Building Verifiable Software Prototypes using Coloured Petri nets

Michael Westergaard

June 10, 2005

Abstract

This progress report outlines the work conducted on part A of my PhD study at the Department of Computer Science, University of Aarhus.

During Part A of my PhD study, I have conducted work in three areas related to modeling of computer systems. Firstly, I have participated in the design and implementation of a state-space tool, which is intended to be easy to extend. As a part of this, I have participated in developing and testing a concrete way to do state-space analysis.

Secondly, I have designed and implemented TIN-CPN, a tool to support animation of Coloured Petri net models, and I have participated in a concrete project using TIN-CPN to create a model-based prototype of a networking protocol.

The third area I have worked on is within the standardization of an exchange format for Petri nets, where I have participated in a number of discussions at standardization meetings, and made a concrete proposal for an exchange format.

The report provides directions for future work during part B of the PhD study.

Contents

Chapter 1

Introduction

One major problem programmers face when writing software, is how to deal with concurrency. Sequential parts of a program are in principle fairly easy to get right using methods such as testing [2], hoare-logic [36] and even automatic or semi-automatic proofs of correctness [30, 76]. Concurrent systems, however, present a large number of new problems, since the number of possible interleavings of parts of the program make manual testing infeasible, in part because of the amount of interleavings and in part because scenarios may be difficult to reproduce. In order to overcome these problems, one may not test the system itself, but a formal model of the system.

During part A of my PhD study, I have conducted work in three areas related to modeling of computer systems. First, a brief overview of the work conducted in each of the three areas is provided.

Towards a New State-space Tool. The idea of the state-space method is to generate a graph with system state as nodes and transitions from one state to another as edges. This graph can be used for analysis. The problem of this method is the state-explosion problem, i.e. that the graph may be very large or even infinite. The problem with many previous tools that support state-space analysis is that they are not written in a modular way. During my PhD study, I have participated in the design and implementation of a new tool to support state-space analysis. The new tool that has been designed and in part implemented defines three interfaces for different parts of a state-space tool, making experimentation with new methods to alleviate the state-explosion problem easier.

I have participated in the development of a particular reduction technique, where the idea is to exploit progress of modeled systems to create a near-optimal representation of the graph. This work has been published at TACAS 2004 as [66].

Model-based Prototyping and Animation. Communicating formal models to software developers and industrial partners is often difficult. Therefore a graphical representation that is easier to understand is often created. I have designed and implemented a tool, TIN-CPN [94] to support this. TIN-CPN is based on a simple plug-in architecture making it easy to extend. TIN-CPN has been used in an industrial case with Ericsson Danmark A/S, Telebit, where we developed a model-based prototype of a network protocol. The result has been shown to project leaders and partners from the Danish, Swedish and English military, all of which had little or no knowledge of formal models. This work has been submitted to IFM 2005 in [60].

Towards an Exchange Format for Coloured Petri Nets. The exchange of models between different tools has several advantages, for example one can make a repository of standard benchmark models or one can design tools that only focus on construction of models and leave verification to other toold. The exchange is made difficult, as different tools use different languages to describe otherwise equal concepts, e.g. declaration of variables. Also, Coloured Petri nets support various composition mechanisms, but different tools may not support the same composition mechanisms.

I have made a concrete proposal for an exchange format for Coloured Petri nets. The format uses a mixture of abstract and concrete syntax in order to allow exchange between different tools while still being easy to extend and usable in practice. The proposal also provides translations of various translations from common composition mechanisms to a simple module concept described in [54]. My proposal has been published at a workshop on exchange formats for Petri nets in [95].

The rest of this report is structured as follows: In Chap. 2, we briefly describe the parts of the concrete modeling language, Coloured Petri nets (CP-nets or CPN) needed to understand the rest of the report. This chapter is provided for background information and to introduce an example, that will be used throughout the report. In Chap. 3, we describe a particular approach to analysis of models, namely the state-space method. We describe the design of a tool to support this kind of analysis and how this design has been implemented and used to test a method to overcome some of the problems with the state-space method. We describe a new, very memory-efficient way to represent a state-space. In Chap. 4, we describe a tool that supports development of model-based software prototypes and animations, and describe how this was used in an industrial case-study to build a prototype of a network protocol. In Chap. 5, we briefly sum up the problems of exchanging Petri nets, and present parts of our concrete proposal. Finally, in Chap. 6, we draw our conclusions and sum up the intended future direction of work. No special knowledge is required to read this report, but some knwoledge of simple formalisms such as labeled transition systems is an advantage.

Chapter 2

A Formal Modeling Language: Coloured Petri Nets

In this chapter, we will briefly introduce the modeling formalism of Coloured Petri nets (CP-nets or CPN) [48–50, 57]. This chapter is provided mainly to introduce the formalism, fusion places, synchronous channels and an example that we shall use throughout the report.

2.1 Generalizing Labeled Transition Systems

We will introduce the concept of CP-nets by starting from the well-known formalism of labeled transition systems, which are deterministic finite automata with no accept or reject states. In their original form, Petri nets [74] can be viewed as a generalization of labeled transition systems (LTS). An LTS is basically a number of states, usually depicted as circles, optionally with a name, and a number of possible transitions between the states, usually depicted as labeled edges between the circles. The system can be in exactly one state at a time. We will depict this by adding a black dot, a token, to the current state. A simple example of an LTS can be seen in Fig. 2.1. This simply models a runner in a race. First, the runner Starts a race. He can then run to a DrinkStand, where he can drink as much water as he desires, and then run to the Goal.

2.1.1 Place-Transition Petri Nets

The example in Fig. 2.1 has some shortcomings and we shall try to solve these. Firstly, the model allow the runner to to drink an arbitrary amount of water. We would like to model that there is only be a finite number of glasses available. Second, the model only allows one runner in the race, and we would like to allow more runners to enter the race. It is possible to model this using LTS, but place-transition Petri nets (PT-net) [24] allow us to do this in an easier way.

The first generalization we will make is that we make it possible to be in more than one state at a time, for example to make a clearer model of the fact that one runner is at the DrinkStand whereas another runner is at the Start of the race. We will draw this by allowing a token in any number of states. When it is possible to be in more than one state at a time, it also makes sense to be in a given state more than once, e.g. both runners can be at the DrinkStand at the same time. We do this by allowing more than one token or by just writing the number of tokens present in a state. We call the n umber of tokens on a place the marking of the place.

It is then natural to allow the modeler to require more than one token from a place at a time for a transition to occur. For example a runner must be at the DrinkStand and a drink must be available for a runner to drink. We do this by changing transitions from normal directed edges to directed hyper-edges [67], i.e. edges that can go between more than two nodes. We thus have a set of source states and destination states rather than just one of each. We also rename states to places. To make drawings easier to read, we will usually draw a line or rectangle for each transitions and draw edges from each source place to the rectangle and draw edges from the rectangle to each of the destination places. We shall then refer to the rectangle or line as the transition, and the edges between the transition and places as arcs. We also allow transitions to require or produce more than one token. This is done by writing the number

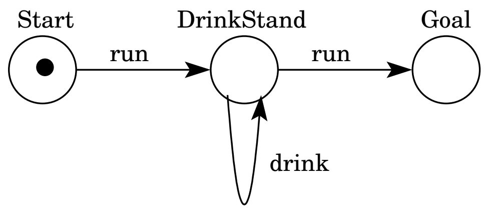


Figure 2.1: A simple Labeled Transition System modeling a runner at a race.

of tokens required from each source place next to the arc from the place to transition. We call this number the arc annotation. We do the same for destination places. An improved model of the race example is seen in Fig. 2.2. We have introduced exactly the two improvements mentioned: we now allow two runners in the race, and we have limited the number of available drinks to two.

The intuitive semantics is that we require that each source place has the required number of tokens, and when a transition occurs the specified number of tokens is removed from the source places and the specified number of tokens are added on the destination places. We then obtain PT-nets.

2.2 Coloured Petri nets

The version of the runner example in Fig. 2.2 also has two problems. Firstly, event though we have introduced two runners, we are not able to see who wins the race, as the two runners are identical. Secondly, one runner is able to drink both glasses of water, leaving the other runner thirsty.

We want both runners to behave the same, yet we still want to distinguish them. This can be modeled using PT-nets, but it will require that we model each process separately, as they may treat different data. Coloured Petri nets solve this problem by distinguishing tokens; intuitively giving them different colors, meaning that rather than allowing a number of black tokens, we allow a multi-set of colored tokens. We must accordingly extend the arc annotation, so we can specify the color of the tokens we wish to consume and produce.

In order to distinguish the runners, but still allow the modeler to specify their behavior uniformly we allow variables on arcs. Variables do not behave like variables in procedural or object-oriented languages, but more like formal parameters to a function in a functional programming language, meaning that variables are local to a single transition, so the value of a variable is forgotten after a transition has occurred. Also, it is not possible to change the value of a variable during transition execution.

An example of a CP-net can be seen in Fig. 2.3. This is a modification of the PT-net in Fig. 2.2. We have changed the runners, so they are no longer identical, but named $\mathsf{r(1)}$ and $\mathsf{r(2)}$. Also, we have introduced a flag, which is up at the start of the race, and taken down when the first runner passes the finish line. This is used to keep track of the first runner to cross the finish line.

As we can now distinguish runners, it makes sense to introduce runner-specific information. We allow arbitrary types of information to be used as tokens. For example, the glasses with the drinks have two pieces of information associated with them, namely the name of the runner who is able to drink it, and information whether the glass is full or empty. We declare types of places, i.e. what kinds of tokens the place can contain, using an annotation near the place. The declarations of the types $\mathsf{RUNNER} = \{\mathsf{r(1)}, \mathsf{r(2)}\}$, $\mathsf{GLASS} = \mathsf{RUNNER} \times \{\mathsf{full}, \mathsf{empty}\}$, and $\mathsf{FLAG} = \{\mathsf{up}, \mathsf{down}\}$ and the variable $\mathsf{x} : \mathsf{RUNNER}$ are not shown in Fig. 2.3 to keep the figure simple. We will furthermore allow arbitrary functions as arc annotations to manipulate the data contained in the tokens. In the example, the arc from the drink transition to the Drinks place will empty the glass.

It is no longer enough to talk about a transition occurring, as a transition's effects depend on which runner we are treating. Rather we introduce the concept of a binding element, which is a transition and an assignment of values to each of its variables. In order to test whether a binding element is enabled, we replace all variables with their assigned value and evaluate the expressions on the arcs. When we in this manner evaluate the variables, the arc annotation will evaluate to a multi-set of typed tokens. In the same way as with PT-nets, we say that a binding element is enabled and can occur if there are "enough" tokens of each kind on all source places. Occurrence will simply lead to a marking where the specified number of tokens of each kind are removed from the source place and new tokens are produced on the destination places.

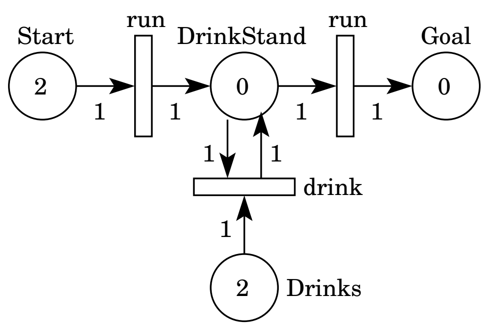


Figure 2.2: A place-transition Petri net. This is an extended version of the system in Fig. 2.1, as we have limited the total number of Drinks to two, and we have introduced an extra runner.

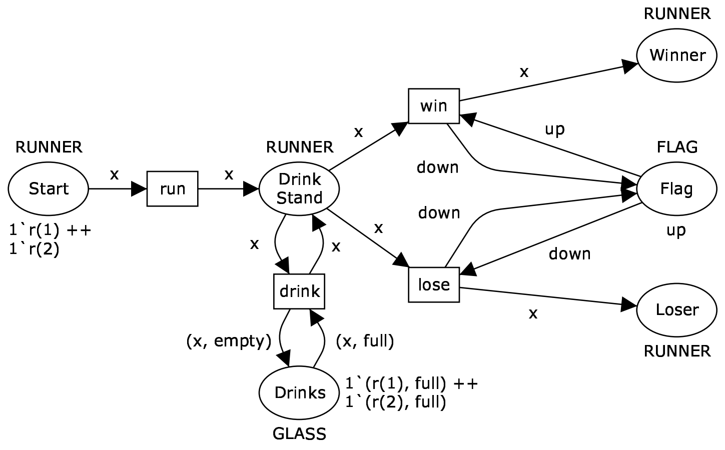


Figure 2.3: A Coloured Petri net. We have changed the runners, so they are no longer identical, and a runner can only take a glass with his name on it and empty it. When a runner leaves the DrinkStand, depending on whether he is first or second, he will either win or lose the race.

2.2.1 Formal Definition of CP-nets

Let us give a formal definition of CP-nets. The definition is similar to the one given in [45, Def. 5.1]. We shall not go into the details, but just mention that the definitions are equivalent, and we will think of CP-nets as described in the previous section when modeling and the following definition when making formal statements.

We will assume that the relations $<$, $=$, $\leq$, $>$, and $\geq$, and operations $+$ and $-$, on multi-sets are defined as usual.

Definition 1 (Coloured Petri net) *A* Coloured Petri net *is a tuple, $\mathcal{CPN} = (\Sigma, P, T, C, I_-, I_+, M_I)$, where*

- $\Sigma \neq \emptyset$ *is a finite set of non-empty* types *of tokens,*
- $P \neq \emptyset$ *is a finite set of* places *such that $\Sigma \cap P = \emptyset$,*
- $T \neq \emptyset$ *is a finite set of* transitions *such that $\Sigma \cap T = \emptyset$ and $P \cap T = \emptyset$,*
- $C : P \cup T \to \Sigma$ *is a* type function *assigning a type to each place and transition,*
- $\forall p \in P\, \forall t \in T. I_-(p,t), I_+(p,t) : C(t) \to \mathbb{N}^{C(p)}$ *are the backward and forward incidence functions, assigning to each arc an arc annotation, and*
- $\forall p \in P. M_I(p) \in \mathbb{N}^{C(p)}$ *is the* initial marking.

The state of a CP-net is given by a *marking* of the places, which is a function over all places, $\forall p \in P. M(p) \in \mathbb{N}^{C(p)}$.

Definition 2 (Binding Element) *A* binding element *for a CP-net $\mathcal{CPN}$, is a pair (t, b), where*

- $t \in T$ *is a transition and*
- $b \in C(t)$ *is the* binding.

We call the set of all binding elements $\mathcal{BE} = \{(t,b) \mid t \in T \wedge b \in C(t)\}$.

Definition 3 *A binding element, $(t,b) \in \mathcal{BE}$, is* enabled *in marking M if $\forall p \in P. M(p) \geq I_-(p,t)(b)$. A transition is enabled if there exists $b \in C(t)$ such that (t,b) is enabled. If (t,b) is enabled in M, it can* occur *and leads to a marking M'. This is written $M\,[(t,b)\rangle\, M'$, where M' is defined by $\forall p \in P. M'(p) = (M(p) - I_-(p,t)(b)) + I_+(p,t)(b)$.*

We will use the notation $M\,[\sigma\rangle\, M'$ for $\sigma = (t_1,b_1)(t_2,b_2)\ldots(t_n,b_n)$ to mean $\exists M_i$ for $i = 1,\ldots,n+1$, $n \geq 0$, such that $M = M_1$, $\forall i = 1,\ldots,n. M_i\,[(t_i,b_i)\rangle\, M_{i+1}$, and $M' = M_{n+1}$. We will also write $M\,[*\rangle\, M'$ to mean $\exists \sigma. M\,[\sigma\rangle\, M'$. We say that a marking M' is *reachable* from another marking M if $M\,[*\rangle\, M'$ and we let $[M\rangle = \{M' \mid M\,[*\rangle\, M'\}$ denote the set of markings reachable from M. When we talk about the set of *reachable markings* of a CP-net, we mean the set of markings reachable from the initial marking, i.e., $[M_I\rangle$.

We can furthermore allow a number of extra conveniences, such as transition guard expressions, which can control in greater detail whether a binding element is enabled, a more modular kind of CP-nets, called Hierarchical Coloured Petri nets, and a version of CP-nets where tokens can carry a timestamp and transitions occurences can take time, called Timed Coloured Petri nets. For a description and formal semantics of these concepts, please refer to [48].

We will consider two extensions of Coloured Petri nets, namely fusion places [48, Chap. 3] and synchronous channels [16], because we shall need them to discuss interfaces to an animation tool (Chap. 4) and to give an example of representations of composition mechanisms in an exchange format for Petri nets (Chap. 5). These extensions basically allow us to draw "shadows" of places and transitions in multiple locations, in part to improve the readability of models, but more importantly to allow different modules to communicate with each other. The formal semantics of these two extensions shall not be given here. We will only note that the formal semantics of the two extensions can be defined by a translation to CP-nets as defined in Def. 1, so formally, we do not need to treat the extensions seperately.

2.2.2 Fusion Places

Suppose we have a large CPN model. It may be difficult to draw the arcs such that the flow of the model is evident. In order to overcome this problem, the model can be split it into smaller modules, which can communicate using fusion places. Fusion places are basically multiple places sharing the same tokens. We can think of fusion places as an asynchronous (unordered) communication channel. Whenever a token is added to one place in the fusion set, it is added (sent) to all the others in the same fusion set, and whenever a token is removed from one place, it is removed from all places in the fusion set. For this reason the places also share the initial token distribution, and must have the same type. We can obtain the model in Fig. 2.4 from the example in Fig. 2.3. We basically make a module containing the exact implementation of what happens when a runner wants to drink water at the DrinkStand. The model is functionally equivalent to the model in Fig. 2.3. The only change is that we now have two places called DrinkStand. Both of these places are members of the fusion set Fusion 1, as indicated by the small box with the text "Fusion 1". Whenever the token representing a runner appears in the right-most DrinkStand place, the same token also appears on the left-most DrinkStand place.

We can remove a fusion set from our model and preserve the behaviour of the model by just creating a new place with the correct type and initial tokens. We then move all arcs attached to a member of the fusion set to the new place, keeping the direction and arc annotation. We then remove all members of the fusion set, and have thus removed the fusion set. A more formal description of this procedure can be found in [48, Chap. 3]. This procedure applied for the example in Fig. 2.4 would lead to the model in Fig. 2.3.

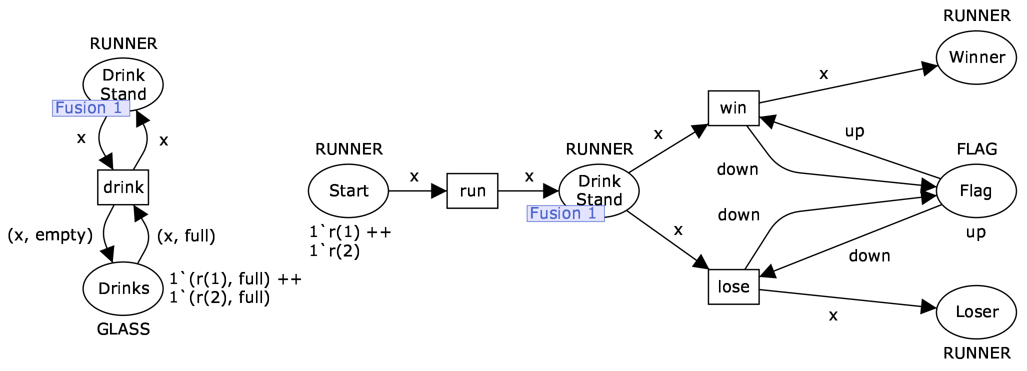


Figure 2.4: A CP-net with fusion places. We thus modularized what happens when a runner drinks water at the DrinkStand, but the model is functionally equivalent to the model in Fig. 2.3.

2.2.3 Synchronous Channels

Sometimes we also want to have multiple representations of the same transition in multiple locations. To do this, we introduce the concept of synchronous channels.

Synchronous channels are well-known from, for example, the π-calculus [70], where one process can send data, synchronously to another process. This is very similar to invoking a method or procedure in a normal programming language. The caller "sends" some data (the parameters) to the receiver (the method/procedure), which can then refer to the received data using its own local name (the formal parameters). Only when the receiver has received the data (the procedure or method returns), can the sender proceed with its own calculations.

In CP-nets, synchronous channels are realized much like place fusion sets, i.e. we have multiple representations of one real transition. We split the transitions participating in a channel into senders and receivers. Each time a sender transition occurs, a receiver transition for the same channel must also occur. In the example in Fig. 2.5, we have one channel ch, with two participants, the two transitions named drink. To the left of the transitions, we see the channel inscription, with a channel name (ch) followed by either .? for a receiver transition or .! for a sender transition. After that is a channel expression, which we can think of as the data being transferred from the sender to the receiver. Informally, the semantics of channels is that for each sender-receiver pair, we create a new transition containing all arcs from the sender and receiver. We furthermore create a condition that the channel expressions of the two transitions evaluate to the same. The details of the construction is provided in [16], where also a more general bi-directional channel is considered.

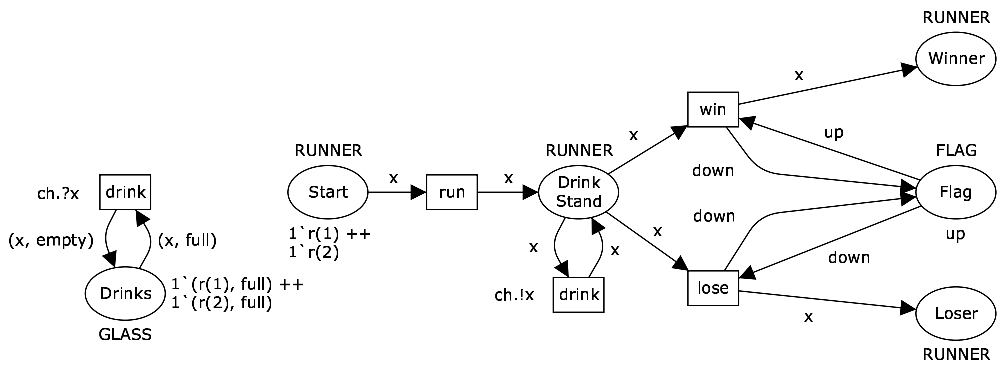


Figure 2.5: A CP-net using synchronous channels. This model is also functionally equivalent to the model in Fig. 2.3. The change is that we have made two copies of the drink transition. The two transitions can only occur if they are both enabled and the x on both transition is bound to the same value. Thus, we have modularized what exactly happens when a runner wants to drink water.

Chapter 3

Towards a New State-space Tool

In Chapter 2.3 we saw how to obtain Coloured Petri nets by adding features to labeled transition systems. Coloured Petri nets clearly have enhanced modeling convenience, as can be seen from even the simple example in Fig. 2.3. It is well-known that many properties of labeled transition systems are easily decidable. Such properties include reachability of a state satisfying a given property or checking properties expressed in Linear Temporal Logic (LTL) [75, 90] or Computation Tree Logic (CTL) [4]. We would obviously like to be able to use this for the analysis of Coloured Petri nets.

The idea of the state-space method is that we construct, at least conceptually, a directed graph, where each node represents a state of the system, and each arc represents an event leading from the source state to the destination state. For CP-nets, we use the reachable markings of the model as states and binding elements as the events.

The major problem of this method is that the state space can become very large, often even infinite. To alleviate this problem, various reduction techniques have been introduced, each of which alleviates the state explosion problem for classes of CP-nets in particular or modeling formalisms in general. In Sect. 3.3 and [66] we describe a particular reduction technique, which provides a very compact representation of the state space for systems with a notion of progress. I has participated in developing and testing this method.

My contribution to this area consists of several things: Firstly, I have made several contributions to the design and implementation of a new state-space tool that makes it easy to implement and experiment with new reduction techniques. As part of this I have implemented a couple of well-known reduction techniques (bit-state hashing [37, 40] and the more general Bloom-filters [9, 27]) and implemented state-space analysis for CCS [68] and Bigraphical Reactive Systems [51, 69]. I have also implemented a user interface for doing state-space analysis, which makes it trivial to do simple analysis and easy to do more complex, customized analysis. Finally, I have participated in the design and test of a particular reduction technique, documented in [66]. Most of the work presented here, is part of the TIN-CPN state-space tool for CP-nets and Bigraphical Reactive Systems.

3.1 The State space of a CP-net

In this section we will define the state space of a CP-net and give the semantics for a logic we can use to query the state space. Introducing Kripke structures [55] for the sake of introducing safety properties may seem like overkill, but this is done in order to be able to easily generalize the results to more interesting modal logics, such as LTL or CTL.

Definition 4 (Kripke structure) *A* Kripke structure *is a tuple,* $\mathcal{K} = (S, L, \delta, AP, \nu, S_I)$*, where*

- $S \neq \emptyset$ *is a set of* states*,*
- $L \neq \emptyset$ *is a set of* labels*,*
- $\delta \subset S \times L \times S$ *is a relation indicating* successor states *(if we are in state* s*, we can take an* a*-step to* s' *iff* $(s, a, s') \in \delta$*),*
- AP *is a set of* atomic propositions*,*
- $\nu : S \to AP \to \{\top, \bot\}$ *is a truth assignment assigning for each state a truth value to each atomic proposition, and*
- $S_I \in S$ *is the* initial state*.*

We say that a Kripke structure is *finite* iff its set of states is finite. Otherwise is is infinite.

Definition 5 (State space) *Given a CP-net,* $\mathcal{CPN} = (\Sigma, P, T, C, I_-, I_+, M_I)$*, we define the* state space *of* $\mathcal{CPN}$ *as a Kripke structure* $\mathcal{SS} = (S, L, \delta, AP, \nu, S_I)$*, where*

- *$S = [M_I\rangle$ the states of $\mathcal{SS}$ are all reachable states of $\mathcal{CPN}$,*
- *$L = \{(t,b) \mid \exists M, M' \in [M_I\rangle . M\,[(t,b)\rangle\, M'\}$ is the set of all binding elements used to move from one reachable state to another,*
- *$\delta = \{(M,(t,b),M') \mid \exists M, M' \in [M_I\rangle, (t,b) \in \mathcal{BE}. M\,[(t,b)\rangle\, M'\}$, successor states are as dictated by the CP-net,*
- *$AP = \{\varphi \mid \varphi : S \to \{\top,\bot\}\}$ are all locally decidable properties,*
- *$\forall s \in S\, \forall \varphi \in AP . \nu(s)(\varphi) = \varphi(s)$ evaluates all propositions of AP, and*
- *$S_I = M_I$ the initial state of the CP-net is the initial state of the state space.*

The definitions of S and δ are straightforward and as one would expect. Examples of locally decidable properties are "the state contains no successors", "the transition t is enabled", or "the place p contains exactly one token with the value 17".

The reason for defining the state space as a Kripke structure is that several interesting modal logics are naturally defined in terms of Kripke structures, and we then get several definitions and results for free. As an example, let us give a semantics for safety properties that is local properties intended to hold in all reachable states. We may think of a safety property as an inveriant. First we give the usual definition of propositional logic:

Definition 6 (Propositional Logic) *Let $AP = \{p, q, r, \ldots\}$ be a set of* atomic properties. *Define*

$$\varphi ::= p \mid \neg\varphi \mid \varphi \to \psi$$

and call the set of all such formulae PL.

We shall of course allow the usual abbreviations,

- $\varphi \vee \psi \equiv \neg\varphi \to \psi$,
- $\varphi \wedge \psi \equiv \neg(\neg\varphi \vee \neg\psi)$, and
- $\varphi \leftrightarrow \psi \equiv (\varphi \to \psi) \wedge (\psi \to \varphi)$.

Now let us define the standard model for propositional formulae.

Definition 7 *Given a function $\nu : AP \to \{\top,\bot\}$, we then define a* truth assignment *for all formulae of PL, $\bar{\nu} : PL \to \{\top,\bot\}$*

$$\bar{\nu}(p) := \nu(p)$$
$$\bar{\nu}(\neg\varphi) := \begin{cases} \bot & \text{if } \bar{\nu}(\varphi) = \top \\ \top & \text{if } \bar{\nu}(\varphi) = \bot \end{cases}$$
$$\bar{\nu}(\varphi \to \psi) := \begin{cases} \bot & \text{if } \bar{\nu}(\varphi) = \top \text{ and } \bar{\nu}(\psi) = \bot \\ \top & \text{otherwise} \end{cases}$$

If $\bar{\nu}(\varphi) = \top$ we say that ν is a model *of φ and write $\nu \models_{PL} \varphi$. We shall say that a Kripke structure, $\mathcal{K} = (S, L, \delta, AP, \nu, S_I)$, is a model of a formula φ iff $\forall s \in S . \nu(s) \models_{PL} \varphi$ and we write $\mathcal{K} \models_{PL} \varphi$.*

In other words, a Kripke structure is a model of a safety property iff the property holds in every state.

Now let us define the problem of checking whether a Kripke structure is a model of a given formula:

Definition 8 (Model-checking Problem) *Given a logic $\mathcal{L}$ with a semantics, $\models_{\mathcal{L}}$, expressed using Kripke structures, a formula φ of $\mathcal{L}$, and a Kripke structure $\mathcal{K}$, we define the* model-checking problem *$MCP_{\mathcal{L}}$ to be*

$$MCP_{\mathcal{L}}(\mathcal{K}, \varphi) \equiv \mathcal{K} \models_{\mathcal{L}} \varphi \tag{3.1}$$

If the logic $\mathcal{L}$ is clear from the context, we shall also use the abbreviations $\mathcal{K} \models \varphi$ rather than $\mathcal{K} \models_{\mathcal{L}} \varphi$ and MCP rather than $MCP_{\mathcal{L}}$. We note that while $\mathcal{K} \models_{PL} \varphi$ means that the property φ must hold in all states of $\mathcal{K}$, we will normally use a modal logic, such as LTL or CTL, which express properties regarding the entire state space.

3.2 Constructing the State space

In this section, we will show how to construct the state space, and indicate how this algorithm can be implemented. We will then extend the construction mechanism to check one of the presented logics, and elaborate on how various reduction techniques easily can be implemented.

It is well known that if a Kripke structure is finite, the model-checking problem is decidable (and even tractable) for several expressive logics, e.g. PL from the previous section, and LTL and CTL. It is therefore interesting to be able to construct the state space of a CP-net.

For the remainder of this chapter, we will assume that we have made sufficient restrictions on CP-nets such that we can always calculate all enabled binding elements for a given marking. We can, e.g., do this by requiring that $\forall t \in T.|C(t)| < \infty$ (exhaustive search on $C(t)$ for all $t \in T$) or requiring that we can calculate I_-^{-1} (calculate $\bigcap_{p\in P} I_-^{-1}(M(p))$ and do exhaustive search). We of course assume that $I_-(p,t)$ and $I_+(p,t)$ are computable functions for all $p \in P$ and all $t \in T$ as well.

If we just want to construct the state space, we can use Algorithm 3.1. It is easy to see that Algorithm 3.1 will terminate if $[M_I\rangle$ is finite. We notice that we do not need to store the successor relation δ and the truth assignment ν as we can, for a given state M, just calculate all enabled binding elements and all successors of M, and gain local knowledge about δ or evaluate all formulae of AP to obtain local knowledge of ν. Thus, in practice, we will rarely store δ nor ν, and we can remove lines 3, 14 and 18 from the algorithm. As we do not store δ, we have little use for the labels, L, so we can also remove lines 2 and 13, and as ν is not stored, we do not need to calculate AP as well, and remove line 17.

Having removed δ, ν, L, and AP, we observe that Algorithm 3.1 uses two data structures, one for the set S and one for the set U, and a single function call, to get all successors of a given state. We will assume three data structures, Storage, Unprocessed, and Formalism, which support the operations of Figures 3.1, 3.2, and 3.3. The data structures are written in the style of Standard ML (SML) [86], as we shall use SML as implementation language for our state-space exploration tool.

The new state-space tool consists of implementations of the Storage, Unprocessed, and Formalism interfaces along with an implementation of the traversal algorithm, Algorithm 3.2. The interfaces and algorithms presented here are slight simplifications only showing the gist of the implementations; the real implementations contain more hooks to e.g. profile the state-space exploration etc.

3.2.1 Different traversal algorithms

Many analysis methods can be implemented more efficiently on-the-fly rather then by first executing Algorithm 3.2 to generate the Kripke structure and then executing a query. For examples we can easily check safety properties (that nothing bad happens in any state) by simply adding a check for the safety property after line 6 in Algorithm 3.2 to obtain Algorithm 3.3. LTL can be checked on-the-fly using nested depth-first traversal [38] and CTL can be checked by storing satisfied sub-expressions [19, Sect. 4.1]. If we obey the interfaces of Storage, Unprocessed, and

```
Require: Coloured Petri net, $\mathcal{CPN} = (\Sigma, P, T, C, I_-, I_+, M_I)$
Ensure: State space of $\mathcal{CPN}$, $\mathcal{SS} = (S, L, \delta, AP, \nu, S_I)$
 1: $S := \{M_I\}$
 2: $L := \emptyset$
 3: $\delta := \emptyset$
 4: $U := \{M_I\}$
 5: while $U \neq \emptyset$ do
 6:     select $M \in U$
 7:     $U := U \setminus \{M\}$
 8:     for all $M', t, b$ such that $M\,[(t,b)\rangle\, M'$ do
 9:         if $M' \notin S$ then
10:             $S := S \cup \{M'\}$
11:             $U := U \cup \{M'\}$
12:         end if
13:         $L := L \cup \{(t,b)\}$
14:         $\delta := \delta \cup \{(M,(t,b),M')\}$
15:     end for
16: end while
17: $AP := \{\varphi \,|\, \varphi : S \to \{\top, \bot\}\}$
18: $\nu(s)(\varphi) := \varphi(s)$ for all $s \in S$ and $\varphi \in AP$
19: return $(S, L, \delta, AP, \nu, M_I)$
```

Algorithm 3.1: Basic state-space algorithm. The algorithm basically maintains two sets, U, the set of unprocessed states, and S, the set of already seen states. Initially we have processed no states and still have to process the initial state. In the loop we pick an unprocessed state, calculate all successor states and add any state we have not yet seen to the set of states we have already seen and not yet processed.

```
signature STORAGE = sig
  type storage
  type element
  val empty_storage: storage
  val insert: element * storage -> storage
  val contains: element * storage -> bool
end
```

Figure 3.1: Signature for the Storage data structure. It contains two type definitions, one for the actual storage and one for the elements stored. It contains one constant value, namely empty_storage, which returns an empty storage, and two functions, insert and contains, for inserting an element into a storage and testing whether an element is already contained in the storage.

```
signature UNPROCESSED = sig
  type unprocessed
  type element
  val empty_unprocessed: unprocessed
  val is_empty: unprocessed -> bool
  val add: element * unprocessed -> unprocessed
  val pick: unprocessed -> element * unprocessed
end
```

Figure 3.2: Signature for the Unprocessed data structure. Like the Storage it contains type definitions for a storage for unprocessed nodes, element, and a constant value for the empty unprocessed storage. The data structure contains three functions, is_empty, which checks whether an unprocessed storage is empty, add, which adds an element to an unprocessed storage, and finally pick, which picks and removes an element from an unprocessed storage and returns the element and an unprocessed storage without the element.

```
signature FORMALISM = sig
  type state
  type event
  val inititial_state: state
  val get_next: state -> (event * state) list
end
```

Figure 3.3: Signature for the Formalism data structure. It contains two type definitions, one for the state, and one for the events leading from one state to another. For CP-nets the events are binding elements. It contains a constant value for the initial_state of the system, and a function, get_next, to get the successors of a given state.

```
Require: Coloured Petri net, $\mathcal{CPN} = (\Sigma, P, T, C, I_-, I_+, M_I)$
 1: $S :=$ storage.insert(formalism.initial_state, storage.empty_storage)
 2: $U :=$ unprocessed.add(formalism.initial_state, unprocessed.empty_unprocessed)
 3: while $\neg$unprocessed.is_empty($U$) do
 4:   $(M, U) :=$ unprocessed.pick($U$)
 5:   for all $((t, b), M')$ in formalism.get_next($M$) do
 6:     if $\neg$storage.contains($M', S$) then
 7:       $S :=$ storage.insert($M', S$)
 8:       $U :=$ unprocessed.insert($M', U$)
 9:     end if
10:   end for
11: end while
```

Algorithm 3.2: Basic state space traversal algorithm without calculation of δ, ν, L and AP. The algorithm uses the data structures of Figs. 3.1, 3.2 and 3.3.

Formalism, it is not difficult to implement such new algorithms, as they often rely on similar primitives. Currently only one implementation of such a traversal exists, namely a version of Algorithm 3.3.

3.2.2 The old state-space tool implementation

CPN Tools [23, 79] is a computer tool for editing and simulating CP-nets, and it contains a state-space tool with an implementation of Algorithm 3.1 (it stores the successor relation δ but not the labels, L, the truth assignment, ν, nor the atomic formulae, AP).

Unfortunately, the implementation is not as clean as desirable. The main problem is that the clean separation between the data structures, Storage, Unprocessed, and Formalism as identified in Algorithm 3.2 is not used in the implementation. Rather, the three data structures are implicitly part of the environment.

This makes it difficult to implement new versions of the various data structures and have them immediately available in any other traversal algorithm we may have written. For example, an old implementation of on-the-fly LTL checking exists for CPN Tools, but as the tree-based storage is so tied to the traversal algorithms, it is very difficult to swap it out and use a more appropriate storage, such as one based on Bloom-filtering.

It is also difficult to make any profiling of state-space tool in CPN Tools, as we cannot easily wrap the Storage, Unprocessed or Formalism with a version tracking how much time is spent on the various operations. This is easy with the cleaner interfaces.

Another, more technical problem is that much of the Storage code is automatically generated and heavily dependent on the CPN model we want to analyze in CPN Tools. As the state-space tool is coded to the implementation and not to an interface, the entire state-space tool becomes dependent on the model and cannot be pre-compiled. It is possible to automatically load the state-space tool when it is needed (and TIN-CPN actually has an implementation of this), but as we have to compile more or less the entire state-space tool every time, this is very time consuming. This is of course annoying, as CPN Tools supports incremental checking and compiling ofCPN models, much like Eclipse [28] does for Java code. Forcing the user to explicitly enter the state-space tool creates a gap between editing a model and analyzing its state space.

3.2.3 Different implementations of the three data structures

Currently CPN Tools uses a quite efficient tree for the implementation of Storage, but if we decide to store states on a hard-disk instead [82], the tree may not be the most efficient way. Also, if we only want to check safety properties using Algorithm 3.3 we do not actually use the values stored in S, as long as we store enough information to keep track of whether we have seen a state before, so we may make an implementation that mainly supports the contains operation, but does not make it possible to actually get the elements of the storage. One way to do this is to implement a storage using bit-state hashing or the more general Bloom-filters. Currently the Storage interface is implemented for a storage using the tree as in CPN Tools, a standard hash table, bit-state hashing, and Bloom-filtering, the two latter implemented by me.

```
Require: Coloured Petri net, $\mathcal{CPN} = (\Sigma, P, T, C, I_-, I_+, M_I)$, safety property, $\varphi$
Ensure: Whether $\forall M \in [M_I\rangle . \varphi(M)$
1:  $S :=$ storage.insert(formalism.initial_state, storage.empty_storage)
2:  $U :=$ unprocessed.add(formalism.initial_state, unprocessed.empty_unprocessed)
3:  while $\neg$unprocessed.is_empty($U$) do
4:     $(M, U) :=$ unprocessed.pick($U$)
5:     for all $((t, b), M')$ in formalism.get_next($M$) do
6:        if $\neg$storage.contains($M', S$) then
7:           if $\neg\varphi(M')$ then
8:              return $false$
9:           end if
10:          $S :=$ storage.insert($M', S$)
11:          $U :=$ unprocessed.insert($M', U$)
12:       end if
13:    end for
14: end while
15: return $true$
```

Algorithm 3.3: Algorithm for checking safety properties. This is a simple extension of Algorithm 3.2 to check MCP_{PL}. We use Algorithm 3.2 to traverse the entire state space and simply check the safety property for each new state (in lines 7-9).

We note that the pick operation of the Unprocessed does not specify which element to pick. For example, if we have implemented Algorithm 3.3, we may be interested in an error trace to the state not satisfying φ. Often we would like this error-trace to be as short as possible. One way to accomplish this is to search the state space using a breadth-first traversal. It is easy to convince oneself that if we implement the Unprocessed interface as a simple FIFO-queue, we will traverse the state space in a breadth-first manner. Other algorithms, such as the nested depth-first traversal used to check LTL on-the-fly, will instead use a depth-first traversal, which can be accomplished by implementing the Unprocessed interface as a LIFO-stack. The sweep-line method [18, 59] uses a special traversal policy that benefits from a priority queue, as is briefly explained in Sect. 3.3. All of the mentioned implementations of the Unprocessed interface currently exist.

Finally, one can also easily imagine other formalisms implementing the Formalism interface, thereby making a state-space tool that is not tied to a specific modeling language. In fact, the Formalism interface has been implemented for CP-nets in CPN Tools, simple CCS-processes and Bigraphical Reactive Systems, the two latter by me.

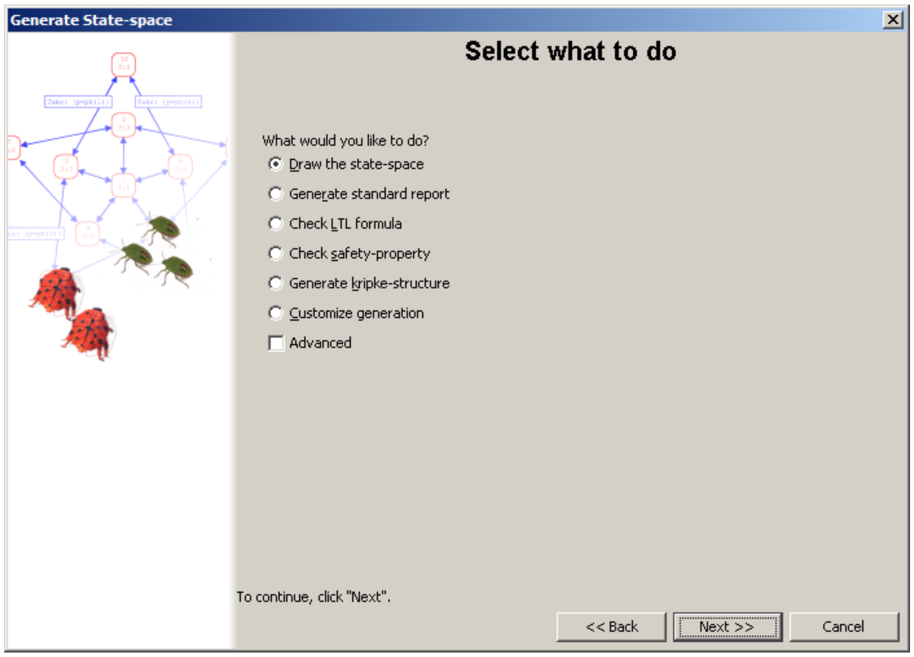
3.2.4 Experimenting with reduction techniques

With the current state-space tool of CPN Tools, experimenting with new reduction methods is tedious and difficult. This problem is not unique to CPN Tools. Actually a similar problem arises if one want to change how states are stored in e.g. SPIN [39], where experimenting with new methods forces you to modify code-generating C-code, which is a difficult task.

The new design, where we have defined the Storage, Unprocessed, and Formalism interfaces and Algorithm 3.3 has made experimenting much easier, one particular example is the example that will be presented in Sect. 3.3.

As a lot of Storages, three different Unprocessed implementations, and two interesting Formalisms have already been implemented, the user is forced to make a lot of choices just to generate and analyze a state space. To circumvent this problem, a wizard has been developed, which makes the decisions for the user for simple tasks such as drawing the state space, generating an informative standard report or making simple queries. The wizard is my idea and designed and implemented by me.

A screen-shot of the possibilities presented to the user in the current prototype can be seen in Fig. 3.4. For example, the inexperienced user, who just wants to draw the state space, simply opens the wizard, checks “Draw the state space”, leaves the “Advanced” box unchecked, and clicks “Next”. The wizard will then choose a storage, e.g. Bloom-filtering, and an appropriate unprocessed storage (standard FIFO queue), and select a suitable version of the traversal algorithm to draw the state space. Likewise reasonable choices are made, which allows the user to generate a report showing simple properties of the system, to check safety properties or to simply create the Kripke structure for further analysis.



What would you like to do?

[✓ Draw the state-space]
Generate standard report
Check LTL formula
Check safety-property
Generate kripke-structure
Customize generation
☐ Advanced

To continue, click "Next".

Figure 3.4: The wizard presented to help users doing common tasks with the new state-space tool. Appropriate choices are made to help the user to e.g. draw the state space, check safety properties, and generate a simple standard report.

The user may also want to change some of predefined decisions, e.g. the safety property checker will currently use Bloom-filtering for storage, but if the user wants to be 100% sure the desired property holds (Bloom-filtering may report no error even if there is an error, and is often used to find errors rather than proving there are none), he may want to use a simple hash table or the tree structure, perhaps even a future disk-based storage if he suspects the state space is large. If the "Advanced" box is checked, the user is allowed to check and change all decisions made by the wizard.

At the end of the wizard, the user is also shown the code that will be executed to generate the state space and do the desired analysis, so he is able to change the code to his desire. This has been used extensively during test and implementation of various instances of the three interfaces.

3.3 Efficient State Representations

As an example of experimenting with the cleaner interfaces, let us briefly sum up the results of a concrete experiment with a very efficient representation of the state space of CP-nets (or any modeling language implementing the Formalism signature).

The problem we address is that we want to store the entire state space explicitly, but we want to use as little space as possible. Often a state space tool will use a quite naïve way to store states, e.g. a pair (n, m) may be stored as two integers. This is done in order to be able to decode and encode values, while at the same time general enough to handle all systems. If $n, m \in \{0, 1, 2, 3, 4\}$, we will use at least $2\lceil\log 5\rceil = 3 + 3 = 6\,bits$ to store a pair (n, m). A better way is to enumerate all possible syntactical states and use only enough bits to distinguish between them, $\lceil\log(5 \cdot 5)\rceil = 5\,bits$. The optimal solution is to enumerate the reachable states only, for example we may know that $n, m \in \{0, 1, 2, 3, 4\}$ and $|n - m| < 2$ for all reachable pairs (n, m), so only 13 states are reachable. We could then store each state using only $\lceil\log 13\rceil = 4\,bits$. The caveat is that most of the time we only know the number of reachable states, $R = |[M_I\rangle|$, when we have generated the state space, so we do not know $\lceil\log R\rceil$ either, and therefore we do not know how many bits are required to store each state. This is the problem we address in [66]. Here, we shall only give the intuition of the algorithm used to create a very efficient representation of the state space.

Let us reconsider the example from Fig. 2.3 on page 5. For simplicity, we will remove the ability to drink water. If we do this, the resulting model will have a state space that looks like Fig. 3.5. In Fig. 3.5, we use the notation (m_1, m_2, b) for each state where m_i is the position of runner r(i) where Start$= 0$, DrinkStand$= 1$, Winner$= 2$, and Loser$= 3$. b indicates whether the flag is up or down.

When we look at Fig. 3.5, we notice several things. Firstly, we can make calculations, like above, of the number of bits required to store each state using the naïve representation, namely $\lceil\log 4\rceil\,bits$ for each runner and $1\,bit$ for the state of the flag, for a total of $5\,bits$, and noticing that only 10 states are actually reachable, so only $\lceil\log 10\rceil = 4\,bits$ are actually needed.

If we take a close look at Algorithm 3.3, we notice that S is only used to ensure termination in the case where we have cycles in the graph and to prevent re-exploring states already visited (if a state is already in S, we will not add it to U, and therefore not re-explore it). If we draw a horizontal line in Fig. 3.5, it will separate all the reachable states into two sets, and all arrows will be directed from the lower set to the upper, and never backwards, as all arrows in the figure point upwards. Thus if we have already visited all states in the lower set, we can safely remove all the states below the line from S; as no arrow points back to the set, we will never be in the case where we should have answered yes to $M' \in S$ in the if statement, but falsely answer no.

The sweep-line [18,59] method defines a simple way to create such a line. We define a so-called *progress measure* to be a function $\varphi : [M_I\rangle \to \mathbb{N}$ (in general the domain is just a partially ordered set) satisfying $M\,[(t, b)\rangle\,M' \implies \varphi(M) \leq \varphi(M')$. Using contraposition, we obtain $\varphi(M') < \varphi(M) \implies \neg M'\,[(t, b)\rangle\,M$, meaning no arrows will go from a state with a higher progress measure to a state with a lower progress measure. This also means that all states in a cycle must have the same progress measure. In the worst case, all states must have the same progress measure. In our example we can use $\varphi(m_1, m_2, b) = m_1 + m_2$.

We can then use a priority queue for our Unprocessed and use $\varphi(M)$ as the priority of M. Whenever we obtain a strict increase of $\varphi(M)$ after the pick in line 4 in Algorithm 3.3, we remove all elements of S with a lower priority. We can directly use this algorithm to check safety properties, but we aim at checking more advanced properties, such as LTL or CTL.

We want to extend the sweep-line method in such a way that we can check more advanced properties. We do this by constructing a condensed representation of the state space, which contains enough information to reconstruct the full structure during analysis.

Our condensed representation is simply a neighbor list representation of a graph, where we only store nodes we have arrows to in our neighbor list. Table 3.1 shows a neighbor list representation of the graph in Fig. 3.5. The first number in the parentheses indicates the number of successors, the second number will be explained shortly, and the numbers after the # indicate the successors. It is no problem to enumerate states on-the-fly in our state-space algorithm, but the problem is that we do not know how many bits are needed to store all the numbers. This is solved

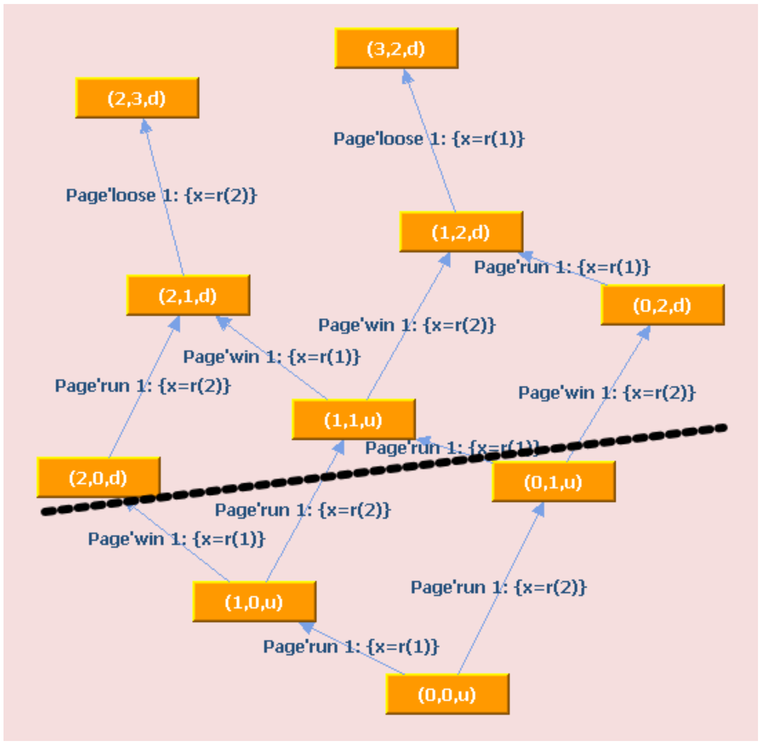


Figure 3.5: The state space of the net from Fig. 2.3 with the drink transition and Drinks place removed. We use the notation (m_1, m_2, b) for each state where m_i is the position of the runners, and b indicates the state of the flag. This figure is generated and automatically drawn using TIN-CPN and the prototypical user interface for the state space tool.

by simply using the minimum bits needed to store the number of all neighbors. We must then store this number in order to be able to decode the data afterwards. This is the second number in the parentheses in Table 3.1.

We combine this with the sweep-line method, so we do not store more than needed of S, and thereby reduce the amount of memory needed to construct the condensed representation. A simplified version of the algorithm presented in [66, Fig. 3] is presented as Algorithm 3.4. The algorithm is a simple extension of Algorithm 3.2. The first change is that we use a priority queue for unprocessed, and we garbage collect all states with lower progress measure (line 8), thus implementing the sweep-line method. Second, we need to enumerate all states. This is done by lines 3, 4, 11, and 12. In line 16 we calculate the number of the successor with the largest number, and in 17 and 18 we construct the header respectively the body of the neighbor list.

As presented here, the progress measure must be *monotone*, i.e. always satisfy $M\,[(t,b)\rangle\,M' \implies \varphi(M) \leq \varphi(M')$. The algorithm presented in [65, 66] works even if this is not always the case, by using the results from [59] at the cost of running time.

In order to validate and evaluate the performance of the new algorithm a proof-of-concept implementation has been developed. The prototype implementation of the new algorithm is slightly simpler than the algorithm described in this paper. We do not implement the variable-length numbers for node indices, but represent each index as a four byte computer word. This greatly simplifies the implementation but uses more memory for smaller systems and limits

state #	neighbor list	state #	neighbor list
0	$(2,2)\#1,2$	5	$(1,3)\#7$
1	$(2,3)\#3,4$	6	$(1,4)\#8$
2	$(2,3)\#4,5$	7	$(1,4)\#9$
3	$(1,3)\#6$	8	ε
4	$(2,3)\#6,7$	9	ε

Table 3.1: Condensed representation of the state space in Fig. 3.5. The construction algorithm has enumerated the elements from right to the left in each sweep. For each node we store the number of successors, then the number of bits used for each successor and finally the successors. States 8 and 9 have no successors, so we create no neighbor list for them.

Require: Coloured Petri net, $\mathcal{CPN} = (\Sigma, P, T, C, I_-, I_+, M_I)$, progress-measure, $\varphi : [M_I\rangle \rightarrow \mathbb{N}$
Ensure: A neighbor list, E

```
1: $S :=$ storage.insert(formalism.initial_state, storage.empty_storage)
2: $U :=$ unprocessed.add(formalism.initial_state, unprocessed.empty_unprocessed)
3: $idx(M_I) := 0$
4: $n := 1$
5: while ¬unprocessed.is_empty($U$) do
6:    $(M, U) :=$ unprocessed.pick($U$)
7:    $X :=$ formalism.get_next($M$)
8:    $S := S \setminus \{M' \in S \mid \varphi(M') < \varphi(M)\}$
9:    for all $((t, b), M')$ in $X$ do
10:      if ¬storage.contains($M', S$) then
11:         $idx(M') := n$
12:         $n := n + 1$
13:         $S :=$ storage.insert($M', S$)
14:         $U :=$ unprocessed.insert($M', U$)
15:      end if
16:      $m := max\{idx(M') \mid ((t, b), M') \in X\}$
17:      $E[idx(M)].header := (|X|, \lceil \log m \rceil)$
18:      $E[idx(M)].data := idx(M')$ for each $((t, b), M') \in X$
19:   end for
20: end while
21: return $E$
```

Algorithm 3.4: A sweep-line method for obtaining a condensed graph representation. This is a simplified version of Fig. 3 in [66].

the prototype to models with less than 2^{32} states, which is no serious limitation.

The first example we consider is a database replication protocol [48, Sect. 1.3]. The protocol describes the communication between a set of database managers for maintaining consistent copies of a distributed database. When a database manager updates its local copy of the database, it broadcasts an update request to all other database managers which then perform the update on their local copies, and then they acknowledge that the update has been performed. The progress measure for the protocol is based on the control flow of the database managers and on an ordering of the database managers. See [59] for details.

All experiments were conducted on a 1 GHz Pentium III Linux PC with 1 Gb of RAM.

Table 3.2 shows the performance of full state space generation compared with the new algorithm. The $|D|$ column shows the number of database managers in the different configurations, the following four columns show the values for the full state space, and the last four columns show the values for the new algorithm. In the full state space columns the *States* column shows the number of states for each configuration, the *Avg* column shows the average number of bytes in the state vector in the different configurations, the *Memory* column shows the total memory usage in bytes for storing all states, and the *Time* column shows the time used for calculating the state space in seconds. In the sweep-line columns the *States* column shows the number of states explored by the sweep-line algorithm, the *Peak* column shows the peak number of states stored during the exploration, the *Memory* column shows the number of bytes used for storing the states in the condensed representation plus the states in *Peak*, the number in the parentheses indicates the memory consumption of the condensed representation as a percentage of the full representation, the *Time*

	Full Reachability Graph				**Sweep-Line based Algorithm**					
$\lvert D\rvert$	States	Avg	Memory	Time	States	Peak	Memory (%)		Time (%)	
5	407	146	59,422	0	813	33	8,070	(14)	0	
6	1,460	169	246,740	0	2,919	88	26,548	(11)	1	(-)
7	5,105	191	975,055	3	10,209	251	88,777	(9)	7	(233)
8	17,498	214	3,744,572	15	34,995	738	297,912	(8)	35	(233)
9	59,051	237	13,995,087	66	118,101	2,197	993,093	(7)	155	(235)
10	196,832	259	50,979,488	286	393,663	6,572	3,276,800	(6)	665	(233)

Table 3.2: Performance of the full state space generation compared to the new algorithm for a database replication protocol. We see that the new protocol uses approximately the double amount of time but only aroundd 10% of the space.

column shows the time used for calculating the condensed graph, and the number in parentheses shows the amount of time used for calculating the condensed representation as a percentage of the amount of time used to generate the full representation.

In the database replication protocol all states but the initial state are explored twice by the sweep-line algorithm, and consequently the condensed graph has twice as many nodes as the full graph and the time for calculating the condensed graph is roughly twice as long as the time for calculating the full state space. The *Memory* in the sweep-line columns is calculated as $4 \cdot \textit{States} + \textit{Avg} \cdot \textit{Peak}$ since one computer word (4 bytes) is used for representing each condensed state and $\textit{Avg} \cdot \textit{Peak}$ bytes are used for representing the states on the sweep-line. We only compare the memory usage for storing the states, as the memory usage for storing the remaining graph structure would be comparable for the two methods. Although the unfolded graph generated by the sweep-line method contains twice as many nodes as the original state space the memory usage—as seen in the two Memory columns—is significantly improved. For five database managers the reduction is down to around 14%, while for ten database managers the reduction is further improved, down to around 6% of the full representation.

The second example is a system with little progress, namely the Dining Philosophers problem. We use the number of eating philosophers as progress measure. The performance is shown in Table 3.3. Here the *# phils* column shows the number of philosophers. The remaining columns have the same meaning as in Table 3.2.

We see that during analysis, we store nearly all reachable states with the new algorithm, so the memory used for storing the compact representation is overhead. Compared to the amount of memory used for storing the full state vectors, this amount is not significant, however, and the only real disadvantage is that we still use extra time for the construction. If the number of reachable states is close to the number of syntactically possible states, the amount of memory used for the condensed representation is comparable to the amount of memory used for the full representation, and little is gained from using the new algorithm.

To sum up the results, the new method, as expected, performs well on systems where the sweep-line method performs well, i.e. systems with a lot of progress. In the case we have seen the method uses approximately twice the time as Algorithm 3.2, but uses only 6-14% of the space. When the sweep-line method performs badly, e.g. in certain reactive systems, the method uses approximately the same amount of memory as Algorithm 3.2 but more time. In a sense, the algorithm trades space for time, so that we can analyze larger systems.

3.4 Contribution and Future Work

In this chapter we have presented the design of a new state-space tool for CP-nets. We have defined three simple interfaces, Storage, Unprocessed, and Formalism, and shown how these interfaces can be used to traverse the state space, to check safety properties, and to generate a condensed representation, which can be used for further analysis.

We have highlighted some concrete implementations of the three interfaces, and shown how we can use this design to easily test new reduction techniques.

The design and implementation of the new state-space tool has been going on for some time. My involvement is primarily with the implementation and test of various storages as well as design and implementation of the state-space wizard, which makes the tool usable for normal users as well as easier to experiment with for researchers working with state space reduction techniques.

Also, I have implemented a couple of different instances of the Formalism interface, one for CCS, mainly for making automatic tests easier to conduct, and one for Bigraphical Reactive Systems, to test the tool with a fairly new

	Full Reachability Graph				**Sweep-Line based Algorithm**					
$\|D\|$	States	Avg	Memory	Time	States	Peak	Memory (%)		Time (%)	
2	4	34	136	0	7	4	164	(120)	0	(-)
4	8	57	456	0	15	7	459	(101)	0	(-)
6	19	82	1,558	0	37	18	1,624	(104)	0	(-)
8	48	106	5,088	0	95	47	5,362	(105)	0	(-)
10	124	130	16,120	0	247	123	16,978	(105)	0	(-)
12	323	154	49,742	0	645	320	51,860	(104)	2	(-)
14	844	177	149,388	6	1,687	841	155,605	(104)	22	(367)
16	2,208	201	443,808	58	4,415	2,205	460,865	(104)	200	(344)
18	5,779	224	1,294,496	4226	11,557	5,773	1,339,38	(103)	1576	(353)
20	15,128	247	3,736,616	2643	30,255	15,105	3,851,95	(103)	9131	(345)

Table 3.3: Performance of the full state space generation compared to the new algorithm for a the dining philosophers problem. We see that the new protocol uses more than three times as much time, but only uses slightly more memory.

formalism. The implementation of state-space analysis for Bigraphical Reactive Systems is also interesting in itself, because, to my knowledge, it has not been possible to do state-space analysis of these systems, until now. and

In part B of my PhD study, I expect to devote most of my time to the state-space tool. One experiment I want to do is to compare the sweep-line method to other well-known reduction techniques, such as bit state hashing and hash compaction [84, 97].

Chapter 4

Model-based Prototyping and Animation

A model of a (computer) system can be compared to drawings and three-dimensional models of new buildings made by architects. A model of a computer system is intended to capture the essentials of the final system, but at the same time be simple enough to allow analysis, e.g. using the state-space method, as described in Chapter 3. An architect will usually first create a drawing of a new building, then create a three-dimensional model, and finally build the building. In the same manner, a computer system might be built by creating a system description as a document written in natural language, then creating a model, a formal, executable, and verifiable description capturing the essentials of the system, and finally implement the system.

Although intriguing, this approach is rarely used in practice. In a real setting, one will often view the modeling step as a waste of time, whereas in an academic setting, one is often not interested in the final product, and therefore stops after the modeling. Even if one goes through both the modeling and implementation of the system, there is no real guarantee that the model really captures the final system; often one has to make slight changes to overcome unforeseen difficulties. These changes are seldom reflected in the model, as one has completed that part of the development process. We may still gain insights from creating the model, but we cannot use the model for verification, as the model no longer reflects the final system. One could try to repair the development model by using a waterfall-like approach, where we go back and update the design documents, such as the model, but this approach is not well tailored to interactive applications [10] and when deadlines approach, going back to repair design flaws is often neglected.

There are, however, a number of areas where modeling has been very successful: within the area of verifying computer processors [26] and within the area of verifying protocols, either computer network protocols [33,34,58,72] or protocols for human interaction or workflows [89]. One reason that the model-based approach is successful here is that the end-product is not an advanced piece of software, but rather the specification, and the model can be viewed as a test rather than an unused intermediate step. Another successful use of models is to observe the success for application of models for protocols, and try to use models for test rather than specification. One then writes a specification, makes an implementation and finally extracts a model (semi-)*automatically* from the final product. The automatic extraction eliminates human errors in the modeling or translation, and errors in the model can often be traced directly back to the software product. This approach is used in various projects, such as Modex [41, 42, 71], a model extractor for C code, or to check contracts for device drivers in Microsoft Windows using SLAM [1,81].

We propose a new way to use models in the software development process. Firstly, we will not look at software in general, but rather at the software most commonly developed, namely software where a user interface shows and manipulates a data model. Such software is designed using the Model-View-Controller design pattern [32]. Thus software often consist of mainly two or three parts, a user interface (the view) and a representation of data we want to manipulate (the model) and how we want to manipulate it (the controller). We will often combine the model and the view, as objects in object-oriented programming languages contain data and methods to manipulate the data. Secondly, we observe that while modeling in the sense we use here, a formal, executable, and verifiable description of a software product is rarely used, a more relaxed version is often applied in practise, namely a prototype. A prototype is written with two things in mind: to discuss the user interface with the user and to understand and capture the logic that needs to be used by the final product. In [60] we give an example of this approach.

The visual part of a prototype can easily be built using a GUI-builder, such as Eclipse [28] or Visual Studio [91]. This part of the prototype can often be used in the final application as well, so this expense is easy to justify. The business logic, or workflow, is often programmed using hastily written code that "works". This code is rarely usable when the prototype is to evolve into the final product, and the expense to write this part is only justified by the need to understand the workflow that needs to be built into the application.

We propose to model the data model part of the prototype using a formal modeling language, such as PROMELA [42, 83], timed automata [5], workflow nets [88, 89], or Coloured Petri nets [48], rather than making a coarse implementation in a standard programming language, such as Java or C#. This has a number of advantages: Firstly, it is often easier to express the data model and the operations that one can make on it, using a higher level language. Secondly, the model may now actually be the final implementation, either by just simulating the model and let the user interact with it, or by automatically generating executable code(-templates) that is guaranteed to capture the logic of the model. Finally, as the model is now closer to the actual implementation, it is now possible to make analysis of the model and directly connect it to the final implementation.

In the paper [60] we describe an industrial case where we build a “prototype” of a network protocol. The prototype is created using a tool, TIN-CPN, which I have designed and implemented during my PhD work. I also participated in the industrial case described in [60]. I mainly created the animation package and the animation we used in the project as well as tying it to the model, but I also participated in various parts of the modeling phase. A lot of the design ideas for the animation tool have evolved from the concrete problems of the industrial case. As part of the design of the animation tool, the idea of a model-driven prototype, was conceived. Presenting the state of a model in a user-friendly way is by no means a new idea [11, 12, 62, 78], but the idea of a model-driven prototype of standard computer software, where the model can be replaced by a real implementation and the view by a test-driver, is.

Figure 4.1 shows the approach taken to use CPN models to develop a prototype. A CPN model (lower left of Figure 4.1) has been developed by modeling the natural language protocol specification (lower right) of the system in question. The modeling activity transforms the natural language specification into a formal executable specification represented by the CPN model. The CPN model captures the essentials of the system we want to implement. The resulting model can already be viewed as an early prototype since it is possible to execute and experiment with the protocol at the level of the CPN model. Since CP-nets is a graphical modeling language, it is possible to observe the execution of the model directly on the CPN model.

The CPN model provides a very detailed view on the execution of the system and it can be an advantage to provide a high-level way of interacting and experimenting with the prototype. Furthermore, when presenting the protocol design to people not familiar with CP-nets, it can be an advantage to be able to demonstrate the prototype without directly relying on the CPN model but more on application and domain specific concepts, such as (images of) computers and routers. To support this, an animation GUI (top left of Figure 4.1) has been added on top of the CPN model. The animation GUI visualizes the execution of the model, by updating the graphics according to the state of the model.

Related Work ExSpect [85], a tool for modeling based on CP-nets, allows the user to view the state by associating widgets with the state of the model, and asynchronously interact with the model, also using simple widgets. In this way, it is easy to create simple user interfaces that support displaying information, but support for creating full applications is not easily available.

MIMIC/CPN [77] makes it possible to animate models within DESIGN/CPN [17, 25], which is another tool for modeling using CP-nets. CPN models are animated by MIMIC/CPN by using function calls that are executed whenever a transition of the CP-net occurs. The animations are drawn using an application that resembles traditional drawing programs. Input from the user is possible by showing a modal dialog, where the simulation of the model is stopped while the user is expected to input information. It is also possible to make click-able regions, and the model can then query if one of these has been clicked.

LTSA [63], a tool for modeling using timed labeled transition systems, allows users to animate models using a library called SceneBeans [64, 80]. In LTSA animations are tied to the models by associating each animation activity with a clock; resetting a clock corresponds to starting an animation sequence. The animation sequence or a user with his mouse can then send events which correspond to the progress of the timer.

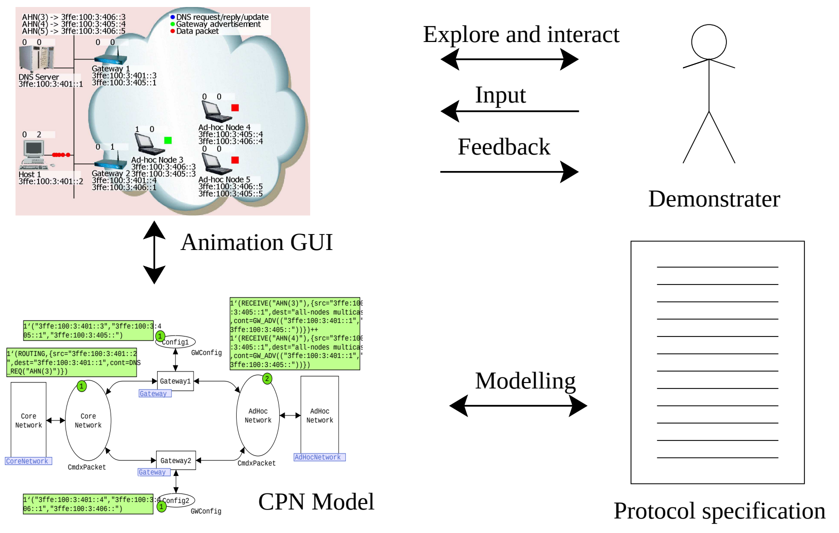


Figure 4.1: Model-based prototyping approach.

Another approach, taken by e.g. the COMMS/CPN [31] library for DESIGN/CPN and CPN Tools, is to provide a TCP/IP [44] abstraction, allowing the user to code the user interface in any language and use RPC to communicate with it. This approach resembles creating real programs quite a lot, but the user has to go through the hassle of implementing RPC himself, making this approach difficult to use in practice.

PNVis [53] is an add-on for the Petri Net Kernel [93], a highly modular tool for editing Petri nets. PNVis associates tokens with 3D objects and certain places with locations in a 3D world. Moving tokens corresponds to moving the associated object in the 3D world. PNVis is suitable for modeling physical systems, but not really useful for creating prototypes of software.

The Play-Engine [35] supports the developer in implementing a prototype by inputting scenarios (play-in) via an application-specific GUI, and then executes the resulting program (play-out). Compared to our approach this makes the model implicit as the model is created indirectly via the input scenarios. In a sense, we create a prototype via direct manipulation, but as the model of the system is created indirectly via the input scenarios it may be difficult to use the model for analysis and as basis for implementation of the final system. The reason is that an implicitly created model is difficult to interpret as it is automatically generated.

We have mentioned a number of libraries, all of which support animation in different ways. Using some libraries, animation is integrated with the modeling formalism, such as the use of timers in LTSA or the ability to view or change the marking of places in ExSpect. Some libraries are easy to extend, such as animations in LTSA, as the SceneBeans library allows users to easily extend it with new animation primitives. Also, animations created using COMMS/CPN can easily be extended, as the "animation" is just a custom (Java) application. Some libraries make it easy to design animations, such as ExSpect and MIMIC/CPN, which both provide a graphical user interface to design animations. All of the libraries are quite low-level, however, and the resulting animation is just that, an animation, and does not really resemble a real software prototype, except for animations created using COMMS/CPN, but in this case you have to code the RPC needed to drive the animation yourself.

The designed animation package aims at providing the good features from all these libraries, and successfully does so. In the rest of this chapter, we will first describe our design of the TIN-CPN animation package, BRITNeY animation, and we will then turn to the description of the industrial case.

4.1 BRITNeY Animation

As part of my PhD part A, I have implemented a tool to support model-driven prototyping and animation.

The earliest version was inspired primarily by COMMS/CPN and MIMIC/CPN. From COMMS/CPN we borrowed the idea of communicating with an external application using some kind of RPC and from MIMIC/CPN we borrowed the idea of making animations using simple procedure calls on transition occurrence. The first version was just a Java application able to draw simple geometric figures based on input on a TCP socket.

The next version switched from a proprietary RPC protocol to the XML-RPC [96] standard protocol. Also, this version introduced the concept of animation objects, a predecessor of animation plug-ins. Animation objects makes it possible to extend the animation tool with new, often higher-level, kinds of animations, such as message-sequence charts or Java dialogs. Animation objects had to be compiled together with the animation tool, making it difficult for other people that the BRITNeY programmers to create new animations such as custom application dialogs.

In the third version, animation objects were put into plug-ins, thereby enabling users to easily create plug-ins themselves. More than 10 animation plug-ins have already been developed, e.g. a plug-in to draw message-sequence charts, a plug-in to draw directed graphs in 2D and in 3D, a plug-in to display simple messages to the user, a plug-in to get simple input from the user, a plug-in to generate different kinds of charts, a plug-in integrating the SceneBeans animation library of LTSA, and a couple of more specialized animation plug-ins. Plug-ins can be created with as few or as many dependencies on services offered by the animation tool as desired. That is, a new animation can be no more than a dialog designed in your preferred GUI builder, or it can be integrated in the animation tool, thereby allowing the user to e.g. print the current state of the animation. The overall architecture of this third version can be seen in Fig. 4.2.

The main idea behind this animation package is the Model-View-Controller (MVC) design pattern, where the CPN model is the MVC-model, the animation is the MVC-view, and the model executor is the MVC-controller. One of the main benefits of using this design pattern is that in principle we can replace any component, and e.g. replace the MVC-controller/MVC-model with a real implementation, which facilitates a model-driven prototype approach. As explained in Sect. 4.1.1, it is also possible to remove the view for analysis.

4.1.1 CPN Tools Specific Enhancements

The abstract view of the animation tool given in Fig. 4.2 is formalism-independent. The animation tool has been designed to seamlessly interact with CPN Tools, and a view of this interaction can be seen in Fig. 4.3. An important thing to notice is that we have added an XML-RPC client to the simulator as well as some stubs [21, Sect. 5.3], i.e. functions encapsulating the RPC calls to the animation tool. This is the standard approach to RPC.

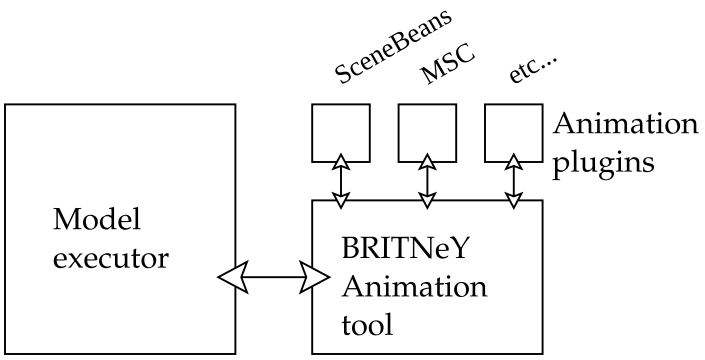


Figure 4.2: The conceptual architecture of the animation tool. On the left, we see the model executor, this can be a CPN simulator or any other application that understands XML-RPC. On the right, we see the animation tool, connected with a number of animation plug-ins.

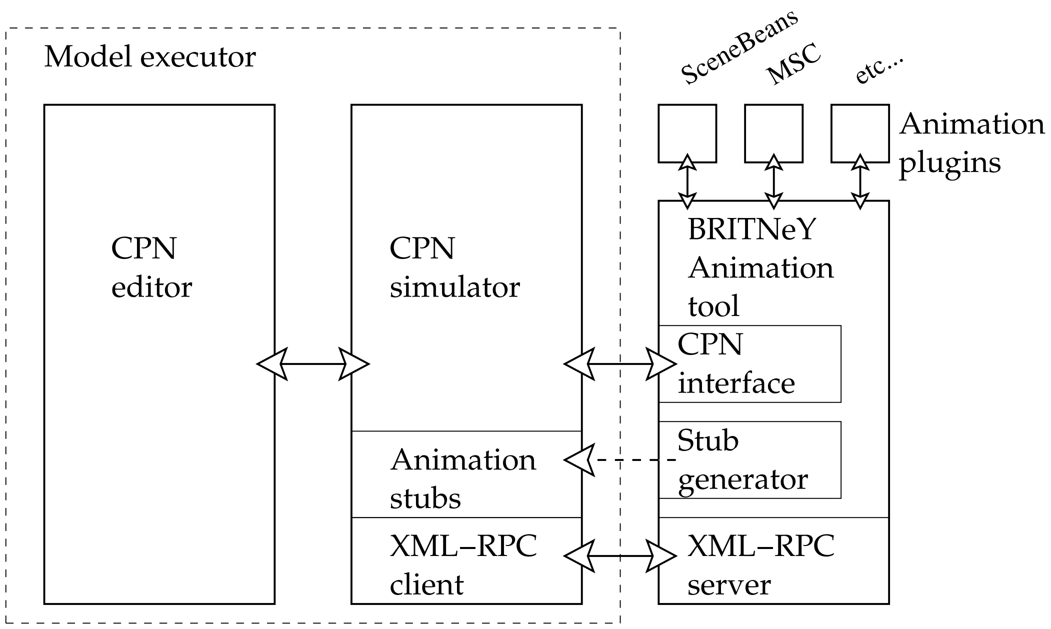


Figure 4.3: A more detailed view of the integration of the animation tool with CPN Tools. Compared to Fig. 4.2, we have a more detailed view of the model executor. Firstly, the executor is split up into two parts, an editor and a simulator. Secondly, we see that the simulator communicates with the animation tool using an XML-RPC client. Furthermore, a number of stubs for the functions in the animation tool exist in the simulator to abstract away the RPC mechanism. We also see two subcomponents of the animation tool, namely a stub generator and a CPN interface.

Automatic Stub Generation

As the functionality of the animation tool can be extended by plug-ins, we may have to dynamically add more stubs to the simulator. In order to make this as simple as possible for the user, this is done automatically.

If we open up the animation tool a bit, we see two subcomponents of the animation tool. One is the Stub generator. This uses Java reflection [47] to inspect the registered plug-ins and generate a stub to access the functionality. This stub is then loaded into the simulator, as indicated by the dashed arrow.

In order to keep the animation tool as independent of CPN Tools as possible while still providing this convenient feature, the stub generator is only invoked if the animation tool detects that the animation is driven by CPN Tools.

CPN Interface

A somewhat subtle problem with directly using the RPC calls in our model, is that the functions are directly tied to the animation, as we explicitly call the functions in the animation tool. This ties our model to the animation, and makes runs without the animation difficult or even impossible. We would rather use mechanisms closer to the model to drive the animation, much like LTSA, where animation "calls" are modeled as operations on clocks, a natural concept in timed labeled transition systems.

We can use synchronous channels as described in Sect. 2.2.3 to simulate calls to the animation tool. If we simply create a send-transition to send the parameters to an animation plug-in followed by a receive transition to retrieve the result, we can communicate with the animation tool by simply sending data on channels. By reversing the order of the sender and the receiver, it is also conceivable that communication is initiated from the animation. This approach provides several advantages. Firstly, we abstract away the concrete animation tool. Secondly, and perhaps more important, we can now remove the animation and instead add a test-driver, e.g. modeled in CPN as well. All the driver has to do is to listen on the correct channels and stimulate other channels. This would enable analysis of the same model that we use to drive our animation. Finally, if we wish to turn off animation altogether, we can just make empty channel endpoints. These can even be auto-generated.

Recall from the introduction that in ExSpect animations are views of places and can affect the model by adding tokens to specific places. The dual concept of synchronous channels is fusion sets, where two or more places share the same tokens. If we allow the animation tool to participate in fusion groups, we can achieve exactly the same behavior as in ExSpect. We add an animation tool equivalent of a member of a fusion set, and allow other objects to register themselves as observers [32] of this member or to modify the member. This provides functionality much like that of synchronous channels, i.e. animation tool independence and ability to easily construct test-drivers, but now the communication is asynchronous rather than synchronous.

As mentioned earlier, animation plug-ins can be as dependent on the services provided by the animation tool as they desire. If a plug-in wants to use the CPN interface, it must state so, and it can then only be used when a CPN simulator is present. Other animation objects may not need these features and can be used without a CPN simulator, thus keeping the animation tool independent from CPN Tools.

4.2 Case Study: Telebit Animation

We present a case study from a joint research project [56] between the Coloured Petri Nets Group [22] at the University of Aarhus and Ericsson Danmark A/S, Telebit [29]. The research project applies formal methods in the form of Coloured Petri Nets and the supporting CPN Tools in the development of Internet Protocol Version 6 (IPv6) [43] based protocols for ad-hoc networking [73]. An ad-hoc network is a collection of mobile nodes, such as laptops, personal digital assistants, and mobile phones, capable of establishing a communication infrastructure for their common use. Ad-hoc networking differs from conventional networks in that the nodes in the ad-hoc network operate in a fully self-configuring and distributed manner, without any preexisting communication infrastructure such as base stations and routers.

Here we will describe a particular case study I have participated in. In this study CP-nets have been used for the specification of an interoperability protocol for routing packets between fixed core networks and mobile ad-hoc networks. The interoperability protocol ensures that a packet flow between a host in a core network and a mobile node in an ad-hoc network is always relayed via one of the closest gateways connecting the core network and the mobile ad-hoc network. [60] shows how integrated use of CP-nets and BRITNeY animation have been applied to build a model-based prototype of the interoperability protocol. The prototype consists of two parts: a CPN model that formally specifies the protocol mechanisms and a graphical user interface for experimenting with the protocol.

Figure 4.4 shows the hybrid network architecture captured by the model-based prototype. The network architecture consists of two parts: an IPv6 core network (left) and a mobile ad-hoc network (right). The core network consists of a Domain Name System (DNS) Server and Host 1. The mobile ad-hoc network contains three mobile nodes (Ad-hoc Node 3-5). The core network and the mobile ad-hoc network are connected by Gateway 1 and Gateway 2. A routing protocol for conventional IP networks (such as OSPF [61]) is deployed in the core network and a routing protocol for ad-hoc networks (such as OLSR [20]) is used in the mobile ad-hoc network. The purpose of the interoperability protocol is to ensure that packets are routed between hosts in the core network and nodes in the mobile ad-hoc network via the closest gateway. For a description of the protocol and how it is implemented, please refer to [60].

The domain-specific GUI makes it possible for the user to observe the behavior of the system and to provide stimuli to the protocol. The use of an underlying formal model is completely hidden when experimenting with the prototype. The domain-specific GUI has been used in the project both internally during protocol design and externally when presenting the designed protocol to management and protocol engineers not familiar with CPN modeling.

We have also included a dynamic generation of message sequence charts, which is a formalism well-known to many protocol engineers. To do this, I implemented an animation plug-in for BRITNeY animation, which is now generally available. A log of how we arrived at the situation in Fig. 4.4 can be seen in the message sequence chart in Fig. 4.5. First the model decided that Gateway 1 should send out advertisements to all Ad-hoc Nodes. Then the user decided to move Ad-hoc Node 3 (simply by dragging the image to its new location). The model now makes Gateway 1 and then Gateway 2 send out advertisements, which causes Ad-hoc Node 3 to update its IP-address and send an update to the DNS Server. Now the user wants Host 1 to transmit data to Ad-hoc Node 3, and clicks on the square next to it. This makes Host 1 look up Ad-hoc Node 3 in the DNS Server, and send the data. In Fig. 4.4 we see the data in transit from the host as dots on the network.

When we started the project, BRITNeY was still in the very early stages, and only supported RPC-based animation facilities. During the project, this proved to be insufficient. For example, we would like the animation to automatically reflect some information about the model, such as the DNS database. We solved this by manually updating the view of the DNS server each time we made a change. This would have been much easier to obtain by just making a view on the place modeling the database. Also, a lot of effort was spent on making somewhat asynchronous input from the animation, such as movement of mobile nodes. Our implementation used a transition to poll the animation, but this would have been much easier, had we just been able to directly add tokens to a place in the model.

Another observation we made during the project was that after integrating the model with the animation, we could not easily turn off the animation and just work with the model. We believe this would be possible, had we abstracted away the direct function-calls from the model and just used synchronous channels.

The project demonstrates that the use of formal modeling combined with the use of domain-specific visualization can be an effective approach to rapidly construct an executable prototype of a communication protocol.

Deploying Animations

Another problem that we also discovered during the project was that deploying the prototype to the industrial partner was quite tedious. We had to give them quite complex instructions on how to load and simulate CPN models using CPN Tools. This is of course not a desired situation, and therefore a lot of work has gone into easing the deployment of animations after the project.

Firstly, it is now possible to run a simulation of a CPN model without ever starting the CPN user interface. BRITNeY animation is able to load and start a model itself, using only the CPN simulator of Fig. 4.3. This is a great advantage as users no longer have to get a long manual on starting CPN Tools, loading a model, preparing to start the

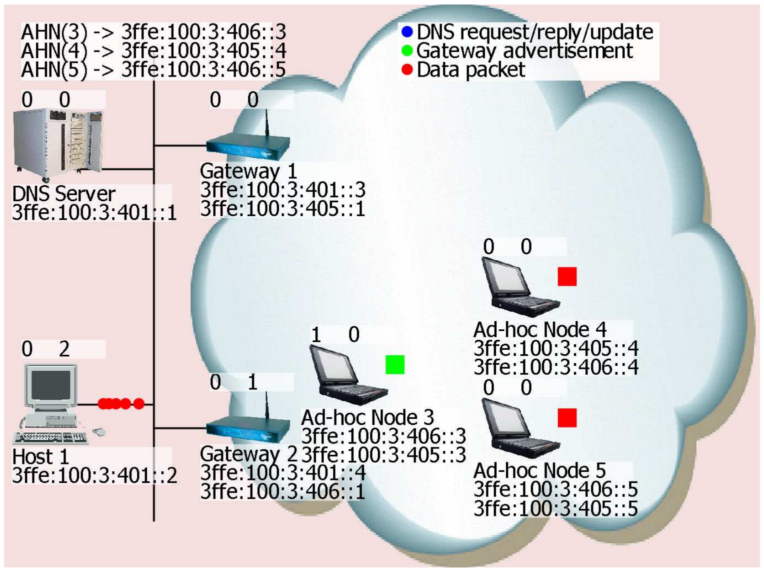


Figure 4.4: The Hybrid Network Architecture.

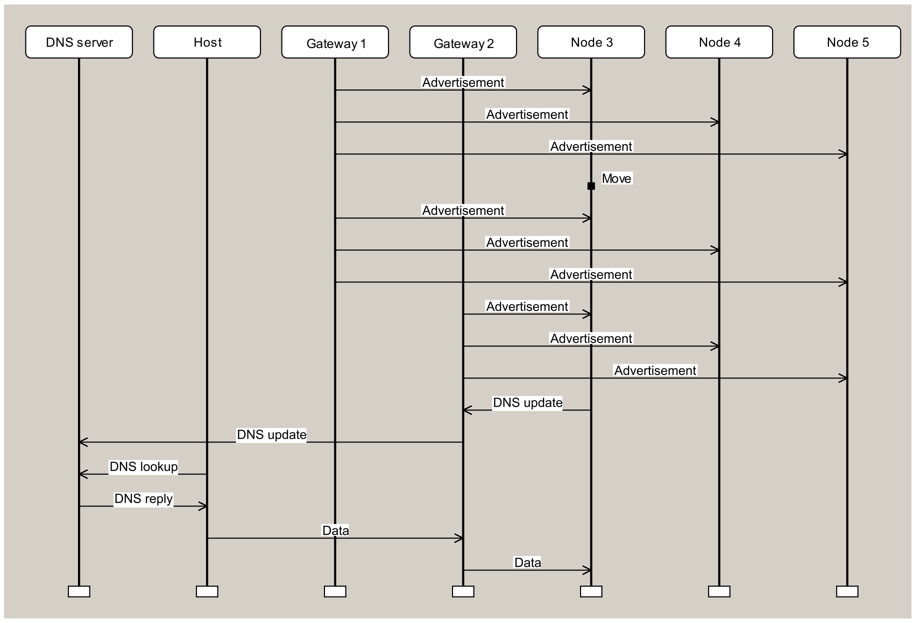


Figure 4.5: Another view of the network in Fig. 4.4. We here see all messages transmitted by the different nodes. This trace shows how we arrived at the situation in Fig. 4.4.

animation, starting the animation and finally interacting with the animation. As the only reason BRITNeY has to do simulation is to drive an animation, we can simplify the loading of a model to a single click on a button.

Secondly, BRITNeY animation supports deployment based on Java Web Start [46], a relatively new technique for deploying Java applications that makes it possible to install and run a new application simply by clicking on a link in a browser. The technology will then check whether the correct version of Java is available and offer to download upgrades if needed. It is definitely easier to just point users to a web page, optionally with some instructions on using the prototype, than to distribute CDs and explain how to use them.

4.3 Contribution and Future Work

In this chapter, we have summarized a number of facilities for animating formal models. We have also described the design of an application supporting animation of CP-net models, and how this tool has been used in an industrial case.

My contribution to visualization of formal models consists of two parts. Firstly, the implementation of BRITNeY makes it possible for users of CPN Tools to create a graphical user interface for their models. Second, the idea of creating prototypes of software, allowing some parts of the Model-View-Controller design pattern to be driven by formal models is new. This approach allows developers to rapidly develop rough prototypes, and through step-wise refinement of the model an prototype to get a formal model of the produced software, and to reuse parts of the prototype in the final product. In some cases it may even be possible to use the model as the implementation or to automatically derive (a skeleton of) the final implementation from the model. The Play-Engine supports an approach similar to this, but the derived model may be difficult to interpret to obtain the final implementation, making this approach less desirable.

My work on BRITNeY is almost complete. It is being cleaned up for a general release, and will then be handed over to the student programmers currently maintaining CPN Tools.

I expect I will use features of BRITNeY in the future, as BRITNeY supports drawing and automatic layout of directed graphs, which has already proven itself a valuable tool for experimentation with the new state-space tool.

Chapter 5

Towards an Exchange Format for Coloured Petri Nets

The ability to exchange Petri net models between different tools has several benefits; among the most important benefits is the ability to make tools that focus on only one aspect of editing, simulation, or analysis. Another important benefit is the possibility of creating a library of common models, which can easily be used when explaining and benchmarking a new analysis method. In order to support such an exchange, a common exchange format should be agreed upon.

The Petri net Markup Language (PNML) [7, 92] provides a meta model of Petri nets and a framework to specify formats. In order to specify an exchange format, the implementer has to write a Petri net Type-Definition (PNTD), which specifies the allowed labels for each type of element, i.e. the allowed annotations, such as names of places and transitions, arc annotations, place types, and initial markings. The PNML framework specifies an eXtensible Markup Language (XML) format, which we shall use as our exchange format.

In the paper [95] we propose and exemplify such a format for high-level Petri nets, such as CP-nets. My contribution to this area is in part the paper [95], as it documents a concrete way to exchange high-level Petri nets and it gives translations of several common composition mechanisms to the module concept of PNML. Furthermore, I have participated in several standardization meetings, and have, along with other members of the CPN group, pushed the standard towards allowing a mixed format for annotations, as shall be motivated and explained later. In this chapter, we will present our work on the concrete proposal.

The PNTD for PT-nets (see Fig. 2.2 on page 4 for an example of a PT-net) is very simple, and except for names of places and transitions, only specifies that places can have an initial marking and that arcs can have an arc annotation. Both initial markings and arc annotations are simply natural numbers. A PNTD for High-level Petri nets (HLPN) must not only specify more labels such as declarations of variables and functions, place types, and transition guard functions, the described labels are also more complex as places can have arbitrary types.

Apart from more advanced labels, CP-nets also allow the modeler to construct models by combining smaller components. We have earlier seen fusion places, in Sect. 2.2.2, and modules communicating using synchronous channels, in Sect. 2.2.3. Both of these constructs (as well other composition mechanisms) can be expressed by adding more labels to the involved places or transitions, e.g. a label for each place indicating the fusion set it is a member of, but PNML already provides a generic module concept, which can be automatically flattened to an equivalent net without modules [54]. By translating to this module concept instead of introducing more labels, it is possible to exchange models between tools understanding elaborate composition mechanisms and tools that only understand flat nets – we still lose some information about the structure, but we preserve the semantics, which is sufficient for e.g. analysis.

5.1 Arc Annotations

In this section, we first describe the main problems with labels, provide solutions for them and exemplify the proposed solution for arc annotations. Labels are the textual annotations of places (e.g. names or types), transitions (e.g. names), arcs (arc annotations), and entire Petri nets (e.g. declarations of types).

A major problem when trying to exchange labels is that different HLPN tools use different annotation languages. For example, to move “one token with value 2 and four tokens with value 7” to or from a place in CPN-AMI or CPN Tools one would use the syntaxes shown in Fig. 5.1. Although the concrete syntax differs, the labels in Fig. 5.1 clearly express the same values and can be exchanged if, instead of exchanging concrete syntax, the tools exchange abstract syntax trees (AST). The problem with similar annotation languages with different concrete syntaxes has been illustrated here using arc annotations, but the problem also arises for other kinds of labels. For more examples, refer to [95].

`<2> + 4*<7>`

(a) CPN-AMI

`1‘2 ++ 4‘7`

(b) CPN Tools

Figure 5.1: The multi set "one token with value 2 and four tokens with value 7" in the syntax of the tools CPN-AMI and CPN Tools.

Even though exchanging everything as ASTs solves the problem with exchanging annotations with different concrete syntaxes, other problems arise. Adding new features to the formalism becomes quite cumbersome because the exchange format has to be changed and all tools have to support the new format. It would be desirable if implementers could easily try out a new feature and, if it persists, the entire community could make an orderly change to the standard rather than encourage implementers to make dozens of independent and incompatible ad-hoc "improvements" as has happened within the Hyper-Text Mark-up Language (HTML) community, where e.g. the introduction of frames was prompted by a vendor-specific "improvement". Therefore we propose allowing, at any point in the saved AST, a leaf containing verbatim text—this leaf can then be used to save concrete syntax when needed. Allowing verbatim text in the AST also offers an important advantage, namely that even though the format is meant to be an exchange format for correct nets, it is also possible to use the format as primary storage format; it is highly likely that a modeler may want to save a net halfway through the modeling, but at this point some annotations may not be syntactically correct and therefore not parsable, so an AST cannot be constructed. If one insists on saving ASTs only, such an erroneous label cannot be saved, but it is easy to save such an annotations as verbatim text.

In Kindler's proposal for ISO/IEC 15909 part 2 [52], a cleaner cut between structured and unstructured labels is suggested: a label is either completely unstructured or completely structured. The advantage of not requiring a clear-cut separation is that it makes it possible to maintain more structure even when using an element that has not yet been standardized. For example, in Fig. 5.2 lines 5-14, we are able to represent a multi set of an integer and a value computed by a function using abstract syntax even though the value of a function application is not yet standardized. If multiple tool implementors decide that function application in multi sets is useful, it is still easy to exchange nets between these tools—this exchange would be difficult had it not been possible to make a partially structured label. Furthermore, allowing partially structured labels does not add any extra complexity to the implementation of tools, as it is very easy to write an eXtensible Stylesheet Language Transformations (XSLT) style-sheet capable of translating (partially) structured labels to unstructured labels.

We also notice that we will have to allow storing labels *both* in concrete and abstract syntax. The abstract syntax is then used for exchanges, and the concrete syntax can then be used by tools to preserve formatting and comments, information that is normally stripped from the abstract syntax. We could just add AST-nodes for comments and preserve formatting by saving positions within the AST-nodes, but this approach is much more complicated and nothing is gained compared to just storing the concrete syntax as well.

Figure 5.3 shows the proposed grammar for multi-set expressions. Line 1 defines arc annotations, line 2 defines initial markings, which can also be expressed using multi sets, and lines 3-8 define multi sets. A multi set is either a value (representing one token with the given value) or a list of cardinality/value pairs. The definition of a value is especially worth noticing. At the moment it only specifies unstructured elements, but it is designed so that future extensions are easy to make, if e.g. one wants to add types such as integer, strings, or enumerated elements. The reason that specific types have not been defined yet is that not all types are well-defined.* If one wants to define data-types, one should take a look at how XML Remote Procedure Call (XML-RPC) [96] or XML Schema [8] handles data-types. Another possibility is to allow more general expressions stored as an AST.

Various tools use different abbreviations. For example, CPN Tools allows specifying the multi set "one token with value 2" as just `2` (as opposed to `1‘2`) and CPN-AMI allows specifying "one black token" using no arc annotation. For exchange to be possible, such abbreviations must be standardized or expanded by the tools. We propose that all abbreviations are removed and that black tokens are not special and must be explicitly declared.

In [95], in addition to the treatment of multi sets, we also give an abstract syntax for declarations (type and variable definitions), types of places and guards of transitions, but the general idea is illustrated in the above examples and we shall therefore not treat these further in this report.

5.2 Fusion Places

In this section, we describe how fusion places can be realized using Modular PNML [54]. To see how other compositional constructions (hierarchical Coloured Petri nets and synchronous channels) can be described using Modular PNML, please refer to [95]. The construction used here is not unique to high-level nets and can easily be used to also introduce advanced composition techniques in other net types, e.g. PT-nets, without breaking compatibility with tools without the features, thanks to the automatic flattening.

*What is e.g. an integer, n? In Java it is $-2^{31} \leq n < 2^{31}$ and in Standard ML it is $-2^{30} \leq n < 2^{30}$

```
1  <annotation>
2    <text>#&amp;*@</text>
3  </annotation>
4
5  <annotation>
6    <multiset>
7      <value cardinality="1">
8        <text>2</text>
9      </value>
10     <value cardinality="3">
11       <text>fac(6)</text>
12     <value>
13   </multiset>
14 </annotation>
```

Figure 5.2: Representations two different arc annotations in XML format. At the top, we see a malformed and therefore un-parsable arc annotation (lines 1-3), and the arc annotation “one token with value 2 and three tokens with the value computed by the `fac` function with the parameter 6” (lines 5-14).

```
 <Arc Inscription> ::= <Multiset> | <Unstructured>
 <Initial Marking> ::= <Multiset> | <Unstructured>
        <Multiset> ::= <Multiset Element>* | <Value>
                     | <Unstructured>
<Multiset Element> ::= <Cardinality> <Value>
           <Value> ::= <Unstructured>
     <Cardinality> ::= <Non Negative Integer>
                     | <Unstructured>
```

Figure 5.3: Grammar for initial place markings and arc annotations.

We want a construction that preserves the semantics of fusion places and preserves as much information as possible about the structure (we could just flatten the entire net and replace the fusion set with a single shared place, but we would then lose the information about the structure). The intuition of our construction is to construct a page, which is a kind of module, for each fusion set, containing only a single globally accessible place. We then translate each participant into a reference to the single global place. Global nodes can be referred to from every module of the net, nicely modeling how we can think of fusion places: a single real place, and a number of participants referring to it. All of these concepts are part of Modular PNML. Figure 5.4 shows how this construction would look like for the net in Fig. 2.4.

This approach has several advantages. When loading a net with fusion places, even in a tool that does not support fusion sets nor Modular PNML, adding and removal of participants to the group is quite easy. Adding a participant amounts to just making a reference (to the place representing the fusion set), whereas removal amounts to removing the reference. This resembles what one would do in a garbage collected programming language: we would just remove the reference to the fusion set and have the garbage collector remove the fusion set when no other participant refers to it.

One might think of other ways to model fusion places if we want to avoid the overhead of a new globally accessible place. For example we could make one participant canonical by converting it to a real place and have the other participants refer to it. This introduces a lot of new overhead though. For example, when the canonical place is removed, we need to make another participant canonical and globally update the references. This also makes removal of the last participant from a fusion set cumbersome, as we have to detect that no other participants exist. Creating a new place and having all others refer to it would solve these problems, but would create problems if the page it resides on is deleted, which brings us back to having to create a special page as described above. This possible but deprecated construction can be seen in Fig. 5.5.

5.3 Contribution and Future Work

In this chapter, we have motivated the need for a common exchange for CP-nets and pointed out some of the main difficulties in designing such a format, namely how to exchange complex labels and how to express compositional

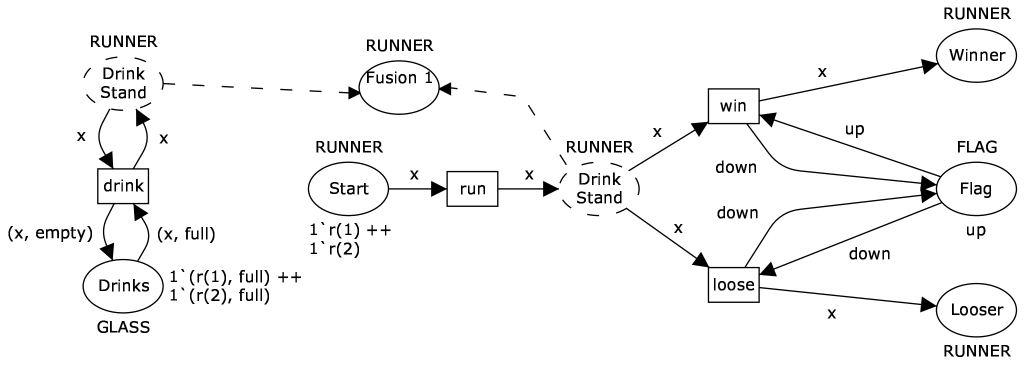


Figure 5.4: How the net in Fig. 2.4 would look like if stored using the proposed Modular PNML construction. The change is that we have added a new place, Fusion 1, with the name of the previous fusion set, and turned all other members of the fusion set into references that simply point to the new place. The dashed ovals are not real places, but merely references that point to a real place. To simplify the drawing, the real Fusion 1 place is placed in the same figure, even though it should be placed in a seperate module.

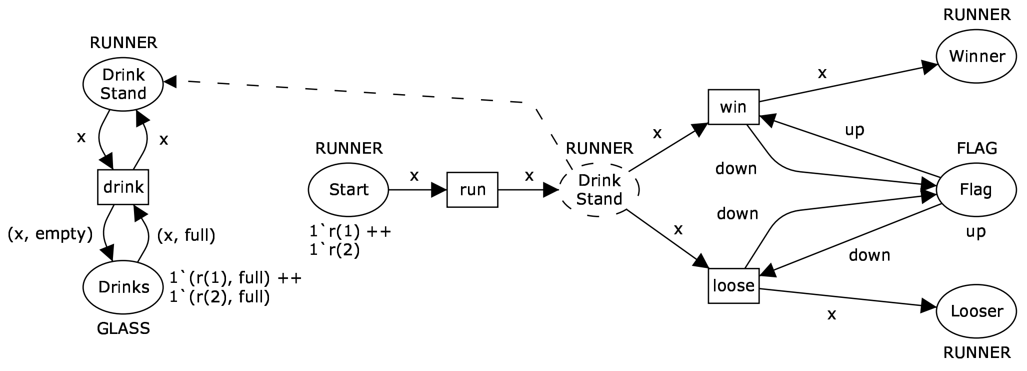


Figure 5.5: How the net in Fig. 2.4 would look like if stored using the deprecated Modular PNML construction. The change is that we have removed both DrinkStand places from the fusion set and turned the right-most place into a reference to the left-most place. The problem with this construction can be seen by imagining that the left-most DrinkStand place is deleted.

constructs as simple as possible. We have also proposed solutions to the problems.

My contribution is two-fold: Firstly, I have given a concrete example of how to exchange the different kinds of labels of Petri nets and, secondly, I have presented a translation of one of the most common composition mechanisms of CP-nets into the existing module concept of PNML, thereby demonstrating the module concept of Modular PNML is indeed a good starting point for an exchange format.

The idea of using abstract syntax to exchange labels is not mine, but the idea of mixing abstract and concrete syntax is new. My main contribution regarding labels is a concrete starting point for the standard, which proved to ignite interesting discussions at the workshop where it was presented.

My translations from compositional constructions typically used in tools to the format of Modular PNML have at least two benefits. Firstly, it simply shows that it can be done, which proves the currently standardized way is “good enough”. Secondly, the constructions used can be used directly in tools, and I have already conducted some work on that, namely a translation from the native format of CPN Tools to the proposed format as well as a prototypical loader for TIN-CPN. This is also mentioned in [95].

This work is long-running, and most of the foundations are in place, so the missing pieces mainly require discussions with people spread around the world, which is a rather slow process. The standard is getting close to a first draft, meaning only details will change. I expect to continue finishing the rough edges of the current representation, but I do not expect to use a lot of energy on the subject.

Chapter 6

Conclusion and Future Plans

In this report I have introduced work in three different areas: state-space analysis, visualizing model execution, and exchange formats for high-level Petri nets.

I have introduced a standard analysis method, the state-space method. I have shown how this can be defined for CP-nets, and how to construct the state space. I have identified the interface of three data structures used during the construction, and I have argued that different implementations of these three interfaces can be used to make different kinds of traversals of the state space, to make different kinds of reductions of the state space, and to analyze entirely different modeling formalisms. My main contributions in this area are:

- participation in design and implementation of a new method based on the sweep-line method to generate a very efficient representation of the state space [66],
- implementation of bit-state hashing and Bloom-filtering for CP-nets,
- design and implementation of state-space analysis and a simple logic for bigraphical reactive systems, and
- design and implementation of a graphical user interface to facilitate easy use of the new state-space tool.

I have also introduced an approach to building prototypes using a model and a visualization tool. I have conjectured that using a model to drive the prototype may improve the final software product, in part because now a formal model is actually built, but also because it may be possible to provide higher-level implementation abstractions to the system developer as the system can be expressed using a modeling language rather than using a standard programming language. My main contributions in this area are:

- design and implementation of a tool to facilitate model driven prototyping using CP-nets and CPN Tools, and
- participation in an industrial case, where we used the described tool to build a model-based prototype of a network protocol.

I have described my work on a standard exchange format. I have identified two problems with the exchange and proposed solutions to the problems. My main contributions to the standard for high-level Petri nets are:

- the idea of mixing concrete and abstract syntax to represent labels,
- a number of translations from common composition constructs to a previously defined module concept, leading to the acceptance of the module concept as the sole composition mechanism for high-level Petri nets,
- a concrete proposal for an exchange format for high-level Petri nets, and
- participation in several standardization meetings, where I have worked, successfully, for the acceptance of the two first points.

6.1 Future Plans

The rest of my PhD work will not concentrate on all these three areas, but mainly on the design, implementation, and experimentation with the state-space tool and new analysis methods. The work on the animation tool is more or less complete and the work on standardization is not very time consuming, so at this moment this seems like the natural road ahead.

Benchmark the Sweep-Line Method

We would like to compare the sweep-line method with bit-state hashing and hash compaction. For this to be possible, I need to implement state caching, which should be relatively simple from the current hash table storage, and bring the current sweep-line method implementation up to date. This is interesting, since the sweep-line method has mainly been compared to calculating the full state space, which is of course not a fair comparison.

Visit to Aalborg

I plan to use the fall and winter at Kim G. Larsen's group in Aalborg, where I intend to participate in projects, development and the like. I will probably work on Timed Automata with UPPAAL [3, 87], but currently I do not know further details.

Define a Modal Logic for Properties of CP-nets

Many formalisms make it possible to specify properties of systems in a formalism-close way. For example (generalized) message-sequence chart models can be verified by checking that certain event sequences can or cannot appear. Mobile ambients [14] can be verified using ambient logic [13].

Currently the only way to check properties of CP-nets in CPN Tools is to write a SML function for checking the property and then check it yourself or using a rather out-dated CTL implementation, ASK-CTL [15].

If we define a logic to express local properties of CP-nets, we can in a canonical way, similar to the approach used in Chapter 3, obtain a logic for specifying safety properties. We can also easily use one of the standard definitions of a modal logic to obtain a modal logic for CP-nets. If the local properties and the modal logic is decidable for finite state spaces, we obtain a decidable logic. To check this we would probably need a checker for the modal logic, such as CTL or LTL.

It is also conceivable that in doing so, we will be able to provide the user with a more friendly way to enter properties, for example one could input temporal properties using message-sequence charts.

Partial State Spaces

Due to the fact that state spaces are often infinite or very large, it is interesting to see what can be checked by generating partial state spaces only, or by only generating coverability graphs [48, p. 146]. It is obvious that we cannot check safety properties, but we may be able to translate some results of bounded model checking [6] to the field of CP-nets.

Telebit Project

I plan to continue my participation in projects with Ericsson Danmark A/S, Telebit, where at least one of the upcoming projects will include state-space analysis. I expect that this will give ideas of how to construct a reasonable logic for CP-nets and also show a need to look at partial state spaces.

Bibliography

[1] T. Ball, B. Cook, V. Levin, and S.K. Rajamani. SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft. In *Proc. of IFM'04*, volume 2999 of *LNCS*, pages 1–20. Springer-Verlag, 2004.

[2] K. Beck. *Test Driven Development: By Example*. Addison-Wesley, 2002.

[3] G. Behrmann, A. David, and K.G. Larsen. A Tutorial on UPPAAL. In *Proc. of SFM-RT'04*, number 3185 in LNCS, pages 200–236. Springer-Verlag, 2004.

[4] M. Ben-Ari, Z. Manna, and A. Pnueli. The temnporal logic of branching time. *Acta Informatica*, 20:207–226, 1983.

[5] J. Bengtsson and W. Yi. Timed Automata: Semantics, Algorithms and Tools. In *Lectures on Concurrency and Petri Nets*, volume 3098 of *LNCS*, pages 87–124. Springer-Verlag, 2004.

[6] A. Biere, A. Cimatti, E.M. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. In *Proc. of TACAS'99*, volume 1579 of *LNCS*, pages 193–207. Springer-Verlag, 1999.

[7] J. Billington, S. Christensen, K. van Hee, E. Kindler, O. Kummer, L. Petrucci, R. Post, C. Stehno, and M. Weber. The Petri Net Markup Language: Concepts, Technology, and Tools. In *Proc. of ICATPN 2003*, volume 2679 of *LNCS*, pages 483–505. Springer-Verlag, 2003.

[8] P.V. Biron and A. Malhotra. XML Schema Part 2: Datatypes. `http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/`.

[9] B.H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, July 1970.

[10] B.W. Boehm. A spiral model of software development and enhancement, may 1988.

[11] C. Bossen and J.B. Jørgensen. Context-descriptive prototypes and their application to medicine administration. In *DIS '04: Proc. of the 2004 conference on Designing interactive systems*, pages 297–306, New York, NY, USA, 2004. ACM Press.

[12] C. Capellmann, S. Christensen, and U. Herzog. Visualising the Behaviour of Intelligent Networks. In *Services and Visualisation, Towards User-Friendly Design*, volume 1385 of *LNCS*, pages 174–189. Springer-Verlag, 1998.

[13] L. Cardelli and A.D. Gordon. Anytime, anywhere: Modal logics for mobile ambients. In *Proc. POPL'00*, pages 365–377. ACM, 2000.

[14] L. Cardelli and A.D. Gordon. Mobile ambients. *Theoretical Computer Science*, 240(1):177–213, 2000.

[15] A. Cheng, S. Christensen, and K.H. Mortensen. Model Checking Coloured Petri Nets Exploiting Strongly Connected Components. Technical report, Department of Computer Science, University of Aarhus, Denmark, 1996. Online version: `http://www.daimi.au.dk/designCPN/libs/askctl/ASKCTLtechreport.pdf`.

[16] S. Christensen and N.D. Hansen. Coloured Petri Nets Extended with Channels for Synchronous Communication. In *Proc. of ICATPN 1994*, volume 815 of *LNCS*, pages 159–178. Springer-Verlag, 1994.

[17] S. Christensen, J.B. Jørgensen, and L.M. Kristensen. Design/CPN—A Computer Tool for Coloured Petri Nets. In *Proc. of TACAS'97*, volume 1217 of *LNCS*, pages 209–223. Springer-Verlag, 1997.

[18] S. Christensen, L.M. Kristensen, and T. Mailund. A Sweep-Line Method for State Space Exploration. In *Proc. of TACAS'01*, volume 2031 of *LNCS*, pages 450–464. Springer-Verlag, 2001.

[19] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. The MIT Press, 1999.

[20] T. Clausen and P. Jacquet. Optimised Link State Routing Protocol (OLSR). RFC 3626, October 2003.

[21] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems - Concepts and Design*. Addison-Weslay, 3rd edition, 2001.

[22] The CPN Group at University of Aarhus. `http://www.daimi.au.dk/CPnets/`.

[23] CPN Tools homepage. `http://www.daimi.au.dk/CPNTools/`.

[24] J. Desel and W. Reisig. Place/Transition Petri Nets. In *Lecture on Petri nets I: basic models*, volume 1491 of *LNCS*, pages 122–173. Springer-Verlag, 1998.

[25] Design/CPN. `http://www.daimi.au.dk/designCPN/`.

[26] P. Deussen. Partial Order Verification of Programmable Logic Controllers. In *Proc. of ICATPN'01*, volume 2075 of *LNCS*, pages 144–163. Springer-Verlag, 2001.

[27] P. Dillinger and P. Manolios. Fast and Accurate Bitstate Verification for SPIN. In *Proc. of SPIN 2004*, volume 2989 of *LNCS*, pages 57–75. Sprinter-Verlag, 2004.

[28] Eclipse. `http://www.eclipse.org/`.

[29] Ericsson Danmark A/S, Telebit. `http://www.tbit.dk/`.

[30] ESC/Java. `http://research.compaq.com/SRC/esc/`.

[31] G. Gallasch and L.M. Kristensen. A Communication Infrastructure for External Communication with Design/CPN. In *Proc. of 3rd Workshop on Practical Use of Coloured Petri Nets and the CPN Tools*, volume PB-554 of *DAIMI*, pages 79–93. Department of Computer Science, University of Aarhus, 2001.

[32] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[33] S. Gordon, L.M. Kristensen, and J. Billington. Verification of a Revised WAP Wireless Transaction Protocol. In *Proc. of ICATPN'02*, volume 2360 of *LNCS*, pages 182–202. Springer-Verlag, 2002.

[34] B. Han and J. Billington. Formalising the TCP Symmetrical Connection Management Service. In *Proc. of Design, Analysis, and Simulation of Distributed Syste ms*, pages 178–184. SCS, 2003.

[35] D. Harel and R. Marelly. *Come, Let's Play*. Springer-Verlag, 2003.

[36] C.A.R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[37] G.J. Holzmann. An Improved Protocol Reachability Analysis Technique. *Software, Practice and Experience*, 18(2):137–161, 1988.

[38] G.J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall International Editions, 1991.

[39] G.J. Holzmann. State Compression in SPIN: Recursive Indexing and Compression Traning Runs. In *Proc. of 3rd SPIN Workshop*, 1997.

[40] G.J. Holzmann. An Analysis of Bitstate Hashing. *Formal Methods in System Design*, 13:289–307, 1998.

[41] G.J. Holzmann. From Code to Models. In *Proc. of ICACSD 2001*, 2001.

[42] G.J. Holzmann. *The SPIN Model Checker*. Addison Wesley, 2003.

[43] C. Huitema. *IPv6: The New Internet Protocol*. Prentice-Hall, 1998.

[44] University of Southern California Information Sciences Institute. Transmission Control Protocol. RFC 793, September 1981.

[45] Software and system engineering – High-level Petri nets – Part 1: Concepts, definitions and graphical notation. ISO/IEC 15909-1:2004, 2004.

[46] Java Network Launching Protocol and API. `http://jcp.org/en/jsr/detail?id=56`.

[47] The Java^{TM} Tutorial: The Reflection API. `http://java.sun.com/docs/books/tutorial/reflect/`.

[48] K. Jensen. *Coloured Petri Nets—Basic Concepts, Analysis Methods and Practical Use. Volume 1: Basic Concepts*. Springer-Verlag, 1992.

[49] K. Jensen. *Coloured Petri Nets—Basic Concepts, Analysis Methods and Practical Use. Volume 2: Analysis Methods*. Springer-Verlag, 1994.

[50] K. Jensen. *Coloured Petri Nets—Basic Concepts, Analysis Methods and Practical Use. Volume 3: Practical use*. Springer-Verlag, 1997.

[51] O.H. Jensen and R. Milner. Bigraphs and Transitions. In *Proc. af POPL 2003*, pages 38–49. ACM Press, 2003.

[52] E. Kindler. High-level Petri Nets—Transfer Syntax. Proposal for the International Standard ISO/IEC 15909 Part 2. Draft Version 0.3.0. `http://wwwcs.uni-paderborn.de/cs/kindler/Publikationen/copies/ISO-IEC-15909-2-Draft.0.3.0.pdf`, April 2004.

[53] E. Kindler and C. Páles. 3D-Visualization of Petri Net Models: Concept and Realization. In *Proc. of ICATPN 2004*, volume 3099 of *LNCS*, pages 464–473. Springer-Verlag, 2004.

[54] E. Kindler and M. Weber. A universal module concept for Petri nets. An implementation-oriented approach. *Informatik-Berichte*, 150, June 2001.

[55] S.A. Kripke. A semantical analysis of modal logic: I. Normal modal propositional calculi. *Zeitschrift fũr Mathematische Logic und Grundlagen der Mathematik*, 9:67–96, 1963.

[56] L.M. Kristensen. Ad-hoc Networking and IPv6: Modelling and Validation. `http://www.pervasive.dk/projects/IPv6/IPv6_summary/`.

[57] L.M. Kristensen, S. Christensen, and K. Jensen. The Practitioner's Guide to Coloured Petri Nets. *Journal on Software Tools for Technology Transfer*, 2(2):98–132, 1998.

[58] L.M. Kristensen and K. Jensen. Specification and Validation of an Edge Router Discovery Protocol for Mobile Ad-hoc Networks. In *Integration of Software Specification Techniques for Application in Engineering*, volume 3147 of *LNCS*, pages 248–269. Springer-Verlag, 2004.

[59] L.M. Kristensen and T. Mailund. A Generalised Sweep-Line Method for Safety Properties. In *Proc. of FME'02*, volume 2391 of *LNCS*, pages 549–567. Springer-Verlag, 2002.

[60] L.M. Kristensen, M. Westergaard, and P.C. Nørgaard. Model-based Prototyping of an Interoperability Protocol for Mobile Ad-hoc Networks. Submitted to Fifth International Conference on Integrated Formal Methods, 2005.

[61] A. Lindem. OSPF for IPv6. Internet-draft, March 2005.

[62] L. Lorentsen, A-P Tuovinen, and J. Xu. Modelling Features and Feature Interactions of Nokia Mobile Phones Using Coloured Petri Nets. In *Proc. of ICATPN 2002*, volume 2360 of *LNCS*, pages 294–313, 2002.

[63] J. Magee and J. Kramer. *Concurrency – State Models and Java Programs*. John Wiley & Sons, 1999.

[64] J. Magee, N. Pryce, D. Giannakopoulou, and J. Kramer. Graphical Animation of Behavior Models. In *Proc. of 22nd International Conference on Software Engineering*, pages 499–508. ACM Press, 2000.

[65] T. Mailund. *Sweeping the State Space — A Sweep-Line State Space Exploration Method*. PhD thesis, Department of Computer Science, University of Aarhus, 2003.

[66] T. Mailund and M. Westergaard. Obtaining Memory-Efficient Reachability Graph Representations Using the Sweep-Line Method. In *Proc. of TACAS'04*, volume 2988 of *LNCS*, pages 177–191. Springer-Verlag, 2004.

[67] mathworld. Hyperedge definition. `http://mathworld.wolfram.com/Hyperedge.html`.

[68] R. Milner. *Communication and concurrency*. Prentice-Hall, Upper Saddle River, NJ, USA, 1989.

[69] R. Milner. Bigraphical Reactive Systems. In K.G. Larsen and M. Nielsen, editors, *Proc. of CONCUR 2001*, volume 2154 of *LNCS*, pages 16–35. Springer-Verlag, 2001.

[70] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes. *Information and Computation*, 100(1):1–77, 1992.

[71] Modex. `http://cm.bell-labs.com/cm/cs/what/modex/`.

[72] C. Ouyang and J. Billington. On Verifying the Internet Open Trading Protocol. In *Proc. of 4th International Conference on Electronic Commerce and Web Technologies*, volume 2738 of *LNCS*, pages 292–302. Springer-Verlag, 2003.

[73] C.E. Perkins. *Ad Hoc Networking*. Addison-Wesley, 2001.

[74] C.A. Petri. *Kommunikation mit Automaten*. PhD thesis, Bonn: Institut fÃ$\frac{1}{4}$r Instrumentelle Mathematik, 1962. Schriften des IIM Nr. 2.

[75] A. Pnueli. The temporal logic of programs. In *Proc. of 18th IEEE Symposium on Foundations of Computer Science*, pages 46–57, 1977.

[76] PVS. `http://pvs.csl.sri.com/`.

[77] J. L. Rasmussen and M. Singh. *Mimic/CPN. A Graphical Simulation Utility for Design/CPN. User's Manual*. `http://www.daimi.au.dk/designCPN/`.

[78] J.L. Rasmussen and M. Singh. Designing a Security System by Means of Coloured Petri Nets. In *Proc. ICATPN 1996*, volume 1091 of *LNCS*, pages 400–419. Springer-Verlag, 1996.

[79] 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 ICATPN 2003*, volume 2679 of *LNCS*, pages 450–462. Springer-Verlag, 2003.

[80] SceneBeans. `http://www-dse.doc.ic.ac.uk/Software/SceneBeans/`.

[81] SLAM. `http://research.microsoft.com/slam/`.

[82] S. Sørensen. I/O Efficient State-Space Storage. Master's thesis, University of Aarhus, 2002.

[83] SPIN Language Reference. `http://spinroot.com/spin/Man/promela.html`.

[84] U. Stern and D.L. Dill. Improved Probabilistic Verification by Hash Compaction. In *Correct Hardware Design and Verification Methods*, volume 987 of *LNCS*, pages 206–224. Springer-Verlag, 1995.

[85] The ExSpect tool. `http://www.exspect.com/`.

[86] J.D. Ullman. *Elements of ML Programming*. Prentice-Hall, 1998.

[87] UPPAAL. `http://www.uppaal.com/`.

[88] W.M.P. van der Aalst. The Application of Petri Nets to Workflow Management. *The Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998.

[89] W.M.P. van der Aalst and K. van Hee. *Workflow Management: Models, Methods, and Systems*. MIT Press, 2002.

[90] M. Vardi and P. Wolper. An Automata-Theoretic Approach to Automatic Program Verification. In *Proc. of IEEE Symposium on Logic in Computer Science*, pages 322–331, 1986.

[91] Visual Studio. `http://msdn.microsoft.com/vstudio/`.

[92] M. Weber. Petri Net Markup Language. `http://www.informatik.hu-berlin.de/top/pnml/`.

[93] M. Weber and E. Kindler. The Petri Net Kernel. In *Petri Net Technologies for Modeling Communication Based Systems*, volume 2472 of *LNCS*, pages 109–123. Springer-Verlag, 2003.

[94] M. Westergaard. TIN-CPN homepage. `http://wiki.daimi.au.dk/tincpn/`.

[95] M. Westergaard. Towards a High-level Petri Net Type Definition. In E. Kindler, editor, *Proc. of Workshop on the Definition, Implementation and Application of a Standard Interchange Format for Petri Nets, Satelite event at the 25th International Conference on Application and Theory of Petri Nets*, pages 71–85, 2004.

[96] D. Winer. XML-RPC Specification. `http://xmlrpc.org/spec`.

[97] P. Wolper and D. Leroy. Reliable Hashing without Collision Detection. In *Proc. of CAV'93*, volume 697 of *LNCS*, pages 59–70. Springer-Verlag, 1993.