Versatile Boxes: a Multi-Purpose Algebra of High-Level Petri nets

Franck Pommereau

▶ To cite this version:

Franck Pommereau. Versatile Boxes: a Multi-Purpose Algebra of High-Level Petri nets. Summer Computer Simulation Conference, 2007, San Diego, United States. hal-02309961

HAL Id: hal-02309961
https://hal.archives-ouvertes.fr/hal-02309961

Submitted on 10 Oct 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Versatile Boxes: a Multi-Purpose Algebra of High-Level Petri nets

Franck Pommereau
LACL, Université Paris 12
61 avenue du général de Gaulle, 94010 Créteil, France
pommereau@univ-paris12.fr

Keywords: algebra of Petri nets, threads, exceptions

Abstract

This paper introduces a model of composable Petri nets, called *Versatile Boxes*, that has all the high-level features already introduced in the family of the Petri Box Calculus (mainly: data and time representation) as well as a new interruption capability. This allows for defining processes that are able to interrupt their execution at any point, just as a program can raise an exception. By choosing a carefully tuned level of generality, we are able to obtain a model that is much simpler than previous approaches, while still allowing to give the semantics of the usual programming language constructs. We believe that our model has the required characteristics for a very general use, hence its name. On the other hand, its is not very complex and so it should be easy to understand or implement and efficient for verification or simulation purpose.

1. INTRODUCTION

The *Petri Box Calculus* (PBC) and its successor PNA [1, 2] are two classes of Petri nets that can be composed like terms in process algebra. Various high-level variants of them have been introduced. The most noticeable one is the model of *M-nets* [6] that can be seen as a high-level version of the Petri net part of PBC. It allows to handle data values directly as tokens and to apply high-level operations on them. However, M-nets lack a syntactic level that allows for building large complicated nets without the need to draw them. (Actually, a syntax exists but is at least as complex has the net model itself [8].) This problem has been addressed through several successive extensions of PBC by introducing classes of high-level Petri nets provided with a user-friendly syntax. In particular: the *Asynchronous Box Calculus* [3] introduces buffered communication; the *Box Calculus with Data* [4] allows for high-level data in the model; the *Causal Time Calculus* [9] addresses timed systems. The model presented in this paper, called *Versatile Boxes* (*v-boxes* for short), will cover all those aspects in a unified presentation and will provide various improvements while keeping the model simple to implement.

The main aim of the present paper is to push further this evolution by introducing an exception-like feature. The goal is to be able to define processes whose execution can be interrupted at any point. This question has been already addressed for M-nets (but not at syntactical level), leading to a very general preemption mechanism [7]. However, it has been necessary to introduce priorities between Petri net transitions in order to be able to preempt an M-net in presence of nested parallelism. The resulting model is an exponential abbreviation of regular Petri nets; moreover, it is not usable with most existing software tools. In this paper, we use a simpler setting inspired from what exists in most programming languages: parallelism is restricted to well identified threads of execution (or tasks), each one being a sequential process. Moreover, we do not consider a general preemption mechanism, where any process can preempt any other, but rather an interrupt mechanism by which a process can terminate itself. This is also what happens in most programming languages where a thread can be stopped only by itself.

As a simple motivating example, let us just consider the *break* or *exit* statement used in many programming languages to stop a loop at any point (we could similarly consider interruptions or the *return* instruction). This commonly used statement cannot be modelled by any of the previous models, except using that from [7] but at the price explained above. We show in the conclusion how this can be modelled in a simple way using v-boxes.

The sequel is organised as follows: we first introduce basic definitions, in particular that of the class of labelled Petri nets used throughout the paper; then, we define the operators that allow to compose these nets; finally, we introduce the syntactic level and its Petri net semantics. As a conclusion, we will summarise the new or improved features of v-boxes and present future directions of work. In order to focus on the new aspects, some technical aspects will be simplified.

2. BASIC DEFINITIONS

A *multiset* over a set X is a function $\mu : X \to \mathbb{N}$. We shall use the following operations on multisets: comparison $\mu \leq \mu'$, element inclusion $x \in \mu$ and sum $\mu + \mu'$. A subset of X may be treated as a multiset over X, by identifying it with its characteristic function. A finite multiset may

be written in extended set notation, *e.g.*, $\{a,a,b\}$. We denote by $\mathbb{N}^X$ the set of all the finite multisets over X.

2.1. Distinguished sets

We consider several sets used in the following; except when specified, they are assumed pairwise disjoint.

$\mathbb{D}$ is the set of *data values*, it includes in particular the integers, the booleans $\top$ and $\bot$, and the Petri net tokens $\bullet$, $\circ$ and $\star$. $\mathbb{D}$ also holds the integer-like value ω that is greater than any integer, and such that $\omega+1 \stackrel{\text{df}}{=} \omega$ and $\omega-1 \stackrel{\text{df}}{=} \omega$. This set of values corresponds to the data stored in a Petri net, $\bullet$ and $\circ$ being specifically used to model the control flow while $\star$ is involved in net compositions. $\mathbb{D}$ is assumed closed under tuple aggregation, that is, $\mathbb{D}^i \subset \mathbb{D}$ for all $i \geq 1$, allowing to model structured data.

A distinguished subset of $\mathbb{D}$ is $\mathbb{C} \stackrel{\text{df}}{=} \mathbb{N} \uplus \{\omega\}$ that is the set of *clock counts*, corresponding to the value associated to a counter of clock ticks. We shall use increasing such counters to measure the passing of time as well as decreasing ones to model timeouts.

$\mathbb{V}$ is the set of *variables*, that are denoted by x, y, x_1, ... In order to allow for renaming variables at will, $\mathbb{V}$ is assumed infinite. These variables will be used to label Petri nets, for instance in order to represent values carried by arcs.

$\mathbb{I} \stackrel{\text{df}}{=} \mathbb{I}_t \uplus \mathbb{I}_s \uplus \mathbb{I}_p \uplus \{\varepsilon\}$ is the set of *identifiers* that are names given to nodes. ε is the *anonymous* identifier. Transitions are named from $\mathbb{I}_t \uplus \{\varepsilon\}$ and places from $\mathbb{I}_s \uplus \{\varepsilon\}$. $\mathbb{I}_p$ is involved in processes naming; for all $\mathsf{name} \in \mathbb{I}_p$ we assume that $\mathsf{name_enter}$ and $\mathsf{name_exit}$ also belong to $\mathbb{I}_p$. We also assume that for all $\mathsf{name} \in \mathbb{I}_s$, there exists a corresponding $name \in \mathbb{V}$, with a mapping var such that $var(\mathsf{name}) \stackrel{\text{df}}{=} name$.

$\mathbb{S} \stackrel{\text{df}}{=} \{\mathsf{e},\mathsf{x},\mathsf{i},\mathsf{b},\mathsf{w},\mathsf{t},\mathsf{v}\}$ is the set of *place statuses*. In a Petri net, they are used to distinguish respectively the entry, exit, internal, buffer, watch, timeout and variable places. The entry, exit and internal places are used to model the control flow of a Petri net and are so called *control flow places*. The others are called *data places* as they store all the data manipulated by the Petri net. More precisely, buffer places will hold multisets of values that can be added or retrieved from buffers. Watch places will model counters of clock ticks used to measure the passing of time. Timeout places will be downward ticks counters used to model timeouts. Last, variable places will hold a single value (but possibly structured) as variables in a program.

$\mathbb{P} \subset \mathbb{D}$ is the set of *process identifiers*, that allows to distinguish several concurrent threads of execution in a Petri net. $\mathbb{P}$ can be assumed finite or infinite depending on the execution policy that one wants to use.

2.2. Labelled Petri nets

A *labelled Petri net* is a tuple $N \stackrel{\text{df}}{=} (S,T,\ell)$ where S is a finite set of *places*, T is a finite set of *transitions* such that $S \cap T = \emptyset$, and ℓ is the *labelling* function on places, transitions and *arcs*, *i.e.*, elements from $(S \times T) \uplus (T \times S)$. This labelling is defined as follows:

- for a place $s \in S$, $\ell(s) \stackrel{\text{df}}{=} \sigma(s)\tau(s)\eta(s)$ where $\sigma(s) \in \mathbb{S}$ is the *status* of s, $\tau(s) \subseteq \mathbb{D}$ is the *type* of s and $\eta(s) \in \mathbb{I}_s \uplus \mathbb{I}_p \uplus \{\varepsilon\}$ is the *name* of s;
- for a transition $t \in T$, $\ell(t) \stackrel{\text{df}}{=} \gamma(t)\eta(t)$ where $\eta(t) \in \mathbb{I}_t \uplus \{\varepsilon\}$ is the *name* of t and $\gamma(t)$ is an evaluable boolean expression, called the *guard* of t (a condition for its execution);
- for an input arc $(s,t) \in S \times T$, $\ell(s,t)$ is a finite multiset over $\mathbb{D} \uplus \mathbb{V}$, representing the values consumed in s when t is executed;
- for an output arc (t,s), $\ell(t,s)$ is a finite multiset of values, variables or evaluable expressions representing the values produced in s when t is executed. The difference with input arcs is that computation of new values is allowed by using expressions.

The usual graphical representation for Petri nets is to have circles for the places, squares for the transitions and directed links for the arcs. We shall omit arcs with empty labels $\emptyset$, brackets $\{\}$, true guards $\top$, $\{\bullet,\circ\}$ place types and $\{\bullet\}$ arc labels. Finally, place statuses will be inscribed inside circular labels on the border of the places.

A *marking* of a Petri net $N \stackrel{\text{df}}{=} (S,T,\ell)$ is a mapping M that associates to each place $s \in S$ a finite multiset $M(s)$ over $\tau(s)$ that represents the tokens held by s. The execution of a labelled Petri net is expected to start from a marking with one token in each entry place and to evolve until it reaches a marking with one token in each exit place (if ever).

3. LABELLED NETS OPERATORS

This section introduces various operators on the labelled Petri nets defined above, allowing to build complex systems from simpler blocks.

Let N_1 and N_2 be two nets, we will define four binary *control flow operators* $\fatsemi$, $\Box$, $\circledast$ and $\triangleright$ with the following intuitive semantics:

- the *sequence* $N_1 \fatsemi N_2$ allows to execute N_1, followed by N_2;
- the *choice* $N_1 \Box N_2$ allows to execute either N_1 or N_2;
- the *iteration* $N_1 \circledast N_2$ allows any number of executions of N_1, followed by one execution of N_2;
- the *trap* $N_1 \triangleright N_2$ allows to execute N_1 then, only if N_1 finishes abnormally, N_2 is executed, *i.e.*, N_2 can trap an error occurring in N_1 (but not outside).

The notion of normal or abnormal termination of a net will correspond to have respectively a token $\bullet$ or $\circ$ in its exit places. Producing a token $\circ$ corresponds to raising an exception in a program: the regular flow of operations is interrupted and the exception may be caught by a handler (a trap in our framework).

The *name hiding* N/name makes anonymous the name-labelled nodes in N. This operator is used to disable the name-based node merges that take place when two nets are combined. For instance, in such a case, the variable places of a given name that come from each net have to be merged in order to ensure that each variable is modelled by only one place. By using the name hiding operator, one can disable further merges of such a place, which results in making a variable local to a sub-net.

The *task declaration* operator $(\!|\mathsf{name}:N|\!)$ turns a net N into a task called name. This allows to start several concurrent executions of N; they are basically independent but may interact through shared data places.

Finally, the *parallel composition* $N_1 \parallel N_2$ allows N_1 and N_2 to evolve concurrently. This operator will never be nested inside another one.

In order to define correctly those operators, we assume that the labelled Petri nets considered as arguments have the following properties (but we may build nets that do not respect these properties):

- R1: exactly one entry place and one exit place;
- R2: control flow places have the type $\{\bullet, \circ\}$;
- R3: control flow places as well as the transitions connected to them are anonymous.

3.1. Operators nets

The four control flow operators will be defined using *operator nets*. They are particular labelled Petri nets used to guide the construction of the net resulting from the application of each operator. Intuitively, an operator net defines the structure of the result that is built by replacing dedicated transitions of the operator net by the operand nets.

Figure 1 shows the four operator nets corresponding to our four control flow operators. In order to perform an operation, each transition $\sharp i$ has to be replaced by the i-th operand net. The labels on the arcs indicate what tokens are allowed: $\bullet$, $\circ$, or any (with $\star$).

Consider for instance the top-left net, that specifies the sequence operator, and take $N_1 \fatsemi N_2$ the sequential composition of N_1 and N_2. $N_{\fatsemi}$ is started by putting one token in its entry place, if it is $\bullet$, transition $\sharp 1$ and thus net N_1 is executed. When it terminates, it produces a token in the internal place. Again, if this is $\bullet$, $\sharp 2$ and thus N_2 is executed until it puts a token in the exit place. But, if N_1 terminates with a token $\circ$, transition $t_2^{\fatsemi}$ is executed

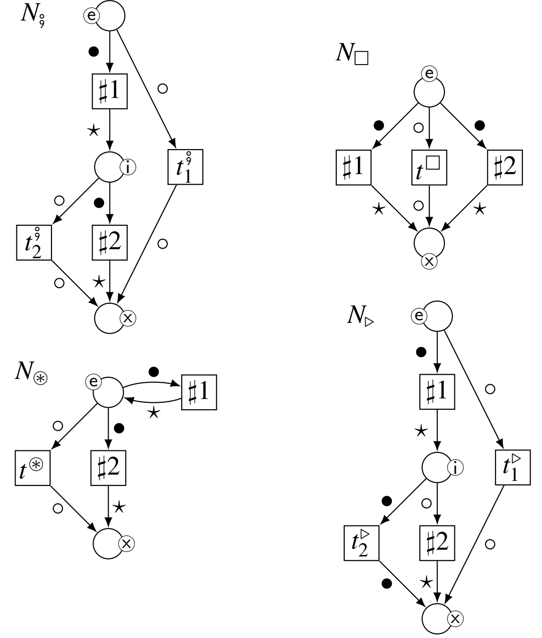

Figure 1. The four operator nets. All the places are anonymous and have the type $\{\bullet, \circ\}$; all the transitions have true guards and are anonymous too.

instead of $\sharp 2$, which results in skipping N_2 because we have to propagate the error until it is trapped. Similarly, if the net is started with a $\circ$ in the entry place, this token is directly propagated to the exit place through transition $t_1^{\fatsemi}$. The situation is similar for the choice and iteration operators. Consider now the trap operator net $N_{\triangleright}$, it has the same structure as $N_{\fatsemi}$ but the tokens on the arcs differ: here, if N_1 terminates with a $\bullet$, this token is directly propagated to the exit place; but if N_1 produces a $\circ$, this token allows to execute $\sharp 2$ and thus N_2, which is the expected behaviour of the trap. Notice that the arcs going out of transitions $\sharp i$ allow any token to be produced as the result of each N_i.

In order to know how the labelling of an arc in an operator net influences the labellings of the arcs in the resulting nets, we define a function θ that maps a pair of arc labels (in operator and operand net) to the label that will be used in the resulting net. This function is defined by the array given in figure 2.

Consider for instance transition $\sharp 1$ in operator net $N_{\triangleright}$ and assume an operand net N_1 that has an entry places e with outgoing arcs to two transitions:

$$e \xrightarrow{\{\bullet\}} t_1 \quad \text{and} \quad e \xrightarrow{\{\circ\}} t_2 \ .$$

When N_1 replaces $\sharp 1$, its place e becomes the entry place of the resulting net that will also include t_1 and t_2. But the operator net allows only $\{\bullet\}$ to enter $\sharp 1$ and thus N_1.

operator label		$\{\bullet\}$	$\{\circ\}$	$\{\star\}$	$\emptyset$
operand label	$\{\bullet\}$	$\{\bullet\}$	$\{\circ\}$	$\{\bullet\}$	$\emptyset$
	$\{\circ\}$	$\{\star\}$	$\{\star\}$	$\{\circ\}$	$\emptyset$
	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
	m	$\{\star\}$	$\{\star\}$	m	$\emptyset$

Figure 2. The array that defines θ, where m is a multiset over $\mathbb{D}$ that is neither $\{\bullet\}$, $\{\circ\}$ nor $\emptyset$.

This is enforced by θ that defines the labels of the arcs outgoing from e in the resulting net as:

$$e \xrightarrow{\theta(\{\bullet\},\{\bullet\})} t_1 \;\Rightarrow\; e \xrightarrow{\{\bullet\}} t_1 \;,$$
$$e \xrightarrow{\theta(\{\bullet\},\{\circ\})} t_2 \;\Rightarrow\; e \xrightarrow{\{\star\}} t_2 \;.$$

So the arc to t_1 is preserved while the arc to t_2 is relabelled with $\{\star\}$. As a result, t_2 becomes a dead transition because $\star$ is not in the type of control flow places (by restriction R2). So, tokens $\circ$ introduced in e will go through transition $t_1^{\triangleright}$ (coming from $N_{\triangleright}$) in the resulting net, which is the expected behaviour. Because any token is allowed to go out of $\sharp1$, the arcs to the exit place of N_1 will not be changed (see the $\{\star\}$ column of the array that defines θ). Consider now transition $\sharp2$, substituted with the same net N_1; we have:

$$e \xrightarrow{\theta(\{\circ\},\{\bullet\})} t_1 \;\Rightarrow\; e \xrightarrow{\{\circ\}} t_1 \;,$$
$$e \xrightarrow{\theta(\{\circ\},\{\circ\})} t_2 \;\Rightarrow\; e \xrightarrow{\{\star\}} t_2 \;.$$

So, the precise meaning of the arc to $\sharp2$ is actually "take one token $\circ$ in the internal place and start the net that replaces $\sharp2$". Token $\circ$ is thus directly consumed by t_1 which corresponds to the regular start of N_1.

3.2. Control flow operators

Let $N_1 \stackrel{\text{df}}{=} (S_1, T_1, \ell_1)$ and $N_2 \stackrel{\text{df}}{=} (S_2, T_2, \ell_2)$ marked respectively by M_1 and M_2. Take $\diamond \in \{\fatsemi, \Box, \circledast, \triangleright\}$ and operator net $N_\diamond \stackrel{\text{df}}{=} (S_\diamond, T_\diamond, \ell_\diamond)$. We assume that S_1, S_2, T_1, T_2, $S_\diamond$ and $T_\diamond$ are pairwise disjoint (nodes in N_1 or N_2 may be renamed if needed). The composition $N_1 \diamond N_2$ is $N \stackrel{\text{df}}{=} (S, T, \ell)$ marked by M defined as follows.

First, the nodes from N_1 and N_2 are copied to N, together with the arcs connecting them. So, for $1 \le i \le 2$, for all $s \in S_i$ and all $t \in T_i$:

- s is also a place in S with $\ell(s) \stackrel{\text{df}}{=} \ell_i(s)$ and $M(s) \stackrel{\text{df}}{=} M_i(s)$;
- t is also a transition in T with $\ell(t) \stackrel{\text{df}}{=} \ell_i(t)$;
- $\ell(s,t) \stackrel{\text{df}}{=} \ell_i(s,t)$ and $\ell(t,s) \stackrel{\text{df}}{=} \ell_i(t,s)$.

The transitions of the operator net that will not be replaced by a N_i are also copied to the result: for all $t \in T_\diamond \setminus \{\sharp1, \sharp2\}$, we also have $t \in T$ with $\ell(t) \stackrel{\text{df}}{=} \ell_\diamond(t)$.

Then, the non-anonymous transitions are merged. For all $\mathsf{tick} \in \mathbb{I}_t$, all the transitions $t_1, \ldots, t_n$ in T such that $\eta(t_i) = \mathsf{tick}$ (for $1 \le i \le n$) are merged, yielding t' such that $\gamma(t') \stackrel{\text{df}}{=} \gamma(t_1) \wedge \cdots \wedge \gamma(t_n)$ and $\eta(t') \stackrel{\text{df}}{=} \mathsf{tick}$. The t_i's ($1 \le i \le n$) are then removed from T.

Moreover, the non-anonymous data places are merged. For all $\mathsf{name} \in \mathbb{I}_s$ and all $a \in \{\mathsf{b}, \mathsf{w}, \mathsf{t}, \mathsf{v}\}$, all the places $s_1, \ldots, s_m$ in S such that $\eta(s_i) = \mathsf{name}$ and $\sigma(s_i) = a$ (for $1 \le i \le m$) are merged, yielding s' such that $\sigma(s') \stackrel{\text{df}}{=} a$, $\eta(s') \stackrel{\text{df}}{=} \mathsf{name}$ and $\tau(s') \stackrel{\text{df}}{=} \tau(s_1) \cup \cdots \cup \tau(s_m)$. The marking of s' is dependent on the status of the merged places: if $a = \mathsf{b}$, then $M(s') \stackrel{\text{df}}{=} M(s_1) + \cdots + M(s_m)$ as a buffer place can accumulate tokens; otherwise, the composition is only allowed if all the places have the same marking. This marking becomes that of s'. The s_i's ($1 \le i \le m$) are then removed from S.

Finally, entry and exit places are combined in order to enforce the desired control flow. To do so, we rely on operator net $N_\diamond$: we consider in turn each place $s_\diamond$ of $N_\diamond$ and collect in a set P places originated from each N_i. The places in P should be merged in order to become the realisation of $s_\diamond$ in N. So, for $1 \le i \le 2$:

- if $\ell_\diamond(s_\diamond, \sharp i) \neq \emptyset$ then the entry place e_i of N_i is added to P;
- if $\ell_\diamond(\sharp i, s_\diamond) \neq \emptyset$ then the exit place x_i of N_i is added to P.

Notice that e_i and x_i exist because of restriction R1. The places in P are merged, yielding a new place $s'_\diamond$ whose label is that of $s_\diamond$. The composition is allowed only if all the places in P have the same marking. This marking becomes that of $s'_\diamond$. The places in P are then removed from S. The last operation related to $s'_\diamond$ is to use θ in order to adjust the labelling of the arcs between $s'_\diamond$ and the transitions originated from each N_i; moreover, $s'_\diamond$ has to be connected to the transitions originated from $N_\diamond$:

- for $1 \le i \le 2$, for all $t \in T_i$ that is connected to the entry e_i or exit place x_i of N_i
  - $\ell(s'_\diamond, t) \stackrel{\text{df}}{=} \theta(\ell_\diamond(s_\diamond, \sharp i), \ell_i(e_i, t))$,
  - $\ell(t, s'_\diamond) \stackrel{\text{df}}{=} \theta(\ell_\diamond(\sharp i, s_\diamond), \ell_i(t, x_i))$;
- for all $t \in T_\diamond \setminus \{\sharp1, \sharp2\}$, $\ell(s'_\diamond, t) \stackrel{\text{df}}{=} \ell_\diamond(s_\diamond, t)$ and $\ell(t, s'_\diamond) \stackrel{\text{df}}{=} \ell_\diamond(t, s_\diamond)$.

Optionally, the transitions in T having an attached arc labelled by $\{\star\}$ being dead, they can be removed.

3.3. Task declaration

Let $N \stackrel{\text{df}}{=} (S, T, \ell)$ be a labelled net and $\mathsf{task} \in \mathbb{I}_p$ be a task identifier, the *task declaration* $(\!|\mathsf{task} : N|\!)$ is a net that is able to concurrently execute instances of N. These instances are identified by a value in $\mathbb{P}$ so that they can run independently. This could not be obtained by just

putting several tokens in the entry place of N because they would be all identical. Instead, (almost) every place $s \in S$ will be given the type $\mathbb{P} \times \tau(s)$ in order to associate each token to a process identifier. The arcs connected to these places will be changed accordingly. However, we do not modify the non-anonymous places that may be merged later on if the task is composed with other nets. These places have to be considered as external shared resources that are not involved in the construction of the task.

When a net is turned into a task, its entry and exit places are changed into buffer places with the type $\mathbb{P} \times \{\bullet, \circ\}$ (because of restriction R2, $\{\bullet, \circ\}$ is the type of these places). So, starting an instance of the task can be achieved by putting in the entry-like place a pair $(p, \bullet)$ where p is a process identifier. The process identifier is carried along the net using a fresh variable $\pi \in \mathbb{V}$, *i.e.*, that is not used in any annotation of N. When the exit-like place becomes marked, the process identifier can be retrieved there, together with the termination status of the corresponding thread ($\bullet$ or $\circ$). In order to find easily the new entry-like and exit-like places, they are named respectively $\mathsf{task_enter}$ and $\mathsf{task_exit}$, where task is the identifier used in the task declaration. Notice that it is possible to start a thread with a token $(p, \circ)$ that may be trapped inside N. Moreover, the user is let responsible for consistently choosing process identifiers.

If N is marked by M, the net $N' \stackrel{\text{df}}{=} (\!|\mathsf{task} : N|\!) \stackrel{\text{df}}{=} (S', T', \ell')$ and its marking M' are defined as follows:

- each transition $t \in T$ is also a transition in T' with $\ell'(t) \stackrel{\text{df}}{=} \ell(t)$;
- each place $s \in S$ such that $\eta(s) \neq \varepsilon$ is also a place in S' with $\ell'(s) \stackrel{\text{df}}{=} \ell(s)$ and $M'(s) \stackrel{\text{df}}{=} M(s)$; moreover, for all $t \in T$, $\ell'(s,t) \stackrel{\text{df}}{=} \ell(s,t)$ and $\ell'(t,s) \stackrel{\text{df}}{=} \ell(t,s)$;
- for each place $s \in S$ such that $\eta(s) = \varepsilon$, there is a place s' in S' with $M'(s') \stackrel{\text{df}}{=} \{(p,v) \mid p \in \mathbb{P}, v \in M(s)\}$ and
  - $\ell'(s') \stackrel{\text{df}}{=} (\mathbb{P} \times \tau(s))\ \mathsf{b}\ \mathsf{task_enter}$, if $\sigma(s) = \mathsf{e}$,
  - $\ell'(s') \stackrel{\text{df}}{=} (\mathbb{P} \times \tau(s))\ \mathsf{b}\ \mathsf{task_exit}$, if $\sigma(s) = \mathsf{x}$,
  - $\ell'(s') \stackrel{\text{df}}{=} (\mathbb{P} \times \tau(s))\ \sigma(s)\ \varepsilon$, otherwise;

  moreover, for all $t \in T$, $\ell'(s',t) \stackrel{\text{df}}{=} \{(\pi, \alpha) \mid \alpha \in \ell(s,t)\}$ and $\ell'(t,s') \stackrel{\text{df}}{=} \{(\pi, \alpha) \mid \alpha \in \ell(t,s)\}$.

It may be remarked that the task declaration produces a net that does not respect restrictions R1 to R3.

3.4. Name hiding, parallel composition

Both operators can work on any labelled Petri net that do not need to respect restrictions R1 to R3.

The net N/name is a copy of N in which the nodes labelled by name are given the anonymous name ε. The name hiding is commutative (and idempotent) and so naturally extends to sets of names.

The parallel composition $N_1 \| N_2$ is defined as a simplified control flow operation: first, the nodes and arcs from both nets are copied, and second, non-anonymous nodes with the same name are merged, exactly as for control flow operators.

4. VERSATILE BOXES

This section introduces a syntax whose semantics is given in terms of a function *box* that constructs a labelled Petri net when given an expression. This syntax can be considered as a kind of assembly language having just the features required to express the constructs usually found in programming languages.

4.1. Syntax

A *box expression* is a term on the syntax given in figure 3. The most basic process is an *atomic action* that corresponds to a test and set, for instance $\langle x > 4 \Rightarrow x := 0 \rangle$ is an atomic action that tests the value of x to be greater than 4 and resets it to 0. Such an atomic action is blocked until its condition becomes true. An *interrupt action* is a variant of an atomic action that can be considered as the raising of an exception: when it is executed, the sequel of the process is by-passed until the exception is trapped. Conditions and assignments in actions are related to *declarations*, inside a *scope* or as a global *resource*. Each declaration in a scope may introduce a *variable*, a *buffer*, a *watch* or a *timeout* with the expected meaning. Moreover, a *clock* may be declared in order to provide the ticks counted by watches and timeouts; each clock is independent from the others. Resources can declare tasks, global data or clocks (scopes declare local objects). Processes can then be combined using operators that correspond to those defined at Petri net level.

4.2. Lexical scoping

Declarations are lexically scoped: an identifier can be used only if it is the name of a previously declared object. Moreover, the reuse of an already declared identifier results in its hiding in favour of the newly declared one. In order to cope with lexical scoping, we will use *environments* allowing to remember the declared objects while computing the semantics of nested processes. Formally, an environment E is a mapping from a subset of $\mathbb{I} \setminus \{\varepsilon\}$ to $(\{\mathsf{v}, \mathsf{b}\} \times 2^{\mathbb{D}} \times \mathbb{N}^{\mathbb{D}}) \uplus (\{\mathsf{t}, \mathsf{w}\} \times \mathbb{I}_t \times \mathbb{C}) \uplus \mathbb{I}_t$. We denote by $\mathsf{name} \in E$ the fact that name is in the domain of E, by $\emptyset$ the environment with empty domain and by $+$ the combination of environments defined by: if $\mathsf{name} \in E_2$ then $(E_1 + E_2)(\mathsf{name}) \stackrel{\text{df}}{=} E_2(\mathsf{name})$, else if $\mathsf{name} \in E_1$ then $(E_1 + E_2)(\mathsf{name}) \stackrel{\text{df}}{=} E_1(\mathsf{name})$, else $\mathsf{name} \notin E_1 + E_2$. Intuitively, $E(\mathsf{name})$ corresponds to how

P	$::=$	S	sequential process
	$\mid$	$R \parallel P$	process with a resource
R	$::=$	$(\mid \mathsf{task} : S \mid)$	task resource
	$\mid$	D	data resource
S	$::=$	$S \mathbin{;} S$	sequence
	$\mid$	$S \circledast S$	iteration
	$\mid$	$S \Box S$	choice
	$\mid$	$S \triangleright S$	trap
	$\mid$	$[\![D \mid S]\!]$	scope
	$\mid$	$\langle B \Rightarrow A \rangle$	atomic action
	$\mid$	$\langle\langle B \Rightarrow A \rangle\rangle$	interrupt action
D	$::=$	$\underline{\mathsf{variable}}\ \mathsf{name} : V := v$	variable declaration
	$\mid$	$\underline{\mathsf{buffer}}\ \mathsf{name} : V := Z$	buffer declaration
	$\mid$	$\underline{\mathsf{watch}}\ \mathsf{name} : \mathsf{tick} := c$	watch declaration
	$\mid$	$\underline{\mathsf{timeout}}\ \mathsf{name} : \mathsf{tick} := c$	timeout declaration
	$\mid$	$\underline{\mathsf{clock}}\ \mathsf{tick}$	clock declaration
	$\mid$	D, D	multiple declarations
B	$::=$	*"boolean expression"*	condition
A	$::=$	$\mathsf{name} :=$ *"expression"*	assignment
	$\mid$	A, A	multiple assignments
	$\mid$	$\emptyset$	no assignment

Figure 3. The syntax of box expressions, where $\mathsf{name} \in \mathbb{I}_s$, $\mathsf{task} \in \mathbb{I}_p$, $\mathsf{tick} \in \mathbb{I}_t$, $V \subseteq \mathbb{D}$, $v \in V$, $Z \in \mathbb{N}^V$, $c \in \mathbb{C}$ and underlined words are keywords. Both *"boolean expression"* and *"expression"* are evaluable expressions.

name was last declared: a variable or a buffer on a given set with an initial value corresponds to a triple from $\{\mathsf{v}, \mathsf{b}\} \times 2^{\mathbb{D}} \times \mathbb{N}^{\mathbb{D}}$; a watch or a timeout corresponds to a triple from $\{\mathsf{w}, \mathsf{t}\} \times \mathbb{I}_t \times \mathbb{C}$; a clock corresponds to its name in $\mathbb{I}_t$.

4.3. Denotational semantics

The denotational semantics of a box expression is defined by the function *box* that takes as arguments an expression and its environment and yields a labelled Petri net. This function is defined recursively on the syntax. The semantics of a process P is the net obtained from $box(P, \emptyset)$ in which one token $\bullet$ is added to each entry place (there will be actually only one such place).

4.3.1. Control flow operators

The semantics of a syntactical control flow operator is simply the application of the corresponding operator at Petri net level. So, for $\diamond \in \{;, \circledast, \Box, \triangleright\}$, we have:

$$box(S_1 \diamond S_2, E) \stackrel{\mathrm{df}}{=} box(S_1, E) \diamond box(S_2, E)$$

4.3.2. Scope

The semantics of a scope consists essentially in enriching the environment with the new declarations and hiding the corresponding names in the Petri net obtained from the process part of the scope:

$$box([\![D \mid S]\!], E) \stackrel{\mathrm{df}}{=} \big(init(D) \mathbin{;} box(S, E + env(D)) \mathbin{;} term(D)\big) \\ / \{\mathsf{name} \mid \mathsf{name} \in env(D)\}$$

where function $env(D)$ is defined as:

$$\begin{aligned}
env(\underline{\mathsf{variable}}\ \mathsf{name} : V := v) &\stackrel{\mathrm{df}}{=} (\mathsf{name} \mapsto (\mathsf{v}, V, \{v\})) \\
env(\underline{\mathsf{buffer}}\ \mathsf{name} : V := Z) &\stackrel{\mathrm{df}}{=} (\mathsf{name} \mapsto (\mathsf{b}, V, Z)) \\
env(\underline{\mathsf{watch}}\ \mathsf{name} : \mathsf{tick} := c) &\stackrel{\mathrm{df}}{=} (\mathsf{name} \mapsto (\mathsf{w}, \mathsf{tick}, c)) \\
env(\underline{\mathsf{timeout}}\ \mathsf{name} : \mathsf{tick} := c) &\stackrel{\mathrm{df}}{=} (\mathsf{name} \mapsto (\mathsf{t}, \mathsf{tick}, c)) \\
env(\underline{\mathsf{clock}}\ \mathsf{tick}) &\stackrel{\mathrm{df}}{=} (\mathsf{tick} \mapsto \mathsf{tick}) \\
env(D_1, D_2) &\stackrel{\mathrm{df}}{=} env(D_1) + env(D_2)
\end{aligned}$$

The Petri nets $init(D)$ and $term(D)$ are intended to initialise and terminate the declared variables, watches and timeouts. Buffers are initialised directly at the level of the marking and they are not terminated as they are allowed to keep their values between two executions of the scope. The net $init(D)$ is defined as:

- it has a single unmarked entry place e_D such that $\ell(e_D) \stackrel{\mathrm{df}}{=} \mathsf{e}\{\bullet, \circ\}\varepsilon$;
- it has a single unmarked exit place x_D such that $\ell(x_D) \stackrel{\mathrm{df}}{=} \mathsf{x}\{\bullet, \circ\}\varepsilon$;
- it has a single transition t_D such that $\ell(t_D) \stackrel{\mathrm{df}}{=} \top\varepsilon$;
- there is one arc from e_D to t_D and one arc from t_D to x_D both labelled by $\{\bullet\}$;
- for each name declared in D such that $\mathsf{name} \in \mathbb{I}_s$ and $env(D)(\mathsf{name}) = (x, y, z)$, there is one place s_{name} defined as:
  - if $x \in \{\mathsf{v}, \mathsf{b}\}$ then $\ell(s_{\mathsf{name}}) \stackrel{\mathrm{df}}{=} x\, y\, \mathsf{name}$, otherwise $\ell(s_{\mathsf{name}}) \stackrel{\mathrm{df}}{=} x\, \mathbb{C}\, \mathsf{name}$,
  - if $x = \mathsf{b}$ then s_{name} is marked by z, else if $x \in \{\mathsf{w}, \mathsf{t}\}$ then s_{name} is marked by $\{\omega\}$, else it is unmarked,
  - if $x \neq \mathsf{b}$ then there is an arc from t_D to s_{name} labelled by z,
  - if $x \in \{\mathsf{w}, \mathsf{t}\}$ there is also an arc from s_{name} to t_D labelled by $\{\omega\}$.

The net $term(D)$ is based on $init(D)$ with the difference that the arcs attached to the places s_{name} are directed the other way and labelled with variables in order to restore the initial markings. More precisely: there is an arc from each s_{name} to t_D labelled by $\{var(\mathsf{name})\}$ if $x \neq \mathsf{b}$ and by $\emptyset$ otherwise; if $x \in \{\mathsf{w}, \mathsf{t}\}$ there is also an arc from t_D to s_{name} labelled by $\{\omega\}$.

4.3.3. Resources

The semantics of a process with a data or task resource is defined as:

$$box((\!|\mathsf{task}:S|\!) \parallel P, E) \stackrel{\text{df}}{=} (\!|\mathsf{task}:box(S,E)|\!) \parallel box(P, E+env((\!|\mathsf{task}:S|\!)))$$
$$box(D \parallel P, E) \stackrel{\text{df}}{=} box(D,\emptyset) \parallel box(P, E+env(D))$$

where $env(D)$ has been defined above and:

$$env((\!|\mathsf{task}:S|\!)) \stackrel{\text{df}}{=} \begin{pmatrix} \mathsf{task_enter} \mapsto (\mathsf{b}, \mathbb{P}\times\{\bullet,\circ\},\emptyset), \\ \mathsf{task_exit} \mapsto (\mathsf{b}, \mathbb{P}\times\{\bullet,\circ\},\emptyset) \end{pmatrix}$$

Moreover, the nets for data resources are as follows:

- $box(\underline{\text{variable}}\ \mathsf{name}:V:=v,\emptyset)$ is composed of a single place labelled by $\mathsf{v}\,V\,\mathsf{name}$ and marked by $\{v\}$;
- $box(\underline{\text{buffer}}\ \mathsf{name}:V:=Z,\emptyset)$ is composed of a single place labelled by $\mathsf{b}\,V\,\mathsf{name}$ and marked by Z;
- $box(\underline{\text{watch}}\ \mathsf{name}:\mathsf{tick}:=c,\emptyset)$, resp. $box(\underline{\text{timeout}}\ \mathsf{name}:\mathsf{tick}:=c,\emptyset)$, is composed of a single place labelled by $\mathsf{w}\,\mathbb{C}\,\mathsf{name}$, resp. by $\mathsf{t}\,\mathbb{C}\,\mathsf{name}$, and marked by $\{c\}$;
- $box(\underline{\text{clock}}\ \mathsf{tick},\emptyset)$ is an empty Petri net;
- $box((D_1,D_2),\emptyset) \stackrel{\text{df}}{=} box(D_1,\emptyset) \parallel box(D_2,\emptyset)$.

As defined in the previous section, the net operation $(\!|\mathsf{task}:N|\!)$ involves a cross-product of $\mathbb{P}$ with the existing marking of N. This potential explosion can be avoided at syntactic level by never nesting a watch, timeout or non-empty buffer declaration inside a task resource (the resulting net in this case has an empty marking).

4.3.4. Atomic action

The semantics of $\langle B \Rightarrow A\rangle$ is mainly one anonymous transition guarded by B and surrounded by the data places corresponding to the current declarations. These places are connected to the transition in order to read the values needed by both the guard and the assignments and to write these values back or assign new ones appropriately. A buffer is treated differently: consulting its value consumes one of its tokens while assigning it adds the result of the assigned expression to the buffer. Finally, non-anonymous transitions are added to model the needed clock ticks.

More precisely, $box(\langle B \Rightarrow A\rangle, E)$ is a labelled Petri net $N \stackrel{\text{df}}{=} (S,T,\ell)$ marked by M such that:

- there is a single entry place e in S such that $\ell(e) \stackrel{\text{df}}{=} \mathsf{e}\{\bullet,\circ\}\varepsilon$ and $M(e) \stackrel{\text{df}}{=} \emptyset$;
- there is a single exit place x in S such that $\ell(x) \stackrel{\text{df}}{=} \mathsf{x}\{\bullet,\circ\}\varepsilon$ and $M(x) \stackrel{\text{df}}{=} \emptyset$;
- there is a transition t in T such that $\ell(t) \stackrel{\text{df}}{=} B\varepsilon$;
- $\ell(e,t) \stackrel{\text{df}}{=} \{\bullet\}$ and $\ell(t,x) \stackrel{\text{df}}{=} \{\bullet\}$;
- for all $\mathsf{name} \in \mathbb{I}_s$ such that $\mathsf{name} \in E$, there is a place s_{name} in S such that:
  - if $E(\mathsf{name}) = (s,V,Z) \in \{\mathsf{v},\mathsf{b}\} \times 2^{\mathbb{D}} \times \mathbb{N}^{\mathbb{D}}$ then $\ell(s_{\mathsf{name}}) \stackrel{\text{df}}{=} s\,V\,\mathsf{name}$ and $M(s_{\mathsf{name}}) \stackrel{\text{df}}{=} \emptyset$,
  - if $E(\mathsf{name}) = (s,\mathsf{name},c) \in \{\mathsf{w},\mathsf{t}\} \times \mathbb{I}_t \times \mathbb{C}$ then $\ell(s_{\mathsf{name}}) \stackrel{\text{df}}{=} s\,\mathbb{C}\,\mathsf{name}$ and $M(s_{\mathsf{name}}) \stackrel{\text{df}}{=} \{\omega\}$;
- for all $\mathsf{tick} \in \mathbb{I}_t$ such that $\mathsf{tick} \in E$, there is a transition t_{tick} in T such that $\ell(t_{\mathsf{tick}}) \stackrel{\text{df}}{=} \top\,\mathsf{tick}$ and, for all place s_{tick} added to S at the previous step (as s_{name} with $\mathsf{name} = \mathsf{tick}$):
  - if $E(\mathsf{tick}) = (\mathsf{w},\mathsf{tick},v)$ then $\ell(s_{\mathsf{tick}},t_{\mathsf{tick}}) \stackrel{\text{df}}{=} \{var(\mathsf{tick})\}$ and $\ell(t_{\mathsf{tick}},\ s_{\mathsf{tick}}) \stackrel{\text{df}}{=} \{var(\mathsf{tick})+1\}$,
  - if $E(\mathsf{tick}) = (\mathsf{t},\mathsf{tick},v)$ then $\ell(s_{\mathsf{tick}},t_{\mathsf{tick}}) \stackrel{\text{df}}{=} \{var(\mathsf{tick})\}$ and $\ell(t_{\mathsf{tick}},s_{\mathsf{tick}}) \stackrel{\text{df}}{=} \{var(\mathsf{tick})-1\}$;
- for all $\mathsf{name} \in \mathbb{I}\setminus\{\varepsilon\}$ involved in A through an assignment "$var(\mathsf{name}) := expr$" and such that $E(\mathsf{name}) \neq (\mathsf{b},V,Z)$:
  - if $\mathsf{name} \in \mathbb{I}_t$ then box fails as a clock can be used only through timeouts or watches,
  - if $\mathsf{name} \notin E$ then box fails as name is used without being declared,
  - otherwise, $\ell(s_{\mathsf{name}},t) \stackrel{\text{df}}{=} \{var(\mathsf{name})\}$ and $\ell(t,s_{\mathsf{name}}) \stackrel{\text{df}}{=} \{expr\}$;
- for all other $\mathsf{name} \in \mathbb{I}\setminus\{\varepsilon\}$ involved in A or in B and such that $E(\mathsf{name}) \neq (\mathsf{b},V,Z)$:
  - if $E(\mathsf{name}) \in \mathbb{I}_t$ or $\mathsf{name} \notin E$ then box fails,
  - otherwise, $\ell(s_{\mathsf{name}},t) \stackrel{\text{df}}{=} \{var(\mathsf{name})\}$ and, $\ell(t,s_{\mathsf{name}}) \stackrel{\text{df}}{=} \{var(\mathsf{name})\}$;
- for all name involved in A or B such that if $E(\mathsf{name}) = (\mathsf{b},V,Z)$:
  - if name appears as "$var(\mathsf{name}) := expr$" then $\ell(t,s_{\mathsf{name}}) \stackrel{\text{df}}{=} \{expr\}$,
  - if $E(\mathsf{name})$ appears in B or in the right side of an assignment then $\ell(s_{\mathsf{name}},t) \stackrel{\text{df}}{=} \{var(\mathsf{name})\}$;
- arcs not defined above are labelled by $\emptyset$.

One can easily check that restrictions R1 to R3 are respected.

4.3.5. Interrupt action

$box(\langle\!\langle B \Rightarrow A\rangle\!\rangle, E)$ is built like $box(\langle B \Rightarrow A\rangle, E)$ except that the arc from transition t to the exit place is labelled by $\{\circ\}$ instead of $\{\bullet\}$.

5. CONCLUSION

We presented a new model of composable high-level Petri nets provided with a syntax that we called *Versatile Boxes* or *v-boxes*. This model unifies and improves various existing evolutions of the Petri Box Calculus.

(1) It features buffers as introduced in [3, 4]. Moreover, v-boxes distinguish multi-valued buffers from one-valued

variables, the latter being more traditional in programming languages.

(2) As in [9], v-boxes are able to model timed systems. However, this new version separates watches from timeouts. It allows to remove the requirement in [9] that watches must be bounded in order to model timeouts. Moreover, v-boxes can have multiple clock ticks defined locally while [9] only allowed for one global clock.

(3) The parallel composition cannot be nested in v-boxes while this is possible in the other models. This allows a definition of a novel interrupt/trap mechanism much more efficient than the approach proposed in [7]. Moreover, this actually does not restrict the expressive power but only enforces a structured programming policy. Indeed the effect of $N_1 \| N_2$ can be simulated by embedding N_1 into a task declaration and enclosing the execution of N_2 in between the spawning of one instance of N_1 and the waiting for the termination of this instance.

(4) Unlike the other models, v-boxes do not propose a transition synchronisation operator. That would be absolutely no problem to have it as it could be directly borrowed from [4] or [9]. We simply felt no need for introducing it. Indeed, it has been possible to define the semantics of all the usual programming constructs without the help of a transition synchronisation. The simpler name-based transition merging scheme introduced in this paper is actually sufficient to ensure the required merges of tick transitions.

Theoretical properties and practical applications of the model are currently being investigated.

(1) Our first studies show that v-boxes basically have all the properties usually required, in particular: clean and safe markings under any evolution (see [3]). By the way, another advantage of removing nested parallelism is that we obtain 1-boundedness of control flow places, instead of 2-boundedness plus auto-concurrency free transitions as found in [3].

(2) Moreover, a prototype implementation is in progress and turns out to be not more difficult to realise than a previous prototype for CTC [10], despite the generalisation and the new features.

(3) Finally, we have already successfully translated to v-boxes significant constructs of the Ada programming language: exceptions, loops with multiple exits, tasks and sub-programs. For instance, an exit (*i.e.*, break) statement can be modelled using an interrupt action with a trap after the loop. A simple example is given in figure 4. In general, there might be other reasons for executing an interrupt action, in particular the execution of a return statement or the occurrence of an exception. These cases can however be distinguished in the right part of each trap operation by consulting a global variable that is set when each interrupt action is executed.

1 **while** $x < 100$ **loop**
2 $x := x + 2;$
3 **exit when** $x = 3;$
4 $x := x + 1;$
5 **end loop**;

$$\left(\begin{pmatrix} \langle x < 100 \Rightarrow \emptyset \rangle \,; \\ \langle \top \Rightarrow x := x+2 \rangle \,; \\ (\langle\langle x = 3 \Rightarrow \emptyset \rangle\rangle \\ \quad \square\, \langle x \neq 3 \Rightarrow \emptyset \rangle) \,; \\ \langle \top \Rightarrow x := x+1 \rangle \end{pmatrix} \circledast \langle x \geq 100 \Rightarrow \emptyset \rangle \right) \triangleright \langle \top \Rightarrow \emptyset \rangle$$

Figure 4. An Ada loop with an exit statement and the corresponding v-box expression.

Future works will address the mechanisms of object-oriented languages by adapting the approach proposed in [5]. Another direction will be to give a SOS semantics to the syntactic level of v-boxes and to prove its consistency with the Petri nets semantics.

REFERENCES

[1] E. Best, R. Devillers and J. Hall. *The Petri Box Calculus: a New Causal Algebra with Multilabel Communication.* APN'92, LNCS 609, Springer, 1992.

[2] E. Best, R. Devillers and M. Koutny. *Petri Net Algebra.* EATCS Monographs on TCS, Springer, 2001.

[3] R. Devillers, H. Klaudel, M. Koutny and F. Pommereau. *Asynchonous Box Calculus.* Fundamenta Informaticae 54(1), IOS Press, 2003.

[4] C. Bui Thanh, H. Klaudel and F. Pommereau. *Box Calculus with Coloured Buffers.* LACL TR, 2002. ⟨http://www.univ-paris12.fr/lacl⟩

[5] C. Bui Thanh. *Modèles Orientés-Objet pour la Vérification de Systèmes Concurrents.* PhD. Thesis, Univ. Paris 12, 2004.

[6] H. Klaudel. *Compositional high-level Petri net semantics of a parallel programming language with procedures.* SCP 41, Elsevier, 2001.

[7] H. Klaudel and F. Pommereau. *A class of composable and preemptible high-level Petri nets with an application to multi-tasking systems.* Fundamenta Informaticae 50(1), IOS Press, 2002.

[8] H. Klaudel and R.-C. Riemann. *High Level Expressions with their SOS Semantics.* CONCUR'97, LNCS 1243, Springer, 1997.

[9] F. Pommereau. *Causal Time Calculus.* FORMATS'03, LNCS 2791, Springer, 2004.

[10] Franck Pommereau. *SNAKES is the Net Algebra Kit for Editors and Simulators.* Comete Procope Workshop, 2004. ⟨http://www.univ-paris12.fr/lacl/pommereau/comete/⟩