

# Parameterised Coloured Petri Nets

Thomas Mailund

Department of Computer Science  
University of Aarhus  
DK-8000 Aarhus C  
Denmark  
mailund@daimi.au.dk

**Abstract.** In this paper we examine Coloured Petri Nets extended with parameters. We characterise three kinds of parameterisation and formally define *Parameterised Coloured Petri Nets*. We then discuss how parameterised Coloured Petri Nets can be used to create libraries of Coloured Petri Nets modules in the same way as libraries for programming languages. Finally we discuss how to implement a simple simulator for such modules.

## 1 Introduction

In Coloured Petri Nets (CPN), as in traditional programming languages, it is infeasible to work on industrial size problems as one single unit. To tackle problems of a certain size, it is necessary to work on smaller units, which can later be composed into the full system.

A modular approach to modelling makes larger systems easier to handle. There is less to validate which reduces the debugging period. Even verification usually takes benefit of modular models, though the situation here is a bit more complex, since the environment a module is put into (i.e. the neighbourhood of the substitution transition) tends to influence the dynamic properties of the module, that one wishes to verify.

But perhaps more importantly, a modular framework opens up for reuse of commonly used constructions. From a modelling point of view, reuse saves time and effort. From a validation point of view, reuse increases faith in correctness, since a module tested in depth in one environment, is likely to work as expected in a similar environment. Needless to say, reusing a verified module is vastly better than creating a new, untested module.

It is often the case that large models contain a number of similar constructions, where only a few details differ. Modularisation alone does not allow for much reuse in such circumstances, however, the constructions can in most cases be changed in a way, such that their differences depend on a set of parameters, which can be assigned when needed. This observations leads to the concept of Parameterised Coloured Petri Nets (PCPN).

In [2] three kinds of parameterisation were identified: type, expression,<sup>1</sup> and net parameterisation, and a short discussion of a possible implementation was given. In this paper we formally define Parameterised CPN, and the three kinds of parameter assignments. We also repeat the discussion on implementation issues, but this time based on an actual implementation of a simulator.

The paper is organised as follows: Sect. 2 introduces the concepts through small toy examples, showing the use of type, expression, and net parameters. In Sect. 3 we formally define Parameterised CPN and the three kinds of parameter assignments. Readers only interested in the practical use of Parameterised CPN can safely skip this section. In Sect. 4 we discuss how Parameterised CPN naturally leads to a module system. In Sect. 5 we show how to implement a simulator to take benefit of this module system. Readers only interested in the use of Parameterised CPN and not tool development can safely skip this section. Finally we consider future work in Sect. 6 and conclude in Sect. 7.

Familiarity with (non-hierarchical) Coloured Petri Nets [5] is assumed throughout this paper, and familiarity with SML, especially the module system [9, 10], is assumed for Sect. 5.

---

<sup>1</sup> In [2] expression parameters were called value parameters, however, we find that the term expression parameters are more true to the definitions given in this paper.

## 2 A small example

In this section, we will introduce the general ideas of parameterised CPN, via a small toy example. Say we want to count the number of times a certain transition fires. We can do this by a construction like the one in Fig. 1. If we substitute the transition whose firings we wish to count with the net in the figure, we get the number of firings by the value of the token on place C.

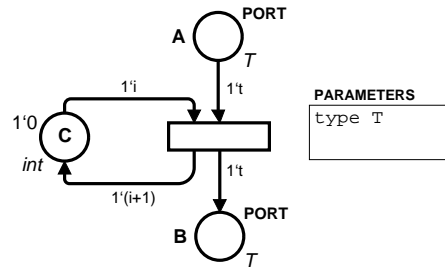


Fig. 1.  $\mathcal{C}$  - Counter net.

### 2.1 Type Parameters

In Fig. 1 we do not know, nor do we care about the type  $T$  of the places  $A$  and  $B$ . We only care about the token on place  $C$  which counts the firings of the transition. Any transition between places  $A$  and  $B$  (with the same colour set) could be substituted with the net in Fig. 1, regardless of the type of  $A$  and  $B$ . We would like to think of the type  $T$  as a parameter to be specified at the location where we use the net.

Consider Fig. 2 where we have places  $A$  and  $B$  with colour set  $\mathbf{int}$ . To count the firings of transition  $Y$  we could substitute  $Y$  with the net  $\mathcal{C}$  where we have assigned type  $T$  to  $\mathbf{int}$ .

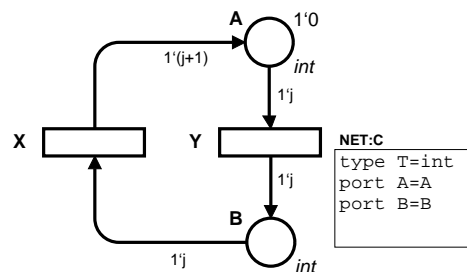


Fig. 2. Using the counter net.

The resulting net is shown in Fig. 3. Here transition  $Y$  has been removed and the counter net  $\mathcal{C}$  has been inserted in its place. The two nets are then connected through the border places of the substituted transition in the first net, and a subset of the nodes of the second net. The connection places of the first net are called the *sockets* of the substituted transition. The connection places of the second net are called the *ports* of the net.

A substitution can be thought of as an operation on nets. Given two nets it creates a new net by replacing a transition in the first net with the second net as described. For a formal description of substitution we refer to Sect. 3.

In practice we do not create the new net in this way, but simply describe the substitution with a *NET* region as seen in Fig. 2. Here we name the net to substitute with, together with assignments of parameters and port/socket assignments, determining how the two nets should be combined.

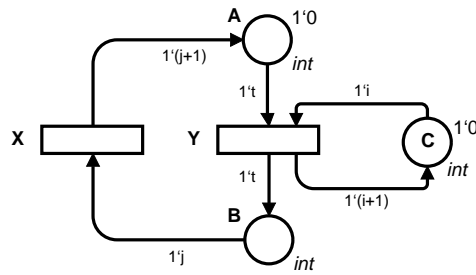


Fig. 3. Fig. 2 and the counter net after substitution.

If more than one transition is substituted with the same net, we create copies of that net, and let a copy substituted each transition. This makes it possible to use the same net in several locations, instantiated with different types. Consider Fig. 4. Here we have places A and B of type **int** separated by transition X, and places A' and B' of type **bool** separated by transition Z. To put a counter on both X and Z we could substitute it with the net in Fig. 1, assigning type  $T$  to be **int** for X and **bool** for Z.

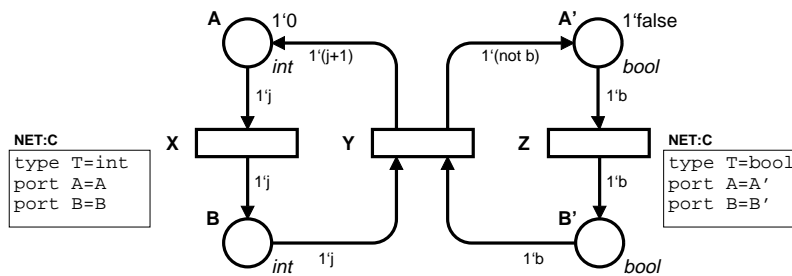


Fig. 4. Using the counter net with different types.

This would then lead to the net shown in Fig. 5.

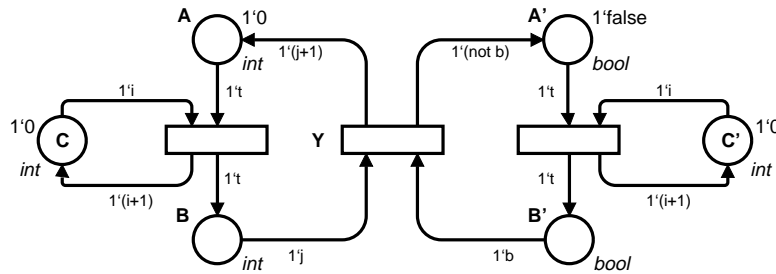


Fig. 5. Using the counter net with different types, after substitution.

## 2.2 Expression Parameters

We might not want to count *all* firings, but only some, depending on the binding, and we might not want to increment the counter with the same value in all cases. We cannot specify this directly in our counter net, since the choice of which firings to count will depend on the specific uses of the net. Instead we can add a predicate to the net to decide when to increment the counter, and a value specifying the step with which to

increment, and let both be parameters of the net. These parameters are called *expression* parameters of the net.

In Fig. 6 we see the counter net from Fig. 1 extended with a predicate  $p$  that, given a value of type  $T$ , returns a boolean, which then determines whether to increment the counter. The predicate is an expression parameter to the net, as indicated in the box to the right of the net. Furthermore we have the expression  $s$  of type **int**. This is used to specify the step with which to increment the counter.

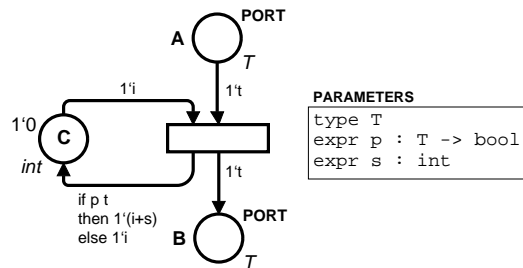


Fig. 6.  $C'$  - Counter with value parameters.

These expression parameters are mentioned in the *PARAMETERS* box, together with the type parameter  $T$ . We have typed the parameters in the declaration, and we only allow expressions with the right type to be assigned to the expression variables.

Fig. 7 shows the new net in use. The left part counts all firings of  $X$  after  $i$  has exceeded 5, and does this with an incrementation step of 2. The right part only counts the firings of  $Z$  when  $b$  is *true*. Since  $b$  is negated in each cycle, only every second firing of  $Z$  is counted. The incrementation step is 1.

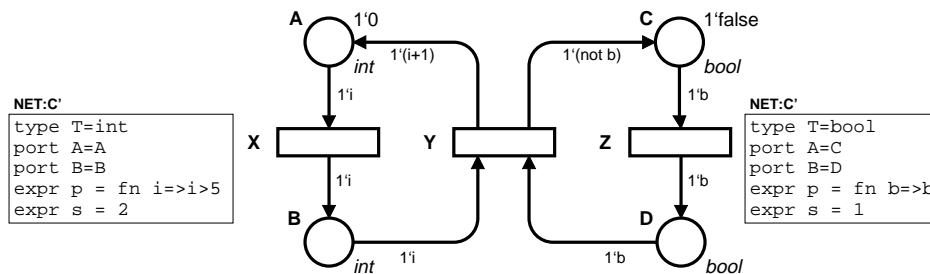


Fig. 7. Using the counter with expression parameters.

### 2.3 Net Parameters

Not only types and values are interesting as parameters. Sometimes, we would like to be able to leave parts of nets undefined, until the net is used. What we need is a way of parameterising parts of a net, and a way of substituting such parts with other nets. For the latter we need to decide exactly how the rim of a parameter part in one net should be glued together with the second net. In this paper we will only allow transitions to be parameters, and glue nets together by fusing the border places of substituted transitions with a subset of places from the second net. For two other ways of substitution (place and arc substitution) we refer to [8].

In a way we have already seen how net parameters should work. In Fig. 2 we said that we substituted the transition  $Y$  with the counter net  $C$ . This was done by simply referring to  $C$  in the *NET* region of  $Y$ . Here we assumed that  $C$  was globally known. Instead we could demand that the net referred to was a parameter.

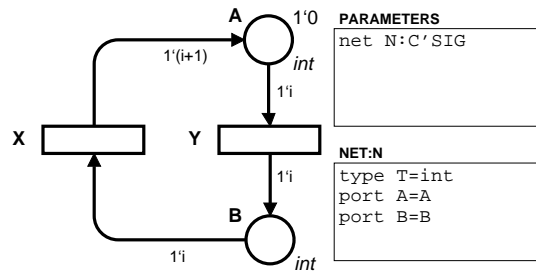


Fig. 8. Net with net parameter.

Consider Fig. 8. This is almost the same net as in Fig. 2. The difference is the *PARAMETERS* box and the name in the *NET* region. In Fig. 8 we substitute *Y* with a net  $\mathcal{N}$  which is given as a parameter to the net.

The net parameter is restricted to a specific *signature*. The signatures do for net parameters what type declarations does for expression variables. It specifies which nets are legal parameters, by fixing how any parameter must look like to ensure a legal substitution. Signatures can be automatically generated from the nets. We will assume this has been done for the nets in this paper, and we will refer to the signature of a net  $\mathcal{N}$  as  $\mathcal{N}'SIG$ .

For a net, the signature is defined by the set of ports (a subset of the places) and the parameters the net takes, i.e., the type, expression, and net parameters. In the case of the counter net  $\mathcal{C}$  (Fig. 1), the signature specifies one type parameter  $T$  and two ports, named *A* and *B*, both with colour set  $T$ .  $\mathcal{C}'SIG$  refers to this exact signature. The net in Fig. 8 accepts any net with type parameter  $T$  and ports *A* and *B* with colour set  $T$ .

Another net with signature  $\mathcal{C}'SIG$  is show in Fig. 9. This is a net that logs all bindings of a transition on the place *L*. The net in Fig. 8 can be instantiated with any net with signature  $\mathcal{C}'SIG$ , and thus with both the counter net and the log net as net  $\mathcal{N}$ .

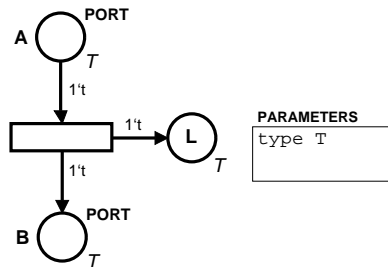


Fig. 9.  $\mathcal{L}$  - Log net.

Of course we did not really need net parameters for switching between  $\mathcal{C}$  and  $\mathcal{L}$ . We could simply change the name in the *NET* region. It gets more interesting when net parameters are nested, that is, when the assignment of net parameters depend on the net they are used in.

Consider Fig. 10. Denote this net  $\mathcal{LR}$ .  $\mathcal{LR}$  chooses randomly whether the left or the right transition fires. The net takes a type parameter  $T$  and a net parameter  $\mathcal{N}$ , with signature  $\mathcal{C}'SIG$ . We can assign both  $\mathcal{C}$  and  $\mathcal{L}$  to  $\mathcal{N}$ . If we assign  $\mathcal{C}$ ,  $\mathcal{LR}$  will count how many times each transition fires, if we assign  $\mathcal{L}$ ,  $\mathcal{LR}$  will log the binding elements.

Now consider Fig. 11. This net takes a net parameter  $\mathcal{M}$  with signature  $\mathcal{LR}'SIG$ . It substitutes the transition *X* with  $\mathcal{M}$  where the net  $\mathcal{N}$  is assigned  $\mathcal{L}$ , and it substitutes transition *Z* with  $\mathcal{M}$  where  $\mathcal{N}$  is assigned  $\mathcal{C}$ . If  $\mathcal{M}$  was assigned  $\mathcal{LR}$  we would log left and right on *X* and count left and right on *Z*.

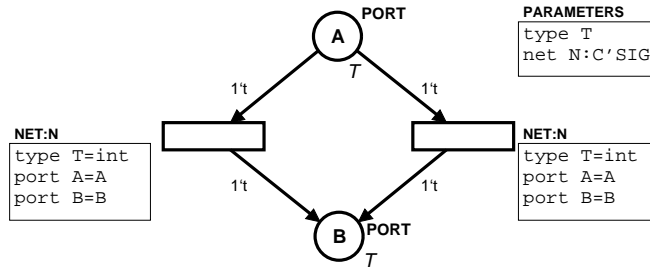


Fig. 10.  $\mathcal{LR}$  - Random left or right.

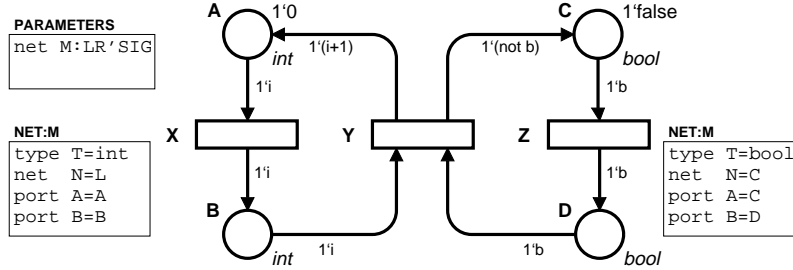


Fig. 11. Nested net parameters.

### 3 Formal Definitions

In this section we will define formally what we mean by Parameterised CPN. We define three different kinds of parameters: type, expression, and net parameters. We then combine these into Parameterised CPN. For net parameters we will only consider transition substitution. Place and arc substitution have been examined in [8]. The important ideas however appear in the definition of transition substitution, and examining all three kinds of substitution will only complicate the definitions. For a more detailed description of the definitions we refer to [8].

#### 3.1 Type Parameters

As we saw in the examples in Sect. 2 we would like to leave some types undefined until a specific type is actually needed. We would like to refer to *type variables* instead of actual types, and assign types to variables when necessary. In the examples we only used type variables for colour sets, but in general we would want to use the type variables when constructing types, e.g., lists or products of a variable type.

We will assume the existence of some type language, e.g., a set of basic types and ways of constructing products, sums etc. of types. We let  $\mathbb{T}$  denote this language, i.e.,  $\mathbb{T}$  denotes the set of expressions created over the type constructions. Furthermore we assume that  $\mathbb{T}$  includes variables from a set of *type variables*  $\mathbf{TV}$  which is disjoint from any other set in the definition.

We assume that no type in  $\mathbb{T}$  is empty, i.e.  $\forall T \in \mathbb{T}. T \neq \emptyset$ .  $\mathbb{T}$  must contain the type  $\mathbf{bool} = \{\text{true}, \text{false}\}$ . This is used in the definition of *guards* (see Def. 6). Finally, we assume that  $\mathbb{T}$  contains a multiset construction  $(\cdot)_{MS}$ . This is used in the definition of *arc functions* and *initialisation functions* (see Def. 6).

**Definition 1 (Type assignment).** A type assignment  $\Gamma : \mathbf{TV} \rightarrow \mathbb{T}$  is a partial mapping from type variables  $\mathbf{TV}$  to type expressions in  $\mathbb{T}$ .  $\square$

For partial mapping  $\Gamma$  we let  $\text{dom}(\Gamma)$  denote the elements on which the mapping is defined.

**Definition 2.** For type expression  $T \in \mathbb{T}$  and type assignment  $\Gamma$ , denote by  $T[\Gamma]$  the expression obtained by replacing all variables in the domain of  $\Gamma$ ,  $v \in \text{dom}(\Gamma)$  appearing in  $T$  with  $\Gamma(v)$ .  $\square$

We will only consider legal type assignments, i.e., for expressions  $T \in \mathbb{T}$  we will only consider  $\Gamma : \mathbf{TV} \rightarrow \mathbb{T}$  such that  $T[\Gamma] \in \mathbb{T}$ .

### 3.2 Expression Parameters

As for type parameters we would like to use *expression variables* in inscriptions, and assign actual expressions to these variables when necessary.

We will assume the existence of an inscription language  $\mathbb{E}$ , with a set of type rules such that all valid expressions in  $\mathbb{E}$  evaluate to values with a type in  $\mathbb{T}$ . We will assume that we can talk about the variables of an expression  $E \in \mathbb{E}$  and that we can talk of the type of a variable. We assume two different kinds (i.e. two disjoint sets of variables) of variables, “ordinary”, or *binding*, variables and *expression variables*  $\mathbf{EV}$ . The former is used in bindings, the later for expression parameters. For variable  $v$  let  $Type(v)$  denote the type of  $v$ . We will demand that  $Type(v) \in \mathbb{T}$  for all variables. By  $Type(E)$  we denote the type of expression  $E$ , and we demand that  $Type(E) \in \mathbb{T}$ . By  $Var(E)$  we denote the binding variables in  $E$ .<sup>2</sup>

**Definition 3 (Expression assignment).** *An expression assignment  $\Delta$  is a partial mapping from expression variables  $\mathbf{EV}$  to expressions in  $\mathbb{E}$ ,  $\Delta : \mathbf{EV} \rightarrow \mathbb{E}$ .*  $\square$

**Definition 4.** *For expression  $E \in \mathbb{E}$  we denote by  $E[\Delta]$  the expression obtained by replacing all variables  $v \in \text{dom}(\Delta)$  appearing in  $E$  with  $\Delta(v)$ .*  $\square$

We will only consider legal value assignments, i.e., for expressions  $E \in \mathbb{E}$  we will only consider  $\Delta : \mathbf{EV} \rightarrow \mathbb{E}$  such that  $E[\Delta] \in \mathbb{E}$ .

### 3.3 Net Parameters

Intuitively net parameters work in much the same way as type and expression parameters. We associate a variable to parts of a net, and allow for this part to be substituted with another net when necessary. Net parameters are a little more complex however, since assigning actual nets to variables requires that we define a way to glue two (or more) nets together. In this paper we only consider *transition substitution* to glue nets together.

In the following we assume we are considering some universe of Parameterised CPN closed under the operations described in this section. Let  $\mathbf{PCPN}$  denote the set of all PCPN in the universe, let  $\mathbf{P}$  denote the union of all place sets in  $\mathbf{PCPN}$ . We define PCPNs shortly (Def. 6), but for now, just think of  $\mathbf{PCPN}$  as a set, where each element has an associated set, called the places of the net. We furthermore assume we have a set of *net variables*  $\mathbf{NV}$  and a set of *port names*  $\mathbf{PN}$ .

**Definition 5 (Net assignment).** *A net assignment is a mapping  $\Theta : \mathbf{NV} \rightarrow \text{Pow}(\mathbf{PN} \times \mathbf{P}) \times \mathbf{PCPN}$ , where  $\text{Pow}(\mathbf{PN} \times \mathbf{P})$  denotes the powerset of  $\mathbf{PN} \times \mathbf{P}$ .  $\Theta$  should satisfy for all  $\text{net} \in \text{dom}(\Theta)$  with  $\Theta(\text{net}) = (R, \mathcal{N})$ , if we let  $P$  denote the set of places of  $\mathcal{N}$  the relation  $R \subseteq \mathbf{PN} \times P$  relates port names to the places in  $\mathcal{N}$ .*  $\square$

A net assignment maps net variables to nets and give a relation which names a subset of the ports of the nets. The naming of ports is used later for gluing nets together.

### 3.4 PCPN

With the three kinds of assignments defined, we are ready to define PCPN, and describe how assignments affects nets.

Let  $\mathbf{TASS}$  denote the set of type assignments, i.e.,  $\mathbf{TASS} = (\mathbf{TV} \rightarrow \mathbb{T})$ , let  $\mathbf{EASS}$  denote the set of expression assignments, i.e.,  $\mathbf{EASS} = (\mathbf{EV} \rightarrow \mathbb{E})$ , and let  $\mathbf{NASS}$  denote the set of net assignments, i.e.,  $\mathbf{NASS} = (\mathbf{NV} \rightarrow \text{Pow}(\mathbf{PN} \times \mathbf{P}) \times \mathbf{PCPN})$ .

**Definition 6 (PCPN).** *A Parameterised CPN (PCPN) is a tuple  $\mathcal{N} = (P, T, A, N, C, G, E, I, \text{Ports}, TS)$  such that*

1.  $P$  is a finite set of places

<sup>2</sup>  $BVar(E)$  would perhaps be a better notation, however  $Var(E)$  is used in [5] so we stick to this convention.

2.  $T$  is a finite set of transitions
3.  $A$  is a finite set of arcs, such that

$$P \cap T = P \cap A = T \cap A = \emptyset$$

4.  $N$  is a node function.  $N : A \rightarrow (P \times T \cup T \times P)$ .
5.  $C$  is a colour function  $C : P \rightarrow \mathbb{T}$ .
6.  $G$  is a guard function  $G : T \rightarrow \mathbb{E}$  such that

$$\forall t \in T. \text{Type}(G(t)) = \mathbf{bool}$$

7.  $E$  is an arc expression function,  $E : A \rightarrow \mathbb{E}$  such that

$$\forall a \in A. \text{Type}(E(a)) = C(p(a))_{MS}$$

where  $p(a)$  is the place of  $N(a)$ , and  $C(p(a))_{MS}$  denotes the set of multi-sets over  $C(p(a))$ .

8.  $I$  is an initialisation function,  $I : P \rightarrow \mathbb{E}$  such that

$$\forall p \in P. (\text{Type}(I(p)) = C(p)_{MS} \wedge \text{Var}(I(p)) = \emptyset)$$

9.  $\text{Ports} \subseteq P$  is a subset of the places.

10.  $TS$  is a substitution mapping  $TS : T \rightarrow \mathbf{NV} \times \text{Pow}(P \times \mathbf{PN}) \times \mathbf{TASS} \times \mathbf{EASS} \times \mathbf{NASS}$  is a partial mapping from the transitions of  $\mathcal{N}$  to pairs of net variables and relations of places and port names. For transition  $t \in T$  let  $S(t) = \bullet t \cup t \bullet$ .  $S(t)$  is called the sockets of  $t$ .  $TS$  should satisfy, that for all  $t \in \text{dom}(TS)$  with  $TS(t) = (\text{net}, R, \Gamma, \Delta, \Theta)$ ,  $R \subseteq S(t) \times \mathbf{PN}$ , that is the relation is a relation between the sockets of  $t$  and port names.  $\text{dom}(TS)$  is called the substitution transitions of  $\mathcal{N}$ .

□

The definition of  $TS$  looks a little ghastly, but will hopefully be more clear after we have discussed the three kinds of assignments related to PCPN.

Intuitively,  $TS$  relates to each substitution transition a net variable and a naming of sockets, and a triple of a type, expression, and net assignment. Given a net assignment, which relates the net variable to an actual net, the assignments are applied to this net and the relation is used to “glue” the two nets together.

In the examples we used the *NET* regions to define  $TS$  (see Fig. 12). The name (after the colon) is the net variable, and refers to the net parameter in the PARAMETERS box. The relation and the three assignments are defined in the box. We use *type* to define type assignments, *expr* to define expression assignments and *net* to define net assignments. The relation is defined by the *port* assignments. Whenever we refer to a net variable which is not a parameter, we assume that the variable refers uniquely to a specific net. We can then think of the net as already being assigned.

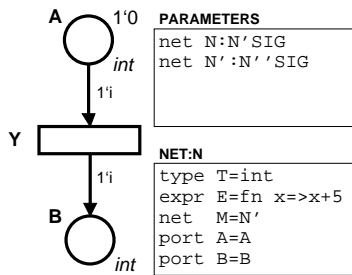


Fig. 12. Example of *NET* region.

If we ignore *Ports* and  $TS$  for a moment, we can see that PCPN looks a lot like non hierarchical CPN as defined in [5] (Def. 2.5). We do not have the set of colour sets  $\Sigma$ . Instead we have the type languages  $\mathbb{T}$ . Furthermore, CPN as defined in [5] does not allow for type nor expression variables.

We will only consider the behaviour of PCPN with no type and expression variables, i.e., for which all variables have been assigned (see Def. 16 and Def. 12). This behaviour is the same as for CPN.

For transition  $t$  let  $Var(t)$  denote the (binding) variables of  $t$  i.e., the union of the binding variables in the guard of  $t$  and the arc expressions on the arcs with source or destination in  $t$ .  $Var(t)$  does not include any expression variables.

$$\forall t \in T. Var(t) = \{ v \mid v \in Var(G(t)) \vee \exists a \in A(t). v \in Var(E(a)) \}$$

where  $A(t)$  denotes the set of arcs with source or destination in  $t$ .

For expression  $E \in \mathbb{E}$  and mapping  $b : Var(E) \rightarrow \mathbb{E}$  denote by  $E\langle b \rangle$  the expression obtained by replacing all variables in the domain of  $b$ , appearing in  $E$  with  $b(v)$ . This closely matches the definition of expression variable substitution, however the different set of brackets  $[-]$  vs.  $\langle - \rangle$  distinguish the two.

**Definition 7 (Binding, [5] Def. 2.6).** A binding of a transition  $t$  is a function  $b$  defined on  $Var(t)$ , such that

1.  $\forall v \in Var(t). b(v) \in Type(v)$
2.  $G(t)\langle b \rangle = true$

By  $B(t)$  we denote the set of all bindings for  $t$ . □

**Definition 8 ([5] Def. 2.7).** A token element is a pair  $(p, c)$  where  $p \in P$  and  $c \in C(p)$ , while a binding element is a pair  $(t, b)$  where  $t \in T$  and  $b \in B(t)$ . The set of all token elements is denoted  $TE$  while the set of all binding elements is denoted by  $BE$ .

A marking is a multi-set over  $TE$  while a step is a non-empty and finite multiset over  $BE$ . □

Let  $E(x, y)$  denote the (multi-set) sum of expressions on arcs between  $x$  and  $y$ . This will be well defined since these expressions will range over the same multi-set. Either  $x$  or  $y$  must be a place, and the expressions will range over the associated colour set.

$$\forall (x, y) \in (P \times T \cup T \times P). E(x, y) = \sum_{a \in A(x, y)} E(a)$$

where  $A(x, y)$  is the set of arcs from  $x$  to  $y$ .

**Definition 9 (Enable, [5] Def. 2.8).** A step  $Y$  is enabled in a marking  $M$  if

$$\forall p \in P. \sum_{(t, b) \in Y} E(p, t)\langle b \rangle \leq M(p)$$

**Definition 10 (Occur, [5] Def. 2.9).** When a step  $Y$  is enabled in a marking  $M$  it may occur, changing the marking to another marking  $M'$ , defined by

$$\forall p \in P. M'(p) = \left( M(p) - \sum_{(t, b) \in Y} E(p, t)\langle b \rangle \right) + \sum_{(t, b) \in Y} E(t, p)\langle b \rangle$$

□

### 3.5 Assignments

We can now define how to assign values to parameters.

**Definition 11.** For PCPN  $\mathcal{N} = (P, T, A, N, C, G, E, I, Ports, TS)$  and type assignment  $\Gamma$ , let  $\mathcal{N}[\Gamma]_T = (P, T, A, N, C', G, E, I, Ports, TS')$  denote the net obtained by replacing in  $\mathcal{N}$  all appearances of type variables  $v \in \text{dom}(\Gamma)$  with  $\Gamma(v)$ . Then

$$\forall p \in P . C'(p) = C(p)[\Gamma]$$

and  $TS'$  defined for all  $t \in \text{dom}(TS)$  by  $TS'(t) = (net, R, \Gamma^*, \Delta, \Theta)$  when  $TS(t) = (net, R, \Gamma', \Delta, \Theta)$  where  $\Gamma^*(v) = (\Gamma'(v))[\Gamma]$  whenever  $v \in \text{dom}(\Gamma')$   $\square$

Substituting types in a net changes the colour function and propagates the substitution to subnets through the transition substitution function. Type substitution is likely to change the type of variables. We do not consider this more formally since it depends heavily on the way  $Type(v)$  is given.

**Definition 12.** For PCPN  $\mathcal{N} = (P, T, A, N, C, G, E, I, Ports, TS)$  and expression assignment  $\Delta$ , we let  $\mathcal{N}[\Delta]_E = (P, T, A, N, C', G, E, I, Ports, TS')$  denote the net obtained by replacing in  $\mathcal{N}$  all appearances of expression variables  $v \in \text{dom}(\Delta)$  with  $\Delta(v)$ , where

1.  $G'(t) = G(t)[\Delta]$  for all  $t \in T$
2.  $E'(a) = E(a)[\Delta]$  for all  $a \in A$
3.  $I'(p) = I(p)[\Delta]$  for all  $p \in P$
4.  $TS'$  defined for all  $t \in \text{dom}(TS)$  by  $TS'(t) = (net, R, \Gamma, \Delta^*, \Theta)$  when  $TS(t) = (net, R, \Gamma, \Delta', \Theta)$  where  $\Delta^*(v) = (\Delta'(v))[\Delta]$  whenever  $v \in \text{dom}(\Delta')$

$\square$

Expression substitution changes the different inscription functions. Again, the substitution is propagated to subnets via the  $TS$  function.

We might need to put some restrictions on the expressions we allow a given variable to be substituted for. In the examples in Sect. 2 we explicitly typed the expression parameters used. This is not explicitly required from the definition, but is used to ensure that all expression assignments used are *legal*. Other restrictions might be needed for other inscription languages or for other purposes. We will not consider this further however.

Before we define how to apply a net assignment (Def. 17), we need to define how we substitute transitions with nets (Def. 16), but first we need to define how to fuse places:

**Definition 13 (Place fusion).** A place fusion set  $F$  of a PCPN  $\mathcal{N} = (P, T, A, N, C, G, E, I, Ports, TS)$  is an equivalence class partition of  $P$  such that

$$\forall p' \in [p]_F . C(p') = C(p) \wedge I(p') = I(p)$$

where  $[p]_F$  denotes the class of  $F$  containing  $p$ .  $\square$

**Definition 14.** For PCPN  $\mathcal{N} = (P, T, A, N, C, G, E, I, Ports, TS)$ , and a place fusion set  $F$  over  $P$ , we can fuse the places and get a net  $[\mathcal{N}]_F = (P', T, A, N', C', G, E, I', Ports', TS)$  given by

1.  $P' = \{ [p]_F \mid p \in P \}$
2.  $N' : A \rightarrow P' \times T \cup T \times P'$  defined by

$$a \mapsto \begin{cases} ([p]_F, t) & \text{if } N(a) = (p, t) \in P \times T \\ (t, [p]_F) & \text{if } N(a) = (t, p) \in T \times P \end{cases}$$

3.  $C' : P' \rightarrow \mathbb{T}$  defined by  $[p]_F \mapsto C(p)$ .
4.  $I' : P' \rightarrow \mathbb{E}$  defined by  $[p]_F \mapsto I(p)$ .
5.  $Ports' = \{ [p]_F \mid p \in Ports \}$

$\square$

A place fusion creates a place for each fusion class and connects a (class-)place and a transition if there is an original place in the class, connected to the transition.

Notice that any class containing a port itself becomes a port.

**Definition 15 (Port/Socket relation).** Let  $\mathcal{N} = (P, T, A, N, C, G, E, I, Ports, TS)$  be a PCPN. For PCPN  $\mathcal{N}$  and  $\mathcal{N}' = (P', T', A', N', C', G', E', I', Ports', TS')$  and transition  $t \in T$  a relation  $R \subseteq Ports' \times S(t)$  for which

$$R(p, s) \Rightarrow C'(p) = C(s) \wedge I'(p) = I(s)$$

is called a port/socket relation (between  $\mathcal{N}'$  and  $t$ ).  $\square$

A port/socket relation is a relation between the ports of a net and the sockets of a transition. The restrictions to the colour set and initial marking ensures that we can fuse ports and sockets.

In the following we use  $\uplus_{i=1, \dots, n} A_i$  to mean the disjoint union of sets  $\{A_i\}_{i=1, \dots, n}$ . We assume we have new symbols  $\mathbf{1}, \mathbf{2}, \dots$ , one for each  $n \in \mathbb{N}$ , and define  $\uplus_{i=1, \dots, n} A_i$  to mean the set  $\bigcup_{i=1, \dots, n} \{(\mathbf{i}, a) \mid a \in A_i\}$ . We define for each  $\mathbf{i}$  a function  $\mathbf{in}_i : A_i \rightarrow \uplus_{i=1, \dots, n} A_i$  by  $\mathbf{in}_i x = (\mathbf{i}, x)$  to map elements in  $A_i$  to the copy in the disjoint union.

Using disjoint union ensures, among other things, that we can have more than one instance of the same nets.

**Definition 16 (Transition substitution).** Let  $\mathcal{N} = (P_1, T_1, A_1, N_1, C_1, G_1, E_1, I_1, Ports_1, TS_1)$  be a PCPN and let  $\Theta$  be a partial mapping  $\Theta : dom(TS_1) \rightarrow (Pow(\mathbf{P} \times \mathbf{P})) \times \mathbf{PCPN}$  such that for all  $t_i \in dom(\Theta)$  with  $\Theta(t_i) = (R_i, \mathcal{N}_i)$ ,  $R_i$  is a port/socket relation between  $\mathcal{N}_i = (P_i, T_i, A_i, N_i, C_i, G_i, E_i, I_i, Ports_i, TS_i)$  and  $t_i$ .

We can substitute with  $\Theta$  and get the net  $\mathcal{N}[\Theta]_{TS}$  constructed in the following way: Let  $\mathcal{N}^* = (P^*, T^*, A^*, N^*, C^*, G^*, E^*, I^*, Ports^*, TS^*)$  be defined by

1.  $P^* = \uplus_{i=1, \dots, n} P_i$
2.  $T^* = (T_1 - dom(\Theta)) \uplus \uplus_{i=2, \dots, n} T_i$
3.  $A^* = (A_1 - A(dom(\Theta))) \uplus \uplus_{i=2, \dots, n} A_i$  where  $A(dom(\Theta))$  denotes the set of arcs connected to any transition in  $dom(\Theta)$
4.  $N^* : A^* \rightarrow (P^* \times T^* \cup T^* \times P^*)$  defined by  $\mathbf{in}_i a \mapsto ((\mathbf{in}_i \times \mathbf{in}_i) \circ N_i)(a)$
5.  $C^* : P^* \rightarrow \mathbb{T}$  defined by  $\mathbf{in}_i p \mapsto \mathbf{in}_i C_i(p)$
6.  $G^* : T^* \rightarrow \mathbb{E}$  defined by  $\mathbf{in}_i t \mapsto G_i(t)$
7.  $E^* : A^* \rightarrow \mathbb{E}$  defined by  $\mathbf{in}_i a \mapsto E_i(a)$
8.  $I^* : P^* \rightarrow \mathbb{E}$  defined by  $\mathbf{in}_i p \mapsto I_i(p)$
9.  $Ports^* = \mathbf{in}_1 Ports_1$
10.  $TS^* : T^* \rightarrow \mathbf{NV} \times Pow(P^* \times \mathbf{PN}) \times \mathbf{TASS} \times \mathbf{EASS} \times \mathbf{NASS}$  defined by  $\mathbf{in}_i t \mapsto TS'_i(t)$  if  $t \in dom(TS_i)$ , where  $TS'_i(t) = (net, \{(\mathbf{in}_i p, pn) \mid (p, pn) \in R_i\})$  when  $TS_i(t) = (net, R)$

Define on  $P^*$  the relation  $F$  as the smallest equivalence relation satisfying  $F(\mathbf{in}_i p, \mathbf{in}_1 s)$  if  $R_i(p, s)$ .

Thus defined,  $F$  will relate all ports related to at least one common socket, and all sockets related to at least one common port, and the closure of this.  $F$  will be an equivalence satisfying the properties needed for a fusion set. Thus we can create the net  $[\mathcal{N}^*]_F$ . Let  $\mathcal{N}[\Theta]_{TS}$  be this net.  $\square$

In short, a transition substitution creates copies for each instance of nets, and fuse ports and sockets related by some port/socket relation.

We relate the substitution mapping  $TS$  to net assignments  $\Theta$  in the following way. Whenever we have PCPN  $\mathcal{N}$  and net assignment  $\Theta$  we can define  $\Theta^* : T \rightarrow Pow(\mathbf{P} \times \mathbf{P}) \times \mathbf{PCPN}$  by

$$t \mapsto \begin{cases} (R' \circ R, \mathcal{N}^*) & \text{when } t \in dom(TS) \text{ and} \\ & TS(t) = (net, R, \Gamma, \Delta, \Theta') \text{ and} \\ & net \in dom(\Theta) \text{ and} \\ & \Theta(net) = (R', \mathcal{N}') \text{ and} \\ & \mathcal{N}^* = ((\mathcal{N}'[T]_T)[\Delta]_E)[\Theta']_N \\ \text{undefined} & \text{otherwise} \end{cases}$$

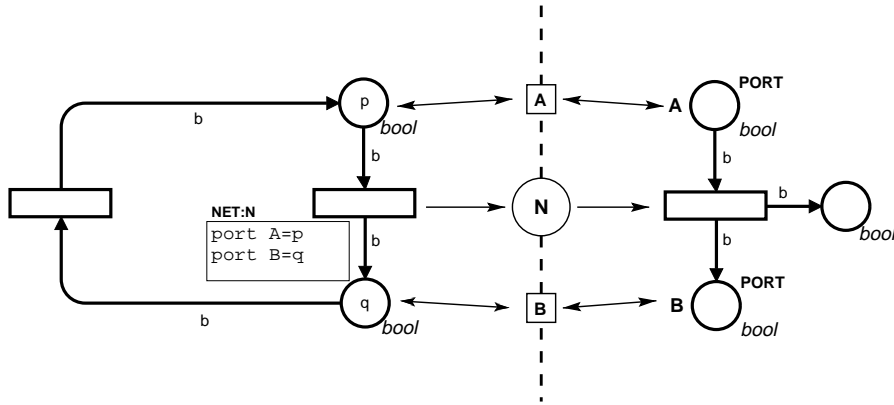
Here  $\mathcal{N}'[\Theta]_N$  is net  $\mathcal{N}'$  after net assignment as defined in Def. 17.

Please also notice the order of assignments in  $\left((\mathcal{N}'[\Gamma]_T)[\Delta]_E\right)[\Theta']_N$ . Expression substitution must always occur after type substitution, to be able to specify values of specific types, and still obey type-rules upon substitution. Placing net substitution last ensures that type and expression substitution only takes place through the substitution mappings in *Sub*.

Notice that the mapping  $\Theta^*$  defined in this way matches the mapping from Def. 16.

Intuitively,  $TS$  maps substitution transitions to variables while  $\Theta$  maps variables to nets.  $\Theta^*$  is the composition that links substitution transitions to nets.

Intuitively  $TS$  relates each substitution transition to a *signature* consisting of the net variable and the naming of the sockets. The net assignment  $\Theta$  then relates a signature to a specific net, by associating the net variable with a net and naming the ports. This is illustrated in Figure 13. On the left we see a substitution transition with two sockets  $p$  and  $q$ .  $TS$  maps this transition to the variable  $\mathcal{N}$ , and relates the socket  $p$  to the name  $A$  and socket  $q$  to the name  $B$ . The signature is drawn on the dashed line. On the right we see a net with two ports. In the figure  $\Theta$  maps variable  $\mathcal{N}$  to this net, and relates the upper port to name  $A$  and the lower port to name  $B$ .  $\Theta^*$  can be thought of as the composition of the arrows in the figure.



**Fig. 13.** Composition of  $TS$  and  $\Theta$

Missing from the figure is the assignments. In this example all assignments are empty, but in general assignments will take place before gluing the nets together.

We will only consider *legal* net assignments, that is, for PCPN  $\mathcal{N}$  and net assignment  $\Theta$ : For  $t \in \text{dom}(\Theta^*)$  and with  $\Theta^*(t) = (R, \mathcal{N}^*)$  the relation  $R$  is a port/socket relation between  $\mathcal{N}^*$  and  $t$ . The signatures used in the examples in Sect. 2 are not part of the definition, but are used to ensure that all net assignments considered are *legal*.

**Definition 17.** For PCPN  $\mathcal{N}$  and net assignment  $\Theta$  we define the net  $\mathcal{N}[\Theta]_N$  to be  $\mathcal{N}[\Theta^*]_{TS}$ .

Notice that Def. 17 is a recursive definition, since it refers to the function  $\Theta^*$  which refers to net assignments.

## 4 PCPN Modules

As defined, PCPN are a kind of *non-hierarchical* CPN with parameters. It could just as well have been defined as a *hierarchical* CPN [5] with parameters. Parameterisation is really orthogonal on the hierarchy concept. There is, however, no need to add hierarchical nets to PCPN, since net parameterisation leads naturally to a module system for PCPN.

We will think of *modules* as self-contained units with a well-defined signature, and constructions for combining modules into other modules. This is exactly what we have with net parameters in PCPN. As

mentioned in Sect. 2 we can talk about the signature of a PCPN, defined from the parameters and the set of ports. Likewise we can talk about the signature of a substitution transition defined by the *NET* region, i.e., the type, expr, net, and port assignments there. Whenever the signature of a net matches that of a substitution transition we can substitute the transition with the net and get a new net.<sup>3</sup>

Nets with signatures work in much the same way as abstract data types. We hide the implementation (the actual net) and only access the functionality through the interface, given by the signature. This allows us to refine one module, or even replace it with a new module, without affecting any other module using the first module.

With Parameterised CPN we can make very general modules, or libraries of modules, and such libraries can then be used in several different models.

Consider a communication protocol, or a protocol suite. Each protocol layer can be modelled as a PCPN with a number of parameters, e.g., a net parameter for the lower layers, a type parameter for the data to be transmitted, and a number of expression parameters for, say, timeout interval or packet sizes. The modules can then be combined in various ways to model different protocol configurations. Furthermore, a library of protocols can be used when modelling distributed applications. Once the protocol library is modelled and validated it can be used in a number of different models.

Tools working with PCPN should of course be built to take advantage of the underlying modularity. For example, tools that generate code from PCPN for simulation, should translate nets into separate units, such that making changes to one net only leads to re-compilation (or re-code-generation) of that particular net. When we write programs in high-level languages, we expect to be able to compile each unit separately, and then link the pieces together. The same should be possible for PCPN.

## 5 Implementation

The modules as described in the previous section can be realized through Standard ML’s module system in a straight forward manner. In this section we will indicate how this can be done. For a discussion of a “proof-of-concept” implementation of a simulator of the kind described in the following, we refer to [7].

We have chosen SML as the implementation language for its module system which closely match our requirements. SML modules consist of *structures*, *signatures*, and *functors*. Structures are used to package up related types, values, and functions. Signatures are then used to specify what components a structure must contain. A *functor* allows structures to be parameterised. In the following we will consider how to translate PCPN modules into SML structures, which can be used for simulation and analysis of the modules.

Let us return to the examples in section 2. The first module we considered was the counter shown in Fig. 14. We will refer to this module as *C*. As we saw in section 2, we can think of the type *T* as a parameter to the module, as indicated in the *PARAMETERS* box. Furthermore we can think of the two port places *A* and *B* as parameters. A net using this module will have to specify both the type *T* and the places to be related to *A* and *B*. Net *C* should therefore be translated into an SML functor which takes *T*, *A*, and *B* as parameters and implements the behaviour of the net.

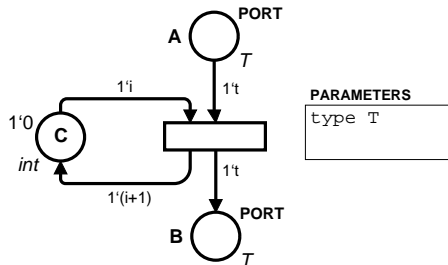


Fig. 14. *C* - Counting transition firings.

<sup>3</sup> In the formal definition of substitution (Def. 17 in the previous section) we only allowed *legal* net assignments. The matching signatures are one way of ensuring this.

A general PCPN module consists of the following components:

**Parameters.** Both the explicit parameters, e.g., the type  $T$  in Fig. 14, and the ports, e.g., the two places  $A$  and  $B$  in Fig. 14.

**Declarations.** Any CPN module can have a number of declarations local to that particular module. We want modules to be self-contained units, so obviously any declaration used in a module should be local.

**The actual net.** This is the most important part for determining the behaviour of the module, and will determine most of the code for executing the module.

**Sub-modules.** A module can contain a number of sub-modules. The sub-modules correspond to substitution transitions, or rather, the nets that substitutes the transitions.

Fig. 15 is a modified version of the module in Fig. 2. It is modified in the way that the arc inscription on the arc from  $X$  to  $A$  has been changed to a call of a function, defined in a *DECLARATIONS* box. This is an example of a module with no parameters, with a non-empty set of declarations, and a non-empty set of sub-modules. Denote this module by  $\mathcal{N}$ .

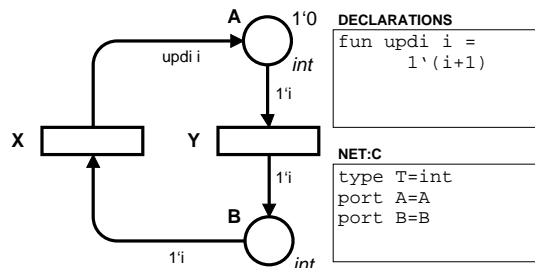


Fig. 15.  $\mathcal{N}$  - Counting transition firings.

This particular module contains one sub-module, which is an instance of  $\mathcal{C}$ , the module in Fig. 14. The module should thus be translated into a SML structure with one sub-structure, corresponding to the module  $\mathcal{C}$ . By implementing  $\mathcal{C}$  as a functor, we get a parameterised structure in SML, and instantiating the sub-module of  $\mathcal{N}$  is a matter of instantiating the SML functor with the appropriate parameters, as specified in the *NET* region.

### 5.1 Translating PCPN modules into SML structures

Before we can decide how to translate a net module into an SML structure, we will have to decide how the structure is going to interact with the tool using the module. In the following we will assume the modules are going to be used in a simulator. Using the modules in other tools, like say a state space tool, should work in a similar fashion.

We will assume that we have a simulator which is responsible for scheduling transition firings, and one global state. Each module will have functions responsible for updating the global state when a firing occurs, and for adding transitions to the scheduling queue when needed.

In general this will work as follows: Initially all transitions will be examined, and the enabled transitions will be added to the scheduling queue. When a transition is scheduled, an enabled binding element will be chosen randomly (if such a binding exists), and the transition will fire and update the state. Changing the state will possibly result in a number of transitions becoming enabled. Each candidate for this will be added to the scheduling queue. We will denote the combination of the selection of binding element, updating state, and scheduling new candidates, as an *event*. We will henceforth refer to the scheduling queue as the *event queue*.

All the actions associated with an event, can be encapsulated in a function or a record of functions, stored in the event queue. Once the initial events are installed in the event queue, the scheduler will only be required to select and execute the next event, if any. No further knowledge of the modules is required.

We will therefore use the signature in Fig. 16 for a PCPN module. Here the function *init* is responsible for initialising the module, i.e., adding all initial events to the event queue. The *init* function is also responsible for initialising any sub-modules, by calling their *init* function.

---

```
signature PCPN_MODULE =
  sig
    val init : unit -> unit
  end
```

---

**Fig. 16.** Signature for PCPN module

A general PCPN module as described above can then be translated to an SML functor of the form seen in Fig. 17.

---

```
functor M(* Parameters *) : PCPN_MODULE =
  structure
    (* — Declarations — *)

    (* — Code for Handling Events — *)

    (* — Sub-modules — *)

    (* — Module Initialisation — *)
    fun init () = (* Initialisation code *)
  end
```

---

**Fig. 17.** Template for PCPN module functor

The functor generated for the module  $\mathcal{C}$  (in Fig. 14) is shown in Fig. 18. In the figure, only the code related to parameterised modules is shown, and the code for interacting with the simulator is omitted. The interesting part in the figure is the parameter part of the functor. The parameters consists of the type parameter  $T$ , and the two port places  $A$  and  $B$ .  $T$  is an SML type parameter, while  $A$  and  $B$  are structures implementing places.

We construct the parameter part of the functor from the *PARAMETERS* box of the net, and we do this in a very straightforward manner. We translate certain PCPN keywords into SML keywords, and insert the rest verbatim into the SML functor. The keywords are translated as follows: *port* is translated to *structure*, *expr* is translated to *val*, and *net* is translated to *functor*. We will return to the latter shortly. This preprocessing is done solely for allowing PCPN keywords in declarations. We could do without it by using the SML keywords. Places marked as ports are also inserted in the parameter part. These are structure parameters with signature *PLACE*.

The last line in the parameter part of the functor in Fig. 18 specifies that the colour set of the port places  $A$  and  $B$  is the same as the parameter type  $T$ . This is implicitly assumed from the colour set specified in the net inscriptions. The line is needed because we implement places as structures, each with a colour set (type)  $C$ .

Figure 19 shows the functor generated for the module  $\mathcal{N}$  in Fig. 15. As can be seen,  $\mathcal{N}$  takes no parameters. In the net there is no *PARAMETERS* box, and in the functor, the parameter part is empty. There is, however, a *DECLARATIONS* box in the net. The declarations there have been inserted verbatim into the functor. The declarations are put at the beginning of the functor, before any code that could possibly need it. In some cases it might be necessary to pre-process a *DECLARATIONS* box before inserting the declarations

---

```

functor C(type T
  structure B : PLACE
  structure A : PLACE
  sharing type T = B.C = A.C) : PCPN_MODULE =
struct
  (* <<snipped open of needed structures>> *)

  (* <<snipped code for handling events>> *)

  fun init () =
    ((* <<snipped code>> *))
end

```

---

**Fig. 18.** Functor for module  $\mathcal{C}$ .

---

```

functor N((* none *)) : PCPN_MODULE =
struct
  (* <<snipped open of needed structures>> *)

  (* DECLARATIONS *)
  fun updi i = 1'(i+1)

  (* <<snipped code for handling events>> *)
  structure Y = C(type T=int structure A=A structure B=B)

  fun init () =
    ((* <<snipped code>> *))
end

```

---

**Fig. 19.** Functor for module  $\mathcal{N}$ .

into the functor in the same way as we process *PARAMETERS* boxes. We will assume this is not necessary for the simulator we are considering.

Notice the sub-module in Fig. 19. This is simply implemented as an SML structure, created by a call to the appropriate functor. In this case the functor  $C$  from Fig. 18. The parameters used in the functor call are created from the *NET* box in the same way as the parameters for the functor  $C$  were created from the *PARAMETERS* box.

In Fig. 19 the functor used to create the sub-module must be globally known, since it is referred to simply by a name. We want to be able to handle net parameters in much the same way, but where we let the functor used to create the module be a parameter of the functor that contains the sub-module. To implement this we can use higher order functors as implemented in SML/NJ [1].

Consider the net in Fig. 20, and denote it  $\mathcal{M}$ . The functor created for this net is seen in Fig. 21. Here  $M$  takes as argument a functor  $N$  and creates the sub-module  $Y$  with this functor. The functor  $N$  is typed with the functor signature  $C'SIG$ . This signature is created automatically from the net  $\mathcal{C}$  we saw in Fig. 14. The definition of  $C'SIG$  is shown in Fig. 22.

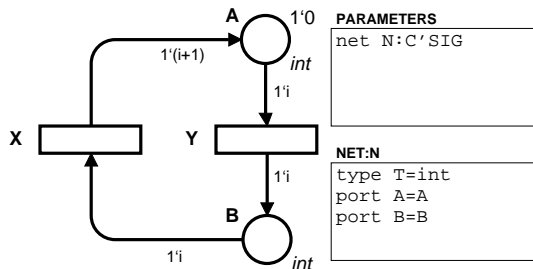


Fig. 20.  $\mathcal{M}$  - Net with net parameter.

## 6 Future work

CPN has powerful and general methods for both validation and verification. Validation is concerned with convincing ourselves that a net behaves as intended, while verification is concerned with proof that a net has a formally stated property. The obvious next step for PCPN is to extend these methods to handle parameters. This is necessary to be able to examine modules independently, and to exploit the underlying modularity.

### 6.1 Validation

Validation is usually done through simulation. In Sect. 5 we examined how to implement a simulator for PCPN. The discussion, however, was only concerned with how to translate PCPN into SML code, and was not concerned with how to actually simulate the nets. With our prototype simulator [7] we can only simulate instantiated nets, and we have only *defined* the behaviour of nets without free type and expression parameters.

Modules, however, are usually self-contained units, and we must expect that modules are primarily developed independently, thus we should be able to validate modules independently. At least, to as high a degree as possible. To do this we have to come up with a meaningful definition of the behaviour of a parameterised net, and tool support for this.

### 6.2 Verification

The primary verification techniques for CPN are state spaces and invariants [6]. Work has been done on exploiting modularity in verification with both techniques in [3] and [4].

The state space method relies heavily on the initial marking of the net in question, and it is thus hard to imagine verification independent of instantiation. Symbolic state spaces, however, could turn out to make this possible in some cases.

Invariants rely on the structure of nets rather than the initial marking, and offers perhaps more promise as a verification technique for PCPN. Type parameters will not influence invariant properties directly, but will determine the weight functions. Expression and net parameters will in general affect properties, but by putting restrictions on the possible nets and expressions assigned to parameters we might be able to prove certain properties.

## 7 Conclusion

In this paper we have formally defined PCPN as CPN with type, expression, and net parameters. We have seen how net parameters lead to a module system for PCPN and examined how a simulator can be implemented such that code for separate modules can be generated independently, and such that instantiated nets can be simulated.

Now, the next step is to create validation and verification techniques for handling parameterised nets individually.

## Acknowledgements

Thanks to Søren Christensen, Bo Lindstrøm, Regnar B. Lyngsø, Kjeld H. Mortensen, and Lisa Wells whose discussion and comments helped improve this paper.

## References

1. Special Features of SML/NJ. WWW online documentation <URL:<http://cm.bell-labs.com/cm/cs/what/smlnj/doc/features.html>>.
2. Søren Christensen and Kjeld H. Mortensen. Parametrisation of Coloured Petri Nets. Technical Report DAIMI PB - 521, Department of Computer Science, University of Aarhus, March 1997.
3. Søren Christensen and L. Petrucci. Modular state space analysis of coloured petri nets. In *Proceeding of the 16th International Conference on Application and Theory of Petri Nets, Turin*, pages 201–217, June 1995.
4. Søren Christensen and Laure Petrucci. Towards a modular analysis of coloured petri nets. In *Proceeding of the 13th International Conference on Application and Theory of Petri Nets, Sheffield, UK*, volume 616, pages 113–133. Springer-Verlag, June 1992.
5. Kurt Jensen. *Coloured Petri Nets - Basic Concepts, Analysis Methods and Practical Use. - Volume 1: Basic Concepts*. Monographs in theoretical computer science, Springer-Verlag, Berlin, 1992.
6. Kurt Jensen. *Coloured Petri Nets - Basic Concepts, Analysis Methods and Practical Use. - Volume 2: Analysis Methods*. Monographs in theoretical computer science, Springer-Verlag, Berlin, 1994.
7. Thomas Mailund. A Simulator for Parameterized CPN Modules. <URL:<http://www.daimi.au.dk/~mailund/pcpn.html>>.
8. Thomas Mailund. Formal Definition of Parameterized CPN. Technical report, Department of Computer Science, University of Aarhus, 1999. Internal Technical Report, <URL:<http://www.daimi.au.dk/~mailund/papers/definitions.ps>>.
9. R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.
10. L.C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 2nd edition, 1996.

---

```

functor M(functor N:C'SIG) : PCPN_MODULE =
  struct
    (* <<snipped code>> *)

    structure Y = N(type T=int structure A=A structure B=B)

    fun init () =
      ((* <<snipped code>> *))
  end

```

---

**Fig. 21.** Functor for module  $\mathcal{M}$ .

---

```

funsig C'SIG (type T
  structure B : PLACE
  structure A : PLACE
  sharing type T = B.C = A.C) = PCPN_MODULE

```

---

**Fig. 22.** Functor signature for net  $\mathcal{C}$ .