Defence Science and Technology Organisation (DSTO) [3] (and later with National Information and Communication Technology Australia [14]) on the topic of operational planning [4, 5, 10, 12, 13, 15, 16], and modelling logistics processes [6–9]. We have used both untimed and Timed CPNs within these projects.

Timed CPNs incorporate a time semantics that causes an *eagerness-to-execute* behaviour to be exhibited by transitions: transitions will occur at the earliest model time that they become enabled (colour enabled and ready [11]) unless conflict results in them becoming disabled. We have observed that when using CPNs to model a process for the scheduling of tasks with fixed durations (as part of the operational planning work) that this eagerness-to-execute property manifests itself as task schedules in which all tasks are executed as early as possible. When producing schedules of tasks that are intended to achieve a particular goal, this behaviour may preclude discovery of the optimal schedule, as it is possible that delaying particular tasks may result in a schedule with a smaller makespan (time taken to achieve the goal). An example of this is given in [5]. We have made similar observations in our modelling work on logistics processes.

Early work (e.g. [2]) on scheduling used Timed Petri Nets, where durations are associated with transitions, and considered cyclic systems. Closest to our work is that of van der Aalst [1] who considered non-cyclic scheduling problems when using a timed Petri net formalism that associates timestamps with tokens. The timestamps are incremented when transitions fire, by a value associated with the transition. This is very close to the semantics of Timed CPNs, which are a generalisation of this formalism. Aalst's formalism thus possesses the same eager-to-execute semantics and he recognised the same problem with failing to discover optimum

* This work was stimulated by the NICTA-UniSA Research Agreement "Modelling and Analysis of Operations Planning using Untimed CPNs" of April 2006.

schedules. He discusses removing the restriction that transitions must fire as soon as possible to allow discovery of optimal schedules from the reachability graph (RG), however, the results in [1] are restricted to eager-to-execute semantics.

As suggested by Aalst [1], in this position paper, we propose a modified semantics for Timed Coloured Petri nets that eliminates the eagerness-to-execute property by removing the restriction that transitions must fire as soon as possible. Section 2 relates our experience with using both untimed and Timed CPN models for the purpose of task scheduling. In Section 3 we present our proposal and illustrate it with a simple example. We discover that in some situations, such as task scheduling, relaxation of the time semantics in this way is problematic, hence we present a refined proposal in Section 4. An attempt to model the effect of our proposal is given in Section 5 in the context of task scheduling. The paper concludes in Section 6.

2 Timed or Untimed?

A simple task-scheduling CPN model, presented in [4], was developed as part of the operational planning work that did not use conventional timestamps but encoded time within tokens. The rationale behind the development of that model was to avoid the eagerness-to-execute property of Timed CPNs, so that the execution of a task could be delayed and not necessarily occur as soon as all of its starting requirements were satisfied.

As we discovered, using untimed CPNs and modelling time within tokens was not restrictive enough. This methodology resulted in sequences of actions in the RG that were, in fact, inconsistent with the encoded timing information in the corresponding nodes and arcs [4], so-called *infeasible* schedules. This stems from the lack of a *synchronisation mechanism* in the untimed model, such as that provided by the global clock in Timed CPNs. Because of this lack, the occurrence of transitions in the untimed model was not forced to obey temporal constraints in the sequencing of task starting and terminating events, as the encoded time information was calculated on a 'per transition' basis and written into tokens, somewhat independently of the sequence of transition occurrences.

One way of addressing this shortcoming of the untimed model is to generate the infeasible schedules but remove them or simply ignore them when analysing the RG. This is undesirable, as it addresses the effect of the problem but doesn't tackle its source. Another, more satisfactory solution is to not generate the infeasible schedules in the first place, but not preclude any feasible sequences. This is the motivation for our proposal in this position paper in Sections 3 and 4.

2.1 Using Untimed CPNs that Encode Time

In order to force transition occurrences in the model to obey temporal constraints, a synchronisation mechanism needs to be implemented, to prevent events occurring before they are 'allowed to', based on the timing information contained in the tokens. For example, if two tasks start at time 0, one with a duration of 5 and the other with a duration of 10, then the task with a duration of 10 cannot terminate before the task with a duration of 5 (when considering fixed durations).

The RG of an untimed model will capture all interleavings of (concurrently) enabled transitions. It is this property of untimed CPNs that makes them desirable for the discovery of all schedules of tasks, as this property manifests itself as the ability to delay the start of

tasks. However, this strength is also a weakness when modelling time within tokens. Because a nondeterministic choice is made whenever more than one transition is (concurrently) enabled, there is nothing to prevent the occurrence of a particular transition from being delayed indefinitely (provided that at least one other transition is always enabled). There is no simple net structure mechanism to force a transition to occur 'at a particular time'.

2.2 An Approach Based on Timed CPNs

Thankfully, however, Timed CPNs *do* provide such a synchronisation mechanism, that *does* force transitions to occur at a particular model time. However, as we know, it forces all transitions to occur as soon as they are able. What we would like to do is to allow *some* transitions to be delayed.

The problem can be summarised as follows: Untimed CPNs provide the flexibility to delay transitions, but not easily enforce time constraints, whereas Timed CPNs enforce timing constraints but do not allow transitions to be delayed. In a sense, these two approaches sit either side of the solution we are looking for. Our investigations in [5] indicate that it is simpler to relax the behaviour of Timed CPNs than introduce rigidity into Untimed CPNs.

3 Removing the Eagerness to Execute Property

In a Timed CPN, in any given marking, the next transition to fire is nondeterministically selected from only those colour-enabled transitions that are ready at the earliest model time. The gist of our proposal is to allow this selection to be made from all colour-enabled transitions. This is essentially how untimed CPNs behave, but with the global clock advancing to the appropriate value (the amount required to enable the selected colour-enabled transition). This follows the proposal given in [1].

This requires but a minor change to the definition of the enabling rule of Timed CPNs as given in [11]. In terms of *steps* (multisets of binding elements), Definition 11.6 in [11] specifies that a step, Y, is enabled at time t' in a timed marking, (M, t^*) (where t^* is the value of the global clock in marking M), if and only if the following properties are satisfied:

1. all binding elements in the step satisfy the guard of the corresponding transition;
2. sufficient tokens exist in all untimed input places to satisfy the untimed input arcs for all binding elements in the step;
3. appropriate timed tokens exist in all timed input places to satisfy the timed input arcs at time t' for all binding elements in the step;
4. the current global clock, t^*, is less than or equal to t'; and
5. t' is the smallest time value for which there exists a step satisfying conditions 1 to 4.

Our proposal modifies the fifth condition so that any step can be considered, not just the steps enabled at the smallest time value. We propose condition 5 to become:

5. t' is the smallest time value that satisfies conditions 1 to 4 for step Y.

Hence, we do not require t' to be the smallest time that satisfies conditions 1 to 4 over all possible steps, thereby allowing any step that is colour enabled to be executed, and the model time to advance only to the necessary time to enable that step. Note that condition 4 is still valid with our proposed time semantics, as t' does not need to be the minimum time at which

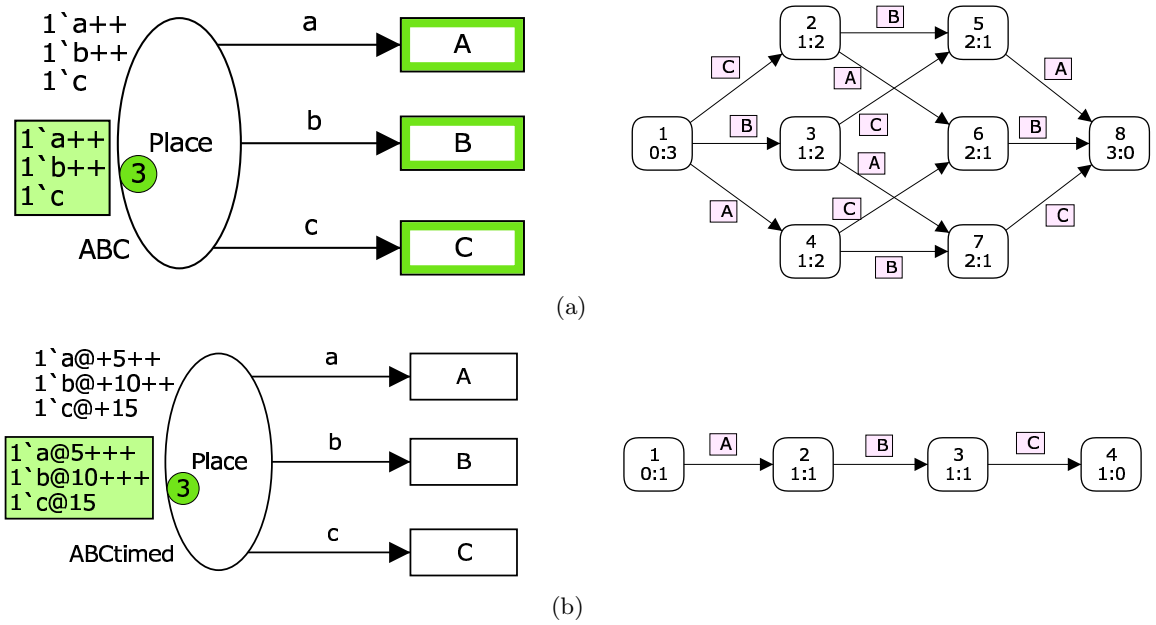


Fig. 1. (a) an Untimed CPN model and its Reachability Graph; and (b) a corresponding Timed CPN model and its Reachability Graph.

step Y is enabled, but rather greater than or equal to that minimum time. Our intention is for the modified condition 5 to allow steps to be enabled at times larger than the minimum required to enable a step, i.e. if step Y_1 is enabled at the earliest at time 5, and step Y_2 is enabled at the earliest at time 10, step Y_2 can occur at time 10, causing step 5 to be delayed until at least time 10. This modification still allows transitions to occur at the time they first become enabled, but it also allows transitions to be delayed. The delay is not arbitrary, but rather it corresponds to the increase in the global clock resulting from the occurrence of one or more other transitions.

As an example, consider the Untimed CPN and its RG in Fig. 1 (a). ABC is an enumerated colour set containing the values a, b and c. The three transitions, A, B and C, are concurrently enabled and can occur in any order, which is reflected in the RG. RGs are generated using interleaving and hence arcs of the RG represent a single binding element. In general a step enabled in the initial marking (marking 1) could comprise, for example, all three transitions. This corresponds to moving from marking 1 to marking 8 in one step.

Figure 1 (b) shows a Timed CPN version of the net in Fig. 1 (a) and its RG, in which the global clock is 0 in the initial marking. From the five conditions given above for the existing time semantics, the only step enabled by the initial marking is the step comprising transition A at time 5: A's guard is satisfied (condition 1); there are no untimed input places (condition 2); an 'a' token with timestamp 5 marks Place (condition 3); the current global clock, 0, is less than or equal to 5 (condition 4); and no other step exists that is enabled at a time less than 5 (condition 5). Transition A will occur at time 5, and for the same reasons B will occur at time 10, and C at time 15. From the firing rule of Timed CPNs, the values for the global clock for markings 2, 3 and 4 are 5, 10 and 15 respectively.

In the case of our relaxed timed semantics, the example in Fig. 1 (b) has 7 steps enabled in the initial marking (we omit the empty binding, $\langle\rangle$, from each binding element): 1‘A, 1‘B, 1‘C, 1‘A++1‘B, 1‘A++1‘C, 1‘B++1‘C, and 1‘A++1‘B++1‘C. Condition 1 is satisfied because

all guards are true; there are no untimed input places so condition 2 is satisfied; there are sufficient timed tokens on all timed input places and t' can be set to 5, 10 or 15 as necessary (condition 3); initially the global clock is zero, $t^* = 0$, so $t^* \leq t'$ (condition 4); and $t' = 5$ for 1`A, $t' = 10$ for 1`B and 1`A++1`B; and $t' = 15$ for the remaining four steps, to satisfy condition 5. When generating the RG we only consider interleaving and hence only the first 3 steps (1`A, 1`B and 1`C). On their occurrence we obtain markings 2, 3 and 4 shown in Fig. 1 (a), except that timing information has been added: timestamps as in Fig. 1 (b) and the value of the global clock being 0 for the initial marking, 15 for marking 2, 10 for marking 3 and 5 for marking 4 according to the firing rule for Timed CPNs. In marking 2, we can see that transitions A and B are concurrently enabled at $t' = 15$. t' must be set to 15 to satisfy both conditions 4 and 5. This leads to the successor markings 5 and 6, and their successor, marking 8, with the global clock remaining at 15 ($t^* = 15$). In marking 3, $t^* = 10$. C is enabled with $t' = 15$, to satisfy conditions 3 and 5 (thus condition 4 also holds), leading to marking 5 with $t^* = 15$. A is enabled with $t' = 10$ (to satisfy conditions 4 and 5) and results in marking 7, with $t^* = 10$. In marking 4, $t^* = 5$. B is enabled with $t' = 10$ (to satisfy conditions 3 and 5), leading to marking 7 and similarly C is enabled with $t' = 15$, resulting in marking 6. In marking 7, C is enabled with $t' = 15$ to satisfy conditions 3 and 5 (condition 4 is satisfied as $t^* = 10$), and when it occurs results in marking 8 ($t^* = 15$). Hence we can see that in this case the RG with relaxed time semantics is isomorphic to the untimed RG. This illustrates how our relaxed time semantics allows transitions to be delayed until after the occurrence of one or more other transitions enabled by a larger global clock.

4 Refining our Relaxed Time Semantics

The proposal in Section 3 follows our own initial belief and coincides with the proposal in [1]. But we have found situations in which this particular relaxation of the time semantics introduces behaviour that we do not desire. For example, our task scheduling models (see e.g. [13]) execute tasks of fixed duration using one transition to represent the start of the task and another transition to represent the termination of the task. If such a task has a fixed duration, d, then we desire the task termination transition to occur exactly d time units after the task start transition, and not be delayed. The time semantics resulting from our proposed change are too relaxed in this sense, as we can no longer enforce the execution of transitions at specific model times.

A refinement to our proposal is thus to allow only a subset of transitions to be delayed within a Timed CPN model. It is possible to modify the definition of Timed CPNs from [11] (Definition 11.4, for non-hierarchical Timed CPNs) so that the definition of transitions recognises two disjoint classes of transition: *fixed* transitions (not delayable) and *delayable* transitions; such that point 2 of Definition 11.4 from [11] becomes:

2. $T = T_f \cup T_d$ is a finite set of **transitions** comprising **fixed transitions**, T_f, and **delayable transitions**, T_d, such that $T_f \cap T_d = \emptyset$ and $P \cap T = \emptyset$.

The enabling rule must then be modified to disallow the enabling of any step that would require the global clock to advance past the model time at which any fixed transition is enabled. To do this, condition 5 of Definition 11.6 from [11] can be further modified from that in Section 3 to become:

5. t' is the smallest time value that satisfies conditions 1 to 4 for step Y, and there does not exist any other step comprising at least one fixed transition that is enabled at a time value less than t'.

By doing so, we are able to prevent certain transitions from being delayed, and hence in the context of task scheduling enforce fixed task durations. This proposal is more general than our first proposal in Section 3 in the sense that we can capture all the behaviour of the first proposal but also enforce the eagerness-to-execute behaviour exhibited by the current time semantics of Timed CPNs, whereas our first proposal cannot enforce such behaviour.

One of the motivations given in [11] for adopting the particular time semantics for Timed CPNs is to preserve occurrence sequences between the timed model and its corresponding untimed equivalent: the occurrence sequences exhibited by a Timed CPN model will be a subset of those exhibited by its corresponding Untimed CPN model [11]. This means that turning an untimed model into a timed model cannot introduce new behaviour in terms of occurrence sequences. A second motivation, for considering that the occurrence of transitions is an instantaneous event, is that the set of reachable markings of the Timed CPN is a subset (when excluding the global clock and timestamps) of those of the corresponding Untimed CPN, i.e. no new markings are introduced when time is introduced into an untimed model.

We conjecture that our proposal for relaxing the time semantics of Timed CPNs also preserves these properties. Further, we contend that the reachability graphs of relaxed timed CPNs (with no fixed transitions) are isomorphic to the corresponding untimed model.

5 Modelling our Proposal

In [5] we have attempted to produce a highly simplified version of the task scheduling model from [13] that exhibits behaviour similar to that which would result from implementation of the proposed time semantics from Section 4. We do so by using additional net structure to *induce* a delay.

Figure 2 shows the simplified task execution engine from [5]. The two transitions, Start and Terminate, model the start and termination of tasks. Tasks to be executed initially reside in the Idle Tasks place, moving to the Executing place while executing, and ending up in the Terminated place once executed. Resources and conditions provide constraints on the specific tasks that can be executed, and when. We wish for Start to be able to be delayed, but not Terminate, so that fixed durations for tasks can be enforced by the model. More details of the basic operation of the complete model, from which this highly simplified version was derived, can be found in [13].

In bold are two places, Semaphore 1 and Semaphore 2, both members of the Semaphore fusion set (but drawn separately for convenience), and arcs that connect these two places to the Start and Terminate transitions. These two places and their associated arcs implement the mechanism that mimics a relaxed time semantics, along with the net shown in Fig. 3, which we now explain.

In Fig. 3 is a place, Semaphore, and a transition, InduceDelay. The sole purpose of this page is to allow the model time to advance instead of a timed transition being forced to occur. The Semaphore place, part of the same fusion set as the two semaphore places in Fig. 2, contains a single token comprising a boolean and a time value. InduceDelay is enabled whenever the boolean part is true, and its occurrence changes the boolean part to false without changing the time value stored in the token.

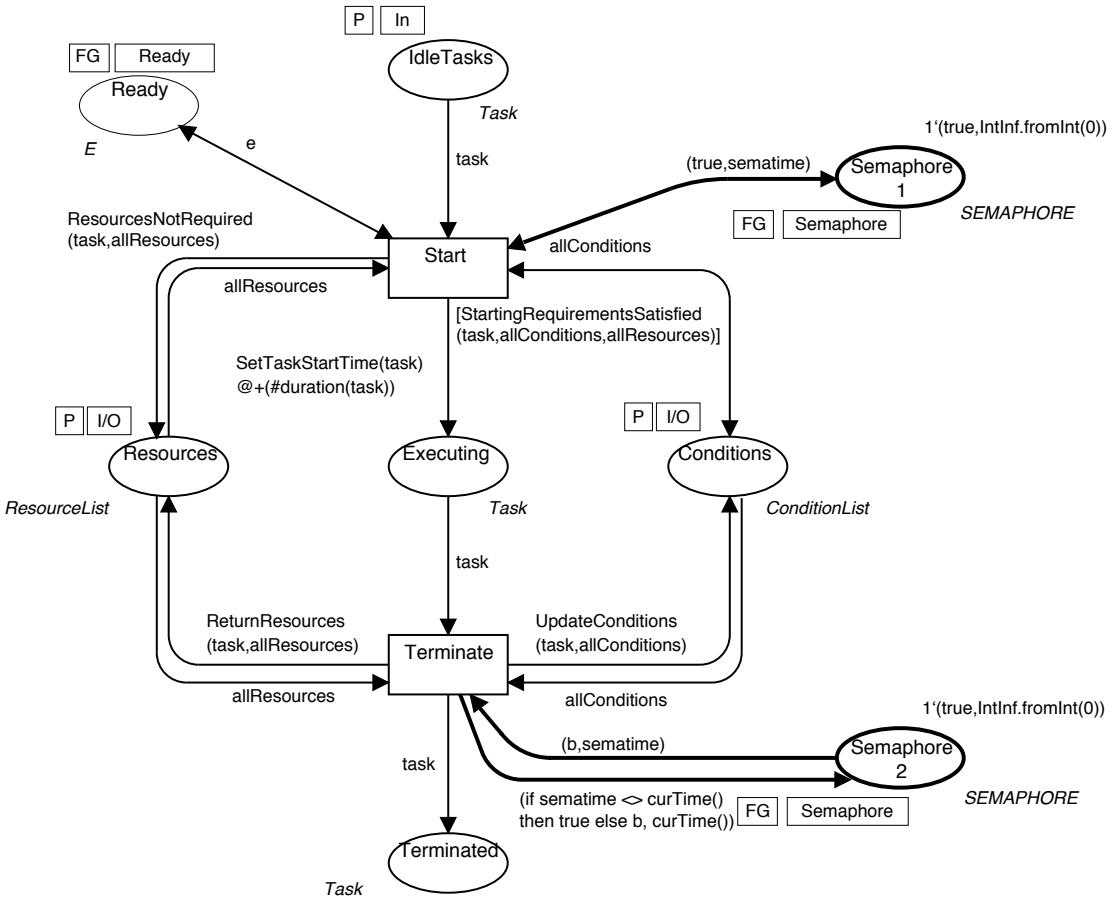


Fig. 2. Part of a Simplifed Task Execution Engine Model from [5].

Figure 2 shows how the token in Semaphore can be used to delay transitions. Transitions that we want to be able to delay are connected to the Semaphore place with an arc inscription requiring the semaphore to be true, as is the case with the Start transition in Fig. 2. InduceDelay is enabled at model time 0, and hence any number of transitions can occur at time 0 (provided they are colour-enabled and ready) before InduceDelay nondeterministically sets the semaphore to false. So, suppose Start was enabled at time 0. If InduceDelay occurs first, Start becomes disabled, and will not be enabled again until the semaphore is again set to true. Alternatively, Start could have occurred at time 0. Transitions that we don't wish to delay are connected to the Semaphore place by arcs that reset the boolean to true and update the time value to the current model time, as is the case with the Terminate transition. Once such a transition occurs, the timed transitions that were previously disabled (delayed) can now occur, if all other enabling conditions are still satisfied.

One emergent property of this implementation was that transitions can be delayed indefinitely, and indeed never occur. This is either a limitation or a bonus, depending on your point of view. For our operational planning work this was a bonus, as we also wanted to model the ability of tasks to be delayed indefinitely. Unfortunately, implementing a solution in this way results in state explosion caused by the additional InduceDelay transition and Semaphore place, and is not straightforwardly extended to larger models with more complex transition interactions.

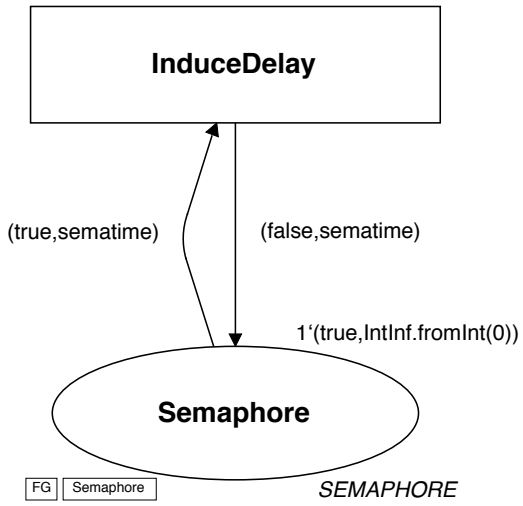


Fig. 3. The InduceDelay Transition to mimic our proposed relaxed time semantics.

6 Conclusions and Future Work

In this paper we propose a relaxation of the time semantics of Timed CPNs to allow some or all transitions to be excluded from the eagerness-to-execute behaviour imposed by the existing time semantics of Timed CPNs. This new time semantics has application in the area of task scheduling, including the modelling of logistics systems and operational planning.

The relaxed semantics is a true relaxation in the sense that it permits all behaviour of the current time semantics of Timed CPNs while also permitting all behaviour allowed by our first proposal in Section 3. Our intial proposal did also capture the eagerness-to-execute behaviour given by the current time semantics but not without also allowing the possibility that all transitions could be delayed, whereas our second proposal has the capability to enforce eagerness-to-execute behaviour.

If our conjecture that occurrence sequences and markings are preserved from Untimed CPNs under our new time semantics, it may follow that many, if not all, of the theory surrounding the properties and analysis capabilities of Timed CPNs will also hold for our new time semantics. Hence, our new time semantics has the potential to be used for the analysis of any system currently analysed by Timed CPNs. Establishing a firm theoretical foundation is thus of high significance and priority. If this can be established, and our proposal given tool support, we would like to investigate the use of this modified time semantics in a case study of significance, such as the operational planning work in [5, 13].

In the previous section we illustrated an attempt to model our proposed time semantics, however this becomes cumbersome and error-prone for large-scale models, as we discovered when we attempted to do something similar to the logistics distribution network model in [6,8]. Implementing the modified time semantics in a tool will allow the operational planning models to be used without any modification.

References

1. W.M.P. van der Aalst. Petri net based scheduling. *OR Spectrum*, 18:219–229, 1996.
2. J. Carlier and P. Chretienne. *Timed Petri Net Schedules*, volume 340 of *Lecture Notes in Computer Science*, pages 62–84. Springer, 1988.

3. Defence Science and Technology Organisation (DSTO). http://www.dsto.defence.gov.au.
4. B. Han G. E. Gallasch and J. Billington. COAST User Interface Design, Integration and Support, and the Development of an Untimed COAST Server. Technical Report CSEC-22, Computer Systems Engineering Centre Report Series, University of South Australia, June 2005, revised July 2005. (93 pages).
5. G. E. Gallasch and J. Billington. Modelling and Analysis of Operations Planning Using Untimed and Timed CPNs. Technical Report CSEC-27, Computer Systems Engineering Centre Report Series, University of South Australia, September 2006. (73 pages).
6. G. E. Gallasch, N. Lilith, and J. Billington. A Coloured Petri Net Model of a Defence Logistics Physical Network. Technical Report CSEC-25, Computer Systems Engineering Centre Report Series, University of South Australia, August 2006. (140 pages).
7. G. E. Gallasch, N. Lilith, and J. Billington. Coloured Petri Net Modelling of Defence Logistics. Technical Report CSEC-33, Computer Systems Engineering Centre Report Series, University of South Australia, July 2008. (198 pages).
8. G. E. Gallasch, N. Lilith, J. Billington, L. Zhang, A. Bender, and B. Francis. Modelling Defence Logistics Networks. *International Journal on Software Tools for Technology Transfer, special section on CPN'06*, 10(1):75–93, 2008.
9. G. E. Gallasch, C. Moon, B. Francis, and J. Billington. Modelling personnel within a defence logistics maintenance process. In *Proceedings of 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems (SIMUTools 2008)*, March 2008. (10 pages).
10. G. E. Gallasch J. Freiheit and J. Billington. About the Use of Untimed CPN Models for COAST and Further COAST Client Development Support. Technical Report CSEC-20, Computer Systems Engineering Centre Report Series, University of South Australia, December 2004. (33 pages).
11. K. Jensen and L.M. Kristensen. *Coloured Petri Nets: Modelling and Validation of Concurrent Systems.* Springer, 2009.
12. L. M. Kristensen. The COAST Server, Design and Implementation. Technical Report CSEC-5, Computer Systems Engineering Centre Report Series, University of South Australia, July 2002.
13. L. M. Kristensen, P. Mechlenborg, L. Zhang, B. Mitchell, and G. E. Gallasch. Model-based Development of a Course of Action Scheduling Tool. *International Journal on Software Tools for Technology Transfer, special section on CPN'06*, 10(1):5–14, 2008.
14. NICTA - Australia's ICT Research Centre of Excellence. http://www.nicta.com.au.
15. L. Zhang, L. M. Kristensen, C. Janczura, G. E. Gallasch, and J. Billington. A Coloured Petri Net based Tool for Course of Action Development and Analysis. In *Formal Methods in Software Engineering and Defence Systems 2002, Proceedings of the* Satellite Workshops on Software Engineering and Formal Methods *and* Formal Methods Applied to Defence Systems, volume 12 of *Conferences in Research and Practice in Information Technology Series*, pages 125–134. Australian Computer Society Inc., 2002.
16. L. Zhang, L. M. Kristensen, B. Mitchell, G. E. Gallasch, P. Mechlenborg, and C. Janczura. COAST - An Operational Planning Tool for Course of Action Development and Analysis. In *Proceedings of the 9th International Command and Control Research and Technology Symposium (ICCRTS), Copenhagen, Denmark.*, 2004.

Search-Order Independent State Caching*

Sami Evangelista[1,2] and Lars Michael Kristensen[3]

[1] Computer Science Department, Aarhus University, Denmark
[2] LIPN, Université Paris 13, France
sami.evangelista@lipn.univ-paris13.fr
[3] Department of Computer Engineering, Bergen University College, Norway
lmkr@hib.no

Abstract. State caching is a memory reduction technique used by model checkers to alleviate the state explosion problem. It has traditionally been coupled with a depth-first search to ensure termination. We propose and experimentally evaluate an extension of the state caching method for general state exploring algorithms that are independent of the search order (i.e., search algorithms that partition the state space into closed (visited) states, open (to visit) states and unmet states).

1 Introduction

Model checking is one of the techniques used to detect defects in system designs. Its principle is to perform an exhaustive exploration of all system states to track erroneous behaviors. Although it provides some advantages compared to other verification methods, its practical use is sometimes prohibited by the well-known state explosion problem: the state space of the system may be far too large to be explored with the available computing resources.

The literature is replete with examples of techniques designed to tackle, or at least postpone, the state explosion problem. While some techniques, like partial order reduction [12], reduce the part of the state space that must be explored while still guaranteeing soundness and completeness, more pragmatic approaches make a better use of available resources to extend the range of systems that can be analyzed. State compression [16], external memory algorithms [1], and distributed algorithms [24] are examples of such techniques. In this paper we focus on the *state caching* method first proposed by Holzmann in [14].

State caching is based on the idea that, in depth-first search (DFS), only the states on the current search path need to be in memory to detect cycles. All states that have been visited but have left the DFS stack can thus be deleted from memory without endangering the termination of the search. This comes at the cost of potentially revisiting states and for many state spaces, time becomes the main limiting factor.

The state caching method has been developed and mostly studied in the context of depth-first search since this search order makes it easy to guarantee

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

termination. In this paper we propose an extension of state space caching to General State Exploring Algorithms (GSEA). Following the definition of [5], we put in this family all algorithms that partition the state space into three sets: the set of *open* states that have been seen but not yet expanded (i.e., some of their successors may not have been generated); the set of *closed* states that have been seen and expanded; and the set of unseen states. DFS, BFS, and directed search algorithms [8] like Best-First Search and A* are examples of such general state exploring algorithms.

The principle of our extension is to detect cycles and guarantee termination by maintaining a tree rooted in the initial state and covering all open states. States that are part of that tree may not be removed from the cache, while others are candidates for replacement. Hence, any state that is not an ancestor in the search tree of an unprocessed state can be removed from memory. This tree is implicitly constructed by the state caching algorithm in DFS, since DFS always maintains a path from the initial state to the current state, while for GSEA it has to be explicitly built. However, our experimental results demonstrate that the overhead both in time and memory of this explicit construction is negligible.

The generalized state caching reduction is implemented in our model checker ASAP [26]. We report on the results of experiments made to assess the benefits of the reduction in combination with different search orders: BFS, DFS, and several variations and combinations of these two; and with the sweep-line method [20] which we show is compatible with our generalized state caching reduction. The general conclusions we draw from these experiments are that (1) the memory reduction is usually better with DFS than with BFS although we never really experienced with BFS a time explosion; (2) BFS is to be preferred for some classes of state spaces; (3) a combination of BFS and DFS often outperforms DFS with respect to both time and memory; (4) state caching can further enhance the memory reduction provided by the sweep-line method.

Structure of the paper. Section 2 presents the principle of a general state exploring algorithm, and Section 3 describes our state caching mechanism for the general algorithm. In Section 4 we put our generalized state caching method into context by discussing its compatibility with related reduction techniques. Section 5 reports on the results of experiments made with the implementation of the new algorithm. Finally, Section 6 concludes this paper.

Definitions and notations. From now on we assume to be given a universe of system states $\mathcal{S}$, an initial state $s_0 \in \mathcal{S}$, a set of events $\mathcal{E}$, an enabling function $en : \mathcal{S} \to 2^{\mathcal{E}}$ and a successor function $succ : \mathcal{S} \times \mathcal{E} \to \mathcal{S}$; and that we want to explore the state space implied by these parameters, i.e., visit all its states. A state space is a triple (S, T, s_0) such that $S \subseteq \mathcal{S}$ is the set of reachable states and $T \subseteq S \times S$ is the set of transitions defined by:

$$\begin{aligned} S &= \{s_0\} \cup \{\, s \in \mathcal{S} \mid \exists s_1, \ldots, s_n \in \mathcal{S} \text{ with } s = s_n \ \wedge\ s_1 = s_0 \ \wedge \\ &\qquad \forall i \in \{1, \ldots, n-1\} : \exists e_i \in en(s_i) \text{ with } succ(s_i, e_i) = s_{i+1}\} \\ T &= \{(s, s') \in S \times S \mid \exists e \in en(s) \text{ with } succ(s, e) = s'\} \end{aligned}$$

Related work. The principle of state caching dates back to an article of Holzmann [14] in 1985. He noted that, in DFS, cycles always eventually reach a state on the stack and, hence, keeping in memory the states on the current search path ensures termination. Forgetting other states comes at the cost of potentially re-exploring them. In the worst case, if any state leaving the stack is removed from memory, a state will be visited once for each path connecting it to the initial state leading to a potential explosion in run-time. Hence, depending on available memory, a set of states that have left the stack are cached in memory.

The question of the strategy to be used for replacing cached states has been addressed in several papers: [11,14,15,17,18,23]. States can be chosen according to various criteria (e.g., in- and out-degree, visit frequency, stack entry time) or in a purely random way. The experiments reported in [23] stress that no strategy works well on all models and that strategies are, to some extent, complementary.

The *sleep-set* reduction technique [12] is fully compatible with state caching [13]. This reduction eliminates most "useless" interleavings by exploiting the so-called *diamond property* of independent transitions: whatever their execution order they lead to the same state. This has the natural consequence to limit revisits of states removed from the cache. For some protocols (e.g., AT&T's Universal Receiver Protocol, MULOG's mutual exclusion protocol), the result of this combination is impressive: at a reasonable cost in time, the cache size can be reduced to less than 3% of the state space.

All the related work discussed above are coupled with depth-first search. To the best of our knowledge, the only work exploring the combination of state caching with BFS is [21] which is more closely related to our work. Termination in [21] is ensured by taking snapshots of the state space, i.e., memorizing full BFS levels. By increasing the period between two snapshots it is guaranteed that cycles will eventually reach a "pictured" state. This approach is in general incomparable with the present work. The algorithm of [21] has to keep full levels in memory while ours stores some states of each level. Besides this algorithm, [21] also introduces some hierarchical caching strategies and learning mechanisms.

Some other reduction techniques share the philosophy of state caching: only store a subset of the state space while still guaranteeing termination. Examples include the "to-store-or-not" method [3] and the sweep-line method [20]. The compatibility of our algorithm with these two works is discussed in Section 4.

2 General State Exploring Algorithm

A general state exploring algorithm is presented in Fig. 1. It operates on two data structures. The set of open states, $\mathcal{O}$, contains all states that have been reached so far, but for which some successor(s) have not yet been computed. Once all these successors have been computed, the state is moved from $\mathcal{O}$ to the set of closed states $\mathcal{C}$. Initially, the closed set is empty, and the open set only contains the initial state. A set of events *evts* is associated with each open state. It consists of its enabled events that have not been executed so far. In each iteration, the algorithm selects an open state s (l. 3), picks one of its executable

events e (if any, since the state may be a terminal state) and removes it from the set of events to execute $s.evts$ (ll. 4–5). The successor state s' of s reached via the execution of e is computed, and if it is neither in the closed nor in the open set (ll. 7–9), it is put in $\mathcal{O}$ to be later visited and its enabled events are computed. Once the successor is computed, we check if all the enabled events of s have been executed (ll. 10–11), in which case we move s from $\mathcal{O}$ to the closed set $\mathcal{C}$.

```
1:  C := ∅ ; O := {s0} ; s0.evts := en(s0)
2:  while O ≠ ∅ do
3:      s := choose from O
4:      if there exists e ∈ s.evts then
5:          s.evts := s.evts \ {e}
6:          s' := succ(s, e)
7:          if s' ∉ C ∪ O then
8:              O := O ∪ {s'}
9:              s'.evts := en(s')
10:     if s.evts = ∅ then
11:         C := C ∪ {s} ; O := O \ {s}
```

Fig. 1. A general state exploring algorithm

In the rest of this paper we shall use the following terminology. *Expanding* a state s consists of executing one of its enabled events and putting its successor in the open set if needed (ll. 6–9). A state s will be characterized as *expanded* if all its successor states have been computed, i.e., $s.evts = \emptyset$, and it has been moved to the closed set; and as *partially expanded* if some of its successors have been computed, but s is still in the open set. A state s *generates* state s' if $succ(s, e) = s'$ for some $e \in en(s)$ and $s' \notin \mathcal{C} \cup \mathcal{O}$ when expanding s. In this case, the transition (s, s') is said to be the *generating transition*.

The algorithm in Fig. 1 differs slightly from explicit state space search algorithms usually found in the literature, e.g., like the GSEA of [5]. In an iteration, the algorithm only executes one event rather than all executable events of a state. This variation is more flexible as it allows us to have open states that are partially expanded. Hence, it naturally caters for search order independence. Depending on the implementation of the open set, the search strategy can be, for instance, depth-first (with a stack), breadth-first (with a queue), or best-first (with a priority queue). Since each search order has its pros and cons, it is of interest to design reduction techniques working directly on the generic search order independent template, e.g., like the partial order reduction proposed in [5], rather than on a specific instance.

3 State Caching for GSEA

The key principle of state caching for depth-first search is that cycles always eventually reach a state on the DFS stack. Hence, it is only necessary to keep this stack in memory to ensure termination of the algorithm. In breadth-first search, or more generally for a GSEA, we do not have such a structure to rely on in order to detect cycles. Hence, a BFS naively combined with state caching may never terminate.

To overcome this limitation, we propose to equip GSEA with a mechanism that allows it to avoid reentering cycles of states and thereby ensures termination.

The principle of this modification is to maintain, as the search progresses, a so-called *termination detection tree* (TD-tree). The TD-tree is rooted in the initial state s_0 and keeps track of unprocessed states of the open set as explained below. To formulate the requirements of the TD-tree we shall use the term *search tree*. The search tree is the sub-graph of the state space which at any moment during the execution of GSEA covers all open and closed states, and contains only generating transitions. In other words, it consists of the state space explored so far from which we remove transitions of which the exploration led to an already seen state, i.e., in the set $\mathcal{C} \cup \mathcal{O}$.

The three following invariants related to TD-tree must be maintained during the state space search:

I1 The TD-tree is a sub-tree of the search tree;
I2 All open states are covered by the TD-tree;
I3 All the leaves of the TD-tree are open states.

A sufficient condition for the modified GSEA to terminate is that we always keep in memory the states belonging to the TD-tree. Intuitively, when expanding a state s picked from the open set, we are sure (provided invariants I1 and I2 are valid) that any cycle covering s will at some point contain a state s' belonging to the TD-tree. States that can be deleted from memory are all closed states that are not part of the TD-tree: their presence is not required to detect cycles. Note that only the two first invariants are required for termination. Invariant I3 just specifies that the TD-tree is not unnecessarily large, i.e., it does not contain states we do not need to keep in the closed set to detect cycles. To sum up, all the states that may not be removed from the cache are (besides open states) all closed states that generated (directly or indirectly through a sequence of generating transitions) a state in the open set.

As an example, let us see see how a BFS extended with this mechanism will explore the state space of Fig. 2(top left). Each state is inscribed with a state number that coincides with the (standard) BFS search order. The TD-tree has been drawn in the right box for several steps of the algorithm. The legend for this box is shown in the bottom left box of the figure. Note that these graphical conventions are used throughout the paper. Some reference counters used to maintain the TD-tree appear next to the states. They will not be discussed now. Their use will become clear after the presentation of the algorithm.

After the expansion of the initial state 0, the queue contains its two successor states 1 and 2 and the TD-tree is equivalent to the search tree (see Step 1). At the next level, we expand open states 1 and 2. State 1 first generates states 3 and 4. As state 4 is already in the open set when state 2 is expanded, this one does not generate any new states, which means that at Step 2, state 2 does not have any successors in the search tree. It is deleted from the TD-tree, and we assume that the algorithm also removes it from the closed set. The expansion of open states 3 and 4 at the next level generates the three states 5, 6 and 7. At Step 3 all the states that were expanded at level 2 generated at least one (new) state. Hence, no state is deleted from the TD-tree. Level 3 is then processed.

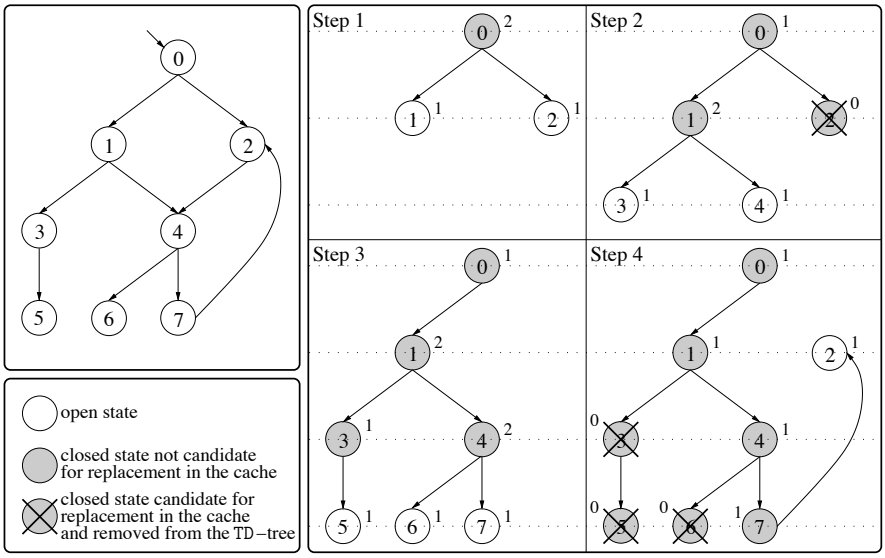


Fig. 2. A state space (top left), the TD-tree at three different stages of a BFS (right) and the legend for the right box (bottom left). Step 1: after the expansion of state 0. Step 2: after the expansion of states 1 and 2. Step 3: after the expansion of states 3 and 4. Step 4: after the expansion of states 5, 6 and 7.

States 5 and 6 do not have any successors and state 7 has a single successor, state 2, which has been visited but deleted from memory. Hence, it is put in the open set. States 5 and 6 can be deleted from the TD-tree at Step 4 since they are terminal states and do not have any successors in the TD-tree. After the deletion of state 5, state 3 is in the same situation and becomes a leaf of the TD-tree. It is thus also deleted. The queue now only contains state 2 that had already been previously expanded. Its only successor, state 4, belongs to the TD-tree and, hence, is present in memory. The algorithm thus detects the cycle 2→4→7→2. After this last expansion, the queue is empty and algorithm terminates.

The operation of our algorithm is similar to the basic state caching reduction for DFS with the difference that the TD-tree is implicitly maintained by the DFS state caching algorithm: it consists of all the open states located on the DFS stack. Closed states have left the stack so they have not generated the states currently on the stack and do not need anymore to be part of the TD-tree.

We now introduce the algorithm of Fig. 3, a GSEA extended with the state caching mechanism we described above. The main procedure (on the left column) works basically as the algorithm of Fig. 1 except that we inserted the lines preceded by a ► to manage the state cache. Apart from the closed set $\mathcal{C}$ and open set $\mathcal{O}$, the algorithm also uses a set $\mathcal{D} \subseteq \mathcal{C}$ that contains all states candidate for deletion if memory becomes scarce, i.e., all the states that left the TD-tree. Hence, the TD-tree is composed of the states in $(\mathcal{C} \cup \mathcal{O}) \setminus \mathcal{D}$.

```
1:  C := ∅ ; O := {s0} ; s0.evts := en(s0)
2:  ► D := ∅ ; s0.refs := 1 ; s0.pred := nil
3:  while O ≠ ∅ do
4:      s := choose from O
5:      if there exists e ∈ s.evts then
6:          s.evts := s.evts \ {e}
7:          s' := succ(s, e)
8:          if s' ∉ C ∪ O then
9:              O := O ∪ {s'}
10:             s'.evts := en(s')
11:             ► s.refs := s.refs + 1
12:             ► s'.refs := 1 ; s'.pred := s
13:             ► garbageCollection()
14:     if s.evts = ∅ then
15:         C := C ∪ {s} ; O := O \ {s}
16:         ► unref(s)

17: procedure unref(s) is
18:     s.refs := s.refs − 1
19:     if s.refs = 0 then
20:         D := D ∪ {s}
21:         if s.pred ≠ nil then
22:             unref(s.pred)
23:
24: procedure garbageCollection() is
25:     if |O| + |C| > MaxMemory then
26:         if D = ∅ then
27:             report "out of memory"
28:         else
29:             s := choose from D
30:             C := C \ {s}
31:             D := D \ {s}
```

Fig. 3. A general state exploring algorithm combined with state caching

After each state expansion, the algorithm calls the *garbageCollection* procedure (l. 13) that checks if the number of states kept in memory exceeds some user-defined limit `MaxMemory`. In that case, one of the candidates for replacement is selected from $\mathcal{D}$ according to a replacement strategy and deleted from both $\mathcal{C}$ and $\mathcal{D}$ (ll. 29–31) to make room for the state newly inserted in $\mathcal{O}$. If there is no candidate (ll. 26–27), the algorithm terminates with a failure: the TD-tree is too large to fit within user-defined available memory.

In order to maintain the TD-tree, two additional attributes are associated with states. The first one, *pred*, identifies the predecessor of the state that previously generated it, i.e., its predecessor in the search tree. It is set at l. 12 when a new state s' is generated from s. The second attribute, *refs*, is a reference counter used by the *garbageCollection* procedure to determine when a closed state leaves the TD-tree and can become a candidate for replacement. The following invariant is maintained by the algorithm for any $s \in \mathcal{C} \cup \mathcal{O}$:

$$\textbf{I4}\ \ s.refs = |\{s' \in \mathcal{C} \cup \mathcal{O} \textbf{ with } s'.pred = s \wedge s'.refs > 0\}| + \begin{cases} 0 \textbf{ if } s \notin \mathcal{O} \\ 1 \textbf{ if } s \in \mathcal{O} \end{cases}$$

In other words, *s.refs* records the number of successors of s in the TD-tree incremented by 1 if s is an open state. It directly follows from invariant I4 that any state s with $s.refs = 0$ must be moved to $\mathcal{D}$ and deleted from the TD-tree in order to satisfy invariant I3. This is the purpose of procedure *unref* called each time a state s leaves the open set (ll. 15–16). Its counter is decremented by 1, and if it reaches 0 (ll. 19–22), s is put in the candidate set and the procedure is recursively called on its predecessor in the TD-tree (if any).

Let us consider again the TD-trees depicted in Fig.2(right). Reference counters are given next to the states. At Step 4, after the expansion of states 5, 6, and 7 of level 3, the reference counter of 5 and 6 reaches 0: they have left the open

set and have no successors in the search tree, i.e., they did not generate any new state. They can therefore be put in the candidate set and leave the TD-tree. *unref* is then also recursively called on states 3 and state 4 and their counters are decremented to 0 and 1, respectively. Hence, state 3 is also put in the candidate set. This finally causes *unref* to decrement the reference counter of state 1 to 1.

Lemma 1. *The algorithm of Fig.3 terminates after visiting all states.*

Proof. Let $\mathcal{T}$ be the (only) sub-tree of the search tree that satisfies invariants I1, I2 and I3 and assume that the states belonging to $\mathcal{T}$ always remain in set $\mathcal{C} \cup \mathcal{O}$. Let $s_1, \ldots, s_n \in \mathcal{S}$ be a cycle of states with $\forall i \in \{1, \ldots, n\} : succ(s_i, e_i) = s_{i \mod n+1}$, and such that s_1 is its first state to enter $\mathcal{O}$. This cycle is necessarily detected, i.e, during the search we reach some $s_i \in \mathcal{C} \cup \mathcal{O}$. Let us suppose the contrary. Then, each state $s_i \in \{s_1, \ldots, s_{n-1}\}$ generates the state $s_j = s_{i+1}$, i.e., $s_j \notin \mathcal{C} \cup \mathcal{O}$ when event e_i is executed from s_i. Hence, after the execution of e_{n-1} by the algorithm it holds from invariants I1 and I2 that $\{s_1, \ldots, s_n\} \subseteq \mathcal{T}$ since $s_n \in \mathcal{O}$ and each $s_j \in \{s_2, \ldots, s_n\}$ was generated by s_{j-1}. Thus, $s_1 \in \mathcal{C} \cup \mathcal{O}$ when event e_n is executed from s_n, which contradicts our initial assumption.

It is straightforward to see from invariant I4 that $s.refs > 0 \Leftrightarrow s \in \mathcal{T}$. After an iteration of the algorithm (ll.3–16), invariant I4 is trivially ensured. This implies that any $s \in \mathcal{C}$ with $s.refs = 0$ ($\Leftrightarrow s \in \mathcal{D}$) can be deleted from $\mathcal{C}$.

The modified GSEA of Fig. 3 consumes slightly more memory per state to represent the *pred* and *refs* attributes. In our implementation *pred* is encoded with a 4 byte pointer and *refs* using a single byte. Nevertheless, these 5 bytes are usually negligible compared to the size of the bit vector used to encode states.

4 Compatibility with Other Reduction Techniques

We discuss in this section several aspects of our algorithm and its combination with some selected reduction techniques.

4.1 Single-Successor States Chain Reduction

Closely related to state caching, the idea of [3] is to use a boolean function that, given a state, determines if the state should be kept in the closed set or not. The paper proposes functions that guarantee the termination of the search. One is to only store states having several successors. The motivation is that the revisit of single-successor and deadlock states is cheap. It consists of reexecuting a sequence until reaching a branching state (which has several successors and which is therefore stored). To avoid entering cycles of single-successor states, the k^{th} state of these sequences is systematically stored.

In order to combine this reduction of [3] with our state caching mechanism we have to carefully reduce chains that are part of the TD-tree. First we notice that we do not have to worry about cycles of single-successor states: they will eventually reach a state of the TD-tree and all their states will immediately leave the TD-tree becoming candidates for replacement. To reduce chains of single-successor

states we associate with each open state s an attribute *ancestor* that points to the branching state that generated the first state of the chain which s belongs to, or is equal to **nil** if s is a branching state. The following piece of code specifies how this attribute is used to remove such chains from the TD-tree. It must be inserted after the generation of state s' from s at line 12.

$$s'.ancestor := \begin{cases} \textbf{if } |en(s')| = 1 \textbf{ and } s.ancestor = \textbf{ nil then } s \\ \textbf{if } |en(s')| = 1 \textbf{ and } s.ancestor \neq \textbf{ nil then } s.ancestor \\ \textbf{else nil} \end{cases}$$

if $s'.ancestor =$ **nil and** $s.ancestor \neq$ **nil then**
$a := s.ancestor$; $s'.pred := a$; $a.refs := a.refs + 1$; $unref(s)$

ancestor is first set when a single-successor state is generated from a branching state and then propagated later along all the states of the chain. If s' is a branching state and $s.ancestor$ points to some state, this means that we just left a chain. The reduction is done by directly linking s' to $s.ancestor$ and removing all the states of the chain from the TD-tree, by unreferencing s.

Fig. 4 shows an example of this reduction. Dotted arcs graphically represent the *pred* pointer of each state in the TD-tree. At Step 1, $s_1, \ldots, s_n$ form a reducible chain of single-successor states. All their *ancestor* field points to a, the branching state that generated the first state of the chain. State s_n then generates s that has several successors (see Step 2). Hence $s.ancestor = \textbf{nil} \neq s_n.ancestor$ and a reducible chain is detected. The consequence is to break the link from s to s_n and make $s.pred$ directly point to $s_n.ancestor = a$. The chain is then removed from the TD-tree by invoking $unref(s_n)$.

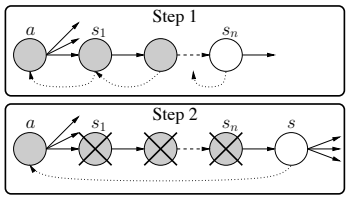


Fig. 4. Reduction of a single-successor state chain

4.2 Distributed Memory Algorithms

Most works in the field of distributed verification follow the seminal work of Stern and Dill [24]. Their algorithm partitions the state space upon several processes using a partition function mapping states to processes. Each process involved in the verification is responsible of storing and exploring the states it is assigned by this function. Whenever a process p generates a state owned by process $q \neq p$ it has to pack it into a message and send it to q that, upon reception, will store it in its open state and expand it later.

Our reduction is compatible with this algorithm. The only issue is raised by the *unref* procedure, used to maintain the TD-tree. This one has to access the ancestors of the unreferenced state — ancestors that may be located on another process — hence generating communications. A possible way to overcome this problem is to only call that procedure when memory becomes scarce. Processes then enter a garbage collection phase where they clean the TD-tree and delete states from memory. Thus, if the aggregated memory is large enough to solve the

problem without any reduction then there is no time overhead. This might also help to group states sent to the same owner and thereby reduce communication.

Another solution is to associate with each new state s received from another process a reference number of 2 (rather than 1) to ensure it will never leave the TD-tree (and that no communication will occur). It is then not necessary that the source state of the transition that generated the message keeps a reference to s. An example of a TD-tree (actually a forest) distributed over two processes can be seen in Fig 5. Dashed transitions are not part of the TD-tree. The reference counter of state s is set to 3 whereas it is closed and only has 2 successors. The reason is, it has been first discovered upon its reception by process 1. Hence, it will never leave the TD-tree guaranteeing that the cycle $t_0.t_1.t_2$ will be detected whatever order states are visited. Another difference with a "sequential" TD-tree is that the counter of s_0 is set to 1 instead of 2 since it only has one successor in the TD-tree on the same process. Hence, although s will remain in the memory of process 1, s_0 will eventually be allowed to leave the cache.

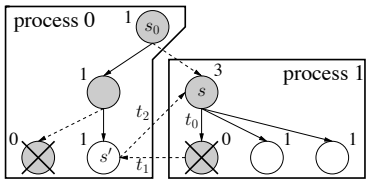


Fig. 5. A distributed TD-tree

Both solutions should benefit from a partitioning exhibiting few cross transitions linking states belonging to different processes. This will limit communications for the first solution and enhance the reduction in the second case.

4.3 Reductions Based on State Reconstruction

Our GSEA is also compatible with the reduction techniques proposed in [10] and [27]. Instead of keeping full state vectors in the closed and open sets, their principle is to represent a state s as a pair $(pred, e)$ where $pred$ is a pointer to the state s' that generated s during the search, and e is the event such that $succ(s', e) = s$. States can be reconstructed from this compressed representation by reexecuting the sequence of events that generated it. At a reasonable cost in time, it allows each state to be encoded by 12–16 bytes whatever the system being analyzed. This reduction fits nicely with the algorithm of this paper. The TD-tree can be compactly stored using this representation of states and, actually, both methods store with each state a pointer to its generating predecessor. The only states that have to be fully stored in memory are those who left the TD-tree since their generating predecessor may not be present anymore in memory.

4.4 The Sweep-Line Method

A sweep-line based algorithm alternates between exploration phases where states are visited and their successor(s) generated; with garbage collection phases where states are removed from memory. A key feature of the method is the progress measure ψ mapping states to (ordered) progress values. It is used to estimate "how far" states are from the initial states and guides the garbage collection

procedure: if the minimal progress value found in the set of open states is $\alpha_{min} = \min_{s \in \mathcal{O}} \psi(s)$ then all closed states s with $\psi(s) < \alpha_{min}$ can be deleted from memory. The underlying idea is that if the progress mapping is monotonic, i.e., all transitions (s, s') are such that $\psi(s) \leq \psi(s')$, then a visited state s with a progress $\psi(s) < \alpha_{min}$ will not be visited again. In [20] the method is extended to support progress measures with regress transitions, i.e., transitions (s, s') with $\psi(s) > \psi(s')$, that with the basic method of [6] cause the algorithm to not terminate. The principle of this extension is to mark destination of regress transitions as persistent to prevent the garbage collector from deleting them.

The sweep-line method can also be used in conjunction with our reduction. This stems from the fact that the algorithm of [20] is also an instance of the GSEA of Fig. 1 that keeps open states in a priority queue (priority being given to states having the lowest progression). However, one has to proceed carefully when combining both methods: the *unref* procedure of our algorithm may not put in the set $\mathcal{D}$ of candidates for replacement an unreferenced state (i.e., with $s.refs = 0$) that has been marked as persistent by the sweep-line reduction. Note that the predecessor of a persistent state may however be unreferenced.

Running the sweep-line algorithm in combination with our state caching algorithm causes the deletion of non-persistent states stored in the `TD`-tree. This means that we only store the parts of the `TD`-tree corresponding to the states determined to be in memory by the sweep-line method. The role of the `TD`-tree (which now becomes a forest) is to ensure termination of each of the phases of the sweep-line method, while the overall termination of the combined search is guaranteed by the persistent states stored by the sweep-line method.

5 Experiments

The technique proposed in this paper has been implemented in the ASAP verification tool [26]. ASAP can load models written in DVE [7], the input language of the DiVinE verification tool [2]. This allowed us to perform numerous experiments with models from the BEEM (BEnchmarks for Explicit Model checkers) database [22] although "Puzzles" and "Planning and scheduling" problems were not considered. These are mostly toy examples having few characteristics in common with real-life models. We performed two experiments, studying the performance of our reduction in combination with basic search algorithms for the first one; and with the sweep-line method for the second one. Due to lack of space, some data has been left out in this section but may be found in [9].

5.1 Experiment 1: State Caching with Basic Search Strategies

State caching is a rather unpredictable technique in the sense that its performance depends on a large range of parameters, e.g., the size and replacement strategy for the cache, the characteristics of the state space. It is usually hard to guess which configuration should be used before running the model checker. State caching must therefore be experimented with in a wide range of settings and with

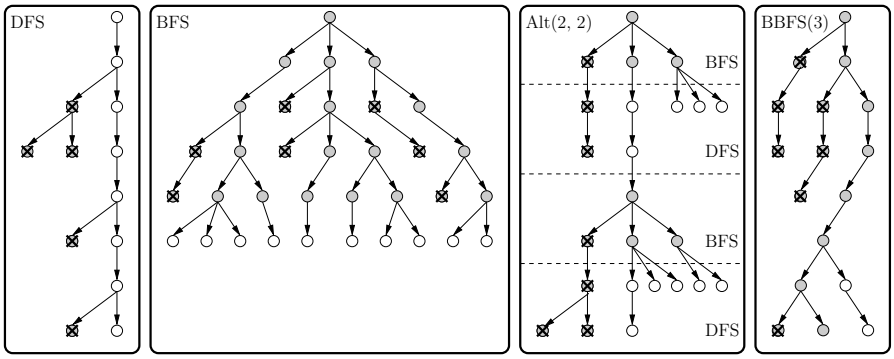


Fig. 6. Snapshot of the search trees with different search strategies.

many different state spaces in order to get a good insight into its behavior. In this experiment we performed more than 1,000,000 runs using different search algorithms, caching strategies, cache sizes, and state space reduction techniques.

Experimentation context

Search strategies. Several preliminary experiments revealed important variations of our state caching reduction when combined with DFS or BFS. Therefore, it seemed interesting to combine both algorithms to observe is such a combination could improve on pure breadth- or depth-first searches. Thus, in addition to DFS and BFS, we also experimented with the two following variations of these search strategies devised for the sake of our experimentation.

- *Alternation of breadth- and depth-first search.* This search strategy is parametrized by two integers b and d. It starts breadth-first on the b first levels. Then for the states at level b the search switches to a DFS until the depth $b + d$ is reached. At that point, the algorithm mutates back to a BFS and so on. This search will be denoted by Alt(b, d) in the following.
- *Bounded-width breadth-first search.* This search strategy, denoted BBFS(w), proceeds as a BFS except that the queue at each level may not contain more than w states (the width of the search). Open states of previous levels are kept in a stack to be expanded later when all next levels have been processed.

Note that DFS and BFS are special instances of these search strategies since it holds that DFS $\equiv$ BBFS(1) $\equiv$ Alt(0, ∞) and BFS $\equiv$ BBFS(∞) $\equiv$ Alt(∞, 0).

A snapshot of the search trees of different state spaces induced by these different strategies can be seen in Fig. 6. With DFS (left), all states outside the stack are candidates for replacement in the cache. With BFS (second left), any ancestor in the search tree of an open state must remain in the cache while others may be replaced. Open states, with algorithm Alt(2, 2) (second right) are those

still present in the stacks and queues used to perform "local" DFSs and BFSs. At last, the tree of BBFS(3) (right) is a BFS tree where each level can contain at most 3 states. Unlike BFS, some previous levels may contain open states as is the case here with the penultimate level.

All these algorithms are implemented using the generic template of Fig. 3 and parametrized by the type of the $\mathcal{O}$ data structure. BFS is implemented with a queue, DFS with a stack, BBFS(w) with a stack of arrays of size w, and Alt(b, d) with a stack containing single states for DFS levels and queues for BFS levels.

Cache replacement strategies. We implemented various strategies from the literature. The garbage collector can select states according to their in- and out-degree, their distance from the initial state (i.e., the depth at which the state has been generated), or in a purely random fashion. Stratified caching [11] has also been implemented. Due to a lack of space, we will not compare these strategies here. For DFS, the reader may consult the large body of work on that subject, e.g., [11,23]. With BFS, distance seems to be criterion with most impact.

State space reduction. Sleep-set reduction has been shown in [13] to drastically reduce the state revisits when using state caching. Rather than sleep-sets, we implemented the reduction of [4] which proposes a sleep-set like technique for both DFS and BFS that has two advantages over it: it does not require any memory overhead whereas the algorithm of [13] associates a set of transitions (the sleep set) with each closed and open set; and it is easier to implement.

We also implemented and experimented with the single-successor state chain reduction of [3] which is compatible with our algorithm as explained in Section 4.1. This reduction will be denoted by CR in the following.

Table 1 summarizes the different parameters and instances we experimented with. Unlike the reduction of [4] which was always turned on, the reduction of [3] was a parameter of each run. A run ended in one of three situations:

success The search could finish within allocated resources.
out of memory The cache was two small to contain the `TD`-tree.
out of time The algorithm visited more states than the specified threshold.

Experimental results

Experimental data is reported in Table 2. Due to space constraints, the table only contains data for 25 selected instances although the average on all instances experimented with is reported on the last line. Under each instance name, we give its number of states |S| and its average degree d as the ratio of transitions over states. All tests performed for each instance were divided into 8 groups according to the search algorithm they used (columns DFS, BFS, BBFS, and Alt) and according to whether or not they used the chain compression reduction (column CR). We then ordered, within each group, all successful runs first by

Table 1. Instances and parameters used during Experiment 1

Selected instances	135 instances with { 1,000, ..., 1,000,000 } states
Maximal state visits	$5 \cdot \|S\|$ (where $\|S\|$ is the state space size)
Cache size	{ 5, 10, 15, 20, ... } (as a % of the state space size) until a successful run could be found
Cache replacement strategy	60 caching strategies selected after experimentation with a small sample of 10 instances
Search strategy	DFS, BFS, BBFS(w) for $w \in \{4, 16, 256\}$, Alt(b, d) for $b, d \in \{1, 4, 8\}$
Reductions used	Reduction of [4] for all runs and CR for some runs

ascending cache size (i.e., memory) and then by ascending number of state visits (i.e., time). Each cell of the table contains data for the best run according to that order: the number of stored states, i.e., the cache size, (column S) and visited states (column V) both expressed as a percentage of the state space. Additionally, for algorithms Alt and BBFS, columns (b, d) and w specify the parameters of the search that the best run used.

State caching apparently provides a better memory reduction when coupled with DFS than with BFS. The size of the TD-tree, with BFS, is lower bounded by the width of the state space, i.e., the size of the largest level, which can be high for some models. The DFS stack can also contain a large proportion of the state space but the reduction of [4] not only reduces interleavings but also the stack size, whereas it is not helpful in BFS.

We still found some models for which BFS outperformed DFS with respect to both time and memory. This is the case for instances `extinction.3`, `firewire_tree.4` and `leader_election.4`. We will see later why BFS is to be preferred for these models.

Some instances like `cambridge.5` have typical characteristics that make state caching inadequate: many cycles and a high degree. This inevitably leads to a time explosion with DFS even using partial order reduction. BFS seems to be more resilient with respect to these instances. We found a couple of similar instances during our experiments.

Although state caching is generally more memory efficient when coupled with DFS, BFS still provides a notable advantage: it is less subject to a time explosion. Even in cases where the cache size was close to its lower bound, i.e., the maximal size of the TD-tree, the time brutally increased in very few cases. With BFS, the distribution of run failures is the following: 98% are "out of memory", and 2% are "out of time". With DFS, these percentages become respectively 61% and 39%. Even in cases where BFS "timed out", increasing the maximal number of state visits from $5 \cdot |S|$ to $20 \cdot |S|$ could turn all these runs into successes. This is, from the user point of view, an appreciable property. With DFS, when the user selects a small cache size, and the search lasts for long, he/she can not know if it is due to a high rate of state revisits or if it is because the state space is very large and state caching is efficient. This situation never occurred with BFS.

Table 2. Summary of data for Experiment 1

Model	CR	DFS		BFS		BBFS			Alt		
		S	V	S	V	S	V	w	S	V	(b,d)
`at.2`	no	35	228	40	130	30	369	4	30	219	(8,4)
|S|=49,443 d=2.9	yes	30	428	40	130	30	373	4	30	222	(8,4)
`bakery.4`	no	20	271	25	216	25	173	4	20	185	(8,4)
|S|=157,003 d=2.6	yes	20	271	25	149	25	168	4	20	185	(8,4)
`bopdp.2`	no	15	215	30	149	15	246	4	15	213	(1,8)
|S|=25,685 d=2.8	yes	10	437	25	162	10	398	4	10	406	(1,8)
`brp.3`	no	5	235	15	112	5	163	256	5	160	(8,8)
|S|=996,627 d=2.0	yes	5	152	10	116	5	168	256	5	153	(8,8)
`brp2.5`	no	5	141	30	112	5	130	4	5	135	(8,1)
|S|=298,111 d=1.4	yes	5	140	20	103	5	131	4	5	137	(8,1)
`cambridge.5`	no	45	285	35	121	45	203	4	45	232	(8,4)
|S|=698,912 d=4.5	yes	45	287	35	120	45	200	16	45	233	(8,8)
`collision.3`	no	10	484	30	119	10	487	16	10	239	(8,1)
|S|=434,530 d=2.3	yes	10	338	25	169	10	412	16	10	286	(8,1)
`extinction.3`	no	10	185	10	100	10	186	4	10	176	(4,4)
|S|=751,930 d=3.5	yes	10	184	10	100	10	187	4	10	176	(4,4)
`firewire_link.7`	no	5	327	25	101	5	278	256	5	214	(8,1)
|S|=399,598 d=2.7	yes	5	321	20	101	5	348	256	5	230	(8,1)
`firewire_tree.4`	no	10	337	10	100	15	190	4	10	313	(4,4)
|S|=169,992 d=3.7	yes	10	345	10	100	15	191	4	10	311	(4,4)
`gear.2`	no	15	109	10	100	10	100	256	10	102	(4,4)
|S|=16,689 d=1.3	yes	15	106	5	102	5	101	256	10	101	(4,4)
`iprotocol.2`	no	5	359	20	132	5	250	4	5	296	(8,1)
|S|=29,994 d=3.3	yes	5	364	20	132	5	245	16	5	294	(1,1)
`lamport_nonatomic.3`	no	45	398	50	120	45	257	4	45	237	(8,8)
|S|=36,983 d=3.3	yes	45	366	50	119	45	260	4	45	232	(8,1)
`leader_election.4`	no	15	450	10	100	15	456	16	15	273	(4,1)
|S|=746,240 d=5.0	yes	15	316	10	100	15	453	16	15	275	(4,1)
`lifts.6`	no	5	123	15	123	5	125	4	5	124	(1,1)
|S|=333,649 d=2.1	yes	5	124	15	112	5	132	4	5	124	(1,1)
`lup.2`	no	30	478	30	142	40	369	4	15	429	(8,8)
|S|=495,720 d=1.8	yes	30	294	30	142	35	324	4	15	428	(8,8)
`needham.3`	no	5	412	30	100	5	409	256	5	435	(1,1)
|S|=206,925 d=2.7	yes	5	415	30	100	5	417	256	5	435	(1,1)
`peterson.3`	no	25	345	35	142	25	374	4	25	329	(1,8)
|S|=170,156 d=3.1	yes	25	331	35	136	25	356	4	25	320	(8,8)
`pgm_protocol.7`	no	10	152	10	106	5	360	16	5	493	(8,1)
|S|=322,585 d=2.5	yes	10	152	5	112	5	392	4	10	145	(4,1)
`plc.2`	no	5	100	30	100	10	101	4	5	100	(1,8)
|S|=130,777 d=1.6	yes	5	100	10	100	5	113	4	5	101	(8,8)
`production_cell.4`	no	10	176	10	100	10	162	4	10	157	(8,1)
|S|=340,685 d=2.8	yes	10	175	10	100	10	162	4	10	158	(8,1)
`rether.3`	no	30	118	40	104	35	132	256	25	113	(8,1)
|S|=305,334 d=1.0	yes	25	152	15	111	15	117	256	15	128	(8,1)
`synapse.6`	no	5	148	30	111	5	133	16	5	129	(8,8)
|S|=625,175 d=1.9	yes	5	118	25	296	5	136	16	5	129	(8,8)
`telephony.3`	no	30	394	50	124	25	482	256	25	368	(8,1)
|S|=765,379 d=4.1	yes	25	491	50	123	25	470	16	25	388	(8,1)
`train-gate.5`	no	10	114	20	100	10	110	4	10	102	(4,1)
|S|=803,458 d=2.1	yes	10	114	20	100	10	105	256	10	102	(4,1)
Average on 135 models	no	18.5	259	30.1	131	19.4	236		16.4	239	
	yes	17.7	251	26.3	143	17.4	241		15.5	249	

In general, we found out that BFS is far less sensitive to the caching strategy than DFS. For a specific cache size, it was not unusual, with DFS, that only one or two replacement strategies could make the run successful. Whereas, with BFS, once a successful run could be found with some strategy, it usually meant that many other runs using different strategies (and with the same cache size) could also terminate successfully. On an average made on all models we calculated that, with DFS and for the smallest cache size for which at least a run turned out to be successful, only 25% of all runs were successful. With BFS, this same percentage goes up to 66%.

These observations are in line with a remark made in [21], p. 226: "*Compared to BFSWS* [BFS With Snapshots]*, the success of DFS setups differs a lot from one case to another.*" One of our conclusions is indeed that BFS has the advantage to exhibit more predictable performance.

The effect of reduction CR is more evident in the case of BFS. The cache size could be further reduced by an average of 4–5% for a marginal cost in time, whereas with DFS the reduction achieved is negligible. On some instances, BFS could, with the help of this reduction, significantly outperform DFS with respect to memory consumption. This is for instance the case for `rether.3` which has a majority of single-successor states.

Lastly, we notice that Alt and BBFS sometimes cumulate the advantages of both BFS (w.r.t. time) and DFS (w.r.t. memory) and perform better than these, e.g., `at.2`, `bakery.4`, `firewire_link.7` and `lup.2`. Search Alt also seems to be more successful than BBFS. It could on average reduce the cache size from 17 to 15% of the state space, when compared to DFS.

Influence of the State Space Structure in BFS. We previously noticed that the width of the graph is a lower bound of the `TD`-tree in BFS. More generally, there is a clear link between the shape of the BFS level graph and the memory reduction in BFS. Figure 7 depicts this graph for several instances that are of particular interest to illustrate our purpose. For each BFS level, the value plotted specifies the number of states (as a percentage of the full state space) belonging to a specific level. For instances `telephony.3` and `synapse.6`, a large proportion of states is gathered on a few neighbor levels. The algorithm will thus have to store most states of these levels and it is not surprising to observe on Table 2 that state caching is not efficient in these cases. Instances `pgm_protocol.7` or `gear.2` have the opposite characteristic: the distribution of states upon BFS levels is rather homogeneous and there is no sequence of neighbor levels containing many states. This explains the good memory reduction observed with BFS on these examples.

With DFS, the time increase is closely related to the average degree of the state space. This factor has a lesser impact with BFS: the proportion of backward transitions[1] plays a more important role. Indeed, the search order of BFS implies the destination state of a forward transition to necessarily be in the open set. Hence, only backward transitions may be followed by state revisits. Instances

[1] (s, s') is a backward transition if the BFS levels d and d' of s and s' are such that $d \geq d'$. The length of (s, s') is the difference $d - d'$.

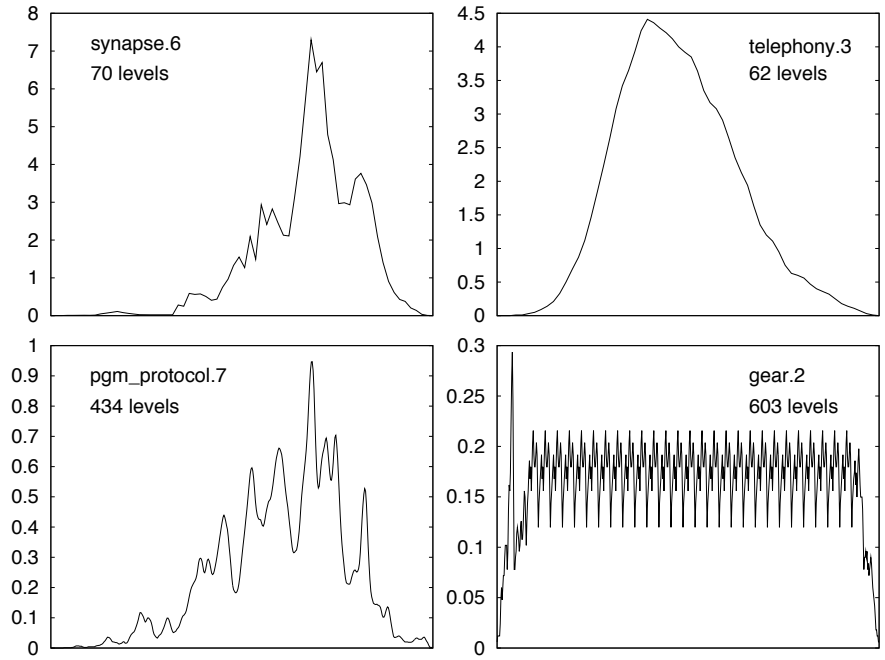


Fig. 7. BFS level graph of some instances.

`extinction.3` and `firewire_tree.4` have rather high (3–5) degrees but few or no backward transitions, which led to very few state revisits with BFS; whereas using DFS we often experienced a time explosion with these instances. The fact that state spaces of real-world problems often have few backward transitions [25] may explain the small state revisit factors usually observed with BFS.

5.2 Experiment 2: State Caching with the Sweep-Line Method

In a second experiment we studied how our state caching algorithm combines in practice with the sweep-line method, as described in Section 4.4. We analyzed the same instances used in the first experiment. We automatically derived from their full state spaces several progress measures identified by a level ranging from 0 to 5. The higher the level, the more precise the progress mapping and the more aggressive the reduction. Our progress measures are abstractions in that they project states to some components of the underlying system. At level 0, the progress mapping is guaranteed to not generate any regress transition since we only consider components having a monotonic progression, e.g., an increasing sequence number; and as the level progresses, the mapping is refined by including in it more and more components in order to multiply the different progress values possible and increase the potential of the reduction (while also introducing more regress transitions). Progress values are then fixed-size vectors

Table 3. Summary of data for Experiment 2

Model	|S|	PM	SL				SL + SC	
			S	V	P	L	S	V
bopdp.2	25,685	3	52.4	231	0.5	51.9	20.4	498
		4	26.8	242	0.8	26.0	14.9	480
		5	13.7	259	1.9	8.3	12.7	482
brp.3	996,627	2	13.3	145	0.1	8.3	4.9	203
		3	10.6	126	0.5	2.9	9.7	154
		4	10.0	122	2.4	0.3	9.9	120
		5	8.1	124	3.8	0.1	8.2	119
collision.3	434,530	4	24.1	100	0.0	24.1	13.5	482
		5	21.1	232	9.0	12.2	16.0	337
iprotocol.1	6,814	5	50.4	100	0.0	47.3	21.0	416
lifts.4	112,792	5	33.3	100	0.0	33.3	8.5	489
needham.3	206,925	5	20.9	100	0.0	0.1	21.0	101
pgm_protocol.3	195,015	3	2.4	144	1.1	0.0	2.1	129
		4	3.2	110	3.1	0.0	3.1	110
		5	9.0	110	8.9	0.0	9.0	108
plc.2	130,777	3	1.8	100	0.0	1.7	0.6	102
		4	1.2	105	1.2	0.0	1.3	105
		5	1.2	105	1.2	0.0	1.3	105
production_cell.4	340,685	4	27.7	100	0.0	22.7	14.1	485
		5	11.1	100	2.0	6.9	5.5	438
rether.3	305,334	4	25.3	161	3.9	21.8	5.0	200
		5	6.3	152	6.0	0.6	6.1	144
synapse.6	625,175	3	26.6	100	0.0	26.6	7.4	477
		4	27.4	316	2.7	16.4	17.2	477
		5	18.1	150	10.8	2.9	17.4	136

of (selected) components that can be compared via a lexical ordering. Each level corresponds to an estimation of the upper bound of the proportion of regress transitions: level 0 → no regress transition, level 1 → at most 1% of regress transitions, level 2 → 2%, level 3 → 5%, level 4 → 10% and level 5 →20%.

Table 3 summarizes the data collected during this experiment for some selected instances and progress measures. For each instance (column Model) and progress measure (of which the level appears in column PM) we performed a first run with the sweep-line algorithm of [20] (column SL) and then several runs with the same algorithm extended with our state caching reduction (column SL + SC) using different cache sizes. We used a variation of the stratified caching strategy [11] that revealed to be the most efficient one during the first experiment. As in the first experiment we only kept, for algorithm SL + SC, the "best" run, that is, the one that used the smallest amount of memory with a state visit factor less than 5. For these two runs columns S and V provide the number of stored and visited states. Additionally, for algorithm SL the table gives in column P the number of persistent states at the end of the search ; and in column L the size of the largest class of states sharing the same progress value and present

in memory at the same time (before being garbage collected). All these values are expressed as a percentage of the state space size given in column |S|. The data of the run that provided the best memory reduction for a model has been highlighted using a gray background.

To have a better understanding of these results it seems necessary to briefly recall the principle of algorithm SL. At each step the algorithm explores a class of states sharing a common progress value ψ. All their successors are put in the priority queue implementing the open set, and once this expansion step is finished, i.e., no state with progress value ψ is in the queue, the algorithm deletes from memory all expanded states. It then reiterates this process with the next progress value found in the queue until it is empty. The size of the largest class of states with the same progress value given in column L is thus a lower bound on the memory consumption of the algorithm. By implementing state caching on top of this algorithm we can hope to reduce only the class of states the algorithm is currently working on. Indeed if we note ψ the progress value of this class then all the states with a progress value $\psi' < \psi$ have been garbage collected by the sweep-line reduction and states with a progress value $\psi' > \psi$ present in memory are necessarily in the open set and hence can not be removed from the cache. The potential of the state caching reduction is thus given by the ratio $\frac{\mathsf{L}}{\mathsf{S}}$ (in column SL). It is therefore not surprising that the gain of using state caching depends on the size L of this largest class. Instances `brp.3`, `plc.2` and `pgm_protocol.3` are the typical examples of models for which the sweep-line method is well suited: they have long state spaces with a clear progression which enables progress mappings to be defined that divide the state space into many small classes. Hence, the sweep-line method used solely can provide a very good reduction that can not be significantly enhanced by state caching.

Increasing the PM level has three noteworthy consequences.

First, for most models, the peak number of states stored by SL decreases although this is not always the case: by refining the progress mapping we usually increase the proportion of regress transitions generating persistent states that will never be garbage collected. Instance `pgm_protocol.3` is a good illustration.

Second, by multiplying progress values we naturally decrease the effect of state caching since, as we previously saw, state caching is helpful to reduce classes of states sharing the same progress value. We indeed observe that the values in column L decrease as we increase the PM level, with the consequence that numbers converge to the same values with both algorithms.

A last observation is that, regarding the number of states stored, SL and SL + SC often follow opposite behaviors. For instances `brp.3`, `collision.3`, `plc.2`, `rether.3` and `synapse.6`, SL consumes less memory when we refine the progress measure whereas SL + SC needs a larger cache. More generally, we observed that the average ratio of the number of states stored by SL over the number of states stored by SL + SC is maximal at level 3 (2.93) and then decreases to 1.97 at level 5. This is an interesting property from a user perspective. This indeed means that he/she does not necessarily have to provide a very fine tuned progress mapping to the model checker. In many cases, a basic mapping that extracts

some monotonic component(s) from the model will be sufficient: the state caching reduction will then fully complement the sweep-line reduction flaws and provide an even better reduction compared to a very precise progress mapping that will cancel the benefits of state caching.

6 Conclusion

In this paper we have proposed an extension of state caching to general state exploring algorithms. Termination is guaranteed by maintaining, as the search progresses, a tree rooted in the initial state that covers all open states. This ensures that any cycle will eventually reach a state of the tree. Closed states that have left the tree can therefore be deleted from memory without endangering the termination of the algorithm. Extensive experimentation with models from the BEEM database has revealed that state caching can reduce the memory requirements of a BFS by a factor of approximately 4. Although this is usually not as good as the reduction observed with DFS, BFS offers an advantage in that the reduction comes almost for free: the average increase in run-time that we observed with BFS was usually around 30–40%, and we observed very few cases of time explosion, whereas this is quite common with DFS even when using partial order reduction. Combining both search strategies can also bring advantages: in some cases, we found that state caching coupled with a combination of BFS and DFS could bring the same (or an even better) reduction as with DFS while limiting the run-time explosion that could occur with this one. Last but not least, our experiments revealed that our reduction can also enhance the sweep-line method as the algorithm it relies on is also an instance of the GSEA.

The algorithm we proposed is fully language-independent in that it only relies on a successor function to explore the state space. Nevertheless, it should be worth experimenting it with other formalisms than DVE and especially Colored Petri Nets (CPN) [19]. It is possible that the high level constructs provided by CPN to express the successor function may impact the structural characteristics of state spaces that, as previous works on state caching and our own experiments revealed, are tightly linked to the performance of state caching. Such an experimentation is therefore the next research direction we focus on.

References

1. T. Bao and M. Jones. Time-Efficient Model Checking with Magnetic Disk. In *TACAS'2005*, vol. 3440 of *LNCS*, pp. 526–540. Springer, 2005.
2. J. Barnat, L. Brim, I. Cerná, P. Moravec, P. Rockai, and P. Simecek. DiVinE - A Tool for Distributed Verification. In T. Ball and R.B. Jones, editors, *CAV'2006*, vol. 4144 of *LNCS*, pp. 278–281. Springer, 2006.
3. G. Behrmann, K.G. Larsen, and R. Pelánek. To Store or Not to Store. In *CAV'2003*, vol. 2725 of *LNCS*, pp. 433–445. Springer, 2003.
4. D. Bosnacki, E. Elkind, B. Genest, and D. Peled. On Commutativity Based Edge Lean Search. In *ICALP'2007*, vol. 4596 of *LNCS*, pp. 158–170. Springer, 2007.

5. D. Bosnacki, S. Leue, and A. Lluch-Lafuente. Partial-Order Reduction for General State Exploring Algorithms. In *SPIN'2006*, vol. 3925 of *LNCS*, pp. 271–287. Springer, 2006.
6. S. Christensen, L.M. Kristensen, and T. Mailund. A Sweep-Line Method for State Space Exploration. In *TACAS'2001*, vol. 2031 of *LNCS*, pp. 450–464. Springer, 2001.
7. DVE Language. `http://divine.fi.muni.cz/page.php?page=language`.
8. S. Edelkamp, S. Leue, and A. Lluch-Lafuente. Directed Explicit-State Model Checking in the Validation of Communication Protocols. *STTT*, 5:247–267, 2004.
9. S. Evangelista and L.M. Kristensen. Search-Order Independent State Caching. Technical report, 2009. `http://daimi.au.dk/~evangeli/doc/caching.pdf`.
10. S. Evangelista and J.-F. Pradat-Peyre. Memory Efficient State Space Storage in Explicit Software Model Checking. In *SPIN'2005*, vol. 3639 of *LNCS*, pp. 43–57. Springer, 2005.
11. J. Geldenhuys. State Caching Reconsidered. In *SPIN'2004*, vol. 2989 of *LNCS*, pp. 23–38. Springer, 2004.
12. P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems*, vol. 1032 of *LNCS*. Springer, 1996.
13. P. Godefroid, G.J. Holzmann, and D. Pirottin. State-Space Caching Revisited. In *CAV'1992*, vol. 663 of *LNCS*, pp. 178–191. Springer, 1992.
14. G.J. Holzmann. Tracing Protocols. *AT&T Technical J.*, 64(10):2413–2434, 1985.
15. G.J. Holzmann. Automated Protocol Validation in *Argos:* Assertion Proving and Scatter Searching. *IEEE Trans. Software Eng.*, 13(6):683–696, 1987.
16. G.J. Holzmann. State Compression in Spin: Recursive Indexing and Compression Training Runs. In *SPIN'1997*, 1997.
17. C. Jard and T. Jéron. On-Line Model Checking for Finite Linear Temporal Logic Specifications. In *Automatic Verification Methods for Finite State Systems*, vol. 407 of *LNCS*, pp. 189–196. Springer, 1989.
18. C. Jard and T. Jéron. Bounded-memory Algorithms for Verification On-the-fly. In *CAV'1991*, vol. 575 of *LNCS*, pp. 192–202. Springer, 1991.
19. K. Jensen and L.M. Kristensen. *Coloured Petri Nets — Modeling and Validation of Concurrent Systems*. Springer, 2009.
20. L.M. Kristensen and T. Mailund. A Generalised Sweep-Line Method for Safety Properties. In *FME'2002*, vol. 2391 of *LNCS*, pp. 549–567. Springer, 2002.
21. R. Mateescu and A. Wijs. Hierarchical Adapatative State Space Caching Based on Level Sampling. In *TACAS'2009*, vol. 5505 of *LNCS*, pp. 215–229. Springer, 2009.
22. R. Pelánek. BEEM: Benchmarks for Explicit Model Checkers. In *SPIN'2007*, vol. 4595 of *LNCS*, pp. 263–267. Springer, 2007.
23. R. Pelánek, V. Rosecký, and J. Sedenka. Evaluation of State Caching and State Compression Techniques. Technical report, Masaryk University, Brno, 2008.
24. U. Stern and D. L. Dill. Parallelizing the Murphi Verifier. In *CAV'1997*, vol. 1254 of *LNCS*, pp. 256–278. Springer, 1997.
25. E. Tronci, G. Della Penna, B. Intrigila, and M. Venturini Zilli. Exploiting Transition Locality in Automatic Verification. In *CHARME'2001*, vol. 2144 of *LNCS*, pp. 259–274. Springer, 2001.
26. M. Westergaard, S. Evangelista, and L.M. Kristensen. ASAP: An Extensible Platform for State Space Analysis. In *ATPN'2009*, vol. 5606 of *LNCS*, pp. 303–312. Springer, 2009.
27. M. Westergaard, L.M. Kristensen, G. Stølting Brodal, and L. Arge. The Comback Method - Extending Hash Compaction with Backtracking. In *ATPN'2007*, vol. 4546 of *LNCS*, pp. 445–464. Springer, 2007.

On Extending the Sweep-Line for Language Equivalence Checking

Guy Edward Gallasch

Computer Systems Engineering Centre
University of South Australia
Mawson Lakes Campus, SA, 5095, AUSTRALIA
Email: guy.gallasch@unisa.edu.au

Abstract. A vital step in formal protocol verification is the comparison of the externally visible behaviour exhibited by a protocol (the *protocol language*) with that allowed by its service (the *service language*). Sometimes, it is sufficient to check for *language inclusion*, which verifies that the protocol language is contained within the service language, i.e. the protocol does not exhibit behaviour that is not in its service. This is a specific instance of model checking a safety property, where the service language becomes the safety property to check. However, it is important to know also if all behaviour allowed by the service is exhibited by the protocol. Thus, the problem becomes one of checking *language equivalence*, which cannot be formulated as safety property. The Sweep-line method, designed to alleviate the problem of state explosion, was recently extended to allow on-the-fly checking of language inclusion (safety properties). This paper presents the genesis of a further extension to allow on-the-fly verification of language equivalence.
Keywords: Language Equivalence, Sweep-line Method, State Space Explosion.

1 Introduction

This paper is motivated by the area of *protocol verification* [3]. Protocols should satisfy a set of properties that encapsulate their desired behaviour, including the service that the protocol should provide to its users. This is known as a service specification. Protocol verification involves proving that a protocol does satisfy its specification. [3] provides a protocol verification methodology that uses Coloured Petri Nets [16] for the specification of protocol behaviour. The 'verification' part of protocol verification can be broken into two main activities. One involves checking that a number of general properties hold for the protocol, such as verifying the absence of undesired terminal states, dead transitions and livelocks. The other, which is of interest to us here, involves checking that all sequences of user-observable actions of the protocol conform to those given in the service specification. Here, the user-observable actions are called *service primitives*, the set of allowable sequences of service primitives as given in the service specification is called the *service language*, $\mathcal{L}_S$, and the set of sequences of service primitives exhibited by the protocol is called the *protocol language*, $\mathcal{L}_P$.

Comparing the protocol and service languages implies a notion of *language equivalence*. As an aside, as pointed out in [3], many notions of equivalence exist (van Glabbeek [13] pointed out 155 in 1993), such as Valmari's Chaos-Free Failures Divergences (CFFD) equivalence [24], that take into account branching behaviour, which allow deadlock and livelock properties to be preserved. However, these properties can be determined from the protocol's Reachability Graph, without reference to another specification. This is normally done first, as they are fundamental to good protocol design. Once these basic properties are proved, then we consider the more specific property of correct sequencing of service primitives. Hence language equivalence is sufficient, as it is precisely the property of interest.

We would like to ensure that both languages are equivalent, i.e. that they capture the same set of sequences of user-observable events. This implies that all user-observable behaviour exhibited by the protocol is allowed by the service ($\mathcal{L}_P \subseteq \mathcal{L}_S$) and that all behaviour defined by the service is actually implemented in the protocol ($\mathcal{L}_S \subseteq \mathcal{L}_P$). There are situations in which the weaker notion of *Language Inclusion* (that $\mathcal{L}_P \subseteq \mathcal{L}_S$) is sufficient for protocol verification, e.g. the Internet Open Trading Protocol [4] implemented an acceptable subset of its service [22, 23], however the key in that situation was knowing what sequences in the service language were missing from the protocol language, and determining what constituted an acceptable subset of behaviour for the protocol.

Languages can be represented by automata. A common approach (e.g. [14,22]) for checking $\mathcal{L}_P = \mathcal{L}_S$ has involved representing the protocol and service languages as deterministic Finite State Automata (FSA) [1] and using language comparison tools such as the FSM Library of AT&T [8]. The protocol language FSA and service language FSA must both be ϵ-free and deterministic in order to use the `fsmequiv` tool of [8]. We denote these $DFSA_P$ and $DFSA_S$ respectively. $DFSA_P$ can be extracted from the Reachability Graph (RG) of the CPN model of the protocol, by firstly defining a mapping from binding elements (arc labels) to service primitives or epsilon (ϵ, the empty move), defining an initial state (usually the initial state of the CPN model), and defining halt states (usually the terminal markings, as a minimum) [3]. ϵ removal and determinisation procedures [1,15] are then used to obtain a deterministic, ϵ-free representation of the protocol language. Unless the service specification is exceedingly simple, a CPN is often constructed in order to produce $DFSA_S$ in the same way.

The downside of this approach is that it requires the complete protocol RG to be generated and in memory at one time. This can be problematic due to the well-known *state explosion* problem [24]. Fortunately, it is usually possible to perform determinisation of automata on-the-fly, e.g. [17, 18] and is certainly possible for Finite State Automata.

The Sweep-line method [6, 20] is a state space exploration technique that uses a notion of *progress* within the system being modelled. The successful application of this method usually involves defining a sensible *progress mapping*, $\psi : M \rightarrow V$, a mapping from states to *progress values*, V, that reflects the notion of progress within the system being modelled. The Sweep-line's algorithm explores states in a least-progress-first manner. The exploration algorithm uses the progress mapping to determine when it is likely that particular states (those with smaller progress values) will not be visited again in the future, and hence can usually be safely deleted from memory. However, it is possible that the notion of progress captured by the progress mapping may be violated, i.e. an arc in the reachability graph moves the system from a state with a higher progress value to one with a lower progress value. This phenomenon is known as *regress*, and the offending edge as a *regress edge*. The generalised algorithm given in [20] is able to cope with this situation, at the cost of potentially exploring parts of the state space more than once.

In [12] we extended the Sweep-line method with the ability to check safety properties on-the-fly by performing an on-the-fly parallel composition of $DFSA_S$ (known a priori) and the protocol RG, by mapping the RG to a FSA on-the-fly. (This was originally presented in [11] in the context of protocol verification, for on-the-fly language inclusion checking.) In this paper we present some preliminary steps toward extending the Sweep-line method for language equivalence checking. Primarily, the contribution of this paper is to show how to combine the Sweep-line method with on-the-fly determinisation, which may also have significance generally in the model checking world beyond our intial application of language equivalence checking. Section 2 gives some further motivation and background details of the extension of Sweep-line

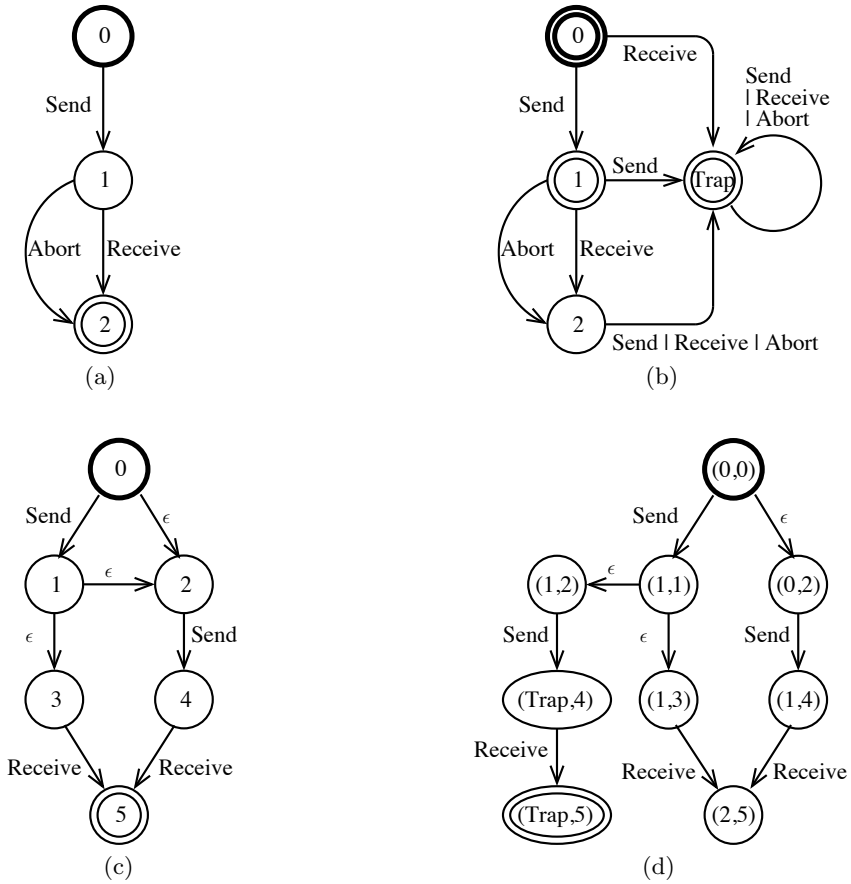


Fig. 1. Finite State Automata depicting (a) $DFSA_S$, representing the service language, $\mathcal{L}_S$; (b) $\overline{DFSA_S}$, representing the complement of $\mathcal{L}_S$; (c) FSA_P, representing the corresponding protocol language, $\mathcal{L}_P$; and (d) the parallel composition of $\overline{DFSA_S}$ and FSA_P.

to checking safety properties on-the-fly. Section 3 states our preliminary ideas for extending this to language equivalence. Finally, Section 4 outlines areas of future work.

Due to length restrictions, we assume that the reader is familiar with the workings of the Sweep-line method. Details can be found in [6, 20].

2 Language Inclusion Checking with the Sweep-line Method

Model checking of safety properties is well known, e.g. [2,7,21,25]. All of these techniques use automata to represent both the system and the safety property to check. Safety properties deal only with finite behaviours [24], hence it was reasoned in [12] that Finite State Automata (FSAs) [1] were adequate for this purpose. To verify a safety property on-the-fly with the Sweep-line method, the method adopted in [12] was to compose [5,19] the FSA representations of the system and the complement of the safety property.

For the sake of continuity with the ideas presented in Section 3, we shall use language inclusion checking as an example of on-the-fly checking of a safety property. We give such an example in Figure 1. Figure 1 (a) presents a simple service language comprising two sequences:

$\mathcal{L}_S = \{$Send Receive, Send Abort$\}$. The deterministic FSA in Figure 1 (a) is $DFSA_S$ and the set of service primitives is Send, Receive and Abort. The initial state is indicated by a bold circle and the halt state by a double circle. Figure 1 (c) presents a FSA produced by applying an appropriate mapping to the binding elements of the RG of an (erroneous) protocol, and identifying the initial state (state 0) and halt states (in this case state 5). We denote this FSA FSA_P, representing the protocol language $\mathcal{L}_P$. Note that FSA_P does not need to be deterministic for language inclusion checking.

To check that language inclusion holds, we can check that no sequences in the protocol language are contained in the complement of the service language, i.e. $\overline{\mathcal{L}_S} \cap \mathcal{L}_P = \emptyset$. Hence, we derive the the FSA representation of the complement of the service language, $\overline{DFSA_S}$, as shown in Figure 1 (b). The process of complementation relies on $DFSA_S$ being deterministic. This complement is then composed (parallel composition) with FSA_P (Figure 1 (c)) with the result shown in Figure 1 (d), to obtain $\overline{\mathcal{L}_S} \cap \mathcal{L}_P$. Essentially, the Sweep-line in [12] sweeps the FSA in Figure 1 (d), so it is possible (likely) that not all of Figure 1 (d) will be in memory at one time.

The parallel composition has an accepting state containing the Trap state of the service, indicating erroneous behaviour on the part of the protocol. Hence $\overline{\mathcal{L}_S} \cap \mathcal{L}_P \neq \emptyset$ and we have detected that our protocol is erroneous. However, we have only checked $\mathcal{L}_P \subseteq \mathcal{L}_S$. Note that the protocol does not exhibit the behaviour corresponding to the sequence Send Abort, so the protocol may be erroneous in this sense also. This cannot be detected by checking language inclusion only.

3 Toward On-the-Fly Language Equivalence Checking with the Sweep-line Method

One solution to performing the additional check of $\mathcal{L}_S \subseteq \mathcal{L}_P$ to verify language equivalence is to compose the service language FSA with the complement of the deterministic protocol language FSA. However, obtaining $\overline{DFSA_P}$ from the RG of the protocol is not straightforward, as often the RG is prohibitively large (hence our desire to apply the Sweep-line method). We can overcome this by extending the Sweep-line method with on-the-fly determinisation and complementation capabilities.

As a first step, we consider the situation in which progress increases monotonically, i.e. there are no regress edges. The ability to cope with regress is discussed in Section 3.3. The approach we propose below is based on the Sweep-line exploration algorithm having explored all states with a given (minimum) progress value and this fragment of RG having been mapped to a FSA (easily done) before the process of determinisation, complementation and parallel composition are undertaken for that fragment of FSA. Mapping the fragment of RG to a FSA, complementation and parallel composition are not discussed below, as they are straightforward processes that can be applied on a state-by-state and arc-by-arc basis once we know that the outgoing arcs of the states involved are deterministic. We hence limit our discussion to the most critical step - on-the-fly determinisation. [1] advocates a two-step method for determinisation: removing empty cycles and empty moves, then eliminating the remaining nondeterminism. This technique is much more suited to on-the-fly determinisation than the subset (power set) construction technique of [15], which requires power sets of states to be generated - a procedure that is counterproductive to state space reduction. However, we discuss the possibility of applying subset construction with *lazy subset evaluation* in Section 4. We illustrate our proposal with a series of RG snapshots.

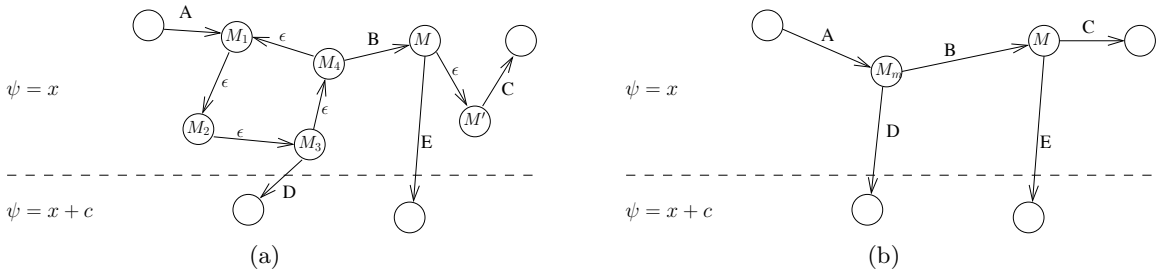


Fig. 2. (a) Epsilon cycles and empty moves within one progress level; and (b) after epsilon removal.

3.1 Removing Empty Moves

Figure 2 (a) presents two scenarios that are straightforwardly dealt with by the algorithms in [1]. On the left is an empty cycle (states M_1, M_2, M_3 and M_4) and on the right is an empty move from M to M' that is not part of a cycle. As is shown in Figure 2 (b), the states forming the cycle are merged into M_m and the other empty move is eliminated by adding appropriately labelled arcs from M to the successors of M' and deleting M'. Although not explicitly pictured in Figure 2, the algorithms in [1] cater for chains of ϵ-moves of arbitrary length, and for situations in which a merged state has predecessors that were not directly involved in the ϵ-removal process. The remaining examples in this paper can be extended in a similar way.

Figure 3 (a) illustrates two situations in which empty moves involve states in two 'progress levels'. On the left is a situation in which the removal of the empty move originating at state M_1 results in a new arc being added that spans two progress values. Given that the successors of M_1' would have already been calculated, M_1'' will be in memory, and hence removal of this ϵ-move causes no problems. On the right is a less benign situation: the empty move from state M_2 corresponds to an increase in progress. In the normal course of events, the Sweep-line method should delete all states with progress value x (once all such states have been explored). However, this is not possible, as at this stage we cannot yet eliminate this ϵ move as we do not know the successors of M_2'. We would like to retain M_2 in memory until the successors of M_2' have been calculated. To do this, we introduce the notion of *transient* states: states that should not be deleted immediately. Unlike the notion of a *persistent* state introduced in [20] to cope with regress edges, a transient state need only be retained in memory until the ϵ-move originating from it can be eliminated - it can be safely deleted at some point in the future. The result of both of these situations is shown in Figure 3 (b), where the transient state is indicated by a shaded circle. Transient states can be used for multiple successive ϵ-moves in a chain spanning any number of progress levels.

3.2 Determinisation of Non-Empty Moves

When all ϵ-moves have been eliminated, with the exception of situations such as that depicted at the right of Figure 3 (a), determinisation of the remaining non-ϵ moves can proceed. Following the procedures of [1], this is done by the merging of states. Figure 4 (a) shows two situations that are handled straightforwardly by these procedures. On the left is the situation where two states, M_1' and M_1'', are to be merged within the same progress level as the originator, M_1. On the right is the situation where two states, M_2' and M_2'', are to be

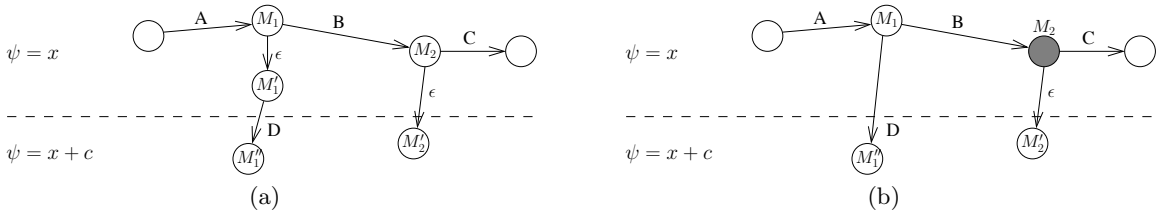


Fig. 3. (a) Epsilon moves involving two progress levels; and (b) removal of the epsilon move from M_1 to M_1', and marking M_2 as *transient*.

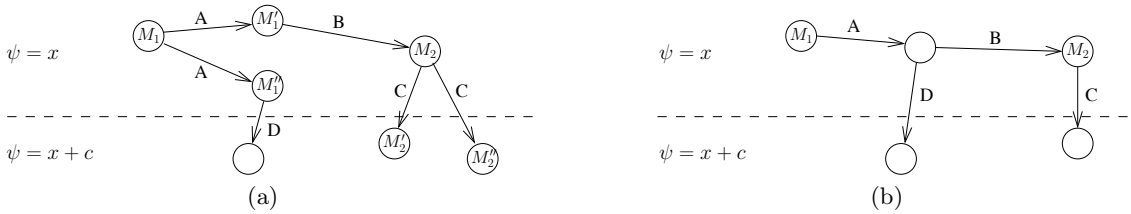


Fig. 4. (a) Nondeterminism involving non-ϵ moves; and (b) merging of states to remove this nondeterminism.

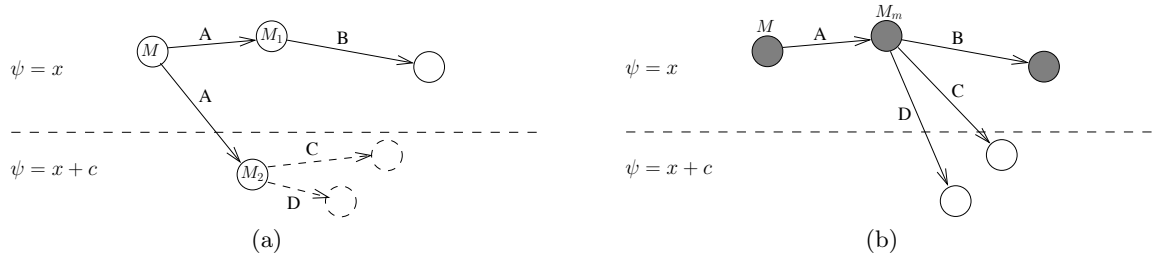


Fig. 5. (a) Nondeterminism of non-ϵ actions across two progress levels; and (b) marking states as *transient* to allow this nondeterminism to be removed once successors of M_2 are known.

merged in the same progress level, but which have a higher progress value than the originator, M_2. The result is shown in Figure 4 (b).

Figure 5 depicts a situation where merging of states occurs across two progress levels. In Figure 5 (a) $\psi(M_1) = x$ but $\psi(M_2) = x + c$. In this situation, it is not possible to merge M_1 and M_2 yet, as M_2 has yet to be explored (the dashed arcs and nodes in Figure 5 (a)). Our proposed solution is to mark M, M_1 and all successors of M_1 as transient, until the successors of M_2 have been calculated. At this point, the merging can occur, and we propose to give the merged state the progress value $min(\psi(M_1), \psi(M_2)) = x$, as illustrated in Figure 5 (b). The reason for this is simple: we wish to avoid regress edges in order to minimise the potential for re-exploration of parts of the state space. This can be generalised to the merging of any number of states in a straightforward manner.

3.3 Coping with Regress

Regress edges complicate matters in two ways. The first is that empty cycles may now exist that span multiple progress levels, as shown at the left of Figure 6 (a), and that ϵ-moves may

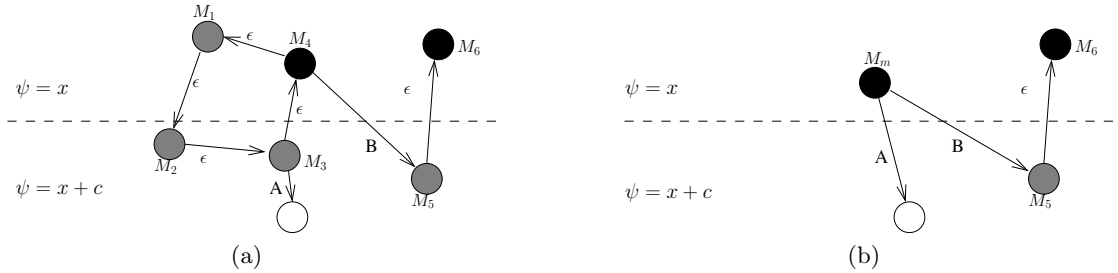


Fig. 6. (a) Empty cycles spanning two progress levels, and an ϵ-move outside a cycle corresponding to a regress edge; and (b) merging of the states in the empty cycle.

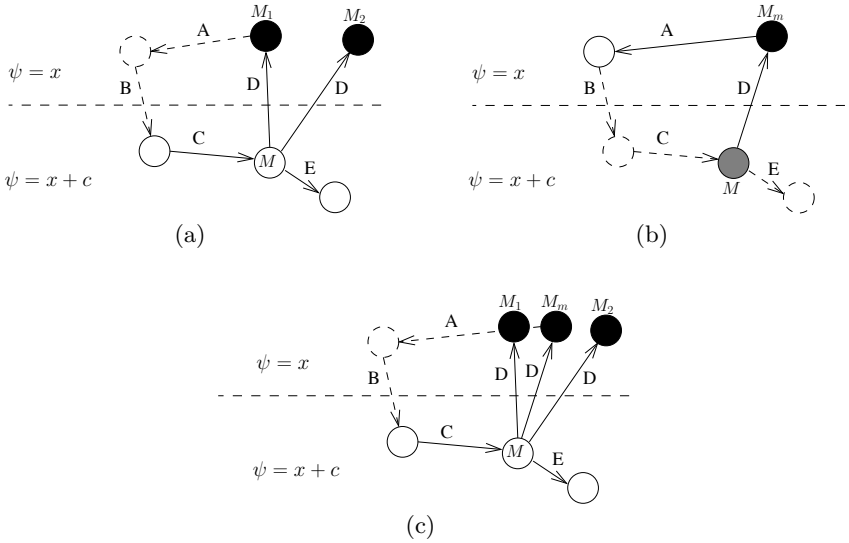


Fig. 7. (a) Two persistent states, M_1 and M_2, to be merged; (b) merging of M_1 and M_2 to M_m on a subsequent sweep; and (c) rediscovery of M_1 and M_2.

correspond to regress edges, as shown at the right of Figure 6 (a). The Sweep-line has marked M_4 and M_6 as persistent (solid circles). Our proposed method for ϵ-removal from Section 3.1 has marked M_1, M_2, M_3 and M_5 as transient. The second complication is that determinisation may require a persistent state to be deleted. For example, the existence of an empty cycle across multiple progress levels implies that at least one state in that cycle has been marked as persistent (in this case M_4). Determinisation will also require M_6 to be deleted (following the algorithm in [1]) as it is the destination of an ϵ-move and has no successors. We propose to treat these two complications by merging the empty cycle as before, but to assign the merged state the minimum progress value over all the states being merged (holding to the heuristic of minimising the number of regress edges) and to mark the merged state as persistent, as shown in Figure 6 (b), in the same way that merged states become halt states if any of their component states are halt states [1]. To eliminate the ϵ-move at the right of Figure 6 (a) it is a matter of exploring the successors of M_6. The only additional step in our proposal is to mark the successors of M_6 as persistent because M_6 was persistent, in the same way that the successors of M_6 would become halt states if M_6 were a halt state [1]. The same procedure

should be applied when merging/deleting persistent states as a result of determinisation of non-empty moves.

As a consequence of our method of coping with regress and the deletion of persistent states, we must consider what will happen if a merged/deleted persistent state is re-explored. Consider Figure 7 (a), in which two regress edges from M induce two persistent states, M_1 and M_2. They cannot be merged until the successors of both are known. We propose that M be marked as transient until such time as M_1 and M_2 can be merged into M_m, as shown in Figure 7 (b). Then, re-exploration from the successors of M_m once again discovers the two persistent states from Figure 7 (a), as shown in Figure 7 (c). Because M_m was marked as persistent, we need to explore M_1 and M_2 again only as far as is necessary to merge them and rediscover M_m. An implementation that would produce this effect would be to mark the immediate successors (once ϵ-removal is complete) of the merged states as persistent, hence re-exploration is limited only to the successor states (exploration is truncated at persistent states visited for the second time).

4 Conclusions and Future Work

In this paper we have presented one possible genesis for an extension of the Sweep-line method to handle language equivalence checking, an extension of importance to protocol verification. However, substantial effort is required to bring this proposed extension to fruition. This work must be formalised to produce an extended Sweep-line algorithm, implemented and evaluated using a case study. There are also many avenues that remain to be investigated.

The discussion of determinisation presented in this paper is based on all states with the minimum progress value having been explored before determinisation/complementation/parallel composition begins. However, it should be possible to interleave the exploration, determinisation, complementation and parallel composition procedures. This may provide a significant saving in the time taken to detect violations of the language equivalence property, particularly if there are many states with identical progress values.

One aspect of this proposal that has not yet been discussed is the choice of progress mapping. Although progress mappings may be arbitrary, previous experience (e.g. [9,10]) has indicated that the choice of progress mapping plays a pivotal role in the performance of the Sweep-line method. The author believes that there is no inherent problem with a progress mapping that results in the merging of states with different progress values, as the progress mapping may depend on internal protocol actions and changes of state within protocol entities that are not part of the protocol's user-observable behaviour.

Although we have used the two-step process of [1] for determinisation, it may be possible to use the method of subset construction with lazy subset evaluation [15] for both epsilon removal and determinisation in a single step. For each new state encountered, its transitive closure comprising only empty moves is calculated. This subset of states becomes the state of the deterministic FSA. However, such a subset may contain states of many different progress values, hence the order of state exploration may violate the least-progress-first policy of the Sweep-line method. This may be able to be overcome by delaying completion of the construction of the subset using transient states in a similar way to the process described in this paper. However, this will reduce the effectiveness of the Sweep-line method as a state space reduction technique, as more transient states will remain in memory for longer.

In terms of simply detecting a violation of language equivalence, it may be possible to take a more direct approach under some conditions. If the parallel composition of $DFSA_S$

and $DFSA_P$ (not its complement) is calculated on-the-fly, it may be possible to examine each composed state to see what actions are allowed by the corresponding states in $DFSA_S$ and $DFSA_P$. If they don't match, we have a violation of language equivalence. This approach relies on knowing that livelocks do not exist in either the protocol or service specifications, but would not require the complement of the protocol FSA to be calculated on-the-fly.

Acknowledgments

The author would like to express his sincere thanks to Prof. Lars Kristensen for productive discussions in 2005 and for originally proposing the idea of transient states, and to Prof. Jonathan Billington for providing valuable comments on preliminary versions of this paper.

References

1. W.A. Barrett, R. M. Bates, D. A. Gustafson, and J.D. Couch. *Compiler Construction: Theory and Practice*. Science Research Associates, 2nd edition, 1986.
2. B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, and Ph. Schnoebelen. *Systems and Software Verification - Model-Checking Techniques and Tools*. Springer, 2001.
3. J. Billington, G. E. Gallasch, and B. Han. A Coloured Petri Net Approach to Protocol Verification. In *Lectures on Concurrency and Petri Nets, Advances in Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 210–290. Springer-Verlag, 2004.
4. D. Burdett. Internet Open Trading Protocol - IOTP Version 1.0. RFC 2801, IETF, April 2000.
5. C. G. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems*. Kluwer Academic Publishers, 1999.
6. S. Christensen, L. M. Kristensen, and T. Mailund. A Sweep-Line Method for State Space Exploration. In *Proceedings of TACAS 2001*, volume 2031 of *Lecture Notes in Computer Science*, pages 450–464. Springer-Verlag, 2001.
7. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. The MIT Press, 2000.
8. FSM Library, AT&T Research Labs. `http://www.research.att.com/~fsmtools/fsm/`.
9. G. E. Gallasch, B. Han, and J. Billington. Sweep-line Analysis of TCP Connection Management. In *Proceedings of the International Conference on Formal Engineering Methods (ICFEM'05)*, volume 3785 of *Lecture Notes in Computer Science*, pages 156–172. Springer-Verlag, 2005.
10. G. E. Gallasch, C. Ouyang, J. Billington, and L. M. Kristensen. Experimenting with Progress Mappings for the Application of the Sweep-Line Analysis of the Internet Open Trading Protocol. In *Fifth Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools*, pages 19–38. Department of Computer Science, University of Aarhus, 2004. Available via `http://www.daimi.au.dk/CPnets/workshop04/cpn/papers/`.
11. G. E. Gallasch, S. Vanit-Anunchai, J. Billington, and L. M. Kristensen. Checking Language Inclusion On-The-Fly with the Sweep-line Method. In *Proceedings of CPN'05*, Department of Computer Science Technical Report, DAIMI PB 576, pages 1–20. University of Aarhus, 2005.
12. G. E. Gallasch, S. Vanit-Anunchai, J. Billington, and L. M. Kristensen. Checking Safety Properties On-The-Fly with the Sweep-line Method. *International Journal on Software Tools for Technology Transfer*, 9(3-4):371–392, 2007.
13. R. J. van Glabbeek. The Linear Time - Branching Time Spectrum II: The semantics of sequential systems with silent moves (extended abstract). In *Proceedings of the 4th International Conference on Concurrency Theory (CONCUR'93)*, volume 715 of *Lecture Notes in Computer Science*, pages 66–81. Springer, 1993.
14. B. Han. *Formal Specification of the TCP Service and Verification of TCP Connection Management*. PhD thesis, Computer Systems Engineering Centre, UniSA, Adelaide, Australia, December 2004.
15. J. E. Hopcroft, R. Motwani, and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 2nd edition, 2001.
16. K. Jensen and L.M. Kristensen. *Coloured Petri Nets: Modelling and Validation of Concurrent Systems*. Springer, 2009.
17. T. Jéron, H. Marchand, and V. Rusu. Symbolic Determinisation of Extended Automata. In *Proceedings of 4th IFIP International Conference on Theoretical Computer Science (TCS 2006)*, volume 209 of *IFIP International Federation for Information Processing*, pages 197–212. Springer Boston, 2006.

18. T. Jussila, K. Heljanko, and I. Niemelä. BMC via On-the-Fly Determinisation. *International Journal on Software Tools for Technology Transfer*, 7(2):89–101, 2005.
19. D. C. Kozen. *Automata and Computability*. Springer-Verlag, 1997.
20. L. M. Kristensen and T. Mailund. A Generalised Sweep-Line Method for Safety Properties. In *Proceedings of FME'02*, volume 2391 of *Lecture Notes in Computer Science*, pages 549–567. Springer-Verlag, 2002.
21. O. Kupferman and M. Y. Vardi. Model Checking of Safety Properties. *Formal Methods in System Design*, 19(3):291–314, 2001.
22. C. Ouyang. *Formal Specification and Verification of the Internet Open Trading Protocol using Coloured Petri Nets*. PhD thesis, Computer Systems Engineering Centre, UniSA, Adelaide, Australia, June 2004.
23. C. Ouyang and J. Billington. On Verifying the Internet Open Trading Protocol. In *Proceedings of 4th International Conference on E-Commerce and Web Technologies (EC-Web)*, volume 2738 of *Lecture Notes in Computer Science*, pages 292–302. Springer, 2003.
24. A. Valmari. The State Explosion Problem. In *Lectures on Petri Nets I: Basic Models*, volume 1491 of *Lecture Notes in Computer Science*, pages 429–528. Springer-Verlag, 1998.
25. M. Y. Vardi and P. Wolper. An Automata-Theoretic Approach to Automatic Program Verification. In *Proceedings of 1st Symposium on Logic in Computer Science, Cambridge, USA*, pages 332–344. IEEE Computer Society Press, 1986.