# Compositional Petri net structures

## Introduction and formal definition

Jörn W. Janneck

TIK Report 60
November 1998

Computer Engineering and Networks Laboratory
Swiss Federal Institute of Technology (ETH) Zurich

# Contents

# 1 Introduction

Petri nets are a useful tool for modeling, simulating, and analyzing dynamic and concurrent systems. They provide powerful means for modeling concurrency and synchronization in a formal yet intuitive manner.

In order to model systems of practical size at a relevant level of detail, one often has to add to the expressiveness of basic Petri nets. One of the key elements in this respect are a system of *values* instead of black tokens, and some way to express computation on them. This has been addressed comprehensively by various high-level Petri net variants, such as PrT nets [4] and Colored Petri Nets [7].

Especially when modeling large systems, a concept of abstraction is an important that allows for structured design and analysis, and fosters composition of model parts. Previous work in this area includes the various techniques for refining places and transitions [1] as well as various object-oriented Petri net approaches (e.g. [3, 8, 9]).

Building on this work, we try to give a compositional approach to constructing individual Petri net components, which can be combined into larger components in a uniform fashion while at the same time facilitating compositional analysis.

## 1.1 Why compositional Petri nets?

Before we address the question of why we think compositionality is crucial for practical applications of Petri nets, we first want to make our notion of compositionality more precise.

When we say that a modeling technique is compositional, we mean that it facilitates the development of individual model parts ('components') that are *closed* in the sense that knowledge of their internal structure is not necessary in order to use them (provided one has a sufficiently precise description of their behavior), while at the same time *open* in the sense that such a component can provide an arbitrary degree of configurability, has documented interfaces which can be connected to the interfaces of other components, can be used independent of its contexts and is free to make use of other components.

Our model of compositional Petri net structures provides a formal framework that caters all these needs: it features a concept of compositional entities which are arbitrarily parametric and which can provide arbitrary runtime-configurability.

Compositionality has many impacts on Petri net modeling. It provides the necessary concepts of hierarchy and abstraction, which are not only necessary for analysis but are obvious requirements for the construction of any sizable model.

Configurability of model components is supported by parametric component descriptions, i.e. the behavior of components may be made dependent on parameters.

Another feature of our approach is that it naturally supports dynamic network structures. It has a well-defined notion of dynamically linking components together, in fact it has only one such notion, and this is based on a dynamic concept of component topology.[1]

## 1.2 Informal description

In this part, we will informally describe the three aspects for which we deem a compositional concept of Petri nets necessary:

- hierarchy and abstraction
- parametric construction
- dynamic structures

First, however, we give a short impression of what our 'components' look like.

[1] Of course, many interesting special cases of this will be networks with a static structure, and fortunately this subclass can be identified by a simple predicate on the static structure of the component network.

### 1.2.1 Components, hierarchy, and abstraction

The absence of any concept of composition and abstraction in pure Petri nets has long been known (see, for instance, [6]) and given rise to several approaches that overcome this deficiency. The usual way to add hierarchical composition to Petri nets is the *refinement* of either or both of its standard atomic building blocks, place and transition [7], [8], [9], [12]. A common problem here is that refining places or transitions destroys some of their key properties [1].

In this paper we will therefore use a different approach based on an structuring concept of high-level Petri nets that was developed in [3]. Instead of trying to directly refine either places or transitions, we will consider components to be subnets with inputs and outputs, which can be embedded into other components. For example, Fig. 1 shows the definition of a component of type `Adder` that has two inputs and one output depicted as triangles (we will denote the direction of the token flow by the direction of the triangle towards the component edge). We will require inputs to be connected to places and outputs to transitions to ensure that any component output can be meaningfully connected to any component input.[2] Also, inputs and outputs (collectively referred to as *interfaces*) are named: each **Adder** component has two inputs named `in1` and `in2`, respectively, and one output `out`.

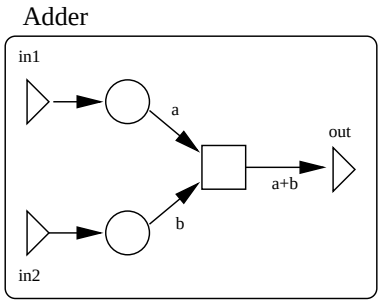


Figure 1: A component.

In Fig. 2 we see the use of an `Adder` instance in the definition of another component, `Nats`, generating the sequence of natural numbers. It has an input (`next`) thru which the next number may be requested and an output (`nats`), at which the natural numbers are delivered. The embedding rules allow regular Petri net places and transitions to be connected to the interfaces of embedded components and these interfaces to be connected to each other or to the interfaces of the embedding component in ways that allow their substitution down to regular place/transition and transition/place connection.

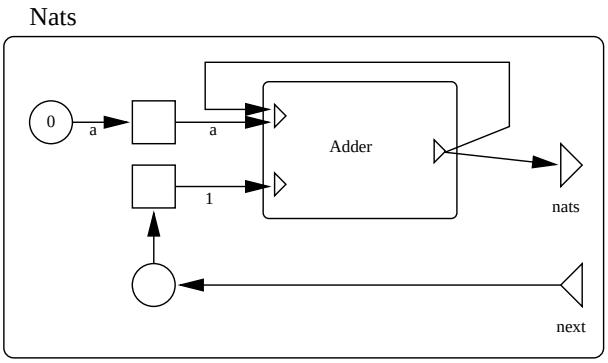


Figure 2: A compound component.

[2] A more detailed examination of possible connections and their relations is given in section 2.2.

The semantics of such a net is defined by substitution rules (which will be given formally in section 2.5), which eventually produce a simple high-level Petri net.

In the case of the natural number generator in Fig. 2, substituting the **Adder** would involve replacing the connections between places and transitions that cross component boundaries (via interfaces) by the corresponding direct connections. The result of flattening the `Nats` component in Fig. 2 can be seen in Fig. 3a. In this way we can recursively flatten any hierarchical system to arrive at a non-hierarchical colored Petri net.

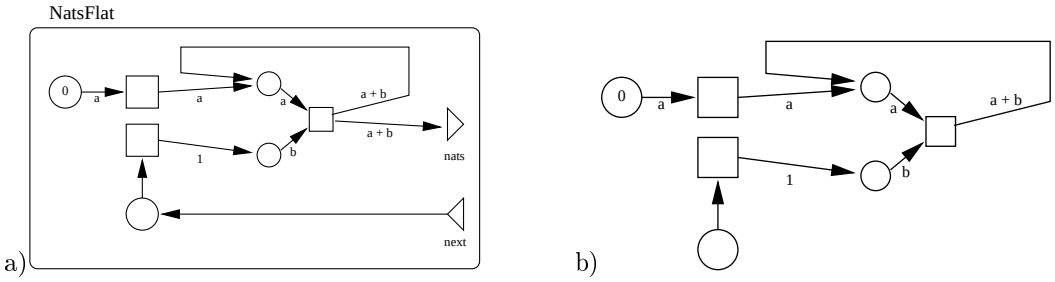


Figure 3: A flat natural number generator.

Once we have obtained such a flat component, we can generate a Petri net by simply removing the interfaces and the arcs to them. The resulting net can be seen in Fig. 3b.

### 1.2.2 Parametric construction

Parametric construction of components enables us to configure component behavior by providing different values for its *instantiation parameters* at the time when it is constructed.

Suppose that we wish to define the `Nats` component class in Fig. 2 in such a way as to make the starting value of the sequence a parameter (instead of a constant 1). While this certainly does not affect its interfaces, we would have to introduce parameters to the class that are defined whenever an instance is created, as shown in Fig. 4.

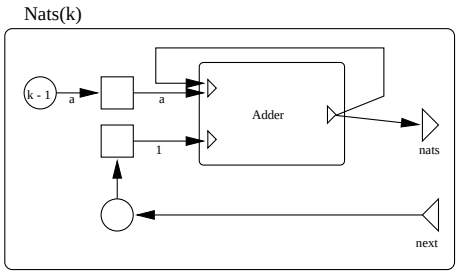


Figure 4: A parametric natural number generator.

Note that the parameter `k` appears in the initialization expression of the top left place, resulting in one token with the value `k-1`. Instantiating a component inside a net might look like Fig. 5.

At this point we can reconsider the question of instantiating an embedded component in the definition of a new component class. Our parametric `Nats` component from Fig. 4 could be used inside a component generating all even numbers starting from 2 as shown in Fig. 5. We instantiate a `Nats` component by `Nats(1)` as, which is an *instantiation expression*. Such an expression could be any expression that evaluates to a component, i.e. which has a component type (`Nats`, in this case) and potentially some parameters. Since it is evaluated at the time when the component is instantiated (just as the expressions describing initial tokens in places), it can only depend on the parameters of the component class and constants. We will now turn to an example illustrating a more sophisticated use of this mechanism.

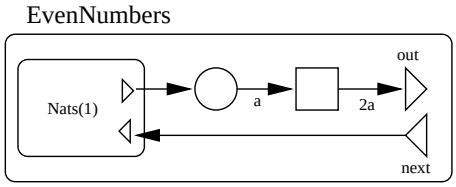


Figure 5: Even numbers generator instantiating `Nats(1)`.

### 1.2.3 Dynamic structures

In this section we will try to illustrate some of the possible uses of dynamic Petri net structures. Many examples are conceivable, where the network contains different levels of activity and one level controls the other – process schedulers in an operating system, production systems with autonomous units, information management systems etc.

However, in order to demonstrate the versatility of the concept of dynamic net structures, we will use a very simple example that nevertheless yields arbitrarily large net structures – i.e. given enough time the net may grow beyond any finite bound – still the model is very small and intuitive: We will build a component that computes the prime numbers in their proper order using the method known as the *Sieve of Erastosthenes*, i.e. by successively eliminating the multiples of the numbers it has so far recognized as primes.

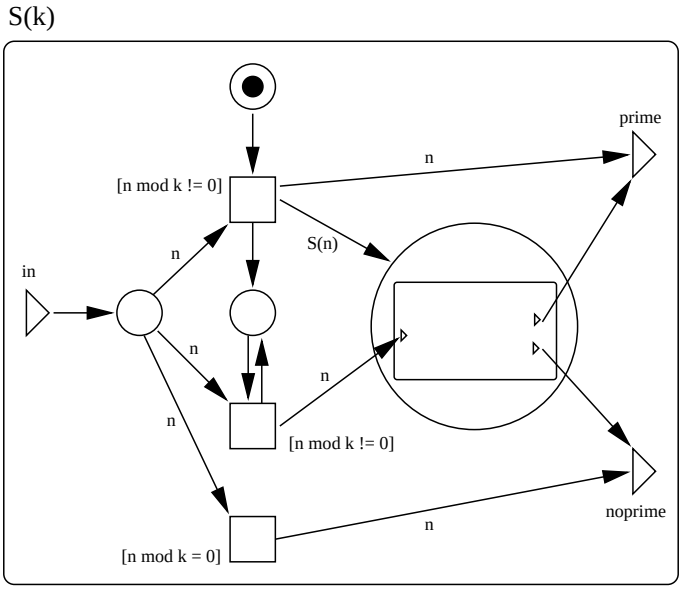


Figure 6: The Sieve of Erastosthenes for prime $k$.

Again, in order to simplify our task we will first consider the generation of a single prime, and later embed this solution into a component that generates a stream of primes. We will build our sieve as a nested succession of components removing from the stream of natural numbers those which are multiples of some given number $k$. For all others, i.e. those which are not divisible by $k$, we have to consider two cases: either we already have a contained component for some $k' > k$, in which case we simply can pass the number to it for further sieving. Or we do not have such a contained sieve – then we have detected the next larger prime and do two things: We create a corresponding new sieve for $k'$, and we output $k'$ as our newly detected prime. The corresponding component could look like the one in Fig. 6.

Note the constructor expression `S(n)` that creates a new instance of `S` with its instantiation parameter `k` bound to the newly found prime.

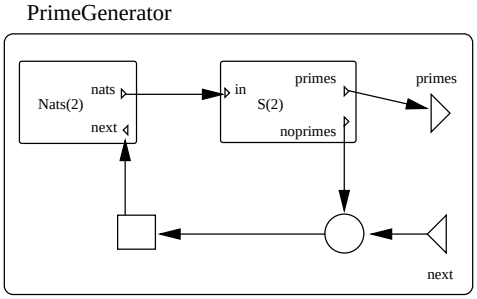


Figure 7: A simple prime generator.

Embedding this sieve into a component generating a stream of primes now is a relatively simple task.

# 2 Formal definition

In the following, we will give a precise formal definition of how a system of compositional Petri net structures (COPS) evolves over time. We will do this in the following steps:

- After a few introductory remarks, we define the structure of Petri net components and thus make our notion of compositional Petri nets structures more explicit.
- From this we can almost deduce the structure of the state space, which differs only slightly from the structure of regular Petri net state spaces, primarily by the fact that its size can be arbitrarily large and dynamic
- Finally, we define state transitions in this framework. We do this in two steps, by first giving a mapping from a given system of compositional Petri net structures into a regular Petri net, and then we can base most of the notion of state transition on the way this has been defined for Petri nets.

## 2.1 Initial remarks

### 2.1.1 Multisets

Just as with CPN, multisets occur in the definition of COPS and the resulting Petri nets. A multiset maps a set to a natural number, denoting the number of occurrences of the respective element in that set. Thus a multiset $M$ of elements in a base set $S$ is a function

$$M : S \to \mathbb{N}$$

We will need two operations on multisets, viz. their addition and the multiplication of a multiset with a natural number:

$$(M_1 + M_2)(e) = M_1(e) + M_2(e)$$
$$(n * M)(e) = n * M(e)$$

### 2.1.2 Types and values

All computation defined by the Petri net happens on a collection of individual and atomic entities. The collection of all such potential entities is the

**Definition 2.1 (Universe).** We assume the existence of a infinite set of possible values, the *universe* $\mathcal{U}$. We assume to have at each instant an infinite set og hitherto 'unused' values, the *reserve* $\mathcal{R} \subseteq \mathcal{U}$.

The reserve is important when new values are allocated during a run. In this work we will not address this issue any further.

In contrast to CPN [7], we will not assume any type system for this work, i.e. there is no structure among the values og the universe that we will take into account – with the notable exception of the values that represent Petri net components (cf. below). A rationale for this decision will be given in section 3.1.

### 2.1.3 Expressions, variables, and evaluation

Since we basically build on Colored Petri Nets as defined in [7], we have variables and expressions to perform atomic computational steps when transitions occur in the Petri net. These are defined as follows:

**Definition 2.2 (Variables, binding).** Given a sufficiently large set $\mathcal{V}$ of *variable symbols*, we call will call any function

$$b : V \to \mathcal{U} \qquad \text{with} \qquad V \subset \mathcal{V}$$

a *binding* of the variables in $V$.

We define the combination $[b_1, b_2]$ of two bindings $b_1$ and $b_2$ as follows:

$$[b_1, b_2] : V_1 \cup V_2 \to \mathcal{U}$$
$$[b_1, b_2](v) \mapsto \begin{cases} b_2(v) & v \in V_2 \\ b_1(v) & \text{otherwise} \end{cases}$$

**Definition 2.3 (Expressions, evaluation).** For the purpose of this work we will leave the precise syntactic structure of *expressions* unspecified. However, we will call the set of all expressions $\mathcal{E}$, and the set of free variables in an expression $E$ will be denoted as $free(E)$.

We call $E$ *closed* iff $free(E) = \emptyset$.

**Definition 2.4 (Evaluation).** Without specifying it in any detail, we will assume that each expression $E$ defines an evaluation $eval_E$ which takes a binding $b$ and computes a value in $\mathcal{U}$. In order for this evaluation to be well-defined, we require that $free(E) \subseteq dom(b)$.

We will write $eval_E(b)$ as $E[b]$ and call it the value of $E$ under $b$.

**Definition 2.5 (Substitution).** Given an expression $E$ and a binding $b$, we can construct a new expression $E' = E \mid b$ by conceptually substituting each free occurrence of some variable $v \in dom(b)$ in $E$ by its value $b(v)$. We will leave the exact nature of the substitution unspecified, but we require that

$$\forall b' : (E|b)[b'] = E[b \circ b']$$

with

$$b \circ b'(v) = \begin{cases} b(v) & v \in dom\ b \\ b'(v) & \text{otherwise} \end{cases}$$

Substituting $b$ into an expression $E$ potentially reduces its set of free variables:

$$free(E \mid b) = free(E) \setminus dom\ b$$

## 2.2 Taxonomy of nodes and arcs

Before we will give a formal definition of component replacement, we first have to investigate COPS networks a little more closely and examine the different types of nodes and connections that occur in them.

COPS networks are basically composed of six different types of nodes: like Petri nets, they contain of course places and transitions (P and T nodes, respectively). They also contain input and output connectors (which we call I and O nodes), as well as the input and output connectors of embedded components (termed PI and PO nodes in this text).

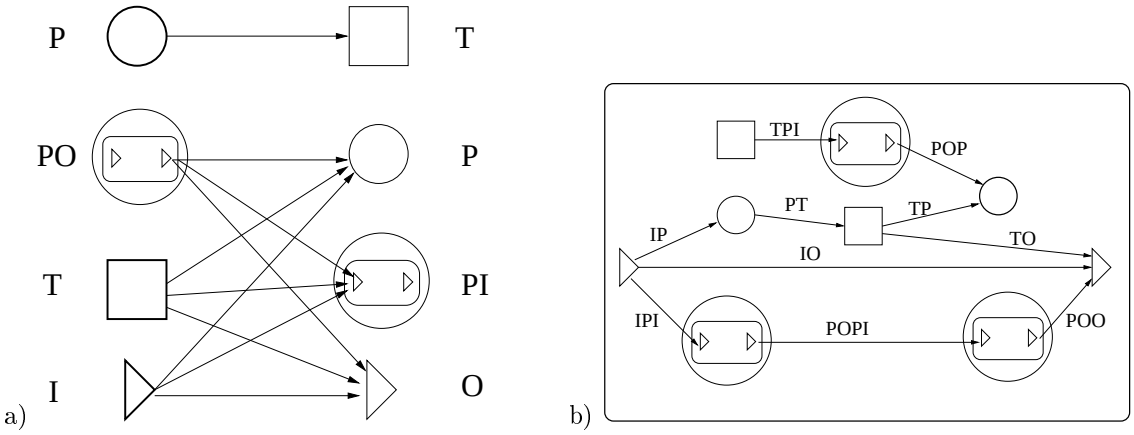


Figure 8: Connections between COPS nodes.

Between these six types of nodes, we have ten different kinds of possible connections, as shown in Fig. 8 (Fig. 8a shows them schematically, while Fig. 8b shows them in context.) Connection types are named by the node types they link, i.e. the connections that link P and T nodes are termed 'PT'-connections.

There are two important classes of connections to be distinguished, viz. those that are associated with expressions (doing computation on the token values) vs those that are not. Fortunately, the classification is very simple: exactly those arcs that are connected to a transition carry expressions, the others are 'simple'. Fig. 9 shows the various connection types and whether they carry expressions: an "E" denotes an arc type that is connected to expressions, while an "S" denotes a simple arc type, i.e. one that is not.

| | P | T | I | O | PI | PO |
|---|---|---|---|---|---|---|
| P | | E | | | | |
| T | E | | | E | E | |
| I | S | | | S | S | |
| O | | | | | | |
| PI | | | | | | |
| PO | S | | | S | S | |

Figure 9: Connections between COPS nodes and their relation to expressions.

Except for the PT connection, which for reasons that become obvious below do not play much of a role in component replacement, the other nine connection types divide the six types of nodes into *sources* and *destinations* according to the following table:

| Source | Destination |
|---|---|
| $T$ | P |
| PO | PI |
| I | O |

## 2.3 Compositional Petri net structures

In the following, we will assume the existence of a few basic sets of abstract 'names' for different kinds of objects such as places and transitions. We will not assume any particular nature of these names, except that they be distinguishable and – for convenience – that the name sets are mutually disjoint. In particular, we define the following sets of names:

- $\mathcal{N}_P$ for places
- $\mathcal{N}_T$ for transitions
- $\mathcal{N}_I$ for input connectors
- $\mathcal{N}_O$ for output connectors

**Definition 2.6 (Compositional Petri net structure COPS).** A *compositional Petri net structure* (COPS or *structure* for short) $C$ is a tuple with the following structure:

$$C = (P, T, I, O, A, E, E_{init}, V)$$

where

$P$, $T$, $I$, $O$ are designators for places, transitions, inputs, and outputs, respectively:
$P \subset \mathcal{N}_P$
$T \subset \mathcal{N}_T$
$I \subset \mathcal{N}_I$
$O \subset \mathcal{N}_O$

$A$ is a group of sets defining the arcs of the component class:
$A = (A_{PT}, A_{TP}, A_{TO}, A_{TPI}, A_{IO}, A_{IP}, A_{IPI}, A_{POO}, A_{POP}, A_{POPI}\}$
$A_{PT} \subseteq P \times T$
$A_{TP} \subseteq T \times P$
$A_{TO} \subseteq T \times O$
$A_{TPI} \subset T \times P \times \mathcal{N}_I$
$A_{IP} \subseteq I \times P$
$A_{IPI} \subset I \times P \times \mathcal{N}_I$
$A_{POO} \subset P \times \mathcal{N}_O \times O$
$A_{POP} \subset P \times \mathcal{N}_O \times P$
$A_{POPI} \subset P \times \mathcal{O} \times P \times \mathcal{N}_I$

$E$ is a group of functions that assign expression multisets to some of the arcs:
$E = (E_{PT}, E_{TP}, E_{TO}, E_{TPI})$
$E_{PT} : A_{PT} \to \mathcal{E}^*$
$E_{TP} : A_{TP} \to \mathcal{E}^*$
$E_{TO} : A_{TO} \to \mathcal{E}^*$
$E_{TPI} : A_{TPI} \to \mathcal{E}^*$

$E_{init}$ assigns an initialization expression to each place:
$E_{init} : P \to \mathcal{E}$

$G$ is a a function assigning guard expressions:
$G : T \to \mathcal{E}$

$V$ is a set of variables:'
$V \subset \mathcal{V}$

When the structure $C$ is not understood from the context, we will write it after the respective set in angle brackets, e.g. $P\langle C\rangle$, or $A_{POP}\langle C\rangle$.

**Definition 2.7 (Well-formed structures).** Well-formedness of a structure involves the well-definedness of the evaluation of its expressions under bindings $b$ with $dom\ b \supseteq V$. Formally, we call a structure *well-formed*, iff

$$\forall p : free(E_{init}(p)) \subseteq V$$

and

$$\forall t \in T : free(G(t)) \subseteq V \cup Var(t)$$

with

$$Var(t) = \left[ \bigcup_{p \in P} [free(E_{PT}(p,t)) \cup free(E_{TP}(t,p))] \cup \bigcup_{(p,i) \in P \times \mathcal{N}_I} free(E_{TPI}(t,p,i) \cup \bigcup_{o \in \mathcal{N}_O} free(E_{TO}(t,o) \right] \setminus V$$

Of course, the expressions bound to the arcs may have free variables outside $V$, since those will be bound in bindings (cf. section 2.6).

In any given application, we will usually deal with collections of these structures, which we will call a system:

**Definition 2.8 (System of structures).** A *system of COPS* is a set $\mathcal{C}$ of compositional Petri net structures.

These structures may be considered *templates* which individual parts of the system are instantiated from. Any such instance is represented by an atomic entity that is element of the universe, *designator*:

**Definition 2.9 (Instance designators).** Assume a system $\mathcal{C}$ of structures $C \in \mathcal{C}$. Assume further an infinite universe of $\mathcal{U}$ of possible values of tokens. Then we denote as $\overline{C}$ the set of all *instance designators* (or also *component designator*) of structure $C$ , with the following properties:

$$\begin{aligned} &\overline{C} \qquad \text{infinite} \\ &\overline{C} \subset \mathcal{U} \\ &\forall C_1, C_2 \in \mathcal{C} : (C_1 \neq C_2) \Rightarrow \overline{C_1} \cap \overline{C_2} = \emptyset \end{aligned}$$

Since it is uniquely defined, we will denote the structure of component designator $c$ as $C_c$. The set of all instance designators is $\overline{\mathcal{C}} = \bigcup_{C \in \mathcal{C}} \overline{C}$.

Of course, such an instance also has a state which forms part of the global state space. Before we can go on we have thus to discuss the structure of that state space.

## 2.4 The state space

As with conventional Petri nets, the state is 'stored' in places. However, in a system of COPS, we have an arbitrarily large number of places, viz. for each instance designator $c$ a local copy for each of its places in $P/langle C_c/rangle$. We therefore define the set of all (possible) places as:

**Definition 2.10 (Set of all Places).** We define the set of all places $\mathcal{P}$ as follows:

$$\mathcal{P} = \{(c,n) \mid \exists\ C \in \mathcal{C}, c \in \mathcal{U}, n \in \mathcal{N}_P : c \in \overline{C} \wedge n \in P\langle C\rangle\}$$

Usually, we will assume each given reachable state to be finite, i.e. we will have a notion of a 'relevant' subset of $\mathcal{P}$ that we are interested in.

Nevertheless, assuming that we maximally might look at all places in $\mathcal{P}$, the global marking thus becomes:

**Definition 2.11 (Global marking).** The *global marking* of the system is a function

$$\mathcal{M} : \mathcal{P} \to U^*$$

The set of all global states $\mathcal{M}$ is written as $\mathbb{M}$.

This, however, does not quite suffice to represent all state information that the execution of the system may depend on. When instantiating a structure, we do this with respect to a binding of its variables, which may influence the evaluation of expressions inside the component. Thus we have to keep a record of the relation between instantiated designators and the binding of their instance variables $V$.

**Definition 2.12 (Instantiation bindings).** The *instantiation bindings* are a function

$$\begin{aligned}&\mathcal{B} : \overline{\mathcal{C}} \to (\mathcal{V} \to \mathcal{U})\\ &\qquad \text{such that} \quad \forall c \in \overline{\mathcal{C}} : dom\ \mathcal{B}(c) \supseteq V\langle C_c \rangle\end{aligned}$$

**Definition 2.13 (Global state).** The *global state* of a system is a tuple $(\mathcal{M}, \mathcal{B})$ of a global marking and instantiation bindings.

The two components of a global state are slightly different in nature: while the marking of a place is subject to changes due to a state transition (cf. section 2.6), the binding associated with a designator remains unchanged once the designator has been instantiated.

Given a global state, create a new instance of some structure $C$ as follows:

**Definition 2.14 (Instance creation operator $\chi$).** Given a component class $C \in \mathcal{C}$ and a binding $b$ such that $dom\ b \subseteq V\langle C \rangle$, one can create an instance of the component class in the state $(\mathcal{M}, \mathcal{B})$ by performing the following two steps:

1. Select a $c_{new} \in \overline{C} \cap \mathcal{R}$, and remove it from $\mathcal{R}$.
2. Update the global marking and the instantiation bindings as follows:

$$\mathcal{M}'(c,n) = \begin{cases} E_{init}\langle C \rangle(n)[b] & c = c_{new} \\ \mathcal{M}(c,n) & \text{otherwise} \end{cases} \qquad \mathcal{B}'(c) = \begin{cases} b & c = c_{new} \\ \mathcal{B}(c) & \text{otherwise} \end{cases}$$

We say that $\chi(C, b, (\mathcal{M}, \mathcal{B})) = (c, (\mathcal{M}', \mathcal{B}'))$.

## 2.5 Constructing a Petri net

We will now define the individual steps needed to produce a Petri net from a system of compositional Petri net structures. The first step will be the construction of a ground Petri net for a given designator $c$, which is basically a 'copy' of the structure $C_c$. Coming back to the natural number generator of section 1.2.1, the result of this step is structurally still looking like the one in Fig. 2 or Fig. 4. The only difference to the latter is that the initialization expressions are evaluated, so instead of a $k$ one would find whatever value was bound to that variable for the instantiation.

**Definition 2.15 (Ground Petri net structure, grounding operator $\gamma$).** The *grounding operator* $\gamma$ serves to create a structure of 'instantiated' nodes, i.e. those which besides their node names are also identified with respect to a given component designator. This structure $G = \gamma(c, \mathcal{B})$ for a designator $c$ is computed from the class $C_c$ as follows:

$$\gamma(c, \mathcal{B}) = (P, T, I, O, A, E)$$

where

$P$, $T$, $I$, $O$ are ground places, transitions, inputs, and outputs, respectively:

$P = \{c\} \times P\langle C_c \rangle$

$T = \{c\} \times T\langle C_c \rangle$

$I = \{c\} \times I\langle C_c \rangle$

$O = \{c\} \times O\langle C_c \rangle$

$A$ is a group of sets defining the ground arcs of the ground component:

$A = (A_{PT}, A_{TP}, A_{TO}, A_{TPI}, A_{IO}, A_{IP}, A_{IPI}, A_{POO}, A_{POP}, A_{POPI}\}$

$A_{PT} = \{((c,p),(c,t)) \mid (p,t) \in A_{PT}\langle C_c \rangle\}$

$A_{TP} = \{((c,t),(c,p)) \mid (t,p) \in A_{TP}\langle C_c \rangle\}$

$A_{TO} = \{((c,t),(c,o)) \mid (t,o) \in A_{TO}\langle C_c \rangle\}$

$A_{TPI} = \{((c,t),(c,p),i) \mid (t,p,i) \in A_{TPI}\langle C_c \rangle\}$

$A_{IO} = \{((c,i),(c,o)) \mid (i,o) \in A_{IO}\langle C_c \rangle\}$

$A_{IP} = \{((c,i),(c,p)) \mid (i,p) \in A_{IP}\langle C_c \rangle\}$

$A_{IPI} = \{((c,i),(c,p),j) \mid (i,p,j) \in A_{IPI}\langle C_c \rangle\}$

$A_{POO} = \{((c,p),o,(c,q)) \mid (p,o,q) \in A_{POO}\langle C_c \rangle\}$

$A_{POP} = \{((c,p),o,(c,q \mid (p,o,q) \in A_{POP}\langle C_c \rangle\}$

$A_{POPI} = \{((c,p),o,(c,q),i) \mid (p,o,q,i) \in A_{POPI}\langle C_c \rangle\}$

$E$ is a group of functions that assign expression multisets to some of the arcs:

$E = (E_{PT}, E_{TP}, E_{TO}, E_{TPI})$

$E_{PT} : ((c,p),(c,t)) \mapsto E_{PT}\langle C_c \rangle(p,t)|\mathcal{B}(c)$

$E_{TP} : ((c,t),(c,p)) \mapsto E_{TP}\langle C_c \rangle(t,p)|\mathcal{B}(c)$

$E_{TO} : ((c,p),(c,o)) \mapsto E_{TO}\langle C_c \rangle(t,o)|\mathcal{B}(c)$

$E_{TPI} : ((c,t),(c,p),i) \mapsto E_{TPI}\langle C_c \rangle(t,p,i)|\mathcal{B}(c)$

$G$ is a a function assigning guard expressions:

$G : (c,t) \mapsto G\langle C_c \rangle(t)|\mathcal{B}(c)$

When the ground structure $G$ is not understood from the context, we will write it after the respective set in angle brackets, e.g. $P\langle G \rangle$, or $A_{POP}\langle G \rangle$.

A Petri net component, in contrast, does not contain any arcs to 'embedded' components, i.e. those whose designators reside on a place in the ground structure.

**Definition 2.16 (Petri net component).** A *Petri net component* $\Gamma$ (or *component* for short) is a tuple as follows:

$$
\begin{aligned}
&\Gamma = (P, T, I, O, A, E) \qquad \text{with} \\
&\quad P \subseteq \overline{\mathcal{C}} \times \mathcal{N}_P \\
&\quad T \subseteq \overline{\mathcal{C}} \times \mathcal{N}_T \\
&\quad I \subseteq \overline{\mathcal{C}} \times \mathcal{N}_I \\
&\quad O \subseteq \overline{\mathcal{C}} \times \mathcal{N}_O \\
&\quad A = (A_{PT}, A_{TP}, A_{TO}, A_{IO}, A_{IP}) \qquad \text{with} \\
&\qquad A_{PT} \subseteq P \times T \\
&\qquad A_{TP} \subseteq T \times P \\
&\qquad A_{TO} \subseteq T \times O \\
&\qquad A_{IO} \subseteq I \times O \\
&\qquad A_{IP} \subseteq I \times P \\
&\quad E = (E_{PT}, E_{TP}, E_{TO}) \qquad \text{with} \\
&\qquad E_{PT} : A_{PT} \to \mathcal{E}^* \\
&\qquad E_{TP} : A_{TP} \to \mathcal{E}^* \\
&\qquad E_{TO} : A_{TO} \to \mathcal{E}^* \\
&\quad G : T \to \mathcal{E}
\end{aligned}
$$

The substitution operator essentially transforms a ground structure into a component. This corresponds to the transition to Fig. 3a for the natural number generator.

**Definition 2.17 (Substitution operator $\sigma$).** Given a ground structure $G$ and a state $(\mathcal{M}, \mathcal{B})$, the substitution operator $\sigma$ computes a corresponding Petri net component $\sigma(G, (\mathcal{M}, \mathcal{B}))$ as follows:

- Let $R = \{(p, c) \mid p \in P\langle G\rangle, c \in \mathcal{M}(p) \cap \overline{\mathcal{C}}\}$ the set of all places/instance designator pairs that have to be replaced.
- Furthermore, to shorten the following, let $S_{XYZ}$ be a group of sets of sequences of direct input/output connections, defined as follows:

$$
\begin{aligned}
S_{IO} = \{&(p_k, c_k, ni_k, no_k)_{k=1\ldots n} \mid ni_k \in I\langle C_{c_k}\rangle \wedge no_k \in O\langle C_{c_k}\rangle \wedge (ni_k, no_k) \in A_{IO}\langle C_{c_k}\rangle \\
&\wedge (p_{k-1}, no_{k-1}, p_k, ni_k) \in A_{POPI}\langle G\rangle\} \\
S_{OI} = \{&(p_k, c_k, no_k, p'_k, c'_k, ni_k)_{k=1\ldots n} \mid ni_k \in I\langle C_{c'_k}\rangle \wedge no_k \in O\langle C_{c_k}\rangle \wedge (p_k, no_k, p'_k, ni_k) \in A_{POPI}\langle G\rangle \\
&\wedge \forall k > 1 : p'_{k-1} = p_k \wedge c'_{k-1} = c_k \wedge (ni_{k-1}, no_k) \in A_{IO}\langle C_{c_k}\rangle\} \\
S_{IOI} = \{&(p_0, c_0, ni_0, no_k, p_k, c_k, ni_k)_{k=0\ldots n} \mid ni_k \in I\langle C_{c_k}\rangle \wedge no_k \in O\langle C_{c_{k-1}}\rangle \\
&\wedge (ni_{k-1}, no_k) \in A_{IO}\langle C_{c_{k-1}}\rangle \wedge (p_{k-1}, no_{k-1}, p_k, ni_k) \in A_{POPI}\langle G\rangle\} \\
S_{OIO} = \{&(p_0, c_0, no_0, p_k, c_k, ni_k, no_k)_{k=0\ldots n} \mid ni_k \in I\langle C_{c_k}\rangle \wedge no_k \in O\langle C_{c_k}\rangle \\
&\wedge (p_{k-1}, no_{k-1}, p_k, ni_k) \in A_{POPI}\langle G\rangle\} \wedge (ni_k, no_k) \in A_{IO}\langle C_{c_k}\rangle\}
\end{aligned}
$$

- Now compute the basic carrier sets of $\sigma(G, \mathcal{M})$ as follows:

$$P = P\langle G\rangle \cup \bigcup_{(p,c)\in R} P\langle\sigma(\gamma(c), \mathcal{M})\rangle$$
$$T = T\langle G\rangle \cup \bigcup_{(p,c)\in R} T\langle\sigma(\gamma(c), \mathcal{M})\rangle$$
$$I = P\langle G\rangle$$
$$O = P\langle G\rangle$$

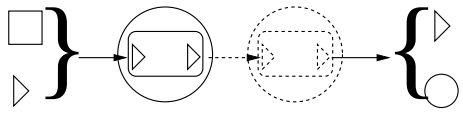

Figure 10: Schema 1.

- The arcs are substituted according to four basic schemata. We call the set of source nodes **S**, the set of destination nodes **D**. The first schema (Fig. 10) then produces the following set of substitution arcs

$$S1_{\mathbf{SD}} = \{(s, d) \mid \exists (p_k, c_k, ni_k, no_k)_{k=1\ldots n} \in S_{IO} : (s, p_1, ni_1) \in A_{\mathbf{S}PI}\langle G\rangle \wedge (p_n, no_n, d) \in A_{PO\mathbf{D}}\langle G\rangle\}$$

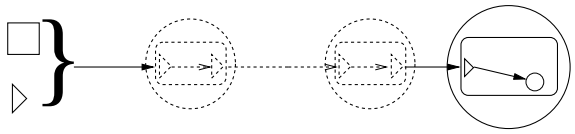

Figure 11: Schema 2.

The second schema (Fig. 11) is defined as

$$S2_{\mathbf{SD}} = \{(s, d) \mid \exists (p_0, c_0, ni_0, no_k, p_k, c_k, ni_k)_{k=0\ldots n} \in S_{IOI} : (s, p_0, ni_0))) \in A_{\mathbf{S}PI}\langle G\rangle \wedge ((c_n, ni_n), d) \in A_{I\mathbf{D}}\langle\gamma(c_n)\rangle\}$$

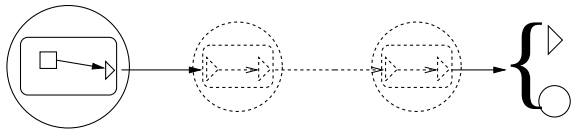

Figure 12: Schema 3.

The third schema (Fig. 12) is defined as

$$S3_{\mathbf{SD}} = \{(s, d) \mid \exists (p_0, c_0, no_0, p_k, c_k, ni_k, no_k)_{k=0\ldots n} \in S_{OIO} : (s, (c_0, no_0))) \in A_{\mathbf{S}O}\langle\gamma(c_0)\rangle \wedge (p_n, no_n, d) \in A_{PO\mathbf{D}}\langle G\rangle\}$$

Finally, the fourth schema (Fig. 13) is[3]

$$S4_{\mathbf{SD}} = \{(s, d) \mid \exists (p_k, c_k, no_k, p'_k, c'_k, ni_k)_{k=1\ldots n} \in S_{OI} : (s, (c_0, no_0))) \in A_{\mathbf{S}O}\langle\gamma(c_0)\rangle \wedge ((c_n, ni_n), p) \in A_{I\mathbf{D}}\langle\gamma(c_n)\rangle\}$$

[3]Actually, this schema will only be instantiated once below, but for reasons of symmetry we presented it here in its more general form.

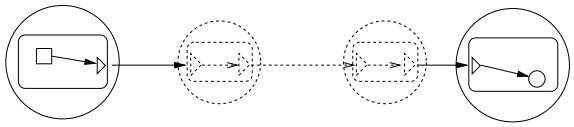

Figure 13: Schema 4.

- Now we are in a position to concisely define the new arc sets:

$$
\begin{aligned}
A_{PT} &= A_{PT}\langle G\rangle \cup \bigcup_{(p,c)\in R} A_{PT}\langle\sigma\gamma(c)\rangle \\
A_{TP} &= A_{TP}\langle G\rangle \cup S1_{TP} \cup S2_{TP} \cup S3_{TP} \cup S4_{TP} \\
A_{TO} &= A_{TO}\langle G\rangle \cup S1_{TO} \cup S3_{TO} \\
A_{IO} &= A_{IO}\langle G\rangle \cup S1_{IO} \\
A_{IP} &= A_{IP}\langle G\rangle \cup S1_{IP} \cup S2_{IP}
\end{aligned}
$$

- Finally, we construct the new expressions for the substituted edges. For PT-arcs, this is rather straightforward:

$$E_{PT} = E_{PT}\langle G\rangle + \sum_{(p,c)\in R} E_{PT}\langle\sigma(\gamma(c),(\mathcal{M},\mathcal{B}))\rangle$$

- For arcs going from transitions to some node type $\mathbf{D}$ (either places or outputs) the construction parallels the construction of the substitution arc sets:

$$
\begin{aligned}
E1_{\mathbf{D}} =&\{(t,d)\mapsto E \mid E = \sum_{(t,p,ni)} E_{TPI}\langle G\rangle * \\
&\#\{(p_k,c_k,ni_k,no_k)_{k=1\ldots n}\in S_{IO} \mid p=p_1, ni=ni_1, (t,p_1,ni_1)\in A_{TPI}\langle G\rangle \wedge (p_n,no_n,d)\in A_{PO\mathbf{D}}\langle G\rangle\} \\
E2_{\mathbf{D}} =&\{(t,d)\mapsto\mid E = \sum_{(t,p,ni)} E_{TPI}\langle G\rangle * \\
&\#\{(p_0,c_0,ni_0,no_k,p_k,c_k,ni_k)_{k=0\ldots n}\in S_{IOI} \mid \\
&\quad p=p_0, ni=ni_0, (t,p_0,ni_0))) \in A_{TPI}\langle G\rangle \wedge ((c_n,ni_n),d)\in A_{I\mathbf{D}}\langle\gamma(c_n)\rangle\}\} \\
E3_{\mathbf{D}} =&\{(t,d)\mapsto E \mid E = \sum_{(t,c,no)} E_{TO}\langle\gamma(c_0)\rangle(t,(c,no)) * \\
&\#\{(p_0,c_0,no_0,p_k,c_k,ni_k,no_k)_{k=0\ldots n}\in S_{OIO} \mid \\
&\quad c=c_0, no=no_0, (t,(c_0,no_0))) \in A_{TO}\langle\gamma(c_0)\rangle \wedge (p_n,no_n,d)\in A_{PO\mathbf{D}}\langle G\rangle\}\} \\
E4_{\mathbf{D}} =&\{(t,d)\mapsto E \mid E = \sum_{(t,c,no)} E_{TO}\langle\gamma(c_0)\rangle(t,(c,no) * \\
&\#\{(p_k,c_k,no_k,p'_k,c'_k,ni_k)_{k=1\ldots n}\in S_{OI} \mid \\
&\quad c=c_1, no=no_1, (t,(c_0,no_0))) \in A_{TO}\langle\gamma(c_0)\rangle \wedge ((c_n,ni_n),d)\in A_{I\mathbf{D}}\langle\gamma(c_n)\rangle\}\}
\end{aligned}
$$

- Now the new arcs expressions look like this:

$$
\begin{aligned}
E_{TP} &= E_{TP}\langle G\rangle + E1_P + E2_P + E3_P + E4_P \\
E_{TO} &= E_{TO}\langle G\rangle + E1_O + E3_O
\end{aligned}
$$

- Finally, the guard expressions are defined as follows:

$$G : (c,t)\mapsto G(t)\langle C_c\rangle$$

Now we are ready to make the final step and transform the top level Petri net component into a simple Petri net. In terms of section 1.2.1 we must make the transition from Fig. 3a to Fig. 3b. First we define our notion of Petri net, however.

**Definition 2.18 (Petri net).** A *Petri net* $\Pi$ is a tuple defined as follows:

$$
\begin{aligned}
&\Pi = (P, T, A, E) \qquad \text{with} \\
&\quad P, T \quad \text{some sets} \\
&\quad A = (A_{PT}, A_{TP}) \qquad \text{with} \\
&\qquad A_{PT} \subseteq P \times T \\
&\qquad A_{TP} \subseteq T \times P \\
&\quad E = (E_{PT}, E_{TP}) \qquad \text{with} \\
&\qquad E_{PT} : A_{PT} \to \mathcal{E}^* \\
&\qquad E_{TP} : A_{TP} \to \mathcal{E}^* \\
&\quad G : T \to \mathcal{E}
\end{aligned}
$$

**Definition 2.19 (Petri net operator $\pi$).** Given a component $\Gamma$, the Petri net operator $\pi$ computes a Petri net $\Pi = \pi(G)$ by simply removing some parts of the ground structure as follows:

$$
\begin{aligned}
P &= P\langle\Gamma\rangle \\
T &= T\langle\Gamma\rangle \\
A_{PT} &= A_{PT}\langle\Gamma\rangle \\
A_{TP} &= A_{TP}\langle\Gamma\rangle \\
E_{PT} &= E_{PT}\langle\Gamma\rangle \\
E_{TP} &= E_{TP}\langle\Gamma\rangle \\
G &= G\langle\Gamma\rangle
\end{aligned}
$$

## 2.6 State transition

We can now describe the concept of state transition for a system of compositional Petri net structures. Since this description will essentially be based on transformation to a Petri net as in Def. 2.18 and subsequently state transition in that Petri net, we will first recapitulate our notion of Petri net state transitions.

### 2.6.1 ... in a Petri net

State transition as defined here is based on the definitions for Colored Petri Nets in [7]. We have simplified the net a little (arcs are simple and we have no types), but the general flavor remains the same.

First we define some notation to describe the neighborhood of a transition.

**Definition 2.20 (Preset, postset).** Given a Petri net as in Def. 2.18, we define for each transition $t \in T$ its preset $\bullet t$ and its postset $t\bullet$ as follows:

$$
\begin{aligned}
\bullet t &= \{p \in P \mid (p, t) \in A_{PT}\} \\
t\bullet &= \{p \in P \mid (t, p) \in A_{TP}\}
\end{aligned}
$$

Since this will be a key notion for binding and enabling, we have to define the free variables of a transition.

**Definition 2.21 (Variables of a transition).** Given a Petri net as in Def. 2.18. Then for each transition $t \in T$ we define the set of free variables of the transitions expressions as

$$
Var(t) = free(G(t)) \cup \bigcup_{p \in \bullet t} free[E_{PT}(p, t)] \cup \bigcup_{p \in t\bullet} free[E_{TP}(t, p)]
$$

Now we can define the valid bindings of those variables with respect to the transition guard:

**Definition 2.22 (Binding element).** A *binding element* $(t, b)$ is a binding of the variables in $Var(t)$ such that $G(t)[b]$ evaluates to *true*. The set of all binding elements is written as $\mathcal{B}$.

A step is simply a set of these bindings. It is enabled, if there are sufficiently many tokens on the respective places for all the binding elements in a step:

**Definition 2.23 (Step, enabled step).** A *step* $Y$ is a multiset in $\mathcal{B}$. The set of all steps is call $\mathbb{Y}$.

A step $Y$ is *enabled* in marking $\mathcal{M}$, iff

$$\forall p \in \mathcal{P} : \sum_{(t,b)\in Y} E(p,t)[b] \subseteq \mathcal{M}(p)$$

Finally, we can say how a state is transformed by the occurrence of a step:

**Definition 2.24 (Occurrence).** An enabled step $Y$ in marking $\mathcal{M}$ may *occur*, yielding a new marking $\mathcal{M}'$ which is defined as follows:

$$\mathcal{M}'(p) = \mathcal{M}(p) - \sum_{(t,b)\in Y} E(p,t)[b] + \sum_{(t,b)\in Y} E(t,p)[b]$$

We say that $\mathcal{M}'$ is *directly reachable* from $\mathcal{M}$ by the (occurrence of) step $Y$, symbolically:

$$\mathcal{M}[Y\rangle\mathcal{M}'$$

The remaining theory (reachability, occurrence sequences, etc.) is of course identical to the conventional notions, which is why we skip them here.

### 2.6.2 ... in a system of compositional Petri net structures

When executing a system of compositional Petri net structures, we start with an initial state $(\mathcal{M}_{init}, \mathcal{B}_{init})$ with $\mathcal{B}_0(c) = \perp$ (i.e. undefined) for all $b \in \overline{\mathcal{C}}$.

Execution of a system needs an anchor point, a basic component that substitution and eventually state transition starts from. We create this initial component (in a sense our 'top level component') with the $\chi$ operator from Def. 2.14, choosing a initial structure $C_{init} \in \mathcal{C}$ and a corresponding binding $b_{init} : V\langle C_{init}\rangle \to \mathcal{U}$.[4]

Now, we initialize the system as follows: $(c_{top}, (\mathcal{M}_0, \mathcal{B}_0)) = \chi(C_{init}, b_{init}, (\mathcal{M}_{init}, \mathcal{B}_{init}))$. This provides us with a top level instance $c_{top}$ and an initial state $(\mathcal{M}_0, \mathcal{B}_0)$.

**Definition 2.25 (State transition in COPS).** Given a top level component $c_{top}$ and a state $(\mathcal{M}, \mathcal{B})$, we say that a state $(\mathcal{M}', \mathcal{B}')$ is *directly reachable* from the former, written as

$$(\mathcal{M}, \mathcal{B}) \xrightarrow{1} (\mathcal{M}', \mathcal{B}')$$

iff

$$\Pi = \pi(\sigma(\gamma(c_{top}, \mathcal{B}), (\mathcal{M}, \mathcal{B})))$$

and $\mathcal{M}'$ is directly reachable from $\mathcal{M}$ in $\Pi$ and evaluation of the expressions in the corresponding step change $\mathcal{B}$ to $\mathcal{B}'$.

## 2.7 Some non-trivial cases

The highly dynamic nature of the ground network topology creates a few situations that deserve special consideration. Some of them actually represent cases of flexible and advanced use, while others are leading to erroneous and undefined behavior. We will now discuss some of them.

[4] Choosing a binding of the variables of the structure amounts to parametrizing the model.

1. **Multiple references to the same component.**

   The designator of a component may be contained in more than one place. This is perfectly legal and (as such) induces no problem to the substitution procedure. A useful interpretation could be that a component is hooked up to different contexts, and in fact it could be connected to the different contexts in different ways.

2. **Multiple refs from same place.**

   The designator of a component could reside more than once on the same place. This is also a legal situation, it simply means that inputs to and outputs from the component are multiplied by the number of occurrences of the designator in the place.

3. **Circular references.**

   A designator might reside on one of the places of its associated ground structure (or any of the places that are substituted into it during the substitution step). This can lead to erroneous situations because the circularity may result in infinitely many ways of reaching a place from a transition or vice versa. In this case, a net is not finitely substitutable and an error should be flagged.[5]

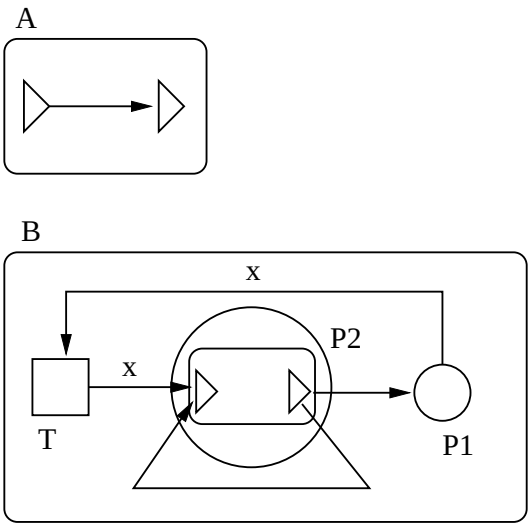


Figure 14: Looping arcs.

4. **Looping arcs.**

   Consider the two structures in Fig. 14. Let us create an $A$ component $a$, and a $B$ component $b$, and put $a$ on the place $(b, P2)$.[6] Now consider the Petri net that results from this system (with $b$ as our top level component).

   Obviously, there are infinitely many paths between $(b, T)$ and $(b, P1)$, i.e. the arc expression $x$ would occur infinitely often in the replaced arc. This situation leads therefore to an infinite inscription and thus an error must be flagged.

# 3 Further enhancements

This section discusses possible ways to extend the basic model presented in this work and gives some rationale why certain aspects have not been dealt with here.

[5] Note, however, that – provided there are always finitely many components – there are always only finitely many places and transitions in the substitution. The infinity potentially results from path multiplicity which results in infinitely large factors in the expression sets that are the arc inscriptions.

[6] This could, e.g., happen by virtue of the initialization expression for $P2$, which is not shown here.

### 3.1 Types

One obvious omission as compared to Colored Petri Nets in [7] are types (called *color sets* in CPN). Except for the part of the universe that serves as designators for components, we do not assume any structure of the value system. The reason is that we felt that the behavioral description of this system should be orthogonal to such issues (similar to the precise nature of the expression language), and that therer are a number of useful type systems that could be added to a system of compositional Petri net structures.

One obvious candidate is certainly some object-oriented system, with inheritance and polymorphism. Not only could this be used to structure the value space, it could also be employed to statically check the network structure: COPS would thus become component classes, their I/O signature would be their public interface, and the type of a place could be used to infer the interfaces that components residing on the place are required to provide.

Various notions of inheritance between component classes are possible, from pure interface towards more behavioral inheritance. Of course, all the usual design problems for type systems in programming languages apply here as well: What should be done about multiple inheritance, parametric polymorphism, how should the inheritance graph look etc.

### 3.2 Time

A notion of time is an important requirement whenever performance issues are involved, but it can of course also affect the functional behavior of a Petri net. Temporal extensions of Petri nets have been discussed in several places (e.g. [11, 10, 7, 3]), so we will deal here only with the special aspects of compositional structures with respect to concepts of time.

Time is usually added to Petri nets by giving places or transitions delays, which may be numbers, intervals, or arbitrary distributions. In the case of components, one might want to control their time axis from the outside, thus specifying the speed that the components runs at relative to its environment. This could either be done only once at instantiation of a component or dynamically at each transition that affects the component. The latter comes into conceptual difficulties when components are running is different places with different contradictory time axis scales.

### 3.3 Runtime execution control

An atemporal cousin of the dynamic skewing of the time axis of a component is controlling *if* a component may make state transitions in a given place, i.e. staring and stopping a component. This would be rather straightforward to define, with two substitution rules for two different kinds of places: the first would substitute as it is done now, the second would just substitute the places (in case we want still inputs to be sent to such a stopped component) or just skip substitution altogether.

### 3.4 Static places

A straighforward extension would be the addition of places which would not be replicated for each instance of a structure, but instead exist only exactly once (similar to fusion places in CPN [7]). Adding this as a special rule t the substitution mechanism is straightforward.

## 4 Future work

The proposed model of compositionality in Petri nets opens several areas for further work. Among them are these:

- Tool support. Clearly, having tools that support COPS for analysis, transformation, execution, and code generation is very important because a key design goal for this approach to compositionality was to enhance the feasibility of Petri net modeles for practical applications. Work on this is done in the context of the **Moses** project, where an object-oriented version of COPS forms the basis of the Petri net formalism that is used.

- Analysis techniques. The strong decoupling of Petri net components makes COPS particularly amenable to compositional analysis techniques. Parametric and recursive construction such as in the case discussed in 1.2.3 make it possible to use inductive proof techniques for some proofs.
- Relation to other hierarchy models in Petri nets. The relations to and possible transformations into other models of hierarchy for Petri nets deserves further investigation. In this way, analysis techniques that have been developed for those apporaches could possibly be used in the COPS context.
- Extensions of the basic model. Extending the model in ways such as have been discussed in section 3 could make it more useful for particular application areas.
- Integration of other models of computation. The strong decoupling of COPS components make it rather straightforward to mix Petri net components with components whose behavior has been specified in another model of computation. While then the semantics of complete model might not be given by substitution to a simple Petri net model, some analysis and proof techniques that rely on the behavioral specification of components might still work.

# References


[1] Wilfried Brauer, Robert Gold, and Walter Vogler. A survey of behaviour and equivalence preserving refinements of Petri nets. In *Advances in Petri Nets*.

[2] D. Buchs and N. Guelfi. CO-OPN: A concurrent object-oriented Petri net approach. In *Proceedings of the 12th International Conference on the Application and Theory of Petri Nets*, 1991.

[3] Robert Esser. *An Object Oriented Petri Net Approach to Embedded System Design*. PhD thesis, ETH Zurich, 1996.

[4] H. J. Genrich and K. Lautenbach. *System Modelling with High-Level Petri Nets*, volume 13 of *Theoretical Computer Science*, pages 109–136. North-Holland, 1981.

[5] ITU-T. Video coding for low bit rate communication. ITU-T Recommendation H.263, March 1996.

[6] Kurt Jensen. Coloured Petri nets: A high level language for system design and analysis. In G. Rozenberg, editor, *Advances in Petri Nets*, volume 483 of *Lecture Notes of Computer Science*, 1990.

[7] Kurt Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use*, volume 1: Basic Concepts of *EATCS Monographs in Computer Science*. Springer-Verlag, 1992.

[8] Charles A. Lakos. Object Petri nets - definition and relationship to coloured nets. Technical Report TR94-3, Computer Science Department, University of Tasmania, 1994.

[9] Charles A. Lakos. From coloured Petri nets to object Petri nets. In *Proceedings of the 15th International Conference on Applications and Theory of Petri Nets*, Lecture Notes of Computer Science, 1995.

[10] P. Merlin and D. J. Faber. Recoverability of communication protocols. *IEEE Transactions on Communications*, 24(9), September 1976.

[11] C. Ramchandani. Analysis of asynchronous concurrent systems by timed petri nets. Technical Report 120, Project MAC, Massachussetts Institute of Technology, February 1974.

[12] Rüdiger Valk. On processes of object Petri nets. Technical Report 185, Fachbereich Informatik, Universität Hamburg, 1996.