
Higher-Order and Symbolic Computation, 17, 207–243, 2004

Implementing Coloured Petri Nets Using a Functional Programming Language

LARS MICHAEL KRISTENSEN* lmkristensen@daimi.au.dk
SØREN CHRISTENSEN schristensen@daimi.au.dk
Department of Computer Science, University of Aarhus, Aabogade 34, DK-8200 Aarhus N, Denmark

Abstract. Coloured Petri Nets (CPNs) are a graphically oriented modelling language for concurrent systems based on Petri Nets and the functional programming language Standard ML. Petri Nets provide the primitives for modelling concurrency and synchronisation. Standard ML provides the primitives for modelling data manipulation and for creating compact and parameterisable CPN models.

Functional programming and Standard ML have played a major role in the development of CPNs and the CPN computer tools supporting modelling, simulation, verification, and performance analysis of concurrent systems. At the modelling language level, Standard ML has extended Petri Nets with the practical expressiveness required for modelling systems of the size and complexity found in typical industrial projects. At the implementation level, Standard ML has been used to implement the formal semantics of CPNs that provide the theoretical foundation of the CPN computer tools.

This paper provides an overview of how functional programming and Standard ML are applied in the CPN modelling language and the supporting computer tools. We give a detailed presentation of the key algorithms and techniques used for implementing the formal semantics of CPNs, and we survey a number of case studies where CPNs have been used for the design and analysis of systems. We also demonstrate how the use of a Standard ML programming environment has allowed Petri Nets to be used for the implementation of systems.

Keywords: distributed and concurrent computation, implementation techniques, programming environments and tools, Coloured Petri Nets, high-level Petri Nets, Petri Nets

1. Introduction

An increasing number of system development projects are concerned with concurrent systems. Examples of these range from large scale systems in the areas of telecommunication and applications based on WWW technology, to medium or small scale systems, in the area of embedded systems. The development of such systems is complex, a main reason being that the execution of a concurrent system consisting of a number of independent but co-operating processes may proceed in many different ways, e.g., depending on messages lost, the speed of the processes involved, and the time at which input is received from the environment. As a result, concurrent systems are difficult to design and test, and can be subject to subtle errors that can go undetected for a long time. This has motivated the development of modelling languages that support computer-aided analysis, validation, and verification of concurrent systems, and that can be used during design to obtain more reliable and trustworthy concurrent systems.

*Supported by the Danish Natural Science Research Council.

One of the most promising ways to increase reliability and reduce design errors of such systems is the use of *formal methods* [9], which are mathematically-based languages, techniques, and tools for specifying, analysing, and verifying systems. The application of formal methods is typically based on the construction of a model, which describes the behaviour of the concurrent system under consideration and which can be manipulated and analysed by a computer tool. The use of formal methods does not automatically guarantee that a system is correct. However, the act of constructing a model and analysing its behaviour is a way to gain greater confidence in the proposed designs and to help reveal inconsistencies, ambiguities, and incompleteness that might otherwise go undetected.

Coloured Petri Nets (CPNs) [33, 34, 36, 40] are a modelling language based on the ideas of *Petri Nets* [60, 64] and *Predicate/Transition Nets* [21]. Petri Nets are traditionally divided into *low-level Petri Nets* [66] and *high-level Petri Nets* [38]. CPNs belong to the class of high-level Petri Nets that are characterised by the combination of Petri Nets and programming languages. Low-level Petri Nets are primarily suited as a theoretical model for concurrency, although certain classes of low-level Petri Nets are often applied for modelling and verification of hardware circuits [80]. High-level Petri Nets are aimed at practical use, in particular because they allow for construction of compact and parameterised models.

CPNs are a combination of Petri Nets and the functional programming language Standard ML (SML) [51, 73]. Petri Nets provide the primitives for describing the concurrency aspects of systems. SML provides the primitives for modelling data values and data manipulation, and for creating compact and parameterisable models. Construction, execution, and formal analysis of CPNs are supported by two computer tools: CPN Tools [72] and Design/CPN [13]. The CPN computer tools have been applied in many industrial projects spanning a broad range of application areas [13, 72].

Functional programming and SML have played a major role in the development of CPNs. One contribution of this paper is to give an overview of how functional programming via SML has been used in the CPN modelling language and the supporting computer tools. A second contribution is a detailed presentation of the so-called enabling inference algorithm used to implement the formal semantics of CPNs. The presented algorithms have been refined and modified over several years based on experiences gained in case studies and industrial projects on how CPNs are being applied in practice. The result is an enabling inference algorithm that has proven to be able to handle CPNs arising in practical modelling of systems. We survey a number of industrial case studies where CPNs have been used for the design and analysis of systems. We also present two industrial projects where CPN models in executable form have been extracted from the CPN computer tools and used as the implementation of a security system and a planning tool. These two case studies demonstrate that having a full programming language environment integrated in CPNs allows the usual gap between system modelling and implementation to be eliminated. They also illustrate how a functional programming language via CPNs has been used to implement real-world systems.

Outline. Section 2 gives an introduction to the syntax and semantics of CPNs showing how SML is used at the modelling language level. Section 3 presents the enabling inference algorithm for implementing the formal semantics of CPNs. Section 4 puts the enabling

inference algorithm and SML into the context of the CPN computer tools. Section 5 surveys a number of case studies where CPNs have been used for the design and analysis of systems. Section 6 presents the two projects where CPN models have been used to obtain the implementation of a system. Section 7 contains a discusssion of related work, and Section 8 sums up the conclusions. The paper does not assume that the reader is familiar with Petri Nets or Coloured Petri Nets. Basic knowledge of functional programming languages and the SML language is assumed.

2. Coloured Petri Nets

This section gives an introduction to the syntax and semantics of CPNs and introduces the notation used in the rest of this paper. The reader is referred to Jensen [33] for a complete and formal definition of the syntax and semantics of CPNs.

2.1. Example: A communication protocol

We consider a small transport protocol from the transport layer of the ISO reference model [12]. The example is a variant of the protocol considered in [37]. The transport layer is concerned with protocols ensuring reliable transmission between sites. The transport protocol does not in any way represent a sophisticated protocol. However, it is complex enough to illustrate the basic CPN constructs and will be used as a running example in this paper.

The transport protocol consists of a *sender* transferring a number of *data packets* to a *receiver*. Communication takes place on a network with packet loss and overtaking. The transport protocol uses sequence numbers, acknowledgements, and retransmissions to ensure that the data packets are delivered once and only once and in the correct order to the receiver. The protocol deploys a stop-and-wait strategy, i.e., the same data packet is transmitted until a corresponding acknowledgment is received.

A CPN model is always created as a graphical drawing. Figure 1 shows the CPN model of the transport protocol. The CPN constructs in Figure 1 will be explained in detail in the following sections.

2.2. Modelling of states

Places. The states of a CPN are represented by means of *places* which are drawn as ellipses or circles. It should be noted that places, in contrast to, e.g., transition systems, are not used to directly describe the possible states of the CPN. The state of a CPN is a *marking* of the places of the CPN model. We explain the concept of markings shortly.

The states of the transport protocol are modelled using eight places. The place Send (top left) models the data packet buffer containing the data packets which the sender is to transmit to the receiver. The places A, B, C and D model buffers in the network. The place NextSend (left) models the internal state of the sender. It will keep track of the sequence number of the data packet to be sent next. The place NextRec (middle right) models the

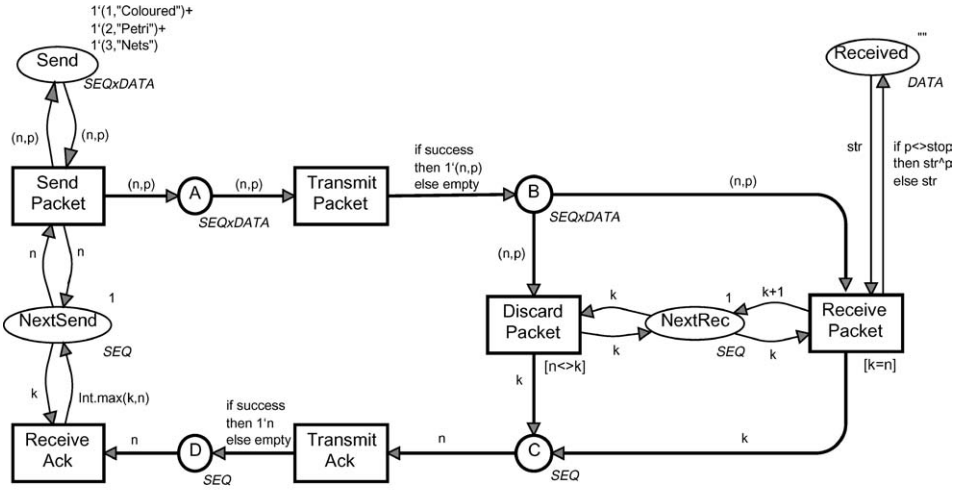

Figure 1. CPN model of a simple transport communication protocol.

internal state of the receiver. It will keep track of the sequence number of the data packet which is expected next. Finally, the place Received (top right) models the data packet buffer containing the data packets received by the receiver.

Types. Each place has an associated *type*[1] determining the kind of data that the place may contain. The type of a place is written in italics, to the lower right of the place. The type of a place can be an arbitrary equality type constructed using, e.g., record, union, product and list type constructors. Figure 2 shows the *type definitions* used for the transport protocol.

Four different types are used to model the transport protocol. The places Send, A, and B have the type `SEQxDATA`. The type is defined as the `product` of the types `SEQ` and `DATA`. The type `DATA` is defined as `string` denoting the set of text strings and will be used to model the contents of data packets. The type `SEQ` is defined as `int` (integers) and will be used to model the sequence numbers of data packets and acknowledgements. The places NextSend, NextRec, C, and D have the type `SEQ`. The place Received has the type `DATA`.

```
color SEQ = int;
color DATA = string;
color SEQxDATA = product SEQ * DATA;
color Bool = bool;
```

Figure 2. Type definitions of the CPN model.

The type `Bool` contains the two values `true` and `false` and will be used to model the possibility of packets being lost on the network.

Multi-sets and markings. A state of a CPN is called a *marking*. It consists of a distribution of *tokens* on the places. Each token carries a *value*,[2] which belongs to the type of the place on which the token resides. The tokens that are present on a particular place are called the marking of that place. The marking of a place is, in general, a *multi-set* of token values. This means that a place may have several tokens with the same token value. Formally, a multi-set ms over a domain X is a function from X into the set of natural numbers $\mathbb{N}$. A multi-set ms can be written as $\sum_{x \in X} ms(x)'x$, where $ms(x)$ is the number of occurrences of the element x in ms. As an example, a possible marking of the place A is the following:

```
2‘(1,"Coloured") + 4‘(2,"Petri")
```

This marking contains two tokens with value `(1,"Coloured")` and four tokens with value `(2,"Petri")`.

A CPN model has a distinguished *initial marking* describing the initial state of the system. If a place contains tokens in the initial marking, then these tokens are specified to the upper right of the place. The place Send has an initial marking consisting of the three tokens `(1,"Coloured")`,`(2,"Petri")`, and `(3,"Nets")`, i.e., three pairs of sequence numbers and data values. The place NextSend initially contains a single token with the value `1`, denoting that the packet to be sent first will be assigned sequence number `1`. The place NextRec initially contains a single token with the value `1`, denoting that the data packet expected first has sequence number `1`. The place Received initially contains a token with the value `""` (empty string) corresponding to the receiver having received no data to begin with. Initially, the remaining four places A, B, C, and D contain no tokens corresponding to the network buffers initially being empty.

2.3. Modelling of actions

Transitions. The actions of a CPN are represented by means of *transitions* which are drawn as rectangles. The CPN model of the transport protocol contains six transitions. The transition SendPacket models the action taken when a data packet is sent from the sender. The transition TransmitPacket models the transmission on the network of a data packet from the sender to the receiver. ReceivePacket models the reception of a data packet at the receiver with the expected sequence number, and the sending of an acknowledgement back to the sender. DiscardPacket models the reception of a data packet at the receiver when the sequence number of the data packet is not being the expected one. TransmitAck models the transmission on the network of an acknowledgement from the receiver to the sender. Transition ReceiveAck models the reception of an acknowledgement at the sender side.

Arc expressions and guards. Transitions and places are connected by *arcs*. The actions of a CPN consist of *occurrences* of *enabled* transitions. A transition is enabled if the required tokens are present on places connected to input arcs (called input places) of the transition. An occurrence of a transition removes tokens from input places, and adds tokens

to places connected to output arcs (called output places) of the transition, thereby changing the marking of the CPN model. As an example, the transition SendPacket has two input arcs and three output arcs. Hence, an occurrence of this transition will remove tokens from the places Send and NextSend, and will add tokens to the places Send, NextSend, and A. The specific tokens added and removed by the occurrence of a transition are determined by the *arc expressions*, which are positioned next to the arcs. An arc expression can be any SML expression provided it has the proper type (as will be explained below). The arc expression may also contain functions, e.g., the arc expression on the arc from ReceiveAck to NextSend uses the function `Int.max` from the SML basis library. User-defined functions are also possible.

The set of *input places* of a transition t is denoted $P_{\mathsf{In}}(t)$, and the set of *output places* of t is denoted $P_{\mathsf{Out}}(t)$. If there is an arc leading from a place p to a transition t, $E(p, t)$ denotes the corresponding arc expression. Similarly, if there is an arc leading from a transition t to a place p, $E(t, p)$ denotes the corresponding arc expression. Since arc expressions are SML expressions, they have a type according to the SML type system. For an SML expression e, Type[e] denotes the type of the expression e.

In addition to the arc expressions, it is possible to attach a boolean SML expression to each transition. This boolean expression is called a *guard*. It specifies an additional requirement for the transition to be enabled. If a transition has a guard, then it is written in square brackets and positioned next to the transition. As an example, the transition ReceivePacket has the guard `[k=n]`. The guard of a transition t is denoted $G(t)$.

2.4. Formal semantics

Variables and bindings. An occurrence of an enabled transition requires data values to be *bound* to the free *CPN variables* appearing in the guard, and in the input and output arc expressions of the transition. This is needed to *evaluate* the arc expression and the guard. The *variable declarations* specifying the CPN variables and their types for the transport protocol are shown in Figure 3. For an expression e, $\mathsf{FV}(e)$ denotes the set of free CPN variables of e. The *scope* of a CPN variable is the input/output arc expressions and the guard of the transition. Hence, the set of free CPN variables $\mathsf{FV}(t)$ of a transition t is given by:

$$\mathsf{FV}(t) = \bigcup_{p \in P_{\mathsf{In}}(t)} \mathsf{FV}(E(p,t)) \cup \bigcup_{p \in P_{\mathsf{Out}}(t)} \mathsf{FV}(E(t,p)) \cup \mathsf{FV}(G(t))$$

```
var n, k : SEQ;
var p,str : DATA;
var success : Bool;
```

Figure 3. CPN variable declarations for the CPN model.

As an example, consider the transition SendPacket to the upper left of Figure 1. The transition SendPacket has two free CPN variables: `n` of type `SEQ` and `p` of type `DATA`. The free CPN variable `n` appearing in all the arc expressions on the arcs connected to the transition is the same variable because of the scope of the CPN variables. In the rest of this paper when we talk about variables of transitions, we implicitly mean free CPN variables.

To evaluate the guard (if any) and the arc expressions on arcs connected to a transition, values must be bound to the variables of the transition. A *binding* of a transition t is by convention written in the form: $\langle v_1 = c_1, v_2 = c_2, \ldots v_n = c_n \rangle$ where $FV(t) = \{v_i \mid i \in 1..n\}$, and c_i is the value bound to v_i. The binding is required to fulfil that $c_i \in \mathsf{Type}[v_i]$ for $1 \leq i \leq n$. For a binding b, $b(v)$ denotes the value bound to the variable v in the binding b. The result $e\langle b \rangle$ of evaluating an expression e (arc expression or guard) in a binding $b = \langle v_1 = c_1, \ldots, v_n = c_n \rangle$ is defined according to the semantics of SML as:

$$e\langle b \rangle \equiv (\mathsf{fn}(v_1, \ldots, v_n) \Rightarrow e)(c_1, \ldots, c_n)$$

Assume that we bind data values to the variables of the transition SendPacket by creating the binding:

$$b_{SP} = \langle \texttt{n = 1, p = "Coloured"} \rangle$$

In Figure 4(a), a dashed box shows the multi-set of tokens obtained by evaluating the corresponding arc expression in the binding b_{SP} for each arc connected to SendPacket. In this case all multi-sets contain a single token.

To ensure that an arc expression evaluates to a multi-set of tokens over the same type as the associated place (input or output) there are type constraints on the arc expressions. These type constraints are formalised below, where $C(p)$ denotes the type of a place p,

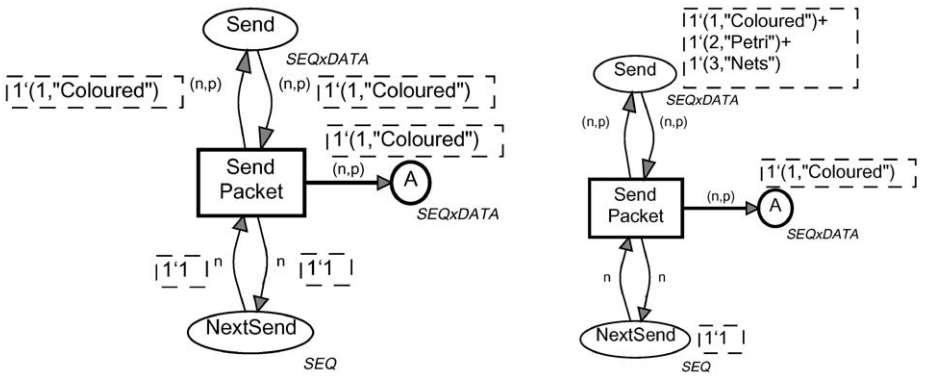

Figure 4. Enabling and occurrence of transition SendPacket.

X_{MS} denotes the type of multi-sets over a set X, and $X \rightarrow Y$ denotes a function type with domain type X and range type Y:

$$\forall p \in P_{\mathsf{In}}(t) : \mathsf{Type}[\mathsf{fn}(v_1, \ldots, v_n) \Rightarrow E(p,t)] = \left(\times_{i=1}^{n} \mathsf{Type}[v_i]\right) \rightarrow C(p)_{MS}$$
$$\forall p \in P_{\mathsf{Out}}(t) : \mathsf{Type}[\mathsf{fn}(v_1, \ldots, v_n) \Rightarrow E(t,p)] = \left(\times_{i=1}^{n} \mathsf{Type}[v_i]\right) \rightarrow C(p)_{MS}$$

Practical experience has shown that many arc expressions arising in practial modelling always evaluate to a singleton multi-set. The function derived from an arc expression is therefore allowed to have $C(p)$ as range type instead of the more general $C(p)_{MS}$ as required above. This is also exploited by the algorithm for enabling inference presented in Section 3. In the CPN model of the communication protocol it is only the outgoing arcs from TransmitPacket and TransmitAck which have the general multi-set type $C(p)_{MS}$. The value empty in the else branch of these arc expressions denotes the empty multi-set.

A type constraint is also imposed on the guard to ensure that it is indeed a boolean expression. The type constraint can be expressed as:

$$\mathsf{Type}[\mathsf{fn}(v_1, ..v_n) \Rightarrow G(t)] = \left(\times_{i=1}^{n} \mathsf{Type}[v_i]\right) \rightarrow \mathsf{bool}$$

Enabling. For a transition to be *enabled* in a marking, i.e., ready to *occur*, it must be possible to bind data values to the variables of the transition such that: (1) each of the arc expressions on input arcs evaluate to a multi-set of tokens which is a subset of the marking of the corresponding input place, and (2) the guard (if any) is satisfied.

Figure 4(a) shows that an occurrence of the transition SendPacket (in the binding b_{SP}) removes a token with value `(1,"Coloured")` from the place Send and a token with value `1` from the place NextSend. The binding of SendPacket is enabled in the initial marking, since the tokens to which the arc expressions evaluate are present on the corresponding input places. There are many other possible bindings for the transition SendPacket. However, none of these are enabled in the initial marking. The possible bindings of data values to the variables of a transition correspond to the possible ways in which a transition can occur. However, as we demonstrated above, only a subset of these will, in general, be enabled in a given marking.

Since the concept of enabling is central to the techniques and algorithms presented in the next sections, we formally define enabling below to resolve any ambiguity in the informal explanation given above. For a marking M, we denote by $M(p)$ the marking of the place p in M. A *binding element* is a pair (t, b) consisting of a transition t and a binding b of the variables of the transition t.

Definition 1. A binding element (t, b) is *enabled* in a marking M if and only if the following condition holds:

$$\forall p \in P_{\mathsf{In}}(t) : (E(p,t)\langle b\rangle \leq M(p)) \wedge G(t)\langle b\rangle$$

In Definition 1 $\leq$ denotes subset for multi-sets. The condition expresses that sufficient tokens are present on each of the input places and that the guard is satisfied. When a binding element (t, b) is enabled in a marking M, the transition t is said to be enabled in the binding b.

Occurrence. The tokens removed from an input place when an enabled binding element occurs are determined by evaluating the arc expression on the corresponding input arc. Similarly, the tokens added to an output place are determined by evaluating the arc expression on the corresponding output arc (see Figure 4(a)). Figure 4(b) shows the marking of the surrounding places of the transition SendPacket resulting from the occurrence of the enabled binding b_{SP} in the initial marking. The marking of a place is shown in the dashed box positioned next to the place. As can be seen, an occurrence of the transition SendPacket leaves the marking of the places Send and NextSend unchanged, and adds a token to place A corresponding to the data packet being sent.

An execution of a CPN model is described by means of an *occurrence sequence*. It specifies the markings that are reached and the *steps* that occur. Above, we have seen an occurrence sequence of length one. It consisted of a single step, the occurrence of SendPacket in the only enabled binding in the initial marking, and leading to the marking shown in Figure 4(b). In general, a step may consist of several enabled binding elements occurring *concurrently*. As an example, consider the marking shown in Figure 5. In this marking there are two tokens with value `(1,"Coloured")` in place A, and there is one token with value `(1,"Coloured")` and one token with value `(2,"Petri")` in place B. Moreover, the token in place NextSend has value 2, the token in place NextRec has value 2, and the marking of place Received is a token with value `"Coloured"`. In this marking

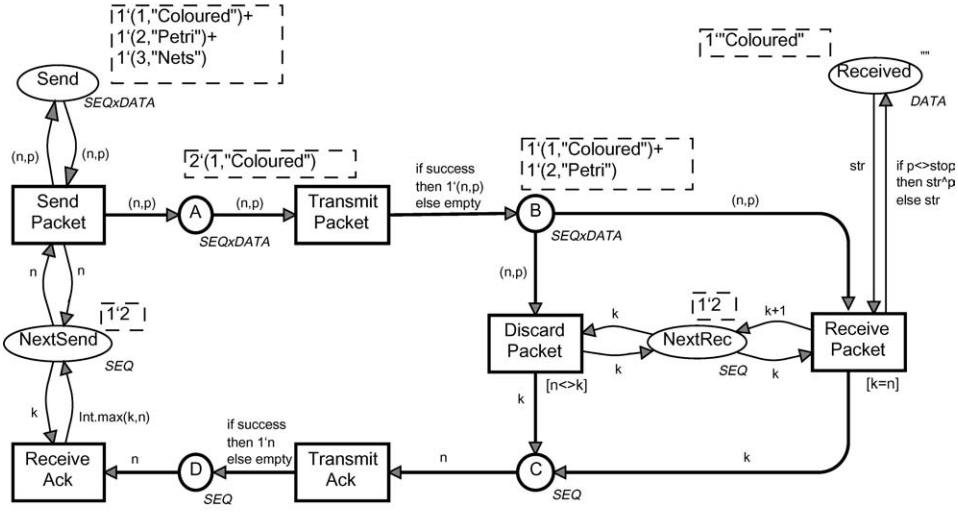

Figure 5. Marking of the CPN model.

there are several enabled binding elements. The transition TransmitPacket is enabled in two different bindings:

$$b_1 = \langle \texttt{n = 1, p = "Coloured", success = true} \rangle$$
$$b_2 = \langle \texttt{n = 1, p = "Coloured", success = false} \rangle$$

The binding b_1 corresponds to a successful transmission of a data packet. The binding b_2 corresponds to a loss of a data packet. The bindings b_1 and b_2 are *concurrently enabled*, since each binding can get those tokens that it needs (i.e., those specified by the input arc expressions)—without sharing the tokens with other bindings. This means that b_1 and b_2 may occur in the same step, corresponding to the two data packets being transmitted simultaneously. An occurrence of b_1 will remove a data packet from place A and add the data packet to place B. An occurrence of b_2 will only remove a token corresponding to the data packet from place A, and it will not add tokens to place B.

The transitions DiscardPacket and ReceivePacket are also enabled in the marking. Transition DiscardPacket is enabled in the binding $b_3 = \langle \texttt{n = 1, p = "Coloured", k = 2} \rangle$, and ReceivePacket is enabled in the binding $b_4 = \langle \texttt{n = 2, p = "Petri", k = 2, str = "Coloured"} \rangle$. However, the bindings b_3 and b_4 are *not* concurrently enabled, since they cannot both get the token with value 2 on place NextRec. These two bindings are in *conflict*.

3. Enabling inference algorithm

The main issue in implementing the formal semantics of CPNs is to be able to infer the set of enabled binding elements in a given marking of the CPN model. From the previous section it follows that computing the set of enabled binding elements amounts to considering the tokens on the input places of transitions and based on this obtain the set of enabled binding elements for each transition.

Finding the set of enabling binding elements in a given marking is hard in its full generality. The reason is that the variables of transitions may range over infinite domains (e.g., lists and integers) and the SML *inscriptions*, i.e., the arc expressions and guards can be arbitrary SML expressions, including recursive functions. One possible way to overcome this problem would be to restrict the variables of transitions to only finite types and to make an exhaustive search to find the enabled binding elements. This, however, is not efficient if the CPN model has large (but finite) types or if the transitions have many variables. Both of these situations occur frequently in practice. Moreover, this approach cannot handle many modelling constructs arising in practice such as using places with a list type to model queues, stacks, and unbounded communication channels. Another possibility would be to apply program analysis techniques combined with imposing syntactical restrictions on the inscriptions such that it is trivial to deduce the inverse function of the arc expressions. In that case, arbitrary recursive functions would have to be disallowed in the inscriptions. Such a restriction is problematic, since CPN models arising in practice often use recursive functions to manipulate highly structured tokens.

A main design criteria for the mechanism computing the set of enabled binding elements in a given marking has been that it should be time efficient and at the same time accommodate enough expressive power and modelling convenience to handle CPN models arising in the practical modelling of systems. This has been achieved using a token-based inference mechanism based on the *pattern matching* capabilities of SML.

3.1. Pattern matching

A pattern in SML is an expression with variables (identifiers) which can be matched with arguments/expressions to give the variables values. The syntactical structure of patterns in SML (taken from [73]) is listed in Figure 6. As can be seen from Figure 6, a rather large set of expressions qualifies as patterns. An expression qualifies as a pattern if it is built of constants (of an equality type), constructors, and identifiers.

In CPN inscriptions we disallow the use of the wild-card symbol _ (underscore) and the wild-card symbol for record fields (`...`). Allowing them in an unrestricted way implies that binding elements are no longer deterministic, i.e., it is possible to have a binding element (t, b) enabled in a marking M_1 such that if (t, b) occurs in M_1 then there are two (or more) possibilities for the resulting marking. The fact that binding elements are deterministic is important for many of the analysis methods of CPNs. The two wild-card symbols can be allowed in inscriptions provided that they are guarded by the `as` constructor. In the rest of this paper we will however disallow wild-card symbols in patterns to simplify the presentation. In the following PATTERN(e) denotes that an expression e qualifies as one of the allowed patterns.

To understand the basic idea of how SML pattern matching can be exploited to infer enabled binding elements, consider the extract of the CPN model of the communication protocol in Figure 7 which shows the SendPacket transition. For simplicity we ignore (for now) that the transition also has NextSend as input place. We consider a marking in which the place Send contains tokens with values `(1,"Coloured")` and `(2,"Petri")`. The pattern `(n,p)` on the input arc from Send contains the two free CPN variables `n` (of type `SEQ`) and `p` (of type `DATA`). Hence, we can *match* the values of the tokens on place Send

PATTERN	::=	ID | ID ATOMICPATTERN | PATTERN INFIXCONS PATTERN | ID **as** PATTERN
ATOMICPATTERN	::=	ID | CONST | _ | TUPLEPATTERN | LISTPATTERN | RECORDPATTERN
TUPLEPATTERN	::=	(PATTERN (,PATTERN)*)
LISTPATTERN	::=	[] | [PATTERN (,PATTERN)*]
RECORDPATTERN	::=	{ ATOMRECPATTERN (,ATOMRECPATTERN)* } | { ATOMRECPATTERN (,ATOMRECPATTERN)*,...}
ATOMRECPATTERN	::=	LABEL = PATTERN | LABEL = ID **as** PATTERN | ID **as** PATTERN | ID | ID **as** PATTERN

Figure 6. Syntactical structure of SML patterns.

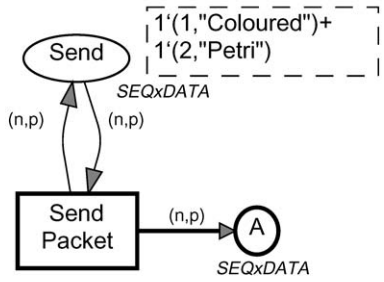

Figure 7. Exploiting pattern matching in CPNs.

and the pattern `(n,p)` causing values to be bound to `n` and `p`. In this way we obtain the two binding elements:

(SendPacket, ⟨`n = 1, p = "Coloured"`⟩)
(SendPacket, ⟨`n = 2, p = "Petri"`⟩)

These two binding elements are the two enabled binding elements of SendPacket in the marking shown in Figure 7, where we have ignored the input place NextSend.

3.2. Pure arc expression based algorithm

To generalise the above idea, it must be ensured that all variables of the transition are bound by matching the tokens on input places against patterns appearing as expressions of input arcs. This requirement is captured by the concept of a *pattern binding basis* which is a set of input arc expressions of a transition qualifying as patterns and which are such that any variable of the transition appears in at least one of the arc expressions.

Definition 2. A *pattern binding basis PBB*(t) for a transition t is a set of input arc expressions of t satisfying:

1. $\mathsf{FV}(t) = \bigcup_{E(p,t) \in PBB(t)} \mathsf{FV}(E(p,t))$
2. $\forall E(p,t) \in PBB(t) : \textsc{Pattern}(E(p,t))$

Item 1 above ensures that all the variables of the transition are covered. Item 2 ensures that all input arc expressions included in the pattern binding basis are patterns. Item 2 is needed since we do not require all input arc expressions of a transition to be patterns.

As an example consider the transition ReceivePacket in Figure 1. The transition has four variables: FV(ReceivePacket) = {`n`, `p`, `k`, `str`}. The input arc expressions: `(n,p)` (from

place B), `str` (from place Received) and `k` (from place NextRec) constitute a pattern binding basis for ReceivePacket.

Matching the value of a token residing on an input place will, in general, only bind a subset of the variables of the transition. As an example, consider the transition ReceivePacket and the marking shown in Figure 5. Matching the token `(1,"Coloured")` residing on place B against the pattern `(n,p)` will bind the variable `n` to `1` and the variable `p` to `"Coloured"`, but it will not bind the variables `k` and `str`. A *partial binding* of a transition with variables $v_1, v_2, \ldots v_n$ is a binding $b = \langle v_1 = c_1, \ldots, v_n = c_n \rangle$ satisfying $c_i \in Type[v_i] \cup \{\perp\}$. As an example, matching the token `(1,"Coloured")` against the pattern `(n,p)` on the input arc from Send results in the partial binding ⟨`n=1, p="Coloured", str=`⊥`, k=`⊥⟩ of the transition ReceivePacket. In the following, PARTIALBIND(e,c) denotes the partial binding obtained by matching a pattern e and a token value c. If the pattern e and the token value c do not match, then we define PARTIALBIND$(e, c) = \perp$.

Since matching a token and a pattern results in a partial binding, each of the patterns in the pattern binding basis is considered in turn to gradually convert the partial binding into a binding where all the variables are bound. Moreover, the partial bindings obtained from one pattern in the pattern binding basis express only the constraints on the enabled bindings with respect to the marking of the corresponding input place, and hence must be *merged* with the partial bindings obtained by pattern matching with respect to the other input places. As an example, consider again Figure 5 and the transition ReceivePacket. Matching the pattern `(n,p)` and the tokens on place B yields the two partial bindings:

⟨`n=1, p="Coloured", str=`⊥`, k=`⊥⟩
⟨`n=2, p="Petri", str=`⊥`, k=`⊥⟩

To bind the variable `str`, the pattern `str` on the input arc from Received must be matched against the tokens on Received yielding the partial binding ⟨`n=`⊥`, p=`⊥`, str="Coloured", k=`⊥⟩. To obtain the combined partial binding, the set of partial bindings arising from pattern matching with respect to pattern `(n,p)` must be merged with the set of partial bindings obtained from pattern matching with respect to pattern `str`. The result of this merging is two partial bindings:

b_1 = ⟨`n=1, p="Coloured", str="Coloured", k=`⊥⟩
b_2 = ⟨`n=2, p="Petri", str="Coloured", k=`⊥⟩

To obtain the complete bindings the k pattern must be matched against the token on NextRec causing k to be bound to 2. Finally, the guard has to be checked causing the binding b_1 to be discarded since it does not satisfy the guard.

To formally define merging of two sets of partial binding elements we define the notion of *compatible* partial bindings. Two partial bindings b_1 and b_2 are compatible (written COMPATIBLE(b_1, b_2)) provided that they agree on the values bound to the variables. For two compatible partial bindings we can construct the *combined* partial binding (written COMBINE(b_1, b_2)) which is the partial binding obtained by inheriting the values bound to

the variables in both partial bindings. Formally, the notion of compatible and combined partial bindings can be defined as follows.

Definition 3. Let t be a transition. Two partial bindings b_1 and b_2 of t are compatible if and only if:

$$\forall v \in FV(t) : b_1(v) \neq \bot \wedge b_2(v) \neq \bot \Rightarrow b_1(v) = b_2(v)$$

For two compatible partial bindings b_1 and b_2, the combined partial binding is the partial binding b satisfying:

$$b(v) = \begin{cases} b_1(v) & : \quad b_1(v) \neq \bot \\ b_2(v) & : \quad b_2(v) \neq \bot \\ \bot & : \quad \text{otherwise} \end{cases}$$

Merging two sets of partial binding elements B_1 and B_2 is defined as:

$$\begin{aligned} &\textsc{Merge}(B_1, B_2) \\ &\quad = \{\textsc{Combine}(b_1, b_2) \mid \exists (b_1, b_2) \in B_1 \times B_2 : \textsc{Compatible}(b_1, b_2)\} \end{aligned}$$

It is worth observing that merging two sets of partial bindings can result in more partial bindings than in B_1 or B_2 as well as fewer bindings depending on which variables are bound in the two set of partial bindings.

The complete algorithm for computing the set of enabled binding elements of a transition in a marking M is listed in Figure 8. The algorithm applies the PARTIALBIND and MERGE procedures introduced above. The algorithm executes a loop (lines 3–12) considering each of the patterns in the pattern binding basis in turn. For each pattern a set of partial bindings C' is computed (lines 4–10) by matching the pattern and the tokens residing on the corresponding input place. The set of partial bindings is then merged with the current set of partial bindings (C) (line 11) and the next pattern is considered. When each of the patterns has been considered, it is checked whether the guard of the transition is satisfied (line 14), and whether sufficient tokens are present on the input places (line 15). It should be noted that in line 15 it suffices to consider only those input places which are not input places associated with one of the patterns in the pattern binding basis. The algorithm in Figure 8 is not specified in SML to abstract from a number of implementation details. We give a further description of how the algorithm is implemented in SML in Section 4.

The potential combinatorial blow-up in the merging of binding elements can be alleviated by creating *binding groups* which is a partitioning of the variables of a transition. For an example, consider the transition ReceiveAck (see Figure 1) which has variables `k` and `n` of type `int`. Assume that the place NextSend contains the multi-set `1‘2 + 1‘3` of two tokens,

```
1:  {PBB(t) = {E_j | 1 ≤ j ≤ l} pattern binding basis for t}
2:  C ← ∅
3:  for all E(p, t) ∈ PBB(t) do
4:     C' ← ∅
5:     for all c ∈ M(p) do
6:        b' ← PARTIALBIND(E(p, t), c)
7:        if b' ≠ ⊥ then
8:           C' ← C' ∪ { b' }
9:        end if
10:    end for
11:    C ← MERGE(C, C')
12: end for
13:
14: C ← { b ∈ C | G(t)⟨b⟩ }
15: C ← { b ∈ C | ∀p ∈ P_In(t) : E(p, t)⟨b⟩ ≤ M(p) }
16:
17: return C
```

Figure 8. Pure arc expression based inference algorithm.

and that the place D contains the multi-set `1‘4 + 1‘5 + 1‘6` of three tokens. This marking is not reachable from the initial marking of the CPN model, but it is still a legal distribution of tokens. Matching the pattern `n` against the tokens on place D yields three partial bindings of the ReceiveAck transition. Matching the pattern `k` against the tokens in place NextSend yields two partial binding elements. When these two sets of partial binding elements are merged we obtain six binding elements all of which are enabled. However, since there are no dependencies (directly or indirectly via guards) between the variable `k` and the variable `n`, the two variables can be bound independently. Hence, we can create two binding groups {`n`} and {`k`} for the ReceiveAck transition and bind the variables independently within each group similar to the algorithm in Figure 8. Merging of binding elements is only necessary within the binding groups, and finding an enabling binding of the transition amounts to picking a partial binding for each binding group of the transition.

3.3. Guard and arc expressions based algorithm

A drawback of the algorithm in Figure 8 is that many bindings may not be discarded until the very last part of the algorithm where those bindings that do not satisfy the guard are discarded. This is unfortunate since the guard of the transition quite often imposes a rather strong requirement on the bindings. It is, therefore, desirable if the guard is taken into account early when computing the enabled bindings.

The observation to make is that quite often the guard of a transition is in *conjunctive form*, i.e., $G(t) = \bigwedge_{i=1}^{m} G_i(t)$. Consider one of these conjuncts $G_i(t)$, and assume that it is of the form $G_{il} = G_{ir}$ where G_{il} and G_{ir} are expressions such that PATTERN(G_{il}). G_{ir} can be evaluated in a partial binding provided that the free CPN variables in G_{ir} are bound in the partial binding. If so, then we can match the value thus obtained with the pattern G_{il} and possibly bind free CPN variables occurring in G_{il}. If a match occurs, then the resulting partial binding will satisfy this guard conjunct. As an example, consider Figure 5 and the transition ReceivePacket. Matching the pattern `(n,p)` and the tokens on place *B* results in the following two partial bindings:

⟨`n=1, p="Coloured", str=`⊥`, k=`⊥⟩
⟨`n=2, p="Petri", str=`⊥`, k=`⊥⟩

The guard `[k=n]` is in conjunctive form, and we can evaluate the left-hand side of the guard `n` in the two partial bindings above and match them against the pattern `k`. This results in the two partial binding elements:

⟨`n=1, p="Coloured", str=`⊥`, k = 1`⟩
⟨`n=2, p="Petri", str=`⊥`, k = 2`⟩

The above idea leads to the concept of an *ordered pattern binding basis* which is similar to the concept of pattern binding basis introduced in Definition 2 except that an order in which variables are bound is imposed to exploit guards conjuncts. A further explanation is given below.

Definition 4. Let t be a transition such that $G(t) \equiv \bigwedge_{i=1}^{m} G_i(t)$. An *ordered pattern binding basis* $OPBB(t) = \{E_j \mid 1 \leq j \leq l\}$ for t is a set of input arc expressions and guard conjuncts of t satisfying:

1. $\mathsf{FV}(t) = \bigcup_{j=1}^{l} \mathsf{FV}(E_j)$
2. $\forall E_j \equiv E(p,t) \in OPBB(t) : \text{PATTERN}(E(p,t))$
3. $\forall E_j \equiv G_i(t) \in OPBB(t) : G_i(t) \equiv G_{il} = G_{ir} \wedge \text{PATTERN}(G_{il})$
4. $\forall E_j \equiv G_i(t) \equiv G_{il} = G_{ir} \in OPBB(t) : \mathsf{FV}(G_{ir}) \subseteq \bigcup_{h=1}^{j-1} \mathsf{FV}(E_h)$

Item 1 above ensures that all the variables of the transition are covered, and item 2 ensures that all input arc expressions included in the ordered pattern binding basis are patterns. Item 3 ensures that all guard conjuncts included have the desired form, and that the left-hand side of the conjunct is a pattern. Item 4 ensures that guard conjuncts included are such that the free CPN variables appearing on right-hand side will be bound by means of the previous members of the ordered pattern binding basis, i.e., the right-hand side of the guard conjunct can be evaluated in the partial bindings. As an example, {`(n,p),k=n,str`} is an ordered pattern binding basis for transition ReceivePacket.

```
1:  {OPBB(t) = {E_j | 1 ≤ j ≤ l} ordered pattern binding basis for t}
2:  C ← ∅
3:  for j = 1 to l do
4:     C' ← ∅
5:     if E_j ≡ G_i(t) ≡ G_il = G_ir then
6:        for all b ∈ C do
7:           b' ← PartialBind(G_il, G_ir⟨b⟩)
8:           if b' ≠⊥ then
9:              C' ← C' ∪ Merge({ b' }, { b })
10:          end if
11:       end for
12:       C ← C'
13:    else {E_j ≡ E(p, t) }
14:       for all c ∈ M(p) do
15:          b' ← PartialBind(E(p, t), c)
16:          if b' ≠⊥ then
17:             C' ← C' ∪ { b' }
18:          end if
19:       end for
20:       C ← Merge(C, C')
21:    end if
22: end for
23:
24: C ← { b ∈ C | G(t)⟨b⟩ }
25: C ← { b ∈ C | ∀p ∈ P_In(t) : E(p, t)⟨b⟩ ≤ M(p) }
26:
27: return C
```

Figure 9. Guard and arc expression based inference algorithm.

The complete algorithm for the enabling inference algorithm which exploits guards in addition to input arc expression is listed in Figure 9. The algorithm executes a loop (lines 3–22) considering each of the members of the ordered pattern binding basis in turn. Lines 5–12 consider the case of a guard conjunct. In this case, each of the partial bindings b in the current set of candidates is considered in turn. The right-hand side of the guard conjunct is evaluated in b (line 7) and matched against the left-hand side. The resulting partial binding b' is merged with b and inserted into the new set of partial bindings C'. The need for merging b and b' is due to the fact that the left-hand side of the guard conjunct may contain free CPN variables already bound in b. Lines 13–20 consider the case of an input arc expression. This case is identical to the first algorithm listed in Figure 8. When each of the members of the ordered pattern binding basis has been considered, it is checked whether the guard of the transition is satisfied (line 24) and whether sufficient tokens are present on the input places (line 25). It

should be noted that in line 25 it suffices to consider only those input places that are not input places associated with any of the patterns in the pattern binding basis, and in line 24 it suffices to check those guard conjuncts which were not part of the ordered pattern binding basis.

3.4. Multi-set arc expressions

Only input arc expressions which qualify as allowed patterns can be exploited by the above algorithms when inferring the set of enabled binding elements. This means that such arc expressions can only remove a single token from the corresponding input place. It is, however, convenient to be able to write *multi-set arc expressions* which can remove several tokens. Multi-set arc expressions have the form $E_{11}{}'E_{12} + \ldots + E_{n1}{}'E_{n2}$ where E_{i1} is an expression of type integer, and the type of E_{i2} is the same as the type of the input place of the arc. Below we sketch how to exploit such multi-set arc expressions when inferring the enabled binding elements.

The first step is to handle arc expressions of the form $k'E$, where $k > 0$ is an integer constant, and E is a pattern. Such an expression can be handled using E as a pattern to be matched against tokens. The coefficient k is then taken into account in line 25 of the algorithm in Figure 9 where it is checked whether sufficient tokens are present on each input place. Handling multi-set arc expressions of the form $E_1{}'E_2$ where E_2 is a pattern, and E_1 evaluates to an integer can be handled similarly using the pattern E_2 to match against tokens. The only modification required is caused by E_1 which may contain free CPN variables. Hence, it has to be ensured that the variables in $\mathsf{FV}(E_1)$ are bound via other input arc expressions. The general multi-set arc expressions of the form $E_{11}{}'E_{12} + \cdots + E_{n1}{}'E_{n2}$ can be handled by splitting the arc expression and treating the transition as if it has n input arcs. The arc expression of the i'th arc is then $E_{1i}{}'E_{2i}$ which can be handled as explained above. Handling multiple input arcs from a given place requires the check in line 25 of the algorithm in Figure 9 to be generalised such that it takes into account tokens reserved by previously checked input arcs from a given place.

4. The CPN computer tools

The enabling inference algorithm presented in the previous section constitutes the core of the two CPN computer tools: CPN Tools [72] and Design/CPN [13]. These tools support construction, simulation (execution), state space analysis, and performance analysis of CPN models. The main difference between CPN Tools and Design/CPN is the graphical user interface; they both have the same semantic foundation.

The overall architecture of the CPN computer tools is shown in Figure 10. Both tools consist of two main components: the Graphical User Interface (GUI) and CPN ML. These two components span the three tightly integrated tools: the CPN Editor, the CPN Simulator, and the CPN State Space Tool. The GUI and CPN ML run as two separate processes communicating via TCP/IP. The CPN ML part is implemented on top of the SML/NJ compiler [56]. The TCP/IP connection between the GUI and CPN ML allows the GUI to access the top-loop of the SML/NJ compiler.

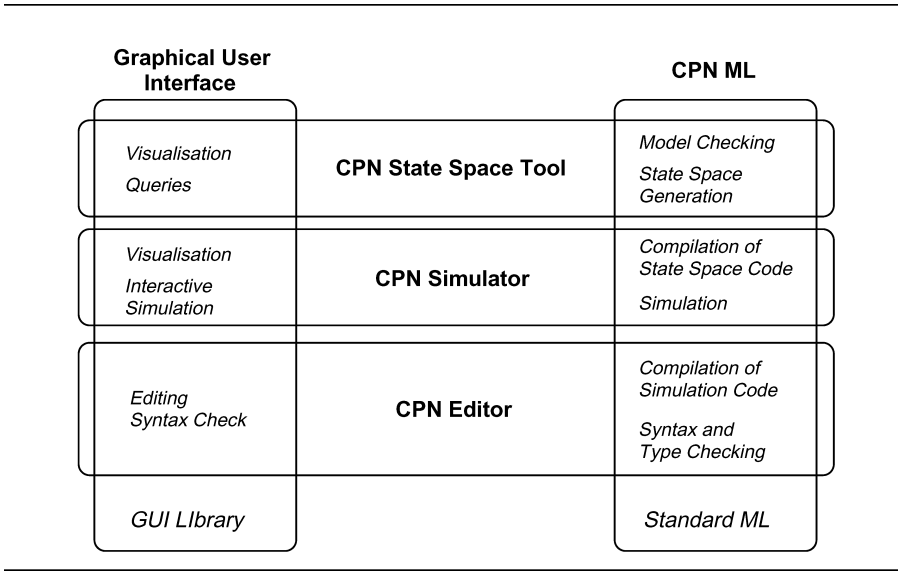

Figure 10. Architectural overview of the CPN computer tools.

4.1. The editor

The CPN editor is used for construction of CPN models. The CPN model in Figure 1 modelling the simple transport protocol, is an example of a CPN model constructed using the CPN editor. The user works directly on the graphical representation of the CPN model.

To simulate a CPN model, it has to constitute a syntactical and type correct CPN. All syntactical checks related to the SML part of CPNs, i.e., the types, functions, and arc expressions are handled using the SML/NJ system. The SML/NJ type system is also used to check that the required type constraints are satisfied, e.g., that the arc expressions satisfy the constraints listed in Section 2.4. Syntax and type errors are reported to the user by intercepting the compiler errors returned by the SML/NJ system and annotating them with information about the source of the problem, e.g., that the arc expression and the corresponding place do not have the same type. The syntactical checks related to the Petri Net part, e.g., that each arc has an arc expression and that each place has a type, are checked in the GUI.

4.2. The simulator

When the CPN model does not contain any syntax or type errors, the user can *switch* to the CPN simulator. The switch to the CPN simulator generates the SML code necessary for simulation (execution) of the CPN model. A minimal ordered pattern binding basis is also determined for each of the transitions in the CPN model. Computing a minimal pattern

binding basis is equivalent to the SET-COVERING problem which is NP-complete. However, computing a pattern binding basis need only be done once, and in practice only relatively small instances of the set covering problem will be considered. In the implementation in the CPN computer tools a greedy algorithm is applied. When computing the ordered pattern binding basis, this algorithm picks the pattern containing the most CPN variables among those patterns not yet included.

Two SML functions are generated for each transition in the CPN model: an *enabling function* for computing the enabled bindings of the transition in a given marking, and an *occurrence function* for computing the marking resulting from an occurrence of the transition in a given marking. The enabling function implements the algorithm presented in Figure 9 based on the ordered pattern binding basis computed prior to simulation code generation. The PARTIALBIND procedure is implemented by generating an SML function for each of the arc expressions in the ordered pattern binding basis. As an example, for the arc expression `(n,p)` on the arc from B to ReceivePacket (see Figure 1) the following function is generated:

```
fun PartialBind_B_ReceivePacket (n,p) =
                    {n=SOME n, p=SOME p, str=NONE, k=NONE}
```

Provided with a token of type `SEQxDATA` this function will return a partial binding element where values have been bound to the variables `n` and `p`. This function is then mapped onto the tokens residing on place B in the current marking to obtain the partial binding elements. The PARTIALBIND procedure for guards is implemented by generating an SML function where the left-hand side of the guard (G_{il}) is used as a pattern in the function.

Simulation of CPN models corresponds to the execution of the generated SML code. An alternative approach would have been to implement a virtual machine for the execution of CPN models. The code generation approach was chosen to make the full SML language and SML/NJ environment available to the modeller for construction of CPN models. If this was to be achieved with a virtual machine approach, then a virtual machine for SML code execution would essentially have to be implemented, in which case it is simpler to rely on an existing environment as is done with the code generation approach. Another reason is simulation speed. An approach based on code generation is likely to be faster than an approach based on introducing an extra layer in the form of a virtual machine. The code generation approach together with the use of the SML/NJ compiler means that the simulation code is executed directly on the underlying machine architecture.

The CPN computer tools implement a slightly more general variant of the algorithm in Figure 9 in that variables of small domains are not required to be bound via patterns. This is for an example how the boolean variable `success` of the transitions TransmitPacket and TransmitAck is handled. Since the domain of the variable success contains only the two values `true` and `false`, the enabling inference algorithm as implemented in the CPN computer tools will simply try both possibilities.

Once the SML code generated for simulation has been compiled, the user is ready to start investigating the behaviour of the system by means of simulations. The CPN computer tools supports two simulation modes: interactive simulation and automatic simulation. Interactive

simulation has the characteristics of single step debugging in which the behaviour of the CPN model is observed in great detail, and the user chooses the next enabled binding element to occur. During such simulations, the markings of the places are shown directly on the CPN diagram similar to Figure 5. In automatic simulation, the CPN model is simply executed without any graphical feedback at the level of the CPN model. Automatic simulations combined with data collection are typically used to evaluate the performance of the system under consideration.

4.3. The state space tool

Verification and validation of CPN models are supported by the CPN State Space Tool. The state space tool supports verification and validation of the functional correctness of systems and is the main formal analysis method for CPNs. The basic idea behind *state spaces* is to construct a directed graph which has a node for each reachable marking and an arc for each occurring binding element. The state space of a CPN model represents all interleaved executions of the CPN model. State spaces of CPNs can be constructed fully automatically, and from a constructed state space it is possible to answer a large set of analysis and verification questions concerning behavioural properties of finite-state systems. Examples of such properties are the absence of deadlocks in the system, the possibility of always reaching a given state, and the guaranteed delivery of a given service. Moreover, if the system does not have a desired property then the state space method can provide debug information in the form of counter examples (error-traces). The counter examples are typically a node, path, or a subset of the state space demonstrating why the desired property does not hold.

The switch from the CPN simulator to the CPN state space tool is similar to the switch from the editor to the simulator, and it consists of a compilation of the necessary SML data structures for working with the state space tool. This is handled by the simulator part of CPN ML. The state space generation is handled by the CPN ML part of the state space tool which uses the enabling inference algorithm to compute the set of enabled binding elements and corresponding successor markings for each marking encountered during the state space construction.

The CPN state space tool supports two modes of state space generation: interactive generation and automatic generation. In interactive generation, the user selects the states to explore next and is in full control of the state space generation. Interactive generation combined with drawing is useful when investigating in detail the states reachable from within a few steps of a given state. In automatic generation, the CPN state space tool conducts a depth-first or breadth-first generation of the state space. Automatic generation can be controlled by giving upper bounds on the generation time and/or the number of states to be generated.

A state space report containing answers to a set of standard dynamic properties of CPNs such as the minimum and maximum number of tokens on the individual places, and reachable markings without enabled binding elements, can be generated fully automatically and saved in a textual file. The user is also able to write queries expressing behavioural properties of the system under consideration using a set of SML functions available for traversing the state

space, or using a state and action oriented variant of the temporal logic CTL (Computation Tree Logic) [4, 10].

The CPN state space tool supports two reduction methods for alleviating the state explosion problem [76]: the symmetry method [15, 35] and the sweep-line method [7, 42].

The basic idea of the symmetry method is to exploit the symmetry present in many systems to construct a condensed state space where each node represents an equivalence class of symmetric states, and each arc represents an equivalence class of symmetric binding elements. The condensed state space is typically orders of magnitude smaller than the ordinary full state space, and verification of properties can be done directly on the condensed state space, i.e., by considering only a representative from each equivalence class. A case study for a flowmeter system [46] showed a reduction in memory usage to as much as 2%, and a reduction in generation time to 25% with the symmetry method.

The sweep-line method exploits a notion of progress present in many concurrent systems. Examples of sources of progress are control flow in processes, sequence numbers and timers in communication protocols, and time in timed CPNs (see below). Exploiting progress makes it possible to delete states from memory during state space generation, thereby reducing the peak memory usage. Verification of properties are then conducted on-the-fly during the state space generation. Practical experiments on the Wireless Application Protocol (WAP) [25] showed a reduction in peak memory usage to 20%, and a reduction in generation time to 25% with the sweep-line method.

4.4. Timed CPNs

The CPN computer tools also support timed CPNs. The time concept of CPNs is based on the introduction of a discrete *global clock*. The clock values represent *model time*, and each token carries a *time value*, also called a *time stamp*. Intuitively, the time stamp describes the earliest model time at which the token can be used, i.e., removed by the occurrence of a binding element. In a timed CPN a binding element is said to be *colour enabled* when it satisfies the enabling rule for untimed CPNs, i.e., when the required tokens are present at the input places and the guard evaluates to true. However, to be enabled, the binding element must also be *ready*. This means that the time stamps of the tokens to be removed must be less than or equal to the current model time. The CPN model remains at the current model time as long as there are enabled binding elements. When no more binding elements are enabled (i.e., colour enabled and ready) at the current model time, the global clock is incremented to the earliest next model time at which transitions are enabled. To model that an activity/event corresponding to the occurrence of transition takes r units of time, the tokens produced on output places of the transition are given time stamps which are r time units higher than the model time at which the transition occurs. The tokens will then be unavailable for r time units as they cannot be removed by occurrences of transitions before the model time has been increased by at least r time units.

Simulation and state space analysis of timed CPNs is achieved by adapting the enabling inference algorithm to also take into account the time stamps of tokens when computing the set of enabled binding elements. The basic idea is to first compute a set of colour enabled

binding elements using the algorithms presented in this paper. For each colour enabled binding element, its enabling time can be computed from the time stamps of the tokens it consumes. The ready binding elements is then the subset of the colour enabled binding elements with minimal enabling time.

The CPN state space tool also supports state spaces for timed CPNs. The problem with timed CPNs and state spaces is however that the absolute notion of time represented by the global clock becomes part of the marking/state of the CPN model. This implies that for cyclic/reactive systems, the state space for timed CPNs becomes infinite. The CPN state space tool therefore implements the time equivalence method [6] to factor out the absolute notion of time and obtain a finite state space. We compare the time concept of CPNs to other time concepts in Section 7.

4.5. Hierarchical CPNs

The CPN computer tools allow CPN models to be constructed as *hierarchical* CPNs by composing a number of smaller CPN models called *pages* similar to the one in Figure 1. The pages are related to each other in a well-defined way by means of *substitution transitions*. The CPN computer tools exploit the *locality* of Petri Nets to ensure that the calculation of enabled binding elements scales for large CPN models. The basic observation is that the occurrence of a binding element affects only the marking of the input and output places of the corresponding transition. Hence when a binding element (t, b) occurs, thus changing the marking of the CPN model, it suffices to recalculate enabling for those transitions with an input place which is also an input or output place of t. This ensures that the enabling inference algorithm is not applied to transitions which are unaffected by the occurrence of a binding element, and it ensures that the time spent on calculation of enabled binding elements is independent of the size of the CPN model. We will give an example of a hierarchical CPN in Section 6.

5. Design and analysis of systems using CPNs

In this section we survey a number of industrial projects in which CPNs and the CPN computer tools have been used for design and analysis of concurrent systems. The aim of this section is not to give a detailed description of the projects and their results, but to give a flavour of the kind of concurrent systems where the techniques and algorithms presented in previous sections have been put into practice. We provide more detail on two case studies in Section 6 where CPNs have been used for the implementation of systems. Further examples of CPN projects and case studies can be found in Jensen [36] and on the Design/CPN homepage [13].

Communication gateways at Australian Defence Forces. In this project [16], CPNs were used to specify and design gateways between Radio Networks and Broadband Integrated Services Digital Networks (B-ISDN) to be used in the Australian Defence Forces Tactical Packet Radio Network (TPRN). A TPRN consists of a number of mobile radio nodes, and the role of the gateway is to make it possible to use a B-ISDN infrastructure as a backbone for connecting

different TPRNs. Starting from an initial behavioural specification of the gateway the design was gradually refined. The CPN state space tool was used to verify the basic behaviour of the gateway such as testing for deadlocks and the correct establishment of calls. The analysis revealed that first attempts to refine the specification did not meet the specification.

Audio/video systems at Bang&Olufsen (B&O). In this project [5], CPNs were used to validate vital parts of the B&O BeoLink system. The BeoLink makes it possible to distribute sound and vision throughout a home via a network. The CPN state space analysis part of the project focused on the lock management protocol of the BeoLink. The protocol is used to grant devices in the system exclusive access to various services in the system. More specifically, the state space analysis established that in the initialisation phase of the system, a single *key* is eventually generated. The key is used by the devices to obtain exclusive access to the services. An interesting aspect of the project was that state spaces were generated for a timed CPN model, i.e., a CPN model which in addition to specifying the logical behaviour of the system also specifies the time taken by different activities. The model was timed since timing is crucial in the communication protocols of the BeoLink system.

Mobile phone software at Nokia Research Center. In this project [79], CPNs were used for modelling and analysis of a new software architecture for a mobile phone family. The purpose of the modelling was to be able to analyse the time and space performance of the software system and for configuring different product family members. The purpose of applying state spaces was primarily to ensure the correctness of the models. State space analysis was conducted on several sub-models and for different scenarios to investigate the properties of the communication protocols including the interaction protocols for the call control system. It was, e.g., established that in a call collision case (in which a user makes a call at the same time as a call comes from the network) the protocols behave correctly in that they have no deadlocks and lead to termination in the desired state. Several non-trivial errors were detected showing that state space analysis is an effective way of debugging a model and a system.

Flowmeter systems at Danfoss A/S. In this project [46], CPNs were used for the modelling and analysis of a flowmeter system. A modern flowmeter system consists of a number of communicating processes cooperating to make various measurements on, e.g., the flow of water through a pipe. The purpose of the project was to investigate the use of CPNs for validating the communication protocols in the flowmeter system. The state space analysis part of the project aimed at applying state spaces for verifying crucial properties of the flowmeter system. The state space analysis identified deadlocks and data consistency problems in the proposed communication protocols. The symmetry method for state spaces was applied for the verification of a modified design proposal which avoided the identified deadlock and data consistency problems. The symmetries were based on the observation that the processes in a flowmeter system behave in a similar way.

Mechatronic systems at Peugeot-Citroën. In this project [52], CPNs were used for the modelling and analysis of a so-called *mechatronic system*. Active suspension, automatic gear

boxes and engine control in modern cars are all examples of mechatronic systems which can be characterised as a hybrid system consisting of a continuous physical system, a discrete control system, and some actuators and sensors allowing the discrete control system to communicate with the physical system. A central activity in designing a mechatronic system is dependability evaluation which is concerned with identifying sequences of events which can lead to so-called feared events or faults. In the project a quite simple mechatronic system consisting of a pressure tank containing oil, a pump supplying oil to the tank and a consumer of oil was considered. State spaces were used to perform qualitative dependability analysis of the system in the presence of re-configurations. The use of state spaces established some of the most important properties of the system including the identification of which sequences of events can lead to the feared events.

6. Implementing systems using CPNs

A CPN model (and models in general) is usually a simplified and abstract representation of the system under consideration. As such, a model is mainly useful for design and evaluation of a system at an abstract level, and there is a gap between the model specifying the system design and the actual implementation of the system.

In this section we present two projects where CPN models have been used to obtain an implementation of a system. The two projects are examples of how the gap between the specification of a system as a CPN model and the actual implementation of the system can be automatically bridged. This effectively eliminates a manual step in which the design as captured by the CPN model is "translated" into an implementation of the system. The fact that the CPN computer tools rely on a full programming language and environment (as offered by SML/NJ) has made this possible.

6.1. COAST: A task planning tool

This project [81] was concerned with the development of a logistics and planning tool by the Australian Defence Science and Technology Organisation (DSTO). The role of a CPN model in the context of the Course of Action Scheduling Tool (COAST) was to provide a sound semantical foundation by formalising the abstract conceptual framework underlying the tool.

The framework underlying COAST is based on the notion of tasks which have associated pre- and postconditions describing the conditions required for a task to start and the effect of executing the task. The key capability of COAST is the computation of task schedules according to available resources and the constraints imposed by pre- and postconditions of tasks. COAST also supports a number of task synchronisation mechanisms, such as requiring a set of tasks to start simultaneously, specifying the absolute start time of a task, and specifying a relative time between the start of certain tasks. The basic idea in COAST is to use a CPN model for modelling the execution of tasks according to the pre- and postconditions of tasks, imposed synchronisations, and available resources. The task schedules are obtained by generating a state space for the CPN model and extracting paths in the state space leading from the initial state to states where certain conditions are satisfied.

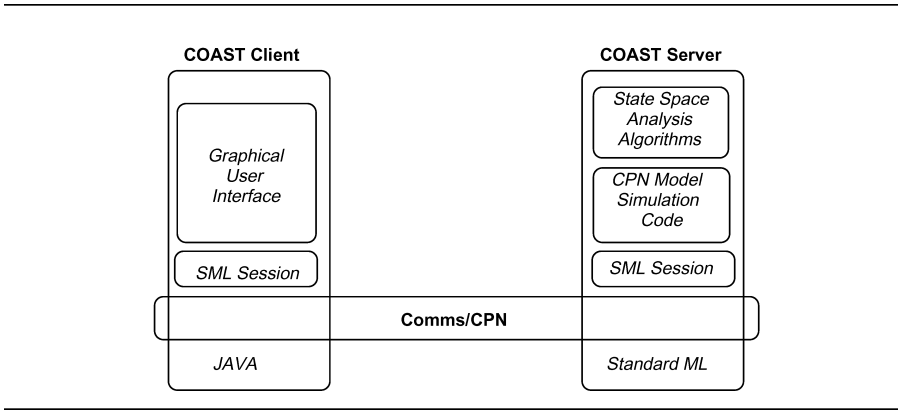

Figure 11. Architectural overview of the COAST tool.

Figure 11 shows the client-server software architecture of COAST. The COAST client, which includes a domain-specific graphical user interface, is implemented in the Java [32] programming language whereas the COAST server is implemented in SML via the embedded CPN model which forms the core of the COAST server. Communication between the client and the server is based on Comms/CPN [20], a library supporting TCP/IP communication between CPN models and external applications. An SML Session layer has been implemented on top of Comms/CPN. This layer allows the client to invoke functions available in the server and receive the corresponding results. The SML Session layer is implemented by allowing the client to submit SML code to the server for evaluation. The received SML code is then executed by the server, and results are sent back to the client. The SML code sent to the server corresponds to the invocation of the SML functions made available by the server via the Analysis module. The Analysis module contains functions that allow the state space of the CPN model to be computed and traversed in order to extract the desired task schedules. As an example, generating of the state space of the CPN model is done by sending the string `Analysis.Generate();` to the server. The server will compute the state space and return the string `GenerationEnd` to the client when generation is completed. This approach essentially gives the COAST client access to the read-eval-print loop of the SML/NJ compiler, and the approach was chosen because of its simplicity and flexibility.

The CPN model has been parameterised with respect to the set of tasks, resources, and task synchronisations. This ensures that a given set of tasks, resources, and task synchronisation can be analysed by setting the initial marking of the CPN model accordingly, i.e., no changes to the structure of the CPN model is required to analyse a different set of tasks. In the following we focus on the CPN model embedded in the COAST server.

Figure 12 shows the *hierarchy page* for the CPN model. Each node in Figure 12 corresponds to a page of the CPN model and an arc between two nodes specifies that the destination page of the arc is a *subpage* of the source page of the arc. The page CoastServer is the highest level page in the CPN model which consists of 22 pages grouped into three

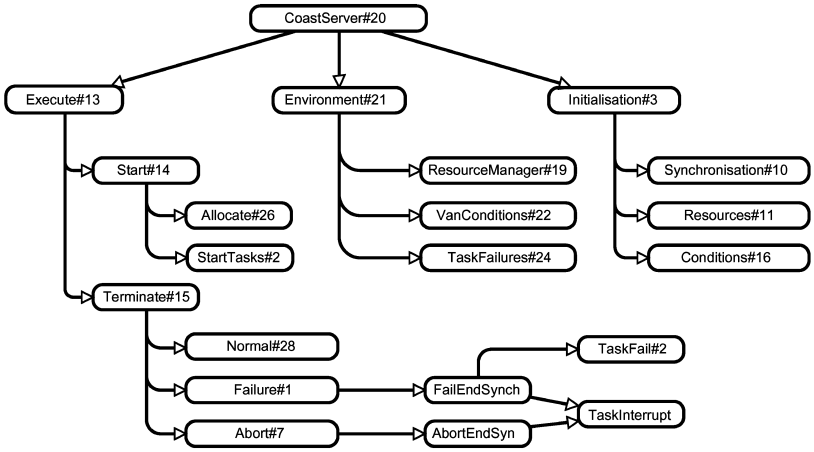

Figure 12. Hierarchy page of the COAST CPN model.

main parts. Page Execute (left) and its subpages model the execution of tasks, i.e, start, termination, abortion, and failure of tasks. Page Environment and its subpages model the environment in which tasks execute, i.e., how external events influence the execution of tasks. Page Initialisation and its subpages are used for the initialisation of the model according to the concrete set of tasks, synchronisation, and resources to be considered. The CPN model is timed since capturing the time taken by executing a task is an important part of the computation of task schedules.

To give an example of a page in the CPN model, consider Figure 13 which shows page TaskFail modelling the failure of a task during execution. The other pages in the CPN model contain more places and transition than page TaskFail, but the arc expressions on these

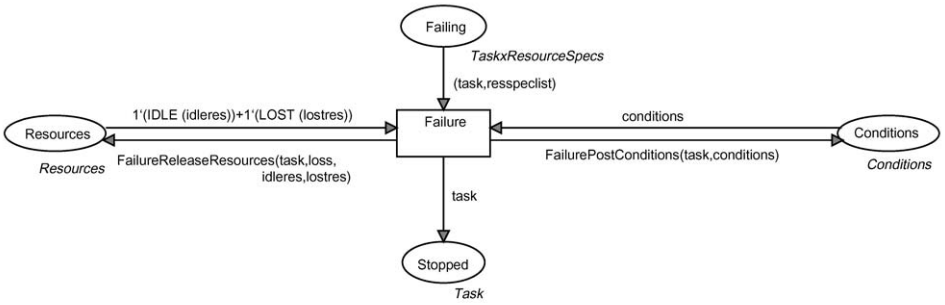

Figure 13. Example page from COAST CPN model.

pages are of a similar complexity as the ones on the TaskFail page. The COAST CPN model contains altogether 32 transitions, 140 places, and 45 type definitions.

The failing task will be represented as a token on place Failing. This will be a token of type TaskxResourceSpecs which is a product type used to describe a task and the resources assigned to the task when it started. Tasks in the CPN model are represented as tokens of a complex record type with 15 fields describing the attributes of the tasks such as pre- and postconditions, required resources, and duration. Place Conditions contains a token of a list type describing the conditions which are valid in the current state. Place Resources always contains two tokens representing the current set of available resources, and the resources that have been lost until now.

The failure of a task corresponds to an occurrence of the transition Failure. An occurrence of this transition will remove a token corresponding to the failing task from place Failing. The failing task corresponding to the value bound to the CPN variable task will be put as a token on place Stopped. An occurrence of Failing removes the list of conditions from place Conditions, and updates them according to the postconditions of the failing task. This is done by the function FailurePostConditions. An occurrence of Failing also removes the lists of idle and lost resources from place Resoures and updates them according to the release of resources for the failing task. This is handled by the function FailureReleaseResources.

The COAST server was obtained from the CPN model by switching the CPN model from the CPN editor to the CPN State Space Tool as described in Section 4. SML files implementing Comms/CPN and the SML session layer were then loaded together with the functions implementing the server loop and the task scheduling analysis algorithms. The resulting executable file constitutes the COAST server and includes the functions required to execute the CPN model and to conduct state space analysis. The COAST server is totally detached from the GUI of the CPN computer tools. When the COAST server is started, it will wait for an incoming TCP connection on a given port, and once the COAST client has established a connection, it can start invoking functions on the COAST server and conduct the task scheduling analysis. That the COAST server uses a CPN model as a basis for the scheduling analysis is fully transparent to the analyst using the COAST client.

The use of SML as part of the CPN computer tools was instrumental in several ways in the development of COAST. It allowed a highly compact and parameterisable CPN model to be constructed, and it allowed the CPN model to become the implementation of the COAST server. The parameterisation is important to ensure that the COAST server is able to analyse any set of tasks, resources, and synchronisations without having to make changes to the CPN model. Having a full programming language available made it possible to extend the COAST server with the specialised algorithms required to extract the task schedules from a generated state space.

Typical planning problems on which COAST is applied consist of 15 to 25 tasks resulting in state spaces with 10,000 to 20,000 nodes and 25,000 to 35,000 arcs. Such state spaces can be generated in less than 2 minutes on a standard PC. The state spaces are relatively small because the conditions, available resources, and imposed synchronisations in practice strongly limit the possible orders in which the tasks can be executed.

6.2. A security system

This project [53, 62] considered the specification, design, and implementation of software to be used in a new version of a security system developed by the Danish engineering company Dalcotech A/S. The security system consists of a central control unit and a number of sensors, actuators, and control panels connected via a dedicated network to the central control unit. The central control unit is in turn connected to an alarm receiving centre via the public telephone network. The project was concerned with the design and implementation of the software running on the central control unit, responsible for controlling the sensors, actuators, and control panels in the system.

Simulation and state spaces were used to validate and verify the designed security system. Crucial properties of the system were verified such as reversibility (the system can always return to its initial state), and if a detector is triggered, then an alarm is generated and sent to the alarm control centre. In spite of the fact that only small configurations of the system were analysed using state spaces, approximately 15 non-trivial errors in the design were identified, some of which would very likely also have existed in the final implementation of the system. The state spaces generated for different configurations of the security system had 50,000 to 250,000 nodes.

Figure 14 shows the hierarchy page for the CPN model of the security system. The CPN model consists of 21 pages which can be divided into three main parts. Page Initialize and its

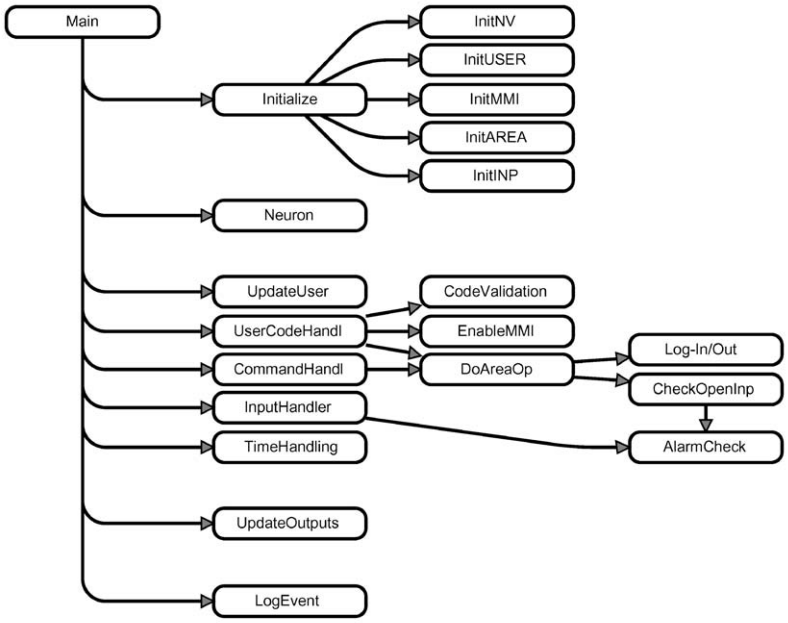

Figure 14. Hierarchy page of the Dalcotech CPN model.

subpages handle the initialisation of the CPN model. Page Neuron models the interface to the underlying network connecting the central control unit of the security system to the external devices in the security system such as sensors and actuators. This part of the model emulates the underlying communication network while analysing the CPN model, and is replaced with the actual communication library when obtaining the running version of the system as will be explained below. The remaining pages specify the software of the central control unit.

The implementation of the central control unit was obtained by extracting the SML simulation code generated by the CPN computer tools during the switch from the CPN Editor to the CPN Simulator. After the code had been extracted, the emulation of the underlying network was also replaced by the communication library supporting the actual communication with the external devices across the network. The extraction of the code was needed since the central control unit was required to run on the MS-DOS platform where Moscow ML [54] at the time of the project was the only available SML system.

Automatically generating the code for the final system offered a number of benefits for Dalcotech A/S in their system development process. Firstly, it reduced development time and cost compared to the traditional manual (and sometimes error-prone) translation from the CPN specification to the final implementation in C++ [70]. Secondly, the simulation and state space analysis results, e.g, absence of deadlocks, obtained during evaluation of the system were valid also for the final implementation of the system. Thirdly, there was always consistency between the CPN model documenting the design of the system and the actual implementation of the system.

7. Related work

Several variants of high-level Petri Nets languages have been developed using the different programming languages for inscriptions and implementation. For an example, Reinke [63] uses the functional programming language Haskell [57] as inscription and implementation language for a high-level Petri Net simulator. In the Renew tool [43], Kummer uses Java [32] as inscription and implementation language.

Calculation of enabling for high-level Petri Nets has also been considered for Well-formed Nets [31] and Stochastic Well-Formed Nets [19] as supported by the GreatSPN tool [27]. Compared to CPNs, both Well-formed Nets and Stochastic Well-Formed Nets have a restricted inscription language that allows only tuples and enumeration types over finite domains as types for places. Mäkelä [47] considers calculation of enabling for a class of high-level Petri Nets called Algebraic System Nets which form the basis of the MARIA tool [48]. The calculation of enabling in [47] is inspired by the calculation of enabling in the CPN computer tools and is based on matching of algebraic terms. Algebraic System Nets and CPNs differ in their formal definition, but the approach based on matching of algebraic terms is similar to the approach based on SML pattern matching presented in this paper. Sanders [67] formulates calculation of enabled binding elements of CPNs as a constraint satisfaction problem by allowing only multi-set input arc expressions with constant coefficients.

Steggles [69] uses rewriting logic and the supporting Elan tool to implement the formal semantics of low-level Petri nets with time. In low-level Petri Nets, tokens do not carry data

values and hence there is no concept of types. The equivalent of an arc expression in a low-level Petri net is an *arc weight* which species the number of tokens removed (added) from an input (output) place when a transition occurs. The basic idea in [69] is to use rewrite rules to model the dynamic behaviour of Petri Nets, where a system state (marking) is represented by a multi-set of algebraic terms derived from the set of places. The built-in support in Elan for Associative-Commutative (AC) matching [14] and strategies is then used to match markings against rewrite rules. Steggles approach could be used for CPNs by unfolding CPNs to behaviourally equivalent low-level Petri Nets. This is, however, problematic, since for real-world systems, the number of places and transitions in the equivalent low-level Petri Nets is huge and can be infinite if the CPN has infinite types. This a problem both with respect to representation of the low-level Petri Nets itself, and with respect to the rewrite rules given in [69] which refer to the individual places. A practical approach for CPNs based on rewrite rules therefore seems to require a more symbolic representation of the rewrite rules than what is given in [69], and a truly high-level approach based on rewriting logic and AC-matching has to the best of our knowledge not been developed for high-level Petri Nets. Another problem is how to handle timed models efficiently. Steggles [69] handles progression of time by single clock ticks which means that there will be an event in the model for each unit of time. An advantage of our approach as implemented in the CPN computer tools, is that the model time can be advanced directly to the earliest next time at which a transition becomes ready. This significantly reduces the number of events that are simulated and hence speeds up simulations.

Our approach presented in this paper imposes some restrictions on the arc expressions in that a pattern binding basis is required to exist for each transition. More advanced techniques for Associative-Commutative (AC) pattern matching and unification [45] could be used to relax these restrictions, but we have chosen to base our approach on what can be implemented directly with the existing pattern matching capabilities of SML.

The PEP [26] tool is another Petri Net based tool which also allows process algebraic specification of systems. In PEP, a system specification is always translated (unfolded) into an ordinary (low-level) Petri Net before verification. A similar approach is taken in the LoLA tool [68]. As discussed above, this unfolding approach often entails a complexity problem with respect to analysis of many real-world systems. A main design philosophy of the CPN computer tools has been to do simulation and analysis directly on the CPNs without unfolding. The many large-scale projects where CPNs have been applied have shown the importance of this approach compared to an approach based on unfolding.

Petri Nets (and CPNs) is one out of many modelling languages for specifying and analysing the behaviour of concurrent systems [17]. Other related modelling languages include *Statecharts* [28] as supported by, e.g., the VisualState tool [78], *Calculus of Communicating Systems* [50] as supported by, e.g., the Edinburgh Concurrency Workbench [71], *timed automata* [2] as supported by, e.g., the UPPAAL tool [44], and *Communicating Sequential Processes* [29] as supported by, e.g., the FDR tool [18]. One difference between Petri Nets and the above modelling languages is that in Petri Nets, actions are concurrent unless explicitly specified otherwise, while most other modelling languages with concurrency have an explicit form of concurrency, i.e., the modeller needs to specify what is concurrent,

e.g., by explicitly using a parallel composition operator. Another difference is that Petri Nets have a true concurrency semantics (in contrast to an interleaved semantics) allowing for modelling of systems with truly concurrent processes and threads.

SPIN [30] is a widely used tool for design and analysis of systems. Like the CPN computer tools, SPIN supports editing, simulation, and state space analysis of models. Input to SPIN is given in the C-like textual language PROMELA. We believe that the approach to modelling by drawing is one of the main virtues of Petri Nets and other graphical modelling languages, such as e.g., Statecharts. The lessons learned from the practical use of CPNs have clearly shown the value of being able to convey design ideas and system behaviour in a graphical form. The SPIN tool implements partial order methods [59, 74] to alleviate the state explosion problem. Some initial developments on using partial order methods for CPNs are discussed in [41, 75]. The main problem with the use of partial order methods for CPNs is to avoid the costly unfolding to low-level Petri Nets.

The time concept of CPNs is a discrete time concept based on the introduction of a single clock. More elaborate time concepts has been developed for Petri Nets. Examples of this includes stochastic Petri Nets [49] aimed at performance analysis of systems, and interval timed Petri Nets [77] based on a continuous time concept. Timed Petri Nets [1] inspired by the continuous time concept of timed automata [2] has also been developed. The time concept of CPNs is relatively simple compared to more elaborate models above. Even so, it has been successfully applied for simulation-based performance analysis in a number of case studies, e.g., in the areas of high-speed interconnects [8] and ATM networks [11], and for verification of real-time systems in, e.g., [5] as discussed earlier in this paper.

CPNs are also related to functional programming languages with concurrency primitives via the use of SML. Some examples of functional programming with concurrency are Concurrent ML [65], Concurrent Haskell [39], Concurrent Clean [55], and Facile [22]. One difference between CPNs and these languages is, of course, the graphical nature of CPNs. Another main difference is that CPNs are aimed at being an implementation-independent language for specification and analysis of concurrent systems at different levels of abstraction rather than being a pure implementation language.

The verification approach taken in CPNs is based on state spaces and is hence aimed at finite-state systems. Another main approach to verification of systems is based on theorem proving and higher-order logics [23]. Examples of tools supporting the theorem proving approach to verification are PVS [61] and the HOL system [24]. The theorem proving approach is not restricted to verification of finite-state system, and is very generally applicable. The use of theorem proving requires, however, considerable mathematical skills by the system designer. A practical advantage of CPNs and many other modelling languages for finite-state systems is that they to a larger extent allow the underlying mathematics to be hidden inside the supporting computer tools.

8. Conclusions

In this paper we have shown how functional programming and SML have been applied in the development and implementation of CPNs and their supporting computer tools. At the

modelling language level, SML has provided CPNs with the concepts of types, variables and expressions, and it has provided the practical modelling convenience and expressiveness required to support modelling of industrial systems. At the implementation level, SML has provided the type system required to check the type requirements of CPNs. Furthermore, it has provided the support for pattern matching which is the key technique used in the token-based enabling inference mechanism that constitutes the semantical core of the CPN computer tools.

The integration of the SML/NJ environment into the CPN computer tools has had a number of advantages. Firstly, the expressiveness of SML is inherited by the CPN computer tools, and a full programming language becomes available for the construction of models. Secondly, SML is strongly typed, allowing many modelling errors to be caught early. Thirdly, polymorphism and definition of infix operators in SML allow inscriptions to be written in a natural, mathematical syntax. Finally, SML is well documented, tested and maintained [3, 58, 73]. The flexibility and extensibility inherited via the SML/NJ environment has also been paramount when developing a number of extensions to the CPN computer tools such as the Comms/CPN library [20]. The two projects reported on in this paper where CPNs have been used for the implementation of systems have also relied heavily on the generality inherited from the SML/NJ environment. This generality has made it possible to adapt the CPN computer tools to specific application domains in a number of cases.

The main challenge in using SML as the inscription language of CPNs has been in teaching engineers how to construct models with the CPN computer tools. Most engineers are unfamiliar with the concepts of functional programming such as type inference, evaluation of expressions, and expressing iterations using recursion. A course on modelling with CPNs must often be combined with a short course on functional programming and SML. However, after a one week course on CPNs and SML, it is our experience that engineers with a background in imperative programming languages can start constructing their own CPN models.

A disadvantage experienced with the use of SML/NJ is related to garbage collection and memory consumption. This is particularly evident in the state space tool where the generation of state spaces in the CPN computer tools consumes more memory (and time) than other verification tools with similar algorithms implemented in languages like C and C++. This disadvantage of using SML in the state space tool has, however, been outweighed by the generality and modelling convenience inherited from SML. As part of future work, we are investigating the feasibility of a hybrid implementation, where the storage mechanism for state spaces is implemented in C, while the enabling inference algorithm is implemented in SML.

In conclusion, the choice of SML has turned out to be one of the most successful design decisions in the development of CPNs and the supporting computer tools.

Acknowledgments

The authors would like to acknowledge the constructive comments of the journal editors and the anonymous reviewers.

Notes

1. A type is also called a *colour set* in CPN terminology.
2. The value carried by a token is also called a *colour* in CPN terminology.

References

1. Abdulla, P.A. and Nylen, A. Timed Petri Nets and BQOs. In *Proc. of 22nd International Conference on Application and Theory of Petri Nets*, vol. 2075 of *Lecture Notes in Computer Science*, Springer-Verlag, 2001, pp. 53–70.
2. Alur, R. and Dill, D. A theory of timed automata. *Theoretical Computer Science*, **126**(2) (1994) 183–235.
3. Appel, A.W. and MacQueen, D.B. Standard ML of New Jersey. In *Proceedings of the Third International Symposium on Programming Language Implementation and Logic Programming*, vol. 528 of *Lecture Notes in Computer Science*, Springer-Verlag, 1991, pp. 1–13.
4. Cheng, A., Christensen, S., and Mortensen, K.H. Model checking coloured petri nets exploiting strongly connected components. In *Proceedings of the International Workshop on Discrete Event Systems, WODES96*, Institution of Electrical Engineers, Computing and Control Division, Edinburgh, UK, 1996, pp. 169–177.
5. Christensen, S. and Jørgensen, J.B. Analysis of Bang and Olufsen's BeoLink audio/video system using coloured petri nets. In *Proceedings of International Conference on Application and Theory of Petri Nets*, vol. 1248 of *Lecture Notes in Computer Science*, Springer-Verlag, 1997, pp. 387–406.
6. Christensen, S., Kristensen, L.M., and Mailund, T. Condensed state spaces for timed petri nets. In *Proceedings of the 22nd International Conference on Application and Theory of Petri Nets*, vol. 2075 of *Lecture Notes in Computer Science*, Springer-Verlag, 2001, pp. 101–120.
7. Christensen S., Kristensen, L.M., and Mailund, T. A sweep-line method for state space exploration. In *Proceedings of TACAS'01*, vol. 2031 of *Lecture Notes in Computer Science*, Springer-Verlag, 2001, pp. 450–464.
8. Ciardo, G., Cherkasova, L., Kotov, V., and Rokicki, T. Modeling a scaleable high-speed interconnect with stochastic petri nets. In *Proceedings of PNPM'95*, IEEE Computer Society Press, 1995, pp. 83–93.
9. Clarke, E.M. and Wing, J.M. Formal methods: State of the art and future directions. *ACM Computing Surveys*, **28**(4) (1996), 626–643.
10. Clarke, E.M., Emerson, E.A., and Sistla, A.P. Automatic verification of finite state concurrent systems using temporal logic. *ACM Transactions on Programming Languages and Systems*, **8**(2) (1986) 244–263.
11. Clausen, H. and Jensen, P.R. Validation and performance analysis of network algorithms by coloured petri nets. In *Proceedings of PNPM'93*, IEEE Computer Society Press, 1993, pp. 280–289.
12. Comer, D.E. *Computer Networks and Internets*. Prentice-Hall, 1997.
13. The Design/CPN Homepage. http://www.daimi.au.dk/designCPN/.
14. Eker, S.M. Associative-commutative matching via bipartite graph matching. *The Computer Journal*, **38**(5) (1995) 381–399.
15. Emerson, E.A. and Sistla, A.P. Symmetry and model checking. *Formal Methods in System Design*, **9**(1/2) (1996) 105–131.
16. Floreani, D.J., Billington J., and Dadej, A. Designing and verifying a communications gateway using colored petri nets and design/CPN. In *Proc. of International Conference on Application and Theory of Petri Nets*, vol. 1091 of *Lecture Notes in Computer Science*, Springer-Verlag, 1996, pp. 153–171.
17. The Formal Methods Homepage. http://www.afm.sbu.ac.uk/.
18. Formal Systems—FDR2. http://www.formal.demon.co.uk/FDR2.html.
19. Gaeta, R. Efficient discrete-event simulation of coloured petri nets. *IEEE Transactions on Software Engineering*, **22**(9) (1996) 629–639.
20. Gallasch, G. and Kristensen, L.M. Comms/CPN: A communication infrastructure for external communication with design/cpn. In *Proceedings of the 3rd Workshop on Practical Use of Coloured Petri Nets and the CPN Tools (CPN'01)*, DAIMI PB-554, University of Aarhus, 2001, pp. 79–93.
21. Genrich, H. Predicate/transition nets. In *High-level Petri Nets*, Springer-Verlag, 1991, pp. 3–43.

22. Giacalone, A., Mishra, P., and Prasad, S. Facile: A symmetric integration of concurrent and functional programming. In *Proc. of TAPSOFT'89*, vol. 352 of *Lecture Notes in Computer Science*, Springer-Verlag, 1989, pp. 184–209.
23. Giunchiglia, F. and Traverso, P. Special issue on theorem proving. *International Journal on Software Tools for Technology Transfer (STTT)*, **3**(1) (2000) 1–92.
24. Gordon, M.J.C. and Melham, T.F. (Eds.). *Introduction to HOL*. Cambridge University Press, 1993.
25. Gordon, S., Kristensen, L.M., and Billington, J. Verification of a revised WAP wireless transaction protocol. In *Proceedings of 23rd International Conference on Application and Theory of Petri Nets*, vol. 2360 of *Lecture Notes in Computer Science*, Springer-Verlag, 2002, pp. 182–202.
26. Grahlmann, B. The PEP tool. In *Proceedings of CAV'97*, vol. 1254 of *Lecture Notes in Computer Science*, Springer-Verlag, 1997, pp. 440–443.
27. The GreatSPN Homepage. http://www.di.unito.it/~greatspn/index.html.
28. Harel, D. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, **8** (1987) 231–274.
29. Hoare, C.A.R. *Communicating sequential processes*. Prentice-Hall, 1985.
30. Holzmann, G.J. *Design and Validation of Computer Protocols*. Prentice-Hall, 1991.
31. Ilié, J.-M. and Rojas, O. On well-formed nets and optimisations in enabling tests. In *Proc. of International Conference on Application and Theory of Petri Nets*, vol. 691 of *Lecture Notes in Computer Science*, Springer-Verlag, 1993, pp. 300–318.
32. The Java Homepage. http://java.sun.com/.
33. Jensen, K. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use.* Vol. 1, *Basic Concepts*. Monographs in Theoretical Computer Science. Springer-Verlag, 1992.
34. Jensen, K. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use.* Vol. 2, *Analysis Methods*. Monographs in Theoretical Computer Science. Springer-Verlag, 1994.
35. Jensen, K. Condensed state spaces for symmetrical coloured petri nets. *Formal Methods in System Design*, **9**(1/2) (1996) 7–40.
36. Jensen, K. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use*. Vol. 3, *Practical Use*. Monographs in Theoretical Computer Science. Springer-Verlag, 1997.
37. Jensen, K. An introduction to the practical use of coloured petri nets. In *Lectures on Petri Nets II*, vol. 1492 of *Lecture Notes in Computer Science*, Springer Verlag, 1998, pp. 237–292.
38. Jensen, K. and Rozenberg, G. (Eds.). *High-level Petri Nets*. Springer-Verlag, 1991.
39. Jones, S.P., Gordon, A., and Finne, S. Concurrent Haskell. In *23rd ACM Symposium on Principles of Programming Languages*, 1996, pp. 295–308.
40. Kristensen, L.M., Christensen S., and Jensen, K. The Practitioner's Guide to Coloured Petri Nets. *International Journal on Software Tools for Technology Transfer*, **2**(2) (1998) 98–132.
41. Kristensen, L.M. and Valmari, A. Finding stubborn sets of coloured petri nets without unfolding. In *Proceedings of ICATPN'98*, vol. 1420 of *Lecture Notes in Computer Science*, Springer-Verlag, 1998, pp. 104–123.
42. Kristensen, L.M. and Mailund, T. A generalised sweep-line method for safety properties. In *Proceedings of Formal Methods Europe*, vol. 2391 of *Lecture Notes in Computer Science*, Springer-Verlag, 2002, pp. 549–567.
43. Kummer, O. Tight integration of JAVA and petri nets. In *Proc. of 6th Workshop on Algorithms and Tools for Petri Nets*, J.W. Goethe-Universität, Institut für Wirtschaftinformatik, Oct. 1999, pp. 30–35.
44. Larsen, K.G., Pettersson, P., and Yi, W. UPPAAL in a Nutshell. *International Journal on Software Tools for Technology Transfer*, **1**(1 + 2) (1997) 134–152.
45. Lincoln, P. and Christian, J.C. Adventures in associative-commutative unification. *Journal of Symbolic Computation*, **8**(1/2) (1989) 217–240.
46. Lorentsen, L. and Kristensen, L.M. Modelling and analysis of a danfoss flowmeter system using coloured petri nets. In *Proceedings of 21st International Conference on Application and Theory of Petri Nets*, vol. 1825 of *Lecture Notes in Computer Science*, Springer-Verlag, 2000, pp. 346–366.
47. Mäkelä, M. Optimising enabling tests and unfoldings of algebraic system nets. In *Proc. of International Conference on Application and Theory of Petri Nets*, vol. 2075 of *Lecture Notes in Computer Science*, Springer-Verlag, 2001, pp. 283–302.

48. Mäkelä, M. Maria: Modular reachability analyser for algebraic system nets. In *Proc. of 23rd International Conference on Application and Theory of Petri Nets*, vol. 2360 of *Lecture Notes in Computer Science*, Springer-Verlag, 2002, pp. 434–444.
49. Marsan, M.A., Balbo, G., Conte, G., Donatelli, S., and Franceschinis, G. *Modelling with Generalized Stochastic Petri Nets*. Series in Parallel Computing. Wiley, 1995.
50. Milner, R. *Communication and Concurrency*. Prentice-Hall, 1989.
51. Milner, R., Harper, R., and Tofte, M. *The Definition of Standard ML*. MIT Press, 1990.
52. Monchelet, G., Christensen, S., Demmou, H., Paludetto, M., and Porras, J. Analysing a mechatronic system with coloured petri nets. *Int. Journal on Software Tools for Technology Transfer*, **2**(2) (1998) 160–167.
53. Mortensen, K.H. Automatic code generation method based on coloured petri net models applied on an access control system. In *Proc. of 21st International Conference on Application and Theory of Petri Nets*, vol. 1825 of *Lecture Notes in Computer Science*, Springer-Verlag, 2000, pp. 367–386.
54. The Moscow ML Homepage. http://www.dina.dk/~sestoft/mosml.html.
55. Nöcker, E., Smethers, S., van Eekelen, M., and Plasmeijer, R. Concurrent clean. In *Proc. of Parallel Architectures and Languages Europe (PARLE '91)*, vol. 505 of *Lecture Notes in Computer Science*, Springer-Verlag, 1991, pp. 202–219.
56. The Standard ML of New Jersey Homepage. http://cm.bell-labs.com/cm/cs/what/smlnj/index.html.
57. The Haskell Homepage. http://www.haskell.org.
58. Paulson, L.C. *ML for the Working Programmer*. Cambridge University Press, 2nd edn., 1996.
59. Peled, D. All from one, one for all: on model checking using representatives. In *Proceedings of CAV'93*, vol. 697 of *Lecture Notes in Computer Science*, Springer-Verlag, 1993, pp. 409–423.
60. Petri, C.A. *Kommunikation mit Automaten.* Bonn: Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2, 1962.
61. The PVS Specification and Verification System Homepage. http://pvs.csl.sri.com/.
62. Rasmussen, J.L. and Singh, M. Designing a security system by means of coloured petri nets. In *Proceedings of 17th International Conference on Application and Theory of Petri Nets*, vol. 1091 of *Lecture Notes in Computer Science*, Springer-Verlag, 1996, pp. 400–419.
63. Reinke, C. Haskell-coloured petri nets. In *Proc. of 11th International Workshop Implementation of Functional Langugages (IFL'99)*, vol. 1868 of *Lecture Notes in Computer Science*, Springer-Verlag, 1999, pp. 165–180.
64. Reisig, W. *Petri Nets*, vol. 4 of *EACTS Monographs on Theoretical Computer Science*. Springer-Verlag, 1985.
65. Reppy, J. *Concurrent Programming in ML*. Cambridge University Press, 1999.
66. Rozenberg, G. and Reisig, W. (Eds.). *Lectures on Petri Nets I: Basic Models*, vol. 1491 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.
67. Sanders, M.J. Efficient computation of enabled transition bindings in high-level petri nets. In *Proc. of IEEE International Conference on Systems, Man and Cybernetics*, Oct. 2000, pp. 3153–3158.
68. Schmidt, K. LoLA: A low level analyser. In *Proc. of 21st International Conference on Application and Theory of Petri Nets*, vol. 1825 of *Lecture Notes in Computer Science*, Springer-Verlag, 2000, pp. 465–474.
69. Steggles, L.J. Rewriting logic and elan: Prototyping tools for petri nets with time. In *Proc. of 22nd International Conference on Application and Theory of Petri Nets*, vol. 2075 of *Lecture Notes in Computer Science*, Springer-Verlag, 2001, pp. 363–381.
70. Stoustrup, B. *The C++ Programming Language*, 3rd edition. Addison-Wesley, 1997.
71. The Edinburgh Concurency Workbench Homepage. http://www.dcs.ed.ac.uk/home/cwb/index.html.
72. The CPN Tools Homepage. http://www.daimi.au.dk/CPNtools/.
73. Ullman, J.D. *Elements of ML Programming*. Prentice-Hall, 1998.
74. Valmari, A. A stubborn attack on state explosion. In *Proceedings of Computer-Aided Verification '90*, vol. 531 of *Lecture Notes in Computer Scienc*, Springer-Verlag, 1990, pp. 156–165.
75. Valmari, A. Stubborn sets of coloured petri nets. In *Proceedings of ICATPN'91*, 1991, pp. 102–121.
76. Valmari, A. The state explosion problem. In *Lectures on Petri Nets I: Basic Models*, vol. 1491 of *Lecture Notes in Computer Science*, Springer-Verlag, 1998, pp. 429–528.
77. Van der Aalst, W.M.P. Interval timed coloured petri nets and their analysis. In *Proceedings of ICATPN'93*, vol. 691 of *Lecture Notes in Computer Science*, Springer Verlag, 1993, pp. 453–472.

78. The VisualState Homepage. http://www.iar.com/.
79. Xu, J. and Kuusela, J. Analyzing the execution architecture of mobile phone software with coloured petri nets. *Int. Journal on Software Tools for Technology Transfer*, **2**(2) (1998) 133–143.
80. Yakovlev, A., Gomes, L., and Lavagno, L. (Eds.). *Hardware Design and Petri Nets*. Kluwer Academic Publishers, 2000.
81. Zhang, L., Kristensen, L.M., Janczura, C., Gallasch, G., and Billington, J. A coloured petri net based tool for course of action development and analysis. In *Proc. of Workshop on Formal Methods Applied to Defence Systems*, vol. 12 of *Conferences in Research and Practice in Information Technology*, Australian Computer Society, 2002, pp. 125–134.