A concurrent semantics of static exceptions in a parallel programming language

Hanna Klaudel, Franck Pommereau

Proc. of ICATPN'01, LNCS 2075, pages 204–223, Springer, 2001.

A Concurrent Semantics of Static Exceptions in a Parallel Programming Language

Hanna Klaudel and Franck Pommereau

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

Abstract. This paper aims at introducing a mechanism of exceptions in a parallel programming language, giving them a formal concurrent semantics in terms of preemptible and composable high-level Petri nets. We show that, combined with concurrency, exceptions can be used as a basis for other preemption related constructs. We illustrate this idea by presenting a generalized timeout and a simple UNIX-like system of concurrent preemptible threads.

Keywords. Exceptions, Petri nets, semantics, parallel programming.

1 Introduction

The starting point of our approach is *B(PN)*^{2} [3, 9] (*Basic Petri Net Programming Notation*) which is a high-level programming language comprising in a simple syntax most traditional concepts of parallel programming. It includes nested parallel composition, iteration, guarded commands, procedures and communications *via* both handshake and buffered communication channels, as well as shared variables. One of the most interesting aspects of B(PN)^2 is its simplicity: it features most classical concepts in a simple syntax. So, it becomes possible to use it as a test language and then to extend or apply the results found for B(PN)^2 to "real-life" languages.

B(PN)^2 has an original formal semantics in terms of *boxes* [1], a class of labelled Petri nets provided with a set of composition operations, and *M-nets* [2], a high-level version of boxes. M-nets are strongly related to boxes by an *unfolding* of M-nets into boxes and allow to represent in a clear and compact way large (possibly infinite) systems. B(PN)^2, boxes and M-nets are implemented in PEP toolkit [7], allowing to simulate a modeled system and also to verify its properties *via* model checking.

Recent works [11] led to the definition of the model of P/M-nets which extends M-nets with preemption, introducing for this purpose a new operator, π. Given a net N, $\pi(N)$ is a net which can be aborted, *i.e.*, it's termination can be forced immediately. Despite this augmented capability, it is proved in [11] that P/M-nets stay strictly equivalent to M-nets in terms of computational power (both may be transformed into 1-safe Petri nets, but P/M-nets lead to much

bigger nets) and have also a concurrent semantics. Having preemption naturally leads to consider enhancing $\mathrm{B(PN)}^2$ with related constructs. This paper proposes a modeling of static exceptions in $\mathrm{B(PN)}^2$, giving their semantics with P/M-nets.

The presented approach allows to propagate exceptions through a nested block structure. However, the *resolution* procedure proposed here is one of the simplest possible : the actually handled exception is choosen arbitrarily between exceptions occurring concurrently. On the top of this system, a more sophisticated resolution system could be introduced, as proposed for instance in [15, 16, 17].

We also show that combining exceptions with parallelism allows to express other constructs like a generalized timeout and a simple multi-threaded system.

2 M-nets, P/M-nets and their Algebras

This section is devoted to introduce *P/M-nets*, high-level composable and preemptible Petri nets [11], which are used as semantic model for exceptions. P/M-nets are an extension of a high-level net model, called M-nets, which are introduced first. P/M-nets (as well as M-nets) may be considered as an efficient abreviation for safe place/transition Petri nets, into which they may be unfolded [2, 11].

2.1 Basic Definitions

Let E be a set. A *multi-set* over E is a function $\mu : E \to \mathbb{N}$, generally denoted with an extended set notation, *e.g.*, $\{a, a, b\}$ for $\mu(a) = 2$, $\mu(b) = 1$ and $\mu(e) = 0$ for all $e \in E \setminus \{a, b\}$. A multi-set μ is finite if so is its support set $E \setminus \mu^{-1}(0)$. We denote by $\mathcal{M}(E)$ (resp. $\mathcal{M}_f(E)$) the set of multi-sets (resp. finite multi-sets) over E, by $\oplus$ and $\ominus$ the sum and difference of multi-sets. We may also use the usual set notations, for example, if μ_1 and μ_2 are two multi-sets over E, $\mu_1 \subseteq \mu_2$ stands for $\forall x \in E : \mu_1(x) \leq \mu_2(x)$.

2.2 M-nets

M-nets (introduced in [2] and developed in [4, 10]) form a class of high-level Petri nets provided with a set of operations giving them a structure of process algebra.

An M-net N is a triple (S, T, ι), where S is the set of places, T is the set of transitions, $(T \times S) \cup (S \times T)$ is the set of arcs, and ι is the annotation function on places, transitions and arcs. The annotation of a place has the form $\lambda.\tau$, where λ is a *label* (entry e, exit x or internal i) and τ is a *type* (a non-empty set of values from a fixed set *Val*). As usual, for each node (place or transition) $r \in S \cup T$, we denote by ${}^\bullet r$ the set of nodes $\{r' \in S \cup T \mid \iota(r', r) \neq \emptyset\}$ and, similarly, $r^\bullet = \{r' \in S \cup T \mid \iota(r, r') \neq \emptyset\}$.

Transitions annotations are of the form $\lambda.\gamma$ where λ is a *label* (which can be hierarchical or for communications) and γ is a *guard* (a finite set of predicates

from a set Pr). Hierarchical labels are composed out of a single hierarchical action (*e.g.*, $\mathcal{X}$) indicating a future refinement (*i.e.*, a substitution) by an M-net. A transition may perform different kind of communications when it fires:

- *synchronous* ones, similar to CCS ones [13], *e.g.*, between transitions labelled by synchronous communication actions such as $A(a_1, \ldots, a_n)$ or $\widehat{A}(a'_1, \ldots, a'_n)$, where A is a *synchronous communication symbol*, $\widehat{A}$ is its *conjugate* and each a_i and a'_i is a value or a variable (belonging to a fixed set *Var*);
- *asynchronous* ones, *e.g.*, between transitions labelled by asynchronous links such as $b^+(a_1)$ or $b^-(a_2)$, where b is an *asynchronous communication symbol* and each a_i is a variable or a value ranging in $type(b) \subseteq Val$. The communication is done *via* a place s_b of type $\tau(s_b) = type(b)$ which plays the rôle of a heap buffer. Link $b^+(a_1)$ means that a_1 can be sent to s_b and $b^-(a_2)$ means that a_2 can be received from s_b;
- or possibly both types at the same time.

Communication labels are then of the form $\lambda = \alpha.\beta$ where α is a finite multi-set of synchronous communication actions and β is a finite multi-set of asynchronous links.

Arcs are incribed by annotations which encode the values consumed or produced in places by a firing of an adjacent transition. If no refinement is concerned, they are simply multi-sets of values or variables; otherwise they are constructed in a systematic way from the arc annotations coming from the refined and refining nets [4, 5].

2.3 Dynamic Behavior and Concurrent Semantics of M-nets

For each transition $t \in T$ we shall denote by $var(t)$ the set of all the variables occurring in the annotations of t and in the arcs coming to and from t. A *binding* for a transition t is a substitution $\sigma : var(t) \to Val$; it will be said *enabling* if it satisfies the guard, if it respects the types of the asynchronous links, and if the flow of tokens it implies respects the types of the places adjacent to t.

A *marking* of an M-net (S, T, ι) is a mapping $M\colon S \to \mathcal{M}(Val)$ which associates to each place $s \in S$ a multi-set of values from $\tau(s)$. In particular, we shall distinguish the *entry marking*, denoted M_{e}, where, for each $s \in S$, $M_{\mathsf{e}}(s) = \tau(s)$ if $\lambda(s) = \mathsf{e}$ and the empty multi-set otherwise; the *exit marking*, M_{x}, is defined similarly.

The transition rule specifies the circumstances under which a marking M' is reachable from a marking M. A transition t is *enabled* at a marking M, this is denoted $M[t\rangle$, if there is an enabling binding σ of t such that $\forall s \in S : \iota(s,t)[\sigma] \subseteq M(s)$, *i.e.*, there are enough tokens of each type to satisfy the required flow. The effect of an occurrence of t is to remove from its input places all the tokens used for the enabling binding σ and to add to its output places the tokens according to σ; this leads to a marking M' such that

$$\forall s \in S\colon \;\; M'(s) = M(s) \ominus \iota(s,t)[\sigma] \oplus \iota(t,s)[\sigma].$$

The above transition rule defines the *interleaving semantics* of an M-net which consists in a set of occurrence sequences. This semantics can be generalized by introducing the *step sequence semantics* [6], which allows any number of transitions to occur simultaneously.

2.4 Algebra of M-nets

For compositionality, we are particularly interested in a sub-class of M-nets: we assume that each M-net has at least one entry and one exit place, that each transition has at least one input and one output place (*T-restrictness* property), and that there are neither arcs going to entry places nor from exit places. Such M-nets are said ex-*good*.

The algebra of ex-good M-nets comprises the operations listed below, where N_1, N_2 and N_3 are M-nets, $\mathcal{X}$ is a hierarchical symbol, A is a synchronous communication symbol, b is an asynchronous link symbol and f is a renaming function on synchronous and asynchronous symbols.

$N_1[\mathcal{X} \leftarrow N_2]$	refinement	$N_1[f]$	renaming
$N_1 \| N_2$	parallel composition	$N_1 \,\mathbf{sy}\, A$	synchronization
$N_1; N_2$	sequence	$N_1 \,\mathbf{rs}\, A$	restriction
$N_1 \Box N_2$	choice	$[\![A : N_1]\!]$	scoping
$[N_1 * N_2 * N_3]$	iteration	$N_1 \,\mathbf{tie}\, b$	asynchronous links

The sequential composition "$N_1; N_2$" means that N_1 is executed first and then N_2. The parallel composition puts nets side by side without any link between them so they can execute in total concurrency. The choice composes nets in such a way that only one of them can be executed. The iteration composes three nets such that the first one is executed once as an initialization part, then the second one is executed an arbitrary number of times as a loop part, and finally the third one is executed once as an exit part. The synchronization w.r.t. a synchronous symbol A adds to a net new transitions anticipating all possible synchronous communications on A. The restriction w.r.t. A removes from the net all unsatisfied communication capabilities on A. The scoping w.r.t. A is defined as a synchronization w.r.t. A followed by a restriction w.r.t. A, it is used to setup all synchronous communications w.r.t. A, making them local to the net and no longer aviable for the other synchronizations. The asynchronous links operation w.r.t. b, applied to a net, adds a new buffer place s_b and arcs between transitions which export or import values, through b^+ or b^-, into or from the buffer, and removes all asynchronous link capabilities w.r.t. b from the inscriptions of the transitions. The refinement of the transitions labelled $\mathcal{X}$ in a net by another net is a kind of substitution which allows the refining net to be executed each time (for every enabling binding) the hierarchical transition in the refined net could fire. Renaming allows to change the names of synchronous or asynchronous communication symbols. Detailed explanations and some examples of these operations are given in [2, 5, 10].

2.5 Pairwise Priorities and Priority M-nets

Let $N = (S, T, \iota)$ be an M-net. A *pairwise priority relation* over T is a binary relation $\rho \subseteq T \times T$. Intuitively, $(t_1, t_2) \in \rho$ means that during an execution of N, the firing of transition t_2 is always preferred to the firing of t_1 when both are possible; in other words, t_1 has a lower priority than t_2. We use standard mathematical notations, in particular, for $\rho \subseteq T \times T$, we denote:

$$dom(\rho) = \{t_1 \in T \mid \exists t_2 \in T \text{ such that } (t_1, t_2) \in \rho\},$$
$$cod(\rho) = \{t_2 \in T \mid \exists t_1 \in T \text{ such that } (t_1, t_2) \in \rho\}.$$

A *priority M-net* is a pair $P = (N, \rho)$ where $N = (S, T, \iota)$ is an M-net (possibly having some non T-restricted communication transitions) and ρ is a pairwise priority relation over T. We call N the *net part* of P.

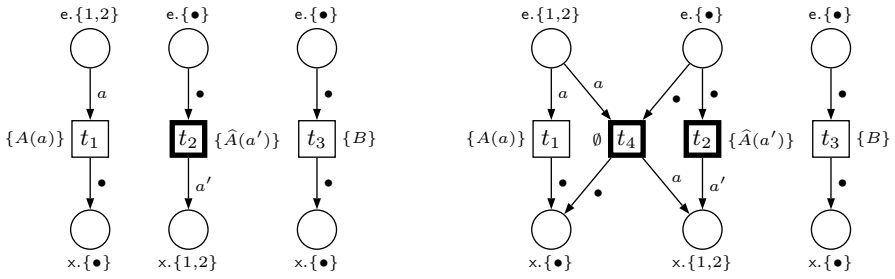
Definition 1. *Let $P = (N, \rho)$ be a priority M-net, M a marking of $N = (S, T, \iota)$ and t a transition of N such that $M[t\rangle$; then t is ρ-enabled in P at M, denoted $M[t\rangle_\rho$ iff $\nexists t' \in T$ such that $M[t'\rangle$ and $(t, t') \in \rho$.*

Notice that ρ allows to disable a transition which would have been enabled with the usual M-nets transition rule, but not the contrary. In other words, we have $M[t\rangle_\rho \Rightarrow M[t\rangle$. As for M-nets, this transition rule can be generalized in order to define the step semantics of priority M-nets [11].

An algebra of priority M-nets can also be considered. The extension of the usual M-net operations to priority M-nets is immediate for most of them. In order to make the paper self-contained, we recall here the definition from [11] which is an important for introducing the preemption operation π and preemptible M-nets.

Definition 2. *Let $P_i = (N_i, \rho_i)$, for $i \in \{1, 2, 3\}$, be priority M-nets, where $N_i = (S_i, T_i, \iota_i)$, and let $\mathcal{X}$ be a hierarchical symbol, A a synchronous communication symbol, b an asynchronous link symbol, and f a renaming function on communication symbols. The usual M-net operations are extended as follows for priority M-nets:*

- $P_1[\mathcal{X} \leftarrow P_2] = (N_1[\mathcal{X} \leftarrow N_2], \rho)$ *where*
  $\rho = \{(t, t') \in \rho_1 \mid \lambda_1(t) \neq \mathcal{X} \neq \lambda_1(t')\}$
  $\quad \uplus \{(t_\mathcal{X}.t, t_\mathcal{X}.t') \mid (t, t') \in \rho_2 \wedge t_\mathcal{X} \in T_1 \wedge \lambda_1(t_\mathcal{X}) = \mathcal{X}\}$
  $\quad \uplus \{(t_\mathcal{X}.t, t') \mid t \notin cod(\rho_2) \wedge (t_\mathcal{X}, t') \in \rho_1 \wedge t_\mathcal{X} \in T_1 \wedge \lambda_1(t_\mathcal{X}) = \mathcal{X}\}$;
- $P_1 \,\mathbf{tie}\, b = (N_1 \,\mathbf{tie}\, b, \rho_1)$;
- $P_1[f] = (N_1[f], \rho_1)$;
- $P_1 \,\mathbf{sy}\, A = (N_1 \,\mathbf{sy}\, A, \rho)$ *where* $N_1 \,\mathbf{sy}\, A = (S, T, \iota)$ *and ρ is the smallest set including ρ_1 such that if $t' \in T$ results from a basic synchronization of t_1 with t_2, and*
  - *if $\exists t''$ such that $(t_1, t'') \in \rho$ or $(t_2, t'') \in \rho$, then $(t', t'') \in \rho$,*
  - *if $\exists t''$ such that $(t'', t_1) \in \rho$ or $(t'', t_2) \in \rho$, then $(t'', t') \in \rho$.*
- $P_1 \,\mathbf{rs}\, A = (N_1 \,\mathbf{rs}\, A, \rho)$, *where* $N_1 \,\mathbf{rs}\, A = (S, T, \iota)$ *and* $\rho = \rho_1 \cap (T \times T)$.



$P = (N, \{(t_3, t_2)\})$ $\qquad$ $P \,\mathbf{sy}\, A = (N \,\mathbf{sy}\, A, \{(t_3, t_4), (t_3, t_2)\})$

Fig. 1. Example of synchronization of a priority M-net. (Only synchronous labels are represented.) Restricting on A would remove from the net transitions t_1 and t_2 (with their surrounding arcs) and pair (t_3, t_2) from the priority relation.

Control flow operators, as sequential composition, iteration, parallel composition and choice, are based on refinement and so they are defined canonically. Scoping, as for M-nets, is defined as a synchronization followed by a restriction: $[\![A : P]\!] = (P \,\mathbf{sy}\, A) \,\mathbf{rs}\, A$.

Figure 1 shows an example of synchronization in a priority M-net. Transitions t_1 and t_2 are synchronized, leading to a new transition t_4 which "inherits" from t_2 its priority over t_3. In figures, transitions with thick borders are those belonging to $cod(\rho)$ and thus have the priority over some other ones. This notation is used in all the sequel.

The definition of operation π is very important in our context because it allows to make abortable an arbitrary priority M-net $P = (N, \rho)$. The definition of π presented here is slightly different from the original one from [11], but the modification only concerns some labels involved in the semantics of exceptions. We use for it the priority M-net $P_\pi = (N_\pi, \rho_\pi)$ where N_π is represented in figure 2, the priority relation being

$$\rho_\pi = \{(t_8, t_2), (t_8, t_5), (t_7, t_2), (t_7, t_5), (t_\mathcal{X}, t_5), (t_3, t_4)\}.$$

In order to produce $\pi(P)$, P is embedded in P_π by refining transition $t_\mathcal{X}$ and the resulting net is synchronized w.r.t. *throw*. This way, if P does not throw any exception, it completes and so does $\pi(P)$ by firing transition t_0 (and no other transition in N_π can fire). However, in the case where P throws an exception, by firing a transition labelled with *throw*, transition t_1 fires too and enables the abortion of $\pi(P)$. This abortion is performed by consuming tokens in P through the loop on t_2 which is synchronized with the "emptying transitions", $t_s \in T_s$, added to P when π is applied. Transitions t_3 to t_7 are used to transmit abortion to all π's nested in P. (More detailed explanations of this mechanism can be found in [11].) Abortion is not elementary but, thanks to priorities, it is atomic in the sense defined in [12]: when started, abortion cannot be interrupted. When

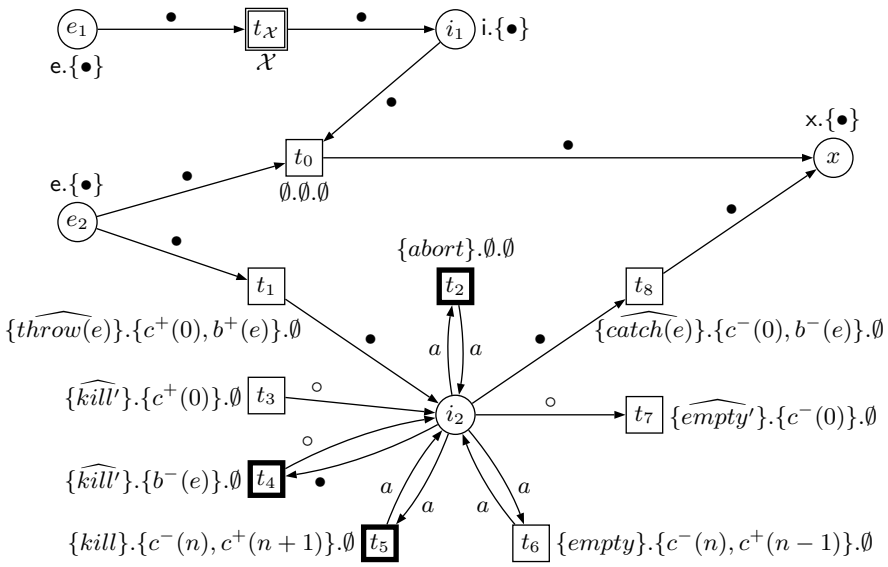


Fig. 2. N_π, net part of P_π where $type(b) = Val$, $type(c) = \mathbb{N}$ and $\iota(i_2) = \mathsf{i}.\{\bullet, \circ\}$.

abortion is terminated, transition t_8 fires, triggering the handler for the raised exception.

Definition 3. *Let P be a priority M-net. Then,*

$$\pi(P) = \Big[\!\Big[\{abort, throw, kill, empty\} : \ Ab\big(P_\pi[\mathcal{X} \leftarrow P]\big) \,\mathbf{tie}\{c, b\} \Big]\!\Big]_{[\widehat{kill'} \mapsto \widehat{kill}, \widehat{empty'} \mapsto \widehat{empty}]}$$

where P_π is the priority M-net defined above and Ab is an auxiliary operation which includes the additional emptying transitions; if $P_\pi[\mathcal{X} \leftarrow P] = P' = ((S', T', \iota'), \rho')$, then $Ab(P') = ((S'', T'', \iota''), \rho'')$ with:

- $S'' = S'$, *and* $\forall s \in S'' : \iota''(s) = \iota'(s)$;
- $T'' = T' \uplus T_s$ *where* $T_s = \{t_s \mid s \in S' \setminus \{x\} \wedge s^\bullet \cap cod(\rho') = \emptyset\}$
  and $\forall t \in T'' : \iota''(t) = \begin{cases} \iota'(t) & \text{if } t \in T', \\ \{\widehat{abort}\}.\emptyset.\emptyset & \text{if } t \in T_s; \end{cases}$
- $\forall (t, s) \in T'' \times S'' : \iota''(t, s) = \begin{cases} \iota'(t, s) & \text{if } t \in T', \\ \emptyset & \text{if } t \in T_s; \end{cases}$
- $\forall (s, t) \in S'' \times T'' : \iota''(s, t) = \begin{cases} \iota'(s, t) & \text{if } t \in T', \\ \{a\} \subset Var & \text{if } t = t_s \in T_s, \\ \emptyset & \text{if } t \in T_s \setminus \{t_s\}; \end{cases}$

– $\rho'' = \rho' \uplus \{(t, t_s) \mid t_s \in T_s \wedge t \in ({}^\bullet t_s)^\bullet\}$.

This mechanism is directly applied to the semantics of exceptions in a programming language. A net semantics P of (a part of) a concurrent program usually contains some $throw(e)$-labelled transitions. It means that if such a transition fires, P should not continue its normal behavior but should start an exceptional one. Operation π embeds P in such a way that the firing of such a transition in $\pi(P)$ is taken into account and brings about abortion of all the part corresponding to P in $\pi(P)$. This abortion is atomic (even if composed of several events) and when it is finished, a $catch(e)$-labelled transition can fire in $\pi(P)$. Also, $catch(e)$ is the only action related to exceptions and visible outside $\pi(P)$. It is used to trigger the handler for the thrown exception.

2.6 Preemptible M-nets: P/M-nets

Preemptible M-nets (*P/M-nets* for short) are defined as a sub-class of priority M-nets with some structural constraints. These constraints allow P/M-nets to have interesting properties, such as to be transformable into safe Petri nets. This sub-class is reasonably wide (it includes ex-good M-nets) and sound with respect to the semantics of preemption [11].

Definition 4. *Let $P = (N, \rho)$ be a priority M-net. P is a P/M-net iff either:*

- *N is an* ex*-good M-net and $\rho = \emptyset$, or*
- *P is defined as $\pi(P_1)$, $P_1[\mathcal{X} \leftarrow P_2]$, $P_1 \| P_2$, $P_1 ; P_2$, $P_1 \,\square\, P_2$, $[P_1 * P_2 * P_3]$, $P_1 \,\mathbf{sy}\, A$, $P_1 \,\mathbf{rs}\, A$, $[\![A : P_1]\!]$, $P_1 \,\mathbf{tie}\, b$ or $P_1[f]$, where P_i, for $i \in \{1, 2, 3\}$, are P/M-nets, $\mathcal{X}$ is a hierarchical symbol, A is a synchronous communication symbol, b is an asynchronous links symbol, and f is a renaming function on communication symbols.*

In the following, we often use some basic P/M-nets as that shown in figure 3. Their net parts are denoted by the label of their unique transition (*i.e.*, $\alpha.\beta.\gamma$ or $\mathcal{X}.\gamma$).

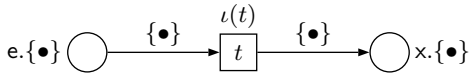


Fig. 3. A basic M-net used in this paper. Transition t may be either a communication transition in which case $\iota(t) = \alpha.\beta.\gamma$ or a hierarchical one with $\iota(t) = \mathcal{X}.\gamma$.

3 Syntax and Semantics of B(PN)²

B(PN)² is a parallel programming language comprising shared memory parallelism, channel (FIFO buffer) communication with arbitrary capacities, and allowing the nesting of parallel operators, blocks and procedures.

The following is a fragment of the syntax of B(PN)^2 (with keywords typeset in bold face, non-terminal in roman face and italic denoting values supplied by the program):

program ::= **program** block
block ::= **begin** scope **end**
scope ::= com | decl ; scope
com ::= ⟨expr⟩ | proc-call
| com ‖ com | com ; com | **do** alt-set **od**
| block | (com)
decl ::= **var** *name*: *set*
| **var** *name*: **chan** k **of** *set*
| **procedure** *name*(formal-parlist) block
| decl, decl
proc-call ::= *name*(effective-parlist)

An atomic action is a B(PN)^2 expression "⟨expr⟩", *i.e.*, a term constructed over operators, constants (here again, *Val* is the set of all possible values) and identifiers of *program variables* and *channels*. A program variable v can appear in an expression as ${}'v$ (pre-value) or v' (post-value), denoting respectively its value just before and just after the evaluation of the expression during an execution of the program. A channel variable c can appear in an expression as $c!$ (sending) or $c?$ (receiving), denoting respectively the value sent or received in a communication on the channel c. An atomic action can execute if the expression evaluates to true. Thus, for example, $\langle {}'v > 0 \wedge v' = c? \rangle$ corresponds to a guarded communication which requires v to be greater than zero and a communication to be available on channel c, in which case the value communicated on c is assigned to variable v.

A command "com" is either an atomic action, a procedure call ("proc-call"), one of a number of command compositions operator or a block comprising some declarations for a command. Parentheses allow to combine the various command compositions arbitrarily.

The domain of relevance of a variable, channel or procedure identifier is limited to the part of a B(PN)^2 program, called "scope", which follows its declaration. As usual, a declaration, in a new block, with an already used identifier results in the masking of the existing identifier by the new one. A procedure can be declared with or without parameters (in which case its "formal-parlist" is empty); each parameter can be passed by *value*, by *result* or by *reference*. A declaration of a program variable or a channel is made with the keyword "**var**" followed by an identifier with a type specification which can be "*set*", or "**chan** k **of** *set*" where *set* is a set of values. For a type "*set*", the identifier describes an ordinary program variable which may carry values within *set*. Clause "**chan** k **of** *set*" declares a channel of capacity k (which can be 0 for handshake communications, 1 or more for bounded capacities, or ∞ for an unbounded capacity) that may store values within *set*.

Besides traditional control flow constructs, sequence and parallel composition, there is a command "**do** ... **od**" which allows to express all types of loops

and conditional statements. The core of statement "**do**...**od**" is a set of clauses of two types: repeat commands, "com; **repeat**", and exit commands, "com; **exit**". During an execution, there can be zero or more iterations, each of them being an execution of one of the repeat commands. The loop is terminated by an execution of one of the exit commands. Each repeat and exit command is typically a sequence with an initial atomic action, the executability of which determining whether that repeat or exit command can start. If several are possible, there is a non-deterministic choice between them.

3.1 P/M-net Based Semantics of B(PN)^2

The definition of the M-net semantics of B(PN)^2 programs (having no preemptible constructs) is given in [3] through a semantical function Mnet. A P/M-net semantics of such programs is easy to obtain through the canonical transformation from M-nets to P/M-nets (as defined in [11]).

In this paper, we introduce new constructs in B(PN)^2 in order to provide it with exceptions. The associated semantics is given through a semantical function PM which extends the canonical semantics obtained from Mnet: function PM maps directly the new B(PN)^2 constructs or overrides the semantics of some existing ones, in particular, of blocks, which may include now the treatment of exceptions.

The semantics of a program is defined *via* the semantics of its constituting parts. The main idea in describing a block is (i) to juxtapose the nets for its local resources declarations with the net for its command followed by a termination net for the declared variables, (ii) to synchronize all matching data/command transitions and (iii) to restrict these transitions in order to make local variables invisible outside of the block.

The access to a program variable v is represented by synchronous action $V(v^i, v^o)$ which describes the change of value of v from its current value v^i (i for *input*), to the new value v^o (*output*).

Each declared variable is described by some *data P/M-net* of the corresponding type, *e.g.*, $N_{Var}(v, set)$ for a variable v of type *set* or $N_{Chan,k}(c, set)$ for a variable c being a channel of capacity k which may carry values of type *set*. The current value of the variable v is stored in a place and may be changed through a $\{\widehat{V}(v^i, v^o)\}$-labelled transition in the data net, while $\{\widehat{C!}(c^!)\}$- and $\{\widehat{C?}(c^?)\}$-labelled transitions are used for sending or receiving values to or from channel c.

Sequential and parallel compositions are directly translated into the corresponding net operations, *e.g.*, $\mathsf{PM}(com_1; com_2) = \mathsf{PM}(com_1); \mathsf{PM}(com_2)$. The semantics of the "**do**...**od**" construct involves the P/M-net iteration operator.

The semantics of an atomic action "$\langle$expr$\rangle$" is $(\alpha.\emptyset.\gamma, \emptyset)$ where α is a set of synchronous communication actions corresponding to program variables involved in "expr", and γ is the guard obtained from "expr" with program variables appropriately replaced by net variables, *e.g.*, v^i for $'v$ and v^o for v'. For instance,

we have:

$$\mathsf{PM}\Big(\langle' v > 0 \wedge v' = c?\rangle\Big) = \Big(\{V(v^i, v^o), C?(c^?)\}.\emptyset.\{v^i > 0 \wedge v^o = c^?\}\ ,\ \emptyset\Big).$$

The P/M-net above has one transition as shown in figure 3. Its synchronous label performs a communication with the resource net for variable v and for channel c: it reads v^i and writes v^o with action $V(v^i, v^o)$, and it gets $c^?$ on the channel with action $C?(c^?)$. The guard ensures that $v^i > 0$ and that v^o is set to the value got on the channel.

4 Modeling Exceptions

In order to model exceptions we introduce in the syntax of B(PN)^{2}a new command, **throw**, which takes one argument which may be either a constant in *Val* in which case it is denoted by w, or a program variable in which case it is denoted by v. It actually represents *the* exception to throw. Moreover, we change the syntax for the blocks as follows:

block	::=	**begin** scope **end**
		| **begin** scope catch-list **end**
catch-list	::=	catch-clause
		| [catch-clause **or**] **catch-others** [v] [**then** com]
catch-clause	::=	**catch** w [**then** com]
		| catch-clause **or** catch-clause

Each catch-clause specifies how to react to an exception w (a value in *Val*). The optional clause **catch-others** can be used to catch any exception uncaught by a previous catch-clause; in this case, it is possible to save the caught exception in a variable v whose type must be *Val*.

The semantics for a block "**begin** *scope* cc_1 **or** ... **or** cc_k **end**" where *scope* is the scope for the block and the cc_i's are the catch-clauses (cc_i handles exception w_i, cc_k may be a clause **catch-others**) is the following:

$$\mathsf{PM}\big(\textbf{begin } \mathit{scope}\ cc_1\ \textbf{or}\ \ldots\ \textbf{or}\ cc_k\ \textbf{end}\big) =$$
$$\Big[\!\Big[\{\mathit{catch}, \mathit{noexcept}\} :\quad \pi\Big(\mathsf{PM}(\mathit{scope})\ ;\ \big(\{\mathit{noexcept}\}.\emptyset.\emptyset\ ,\ \emptyset\big)\Big)$$
$$\Big\|\ \Big(\mathsf{PM}(cc_1) \,\square\, \cdots \,\square\, \mathsf{PM}(cc_k) \,\square\, P_{transmit} \,\square\, \big(\{\widehat{noexcept}\}.\emptyset.\emptyset\ ,\ \emptyset\big)\Big)\Big]\!\Big]$$

where $P_{transmit}$ and *noexcept* are explained below.

If the block finishes without throwing any exception, action $\{\mathit{noexcept}\}.\emptyset.\emptyset$ is reached in $\pi\big(\mathsf{PM}(\mathit{scope})\,;\,(\{\mathit{noexcept}\}.\emptyset.\emptyset\,,\,\emptyset)\big)$ and the block can exit by firing the transition which results from the synchronization w.r.t. *noexcept*. If an exception e (e is a net variable) is thrown in the block, it is either caught by one of the catch-clauses cc_i in the block and then a corresponding $\mathsf{PM}(cc_i)$ is executed, or there is no *specific* catch-clause for it and there are still two cases:

– a catch-clause has been specified explicitly in cc_k using **catch-others**, in which case the corresponding $\mathsf{PM}(cc_k)$ is executed;
– there is no **catch-others** specified in the block and so, the uncaught exception e is simply re-thrown by P/M-net $P_{transmit}$.

P/M-net $P_{transmit}$ is defined has follows:

$$P_{transmit} = \begin{cases} (N_{stop}, \emptyset) \\ \qquad \text{if } cc_k \text{ is a clause } \textbf{catch-others}; \\ \big(\{catch(e), throw(e)\}.\{e \neq w_1 \wedge \cdots \wedge e \neq w_k\}.\emptyset\ ,\ \emptyset\big) \\ \qquad \text{otherwise.} \end{cases}$$

where N_{stop} is shown in figure 4 and w_1, …, w_k are the exceptions caught in the catch-clauses of the block.

Fig. 4. The net N_{stop} used in the semantics of blocks.

The new constructs added to the syntax have the following semantics:

$\mathsf{PM}(\textbf{throw}(w)) = \big(\{throw(w)\}.\emptyset.\emptyset\ ,\ \emptyset\big)$
$\mathsf{PM}(\textbf{throw}(v)) = \big(\{V(v^i, v^o), throw(e)\}.\emptyset.\{e = v^i \wedge v^o = v^i\}\ ,\ \emptyset\big)$
where e is a net variable
$\mathsf{PM}(\textbf{catch}\ w) = \big(\{catch(w)\}.\emptyset.\emptyset\ ,\ \emptyset\big)$
$\mathsf{PM}(\textbf{catch}\ w\ \textbf{then}\ com) = \mathsf{PM}(\textbf{catch}\ w)\ ;\ \mathsf{PM}(com)$
$\mathsf{PM}(\textbf{catch-others}) = \big(\{catch(e)\}.\emptyset.\{e \neq w_1 \wedge \cdots \wedge e \neq w_{k-1}\}\ ,\ \emptyset\big)$
where w_1, …, w_{k-1} are the exceptions caught in the previous catch-clauses
$\mathsf{PM}(\textbf{catch-others}\ v) = \big(\{catch(e), V(v^i, v^o)\}.\emptyset .\{e \neq w_1 \wedge \cdots \wedge e \neq w_{k-1} \wedge v^o = e\}\ ,\ \emptyset\big)$
$\mathsf{PM}(\textbf{catch-others}\ \textbf{then}\ com) = \mathsf{PM}(\textbf{catch-others})\ ;\ \mathsf{PM}(com)$
$\mathsf{PM}(\textbf{catch-others}\ v\ \textbf{then}\ com) = \mathsf{PM}(\textbf{catch-others}\ v)\ ;\ \mathsf{PM}(com)$

The propagation of exceptions is ensured by alternating scoping w.r.t. *throw* and *catch*, as shown in figure 5. First a $throw(e)$ is "emitted" somewhere in *scope*. Operation π aborts the scope and "converts" the $throw(e)$ into a $\widetilde{catch}(e)$ which synchronizes with an appropriate $catch(w_i)$, then the associated com_i is executed.

In figure 5, we assume that cc_k is a clause **catch-others** which re-throws the exception outside the block. Otherwise, the semantics of the blocks would have ensured this behavior. For $1 \leq i < k$, we assume that cc_i is a clause "**catch** w_i **then** com_i".

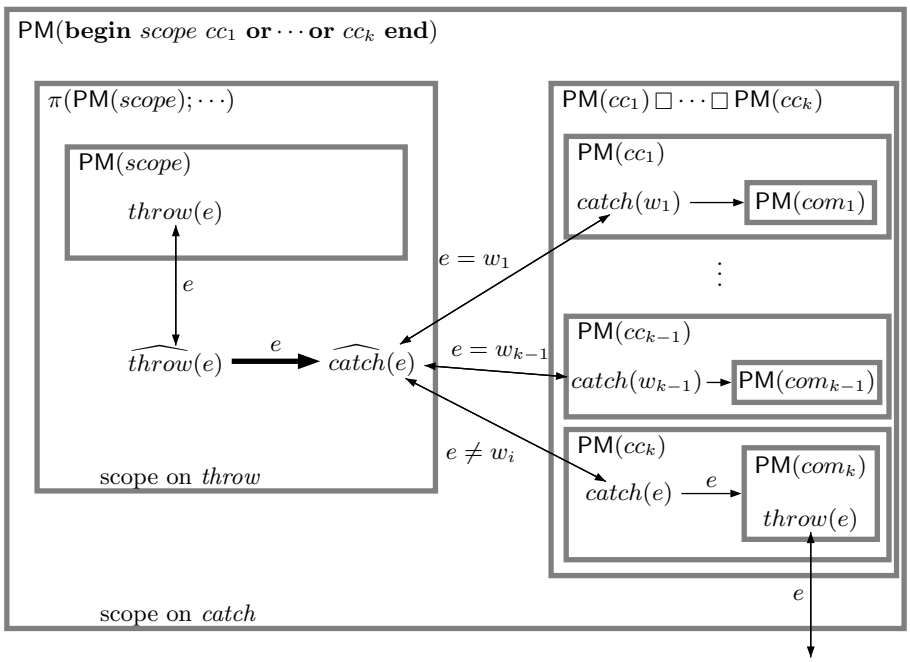


Fig. 5. The semantics of a block with exceptions. A simple arrow denotes a causal dependence between two actions, a double arrow links two synchronized actions. The thick arrow denotes that abortion is performed between the occurrence of the two linked actions.

Notice that several exceptions may be thrown concurrently (from different concurrent parts of the block); in such a case the choice operation in the semantics of the blocks ensures that only one of them may be caught and the others are ignored (this choice is non deterministic).

4.1 Preprocessing

On the top of the semantics given above, we build a preprocessor which rewrites programs, before PM is applied, in order to enforce a more intuitive behavior. Until now:

1. The variables declared in a block are not visible from the commands given in the catch-clauses of this block. The reason is that the declarations for the block are made in a scope nested in π and so they are local to it.
2. Exceptions at the top level of the program are not handled.

The first rewriting rule fixes the first point. It applies when a block comprises some declaration followed by a command and some catch-clauses (all three at the same time):

$$(R1): \quad \begin{array}{l}\textbf{begin}\\ \quad \textit{declarations}\ ;\\ \quad \textit{command}\\ \quad \textit{catch-clauses}\\ \textbf{end}\end{array} \longrightarrow \begin{array}{l}\textbf{begin}\\ \quad \textit{declarations}\ ;\\ \quad \textbf{begin}\\ \quad\quad \textit{command}\\ \quad\quad \textit{catch-clauses}\\ \quad \textbf{end}\\ \textbf{end}\end{array}$$

Then, we use a simple rule for the second point:

$$(R2): \quad \textbf{program}\ \textit{name}\ \textit{block} \longrightarrow \begin{array}{l}\textbf{program}\ \textit{name}\\ \textbf{begin}\\ \quad \textit{block}\\ \textbf{catch-others}\\ \textbf{end}\end{array}$$

In order to avoid a recursive application of this rule, the preprocessor has to jump to *block* after the first application of $(R2)$.

Notice that this rules changes the behavior of the program since it now silently discards an unhandled exception. This behavior may be undesirable and one may prefer a more sophisticated rule which would warn about the problem. Our purpose is just to show that embedding the whole program into a generic environment is an easy solution of this problem.

4.2 Semantics of Procedures in the Context of Static Exceptions

It is well known that static exceptions may lead to an unexpected behavior of procedures when they raise an exception. Consider for example the following block and its sub-block (where w is a given value):

```
begin
  procedure P() begin throw(w) end ;
  begin
    P()
  catch w
  end
end
```

One could expect the clause "**catch** w" to catch the exception thrown by procedure call. But it is not the case with static exceptions: a **throw** occurs "physically" where it was declared, and so, not inside the sub-block.

In order to have the intuitively expected behavior, we extend the preprocessor in such a way that it encapsulates procedure declarations and procedure calls into some additional $\mathrm{B(PN)}^2$ constructs. The usual way to solve this is to consider an exception coming out of a procedure call as a hidden return value. If this

value is set when the procedure returns, then this value is re-thrown at the call point. This way, the thrown exception continues to be propagated from the point where the procedure was called and not from where it was declared. To do this, the preprocessor adds two additional parameters to all procedure call and declaration, one is used to know if an exception was thrown during the procedure call and the other carries the value of the exception when needed.

A call to a procedure P is encapsulated into a block which declares two additional variables, ex and v, which are assumed not to be already used as parameters for P nor as variables already visible from P (in such a case, we just need to choose other names). Variable ex is set to $\bot$ if no exception is thrown in P, otherwise it is set to $\top$ and, in this case, v stores the value of the thrown exception. So, for procedure calls, we have:

$$(R3): \quad P(\textit{effective-parlist}) \quad \longrightarrow \quad \begin{array}{l} \textbf{begin} \\ \quad \textbf{var } ex : \{\top, \bot\} \ , \ \textbf{var } v : \mathit{Val} \ ; \\ \quad P(\textit{effective-parlist}, ex, v) \ ; \\ \quad \textbf{do} \\ \quad\quad \langle {}'ex = \top \rangle \ ; \ \textbf{throw } v \ ; \ \textbf{exit} \\ \quad\quad \langle {}'ex = \bot \rangle \ ; \ \textbf{exit} \\ \quad \textbf{od} \\ \textbf{end} \end{array}$$

where ex and v are fresh identifiers and *effective-parlist* is the list of effective parameters for the procedure call.

For a procedure declaration, we have:

$$(R4): \quad \begin{array}{l} \textbf{procedure } P(\textit{formal-parlist}) \\ \textit{block} \end{array} \quad \longrightarrow \quad \begin{array}{l} \textbf{procedure } P(\textit{formal-parlist}, \\ \qquad\qquad \textbf{ref } ex, \textbf{ref } v) \\ \textbf{begin} \\ \quad \textit{block} \ ; \ \langle ex' = \bot \rangle \\ \textbf{catch-others } v \\ \qquad \textbf{then } \langle ex' = \top \rangle \\ \textbf{end} \end{array}$$

where ex and v are fresh identifiers not already used in *formal-parlist* (the list of formal parameters). These two new parameters are passed by reference.

Since these rules could be applied recursively, the preprocessor uses the following additional directives: for ($R3$), it jumps directly after the text produced since it does not match any other rule; for ($R4$), the preprocessor just has to jump to *block* since no other rule matches the rest.

5 Applications

Combined with concurrency, exceptions allow to express some other preemption related constructs. As an illustration, we give in this section two applications of the exceptions introduced in the paper. First, we use the exceptions in order to

introduce in the language a generalized timeout. Second, we show how to model systems composed of concurrently running blocks, called *threads*, which can be killed from other parts of the program, in particular from the other threads.

5.1 Generalized Timeout

A timeout is usually expressed through a construct such as:

run com_1 **then** com'_1
before com_2 **then** com'_2

which intuitively means "start concurrently commands com_1 and com_2, if com_1 finishes before com_2, then run com'_1 else run com'_2". Usually, com_2 just waits for a timeout event. This may be expressed using exceptions: the command, which finishes first throws an exception which is caught in order to run either com'_1 or com'_2. So, the syntax given above may be rewritten as the following B(PN)^2 block:

begin
(com_1 ; **throw**(1)) $\|$ (com_2 ; **throw**(2))
catch 1 **then** com'_1
or catch 2 **then** com'_2
end

This construct can be easily generalized to an arbitrary number of commands running concurrently, each one trying to finish first. The "winner" kills the others and is the only one allowed to execute its clause **then**. It would also be useful to allow one of the clauses to be a *timeout*. This may be made easily using, for instance, the *causal time model* introduced in [10]. Thus, the syntax would become:

run com_1 **then** com'_1
and com_2 **then** com'_2
$\vdots$
and com_n **then** com'_n
[**timeout** d **then** com'_0]
end run

where d is the number of clock ticks to be counted before timeout occurs. This generalizes the **run**/**before** construct given above and its semantics is easy to obtain: all com_i and, optionally, a chronometer for at most d clock ticks run concurrently; the first which finishes stops the chronometer and throws an exception which is caught in order to execute the corresponding com'_j.

5.2 Simple Threads

As they are defined above, exceptions model what we could call "internal abortion": an exception is propagated through the nesting of blocks, from internal to external ones. In the following, we show that exception can be used in order to model "external abortion" where a block can be aborted by another (non nested)

one. For this purpose, we model a simple multi-threaded system in which *processes* (or *threads*), identified by *process identifiers* (*pid* for short), are able to be killed from any part of the system. The execution of a command "**kill**(s, p)" somewhere in the program has the effect to send signal s to the thread identified by p. When it receives a signal, a thread is allowed to run a command and then it finishes. This behavior is a simplification of what happens in UNIX-like systems.

We use the following syntax for threads:

thread
  declarations for the thread ;
  command for the thread
signal sig_1 **then** com_1
  ⋮
signal sig_n **then** com_n
end thread

Constants sig_1 to sig_n are the signals captured by the thread; we assume that there exists a reserved constant *SIGKILL* which cannot be used in a clause **signal** (in UNIX, it is the name of a signal which cannot be captured). This restriction can be checked syntactically and will be useful in the following.

The programmer is also provided with a new command "**kill**(s, p)" which may be used to send a signal s (any constant in *Val*) to a thread identified by p. One could prefer to restrict signals to a predefined set but this is not necessary for our purpose. The semantics for this new command is simply

$$\mathsf{PM}\big(\mathbf{kill}(s,p)\big) = \big(\{kill(s,p)\}.\emptyset.\emptyset\ ,\ \emptyset\big).$$

Inside each thread, a variable called *pid* is implicitly declared, it contains the pid allocated for the thread. This variable *must not be changed* by the program (this is easy to check syntactically).

In order to attribute pids and to transmit signals, we use a *pid server*, which is a kind of data P/M-net, its priority relation is empty and its net part is defined by the following expression:

$$\Big[\ \emptyset.\{b^+((p_1,\bot)),\ldots,b^+((p_k,\bot))\}.\emptyset$$
$$*\quad \{\widehat{kill}(s,p),\widehat{transmit}(s,p)\}.\emptyset.\emptyset$$
$$\Box\{\widehat{allocpid}(p)\}.\{b^-((p,\bot)),b^+((p,\top))\}.\emptyset$$
$$\Box\{\widehat{freepid}(p)\}.\{b^-((p,\top)),b^+((p,\bot))\}.\emptyset$$
$$*\ \{\widehat{PS_t}\}.\{b^-((p_1,x_1)),\ldots,b^-((p_k,x_k))\}.\emptyset\ \Big]\ \mathbf{tie}\, b$$

This iteration is composed of three parts (separated by stars):

- $\emptyset.\{b^+((p_1,\bot)),\ldots,b^+((p_k,\bot))\}.\emptyset$ is the initialization which sets up the server by filling the heap buffer represented by the asynchronous links on b. It is filled with pairs $(p_i,\bot)$, for $1 \le i \le k$, where the p_i's are the pids and $\bot$ mark them free;

- conversely, the termination, $\{\widehat{PS_t}\}.\{b^-((p_1,x_1)),\ldots,b^-((p_k,x_k))\}.\emptyset$, clears the buffer; it can be triggered from the outside with a synchronization on action $\widehat{PS_t}$;
- the repeated part is the most complicated. It is a choice between three actions, this choice being proposed repeatedly as long as the iteration is not terminated. It offers the following services:
  - allocation of a pid when a thread starts: when the transition labelled by $\{\widehat{allocpid}(p)\}.\{b^-((p,\bot)),b^+((p,\top))\}.\emptyset$ fires, a token $(p,\bot)$ is chosen through asynchronous links on b and marked used (with $\top$ on its second component);
  - symmetrically, when a thread terminates, it frees its pid by synchronizing with $\{\widehat{freepid}(p)\}.\{\widehat{b^-}((p,\top)),b^+((p,\bot))\}.\emptyset$;
  - part $\{\widehat{kill}(s,p),\widehat{transmit}(s,p)\}.\emptyset.\emptyset$ is used to transmit a signal to a thread identified by p. It just "converts" a $\widehat{kill}$ into a $\widehat{transmit}$;

The iteration is under the scope of a **tie** b which sets up the asynchronous links.

Notice that because of the choice in the loop part of the iteration, only one thread action (starting, terminating or killing a thread) can be executed at one time, allowing in this way to avoid, for instance, mutual killings. However, a server with more concurrent behavior may be designed.

Provided this server, we define three internal commands (*i.e.*, not available for the programmer) with the following semantics:

- "**alloc-pid**(v)" asks the server to allocate a pid which is written in variable v, so we have
  $\mathsf{PM}(\textbf{alloc-pid}(v)) = \big(\{allocpid(p),V(v^i,v^o)\}.\emptyset.\{v^o=p\}\ ,\ \emptyset\big)$
- "**free-pid**(v)" frees an allocated pid, reading it in v and so
  $\mathsf{PM}(\textbf{free-pid}(v)) = \big(\{freepid(p),V(v^i,v^o)\}.\emptyset.\{p=v^i\}\ ,\ \emptyset\big)$
- "**capture-signals**" receives a signal relayed by the server and converts it into an exception:
  $$\mathsf{PM}(\textbf{capture-signals}) = \Big(\{transmit(s,p),throw(s),PID(pid^i,pid^o)\}.\emptyset.\{p=pid^i\wedge pid^o=pid^i\} \\ \Box\,\{transmit(s,p),throw(SIGKILL),PID(pid^i,pid^o)\}.\emptyset \\ .\{s\neq sig_1\wedge\cdots\wedge s\neq sig_n\wedge p=pid^i\wedge pid^o=pid^i\}\ ,\ \emptyset\Big)$$
  where $sig_1,\ \ldots,\ sig_n$ are the signal already captured by the thread.

Then, the semantics for the threads given above is a nested block structure as follows:

begin
  var $pid : \{p_1,\ \ldots,\ p_k\}$,
  var $ex : Val$,
  declarations for the thread ;
  alloc-pid(pid) ;
  begin
    (*command for the thread* ; **throw**$(SIGKILL)$) $\|$ **capture-signals**
  catch sig_1 **then** com_1

```
    ⋮
  or catch sig_n then com_n
  or catch SIGKILL
  end ;
  free-pid(pid) ;
catch-others ex then free-pid(pid) ; throw(ex)
end
```

First, we declare fresh variables *pid* (whose name is reserved for threads and must not be declared by the programmer) and *ex* (used to re-throw an exception thrown by the thread). Then we make the declarations for the thread in such a way that they are visible from clauses **signal**. The first instruction initializes *pid* with a call to the pid server. Then the commands (i) which forms the body of the thread and (ii) **capture-signals**, areput in parallel. Command **capture-signals** waits for any signal coming from outside. If this happens, the signal is converted into an exception which is caught accordingly to what is specified in the thread. If a signal which is not handled by the thread comes, it is converted into a *SIGKILL*. If the command for the body of the thread terminates, command **throw**(*SIGKILL*) is used to abort command **capture-signals** and to terminate the block. When the internal block is finished, the pid for the thread is freed and, if the termination comes from an unexpected exception, the first command **free-pid** is by-passed. A **catch-others** allows to free the pid in this case and to re-throw the unexpected exception so it is propagated to the block which declared the thread.

Finally, the semantics for the program just put the pid server in parallel to the most external block (as for a global variable declaration), with the scoping on actions *transmit*, *kill*, *allocpid*, *freepid* and PS_t.

6 Conclusion

Concurrent exceptions has been addressed in literature, for instance in the context of *Coordinated Atomic Actions* [14] or *Place Charts Nets* [8]. In this paper, we introduced static exceptions in a parallel programming language, B(PN)^2, which is provided with a concurrent semantics based on Petri nets and for which implemented tools can be used [7].

It turned out that combining these exceptions with concurrency allowed to express other preemption related constructs like a generalized timeout and a simple multi-threading system.

Future works may emphasize the links with real-time, for instance by introducing *causal time*, already defined in [10] for M-nets, at the level of B(PN)^2. This would allow one to express timed systems using statements like delays and deadlines, and thus would turn B(PN)^2 into a full featured real-time language. Another interesting work would be to apply this kind of semantics to other languages. We believe that, in the present state of the development, these ideas could be used to give a semantics for a reasonably rich (even if not fully general) part of the Ada programming language.

Acknowledgements

We are very grateful to Tristan Crolard: our discussions were fruitful and helped us to clarify our mind about exceptions. We also thank the anonymous referees who pointed out some mistakes and missing references.

References

[1] E. Best, R. Devillers, and J. G. Hall. The box calculus: a new causal algebra with multi-label communication. *LNCS 609:21–69*, 1992.
[2] E. Best, W. Fraczak, R. Hopkins, H. Klaudel, and E. Pelz. M-nets: An algebra of high-level Petri nets, with an application to the semantics of concurrent programming languages. *Acta Informatica*, 35, 1998.
[3] E. Best and R. P. Hopkins. B(PN)^2 — A basic Petri net programming notation. *PARLE'93*, *LNCS 694:379–390*, 1993.
[4] R. Devillers, H. Klaudel, and R.-C Riemann. General refinement for high-level Petri nets. FST&TCS'97, *LNCS 1346:297–311*, 1997.
[5] R. Devillers, H. Klaudel, and R.-C. Riemann. General parameterised refinement and recursion for the M-net calculus. *Theoretical Computer Science*, to appear (available at `http://www.univ-paris12.fr/klaudel/tcs00.ps.gz`).
[6] H. J. Genrich, K. Lautenbach, and P. S. Thiagarajan. Elements of General Net Theory. *Net Theory and Applications*, Proceedings of the Advanced Course on General Net Theory of Processes and Systems, *LNCS 84:21–163*, 1980.
[7] B. Grahlmann and E. Best. PEP — more than a Petri net tool. *LNCS 1055*, 1996.
[8] M. Kishinevsky, J. Cortadella, A. Kondratyev, L. Lavagno, A. Taubin and A. Yakovlev. Coupling asynchrony and interrupts: place chart nets and their synthesis. ICATPN'97, *LNCS 1248:328-347*, 1997.
[9] H. Klaudel. Compositional high-level Petri net semantics of a parallel programming language with procedures. *Science of Computer Programming*, to appear (available at `http://univ-paris12.fr/lacl/klaudel/proc.ps.gz`).
[10] H. Klaudel and F. Pommereau. Asynchronous links in the PBC and M-nets. *ASIAN'99*, *LNCS 1742:190–200*, 1999.
[11] H. Klaudel and F. Pommereau. A concurrent and compositional Petri net semantics of preemption. *IFM'2000*, *LNCS 1945:318–337*, 2000.
[12] P. A. Lee and T. Anderson. Fault tolerance: principle and practice. Springer, 1990.
[13] R. Milner. A calculus of communicating systems. *LNCS 92*, 1980.
[14] B. Randell, A. Romanovsky, R. J. Stroud, J. Xu and A. F. Zorzo. Coordinated Atomic Actions: from concept to implementation. Submitted to *IEEE TC Special issue*.
[15] A. Romanovsky. Practical exception handling and resolution in concurrent programs. *Computer Languages*, v. 23, N1, 1997, pp. 43-58.
[16] A. Romanovsky. Extending conventional languages by distributed/concurrent exception resolution. *Journal of systems architecture*, Elsevier science, 2000
[17] J. Xu, A. Romanovsky and B. Randell. Coordinated Exception Handling in Distributed Object-Oriented Systems: Improved Algorithm, Correctness and Implementation. Computing Dept., University of Newcastle upon Tyne, TR 596, 1997.