A class of composable and preemptible high-level Petri nets with an application to multi-tasking systems

Hanna Klaudel, Franck Pommereau

A class of composable and preemptible high-level Petri nets with an application to multi-tasking systems

Hanna Klaudel

Franck Pommereau

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

Abstract. This paper presents an extension of an algebra of high-level Petri nets with operations for suspension and abortion. These operations are sound with respect to the semantics of preemption, and can be applied to the modelling of the semantics of high-level parallel programming languages with preemption-related features. As an illustration, the paper gives an application to the modelling of a multi-tasking system in a parallel programming language, which is provided with a concurrent semantics based on Petri nets and for which implemented tools can be used.

Keywords: Petri nets, compositionality, preemption, tasks.

1. Introduction

Preemption relates to controlling the execution of the processes composing a concurrent system. Such processes are called *preemptible* if they support interruption during their execution. Usually, when dealing with preemption, one distinguishes a *suspension*, which freezes a process but keeps it alive for a possible resume, from an *abortion*, which kills a process definitively.

Preemption is an essential feature of *reactive systems* which can efficiently be addressed using *synchronous* models and languages [2, 15]. However, very often, synchronous approaches cannot be used for mixed control- and data-flow applications since they only deal with the control structure. Also, in most cases, the underlying semantics is *sequential* (*i.e.*, parallel events are either simultaneous or interleaved), which is well suited to the modelling of systems in which the computation, performed in response to an input coming from the environment, is relatively simple. But when the structure of the computation becomes more important than the structure of the reaction, the sequential semantics may not be sufficient. A *concurrent* semantics is often more suitable to the modelling of *heterogeneous* architectures which combine software (distributed on several processors) and specialised hardware components. In particular, in distributed systems, the synchronous paradigm is very difficult to implement.

In this paper we address the problem of modelling preemption (abortion and suspension) in a Petri net framework, with the aim to give a concurrent semantics to parallel programming languages. A treatment of preemption (but only abortion) has been introduced in the theory of Petri nets with Place Chart Nets [16] which are hierarchical and whose hierarchy is completely determined by preemption. The originality of our approach with respect to Place Chart Nets is in providing a complete algebra of Petri nets with a number of control-flow and communication operators.

Our starting point is a compositional model defined by the *Petri Box Calculus* (*PBC*) [4] in which concurrency, non-determinism, causality and a treatment of data, can be represented explicitly. The semantic domain of PBC forms a class of labelled safe Petri nets, called *boxes*, provided with a set of composition operations giving them an algebraic structure. In order to cope with the possibly large size of the nets, higher level versions of PBC have been considered, and in particular an algebra of *M-nets* (high-level Petri net version of boxes [5]) which allows one to represent large systems in a clear and compact way. The high- and low-level domains are related by an operation of *unfolding* which associates a box to each M-net. The PBC framework also features a parallel programming language, *B(PN)*^{2} [6], which can be seen as a "user friendly" syntax on the top of both, high- and low-level process algebras. This framework is implemented in PEP toolkit [14, 24], allowing one to simulate the modelled systems and to verify their properties *via* model checking [3, 13]. This paper aims at providing a basis for introducing preemption in this framework.

A *preemptible Petri net* is expected to be able to run under a "standard mode", which corresponds to the normal activity of the modelled system, or under a "preemption mode", which corresponds to the situation in which the modelled system is interrupted (suspended or aborted) and must stop its normal activity, possibly for another one. The two modes are mutually exclusive and the preemption mode has the priority over the normal mode (we treat here the case which, in the terminology of [1], is called "must" preemption). More precisely, if two transitions t_n, for normal mode, and t_p, for preemption mode, are both enabled, t_p should be always preferred. This point of view naturally leads to consider priorities between transitions, and allows us to bring to the theory of *priority systems* as presented in [7]. In this paper, the M-net model is enriched by considering *M-nets with priorities* as pairs (N, ρ), where N is an M-net and ρ a binary priority relation between its transitions. The M-net algebra is then extended by two operations, which allow one to make suspendable or abortable any M-net with priorities and which can be arbitrarily nested. These operations are orthogonal with respect to each other since it makes sense to suspend a net comprising aborted parts or to abort a suspended net. Moreover, they are independent of the rest of the algebra as advocated in [1]. We are particularly interested in a *sub-class* of M-nets with priorities, called *preemptible M-nets* (*PM-nets*), which fulfil some structural constraints. (Each PM-net is an M-net with priorities where the priority relation has some suitable properties.) The behaviour of PM-nets is sound with respect to the semantics of preemption. The proposed approach tends to be as conservative as possible with respect to the existing framework of M-nets. The goal is to minimise the changes necessary in order to adapt the existing software tools to the proposed model. Notice that we do not propose, at the level of the algebra, a way to prevent a net from being preempted. Instead, we advocate that this kind of feature could be realized on the top of the low-level operations we provide. The model presented in this paper is largely improved and extended with respect to its first version from [19]. It was applied in [20] to a semantics of exceptions in $\mathrm{B(PN)}^2$. In this paper, we outline another possible application of the introduced preemption operations by extending the $\mathrm{B(PN)}^2$ language with tasks.

The paper is structured as follows. First, we recall briefly the definition of M-nets [5, 9, 18], the associated algebra and their dynamic behaviour (step sequence semantics). We define then an auxiliary class of nets called *M-nets with priorities*, analogous to *priority systems* from [7]. It consists of M-nets equipped with a binary priority relation between their transitions. The transition rule of these nets takes into account the information about priorities and so does their step sequence semantics. We then extend M-net operations to M-nets with priorities, giving them an algebraic structure. Next, we define new operations for M-nets with priorities, π_s and π_a, which allow one to make suspendable or abortable, respectively, any M-net with priority. Finally, we introduce PM-nets (preemptible M-nets) as a sub-class of M-nets with priorities having interesting structural properties. The last section is dedicated to an application in which we extend the language $\mathrm{B(PN)}^2$ with tasks.

2. M-nets

An M-net N is a triple (S, T, ι), where S is the set of places, T is that of transitions, $(T \times S) \cup (S \times T)$ is that of arcs, and ι is the annotation function on places, transitions and arcs.

The annotation of a place s is of the form $\iota(s) = \lambda(s).\tau(s)$. $\lambda(s)$ is a *label*, corresponding to its *status* which may be: entry e, exit x or internal i. $\tau(s)$ is a *type*, a non-empty set of values, from a finite set *Val*, which the place is allowed to carry.

The annotation of a transition t is of the form $\lambda(t).\gamma(t)$ where $\lambda(t)$ is a *label*, which can be hierarchical or used for communication, and $\gamma(t)$ is a *guard* (a finite set of predicates). 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. Communications are similar to CCS ones [22], *e.g.*, between transitions labelled by actions such as $A(a_1, \ldots, a_n)$ or $\widehat{A}(a'_1, \ldots, a'_n)$, where A is an *action symbol*, $\widehat{A}$ is its *conjugate* and each a_i and a'_i is a value (in *Val*) or a variable (belonging to a set *Var*).

Arcs are inscribed by sets of values or variables, representing what is transported by an arc during the firing of a transition[1]. As usual, for any place or transition $r \in S \cup T$, we denote by ${}^\bullet r$ its pre-set $\{r' \in S \cup T \mid \iota(r', r) \neq \emptyset\}$ and, similarly, by $r^\bullet = \{r' \in S \cup T \mid \iota(r, r') \neq \emptyset\}$ its post-set.

M-nets are represented as labelled high-level Petri nets with the following simplifications in order to keep the figures as clear as possible. Arcs with empty annotations are omitted. An annotation $\{\bullet\}$ on an arc is omitted most of time, as well as empty labels or guards. A

[1] Actually, more complex *structured annotations* are generated by the refinement [9], but their introduction with all details could be harmful for the intuition, so we omit them in order to streamline the presentation.

place type $\{\bullet\}$ is also omitted. Moreover, the brackets enclosing sets are omitted when no confusion is possible. A double-arrowed arc stands for two opposite direction arcs having the same annotation (side-loop). Hierarchical transitions (i.e., transitions labelled by a hierarchical action) are represented by squares with a double border. Finally, in order to reduce the number of diagrams, we denote by $\lambda.\gamma$ the simple M-net depicted in figure 1.

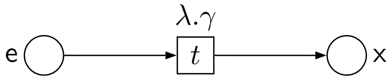


Figure 1. M-net $\lambda.\gamma$.

2.1. Dynamic behaviour 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 its adjacent arcs. A *binding* for a transition t is a substitution $\sigma : var(t) \rightarrow Val$; it will be called *enabling* if it satisfies the guard and if the flow of tokens it implies respects the types of the places adjacent to t. We denote by $\nu[\sigma]$ the evaluation of an object ν (which may be a transition or arc annotation) under σ.

A *marking* M of an M-net (S, T, ι) is a mapping which associates to each place $s \in S$ a set[2] 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 set otherwise; the *exit marking* M_{x} is defined analogously. The dynamic behaviour of an M-net starts with its entry marking; the exit marking corresponds to the final state.

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 in order to satisfy the required flow) and if its guard evaluates to *true* through σ. 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:

$$\forall s \in S\colon \quad M'(s) = M(s) - \iota(s,t)[\sigma] + \iota(t,s)[\sigma].$$

The above transition rule defines the *interleaving semantics* of M-nets which is the set of all the possible occurrence sequences. This semantics can be generalised by introducing the *step sequence semantics* [12], which allows any number of transitions to occur simultaneously. Given an M-net $N = (S, T, \iota)$, a set[3] δ of bound transitions $t[\sigma]$ (where t is a transitions and σ and enabling binding of t) is called *concurrently enabled* at a marking M if there are enough tokens to allow the simultaneous firing of all the transitions in δ. Such a δ is called a *step*. A *step sequence* of N is a sequence $D = \delta_1\delta_2\ldots$ such that there are markings M_1, M_2, …, where $M_1 = M_{\mathsf{e}}$, which satisfy $M_i[\delta_i\rangle M_{i+1}$ for $i \geq 1$. The set of all the step sequences of N is its *step*

[2] As we explain below, we consider only safe M-nets; this allows, in particular, to simplify some definitions. For instance, we may use sets instead of multi-sets which are usually required.

[3] There is no auto-concurrency because only safe M-nets are considered.

sequence semantics and is denoted by $\mathit{steps}(N)$. It is easy to see that $\mathit{steps}(N)$ is closed under linearisation, *i.e.*, if δ belongs to a step sequence $D \in \mathit{steps}(N)$, then, replacing δ by any of its linearisation gives a step sequence which is also in $\mathit{steps}(N)$. For instance, a step $\{t_1[\sigma_1], t_2[\sigma_2]\}$ can be replaced by $\{t_1[\sigma_1]\}\{t_2[\sigma_2]\}$ or $\{t_2[\sigma_2]\}\{t_1[\sigma_1]\}$. In the following, a step $\{t_1[\sigma_1], t_2[\sigma_2]\}$ will be denoted $\{t_1, t_2\}$ if σ_1 and σ_2 are the only enabling bindings of t_1 and t_2, respectively.

2.2. Unfolding

The unfolding of an M-net $N = (S, T, \iota)$ is the labelled Petri net $\mathcal{U}(N) = (\mathcal{U}(S), \mathcal{U}(T), W, \lambda)$, where $\mathcal{U}(S)$ is the set of places, $\mathcal{U}(T)$ the set of transitions, W the weight function on arcs and λ the labelling function on places and transitions, defined as follows:

- $\mathcal{U}(S) = \{(s, v) \mid s \in S \text{ and } v \in \tau(s)\}$;
- $\forall (s, v) \in \mathcal{U}(S) : \lambda((s, v)) = \lambda(s)$;
- $\mathcal{U}(T) = \{(t, \sigma) \mid t \in T \text{ and } \sigma \text{ is an enabling binding of } t\}$;
- $\forall (t, \sigma) \in \mathcal{U}(T) : \lambda((t, \sigma)) = \begin{cases} \lambda(t)[\sigma] & \text{if } t \text{ is a communication transition,} \\ \lambda(t) & \text{if } t \text{ is a hierarchical transition;} \end{cases}$
- $\forall (t, \sigma) \in \mathcal{U}(T), \forall (s, v) \in \mathcal{U}(S) : W((s, v), (t, \sigma)) = \sum_{x \in \iota(s,t)} \iota(s, t)(x) \cdot x[\sigma](v)$, where $x[\sigma](v)$ is the number of instances of value v occurring in the arc inscription x evaluated under σ. In other words, the weight of the arc is the number of occurrences of value v taken from s during a firing of t under binding σ. $W((t, \sigma), (s, v))$ is defined analogously.

If M is a marking of N, the marking $\mathcal{U}(M)$ of $\mathcal{U}(N)$ is such that each low-level place (s, v) contains as many tokens as the number of occurrences of v in s.

A labelled Petri net (S, T, W, λ) is called *T-restricted* if, for all transition $t \in T$, we have ${}^\bullet t \neq \emptyset \neq t^\bullet$; *i.e.*, each transition in T has at least one input and one output place. An M-net is called *T-restricted* if so is its unfolding.

The unfolding can easily be extended to steps and step sequences by replacing in each step δ, each high-level bound transition $t[\sigma]$ by its unfolding (t, σ). Moreover, by definition of the unfolding of a marking of N, it may easily be shown that unfolding the step sequence semantics of an M-net N gives exactly the step sequence semantics obtained from $\mathcal{U}(N)$.

Proposition 2.1. Let N be an M-net. Then, $\mathcal{U}(\mathit{steps}(N)) = \mathit{steps}(\mathcal{U}(N))$. □

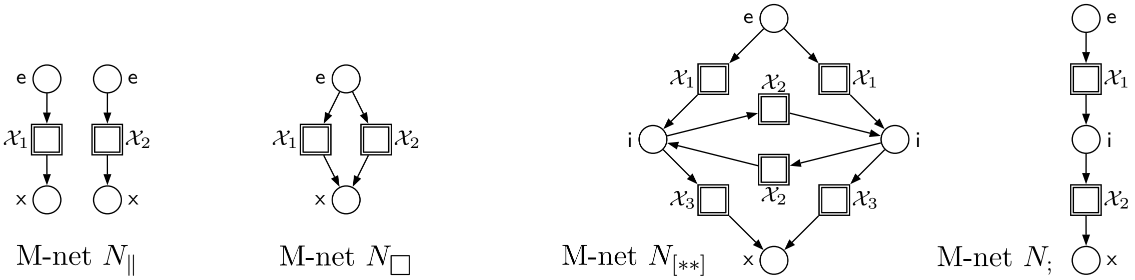
An M-net N in entry marking is called *safe* if so is its unfolding, *i.e.*, every marking M of $\mathcal{U}(N)$, reachable from $\mathcal{U}(M_\mathsf{e})$ holds at most one token per place. Traditionally, and it is also the case in this paper, only safe M-nets are considered since this class happens to be powerful enough for most practical applications while guaranteeing efficient verification algorithms [11].

2.3. Algebra of M-nets

For compositionality, we are particularly interested in a sub-class of M-nets, called ex-*good* M-nets, which have at least one entry and one exit place, which are T-restricted, and such that there are neither in-going arcs to entry places nor outgoing arcs from exit places (ex-directedness property). The algebra of unmarked ex-good M-nets comprises the operations listed below[4], where N_1, N_2 and N_3 are M-nets, $\mathcal{X}$ is a hierarchical symbol, and A is an action symbol.

$N_1[\mathcal{X} \leftarrow N_2]$	refinement	$[N_1 * N_2 * N_3]$	iteration
$N_1 \| N_2$	parallel composition	$N_1\,\mathbf{sy}\,A$	synchronisation
$N_1 ; N_2$	sequence	$N_1\,\mathbf{rs}\,A$	restriction
$N_1 \,\Box\, N_2$	choice	$N_1\,\mathbf{sc}\,A$	scoping

The refinement of a hierarchical transition (labelled $\mathcal{X}$) by a net is a transition substitution. It allows the refining net to be executed each time (for every enabling binding) a hierarchical transition in the refined net could fire[5]. The parallel composition puts nets side by side without any link between them so they can execute in total concurrency. The sequential composition allows N_1 to be executed first and be followed by N_2. 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 (initialisation), the second one is executed an arbitrary number of times (loop), and is followed by one execution of the third one (exit). These four operators are called the *control flow* ones and are defined using the refinement and special *operator nets* (shown in figure 2) in which each transition is substituted by one of the arguments of the operation.



M-net $N_{\|}$ M-net $N_{\Box}$ M-net $N_{[**]}$ M-net $N_{;}$

Figure 2. The operator nets used to synthesise the control flow operators. For each operation, each $\mathcal{X}_i$-labelled transition is refined by the ith argument of the operation.

The synchronisation w.r.t. an action 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 (*i.e.*, it removes transitions having an A or an $\widehat{A}$ in their label). The scoping w.r.t. A is defined as a synchronisation w.r.t. A, followed by a

[4] Actually, the algebra of M-nets comprises also operations of renaming and asynchronous links [18]. We omit them here for the sake of simplicity. All the enrichments we introduce in this paper easily extend to these operations.

[5] In order to preserve the uniqueness of names, the names of nodes in the refining net are prefixed by the name of the refined transition (for instance, if transition t belongs to a net refined into a transition $t_{\mathcal{X}}$, its name becomes $t_{\mathcal{X}}.t$).

restriction w.r.t. A. It is used to setup all synchronisations w.r.t. A, making them local to the net and no longer available for a further synchronisation. Thanks to commutativity properties concerning synchronisation, restriction and scoping, these operators can be extended to sets of action symbols. For instance $N_1\,\mathbf{sc}\{A, A'\}$ stands for $(N_1\,\mathbf{sc}\,A)\,\mathbf{sc}\,A'$ (or the other way around). Detailed explanations and some examples of these operations are given in [5, 18, 9].

In the following, we only use iteration in very restricted cases, so we define a binary version of iteration as:

$$N_1 * N_2 = N_2 \,\Box\, [N_1 * N_1 * N_2].$$

3. M-nets with priorities

Let $N = (S, T, \iota)$ be an M-net. A *priority relation* $\rho \subseteq T \times T$ is a binary relation on transitions of N. Intuitively, $(t_1, t_2) \in \rho$ means that during an execution of N, the firing of transition t_2 is always preferred to that of t_1 when both transitions are enabled. In other words, t_1 has a lower priority than t_2. We shall denote this by $t_1 \prec_\rho t_2$ (or simply $t_1 \prec t_2$ if ρ is understood from the context).

A priority relation $\rho \subseteq T \times T$ is called *well-formed* if $\rho \cap \{(t,t) \mid t \in T\} = \emptyset$ and if it is included in a partial order over T (this amounts to say that the graph of ρ has no cycle). Intuitively, a priority relation is meaningful if it is well-formed, i.e., if it does not specify contradictions in the priority between transitions.

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

Definition 3.1. Let $P = (N, \rho)$ be an M-net with priorities, 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 $M[t\rangle_\rho$, if $\nexists t' \in T$ such that $M[t'\rangle$ and $t \prec t'$.

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 $M[t\rangle_\rho \Rightarrow M[t\rangle$.

Notice also that we do not require the priority relation ρ of M-nets with priorities to be well-formed. It may contain, for instance, inconsistencies such as $t \prec t$ which always disable t. It may also contain cycles such that $t_1 \prec t_2 \prec \cdots \prec t_1$ in which case all t_i may be enabled (with the usual transition rule) and thus none of them would be ρ-enabled. We neither require ρ to be transitive. However, the well-formedness of the priority relation will be a property of PM-nets: a sub-class of M-nets with priorities, which are of major interest for our purpose.

The notion of steps and step sequences defined for M-nets cannot be directly reused for M-nets with priorities because it would lead to inconsistencies in the semantics. Consider, for example, the M-net with priorities $P = (N, \rho)$ shown in figure 3 (taken from [7]); if we do not take ρ into account, we have the step sequence semantics:

$$\mathit{steps}(N) = \{\emptyset, \{t_1\}, \{t_3\}, \{t_1, t_3\}, \{t_1\}\{t_3\}, \{t_3\}\{t_1\}, \{t_1\}\{t_2\}\},$$

where $\emptyset$ is the empty step sequence. We can see that it contains the sequence $\{t_1\}\{t_3\}$ which violates ρ (because, after the firing of t_1, t_3 and t_2 are both enabled and t_2 has the priority). Removing this sequence is necessary but not sufficient since some inconsistency remains. Actually, the semantics cannot contain $\{t_1, t_3\}$ because $\{t_1\}\{t_3\}$ is one of its linearisations. The *consistent step sequence semantics* of P, denoted $\mathit{steps}(P)$, is thus the largest sub-set of $\mathit{steps}(N)$ such that each step sequence $D \in \mathit{steps}(P)$ and each of its linearisations respects ρ. For the above example, we have:

$$\mathit{steps}(P) = \{\emptyset, \{t_1\}, \{t_3\}, \{t_3\}\{t_1\}, \{t_1\}\{t_2\}\}.$$

According to [7], this consistent step sequence semantics is one of the most concurrent semantics that one can expect for priority systems.

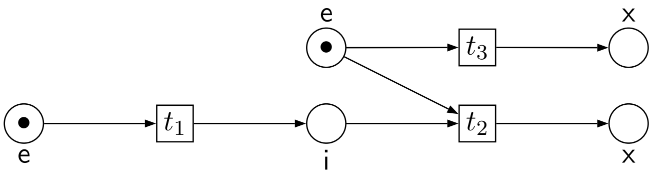


Figure 3. An M-net with priorities and its entry marking, with $\rho = \{(t_3, t_2)\}$.

The unfolding of M-nets with priorities is a natural extension of the unfolding of M-nets.

Definition 3.2. Let $P = (N, \rho)$ be an M-net with priorities. The unfolding of P, $\mathcal{U}(P)$, is a pair $(\mathcal{U}(N), \mathcal{U}(\rho))$ where $\mathcal{U}(N)$ is the usual M-net unfolding of N and

$$\mathcal{U}(\rho) = \{((t, \sigma_i), (t', \sigma'_j)) \mid t \prec_\rho t' \text{ and } (t, \sigma_i) \in \mathcal{U}(t) \text{ and } (t', \sigma'_j) \in \mathcal{U}(t')\}$$

where $\mathcal{U}(t) = \{(t, \sigma_i) \mid \sigma_i$ is an enabling binding of $t\}$ and $\mathcal{U}(t') = \{(t', \sigma'_j) \mid \sigma'_j$ is an enabling binding of $t'\}$. The unfolding of markings is defined as for M-nets.

As before, an extension of the unfolding to consistent steps and consistent step sequences is straightforward. By definition of the unfolding of the M-nets with priorities and of the priority relation, we obtain easily the following proposition.

Proposition 3.1. Let P be an M-net with priorities. Then, $\mathcal{U}(\mathit{steps}(P)) = \mathit{steps}(\mathcal{U}(P))$. □

3.1. Algebra of M-nets with priorities

The extension of the M-net operations to M-nets with priorities is immediate for most of them. However, in the case of synchronisation or refinement, several possible definitions of the resulting priority relations may be considered. Our choice is clearly not the most general (in particular for the refinement) but it is suitable for the definitions of the preemption operations and for that of the sub-class of PM-nets. However, the definition of refinement allows one to refine several $\mathcal{X}$ labelled transitions simultaneously as it is already the case in M-nets [9].

Definition 3.3. Let $P_i = (N_i, \rho_i)$, for $i \in \{1, 2\}$, be M-nets with priorities, where $N_i = (S_i, T_i, \iota_i)$. Moreover, let $\mathcal{X}$ be a hierarchical symbol, and A an action symbol. The usual M-net operations are extended as follows for M-nets with priorities (see footnote 5 about notation $t_{\mathcal{X}}.t$):

- $P_1[\mathcal{X} \leftarrow P_2] = (N_1[\mathcal{X} \leftarrow N_2], \rho)$ where
  $$\begin{aligned}\rho = &\{(t,t') \in \rho_1 \mid \lambda_1(t) \neq \mathcal{X} \neq \lambda_1(t')\}\\ &\uplus \{(t_{\mathcal{X}}.t, t_{\mathcal{X}}.t') \mid t \prec_{\rho_2} t' \text{ and } t_{\mathcal{X}} \in T_1 \text{ and } \lambda_1(t_{\mathcal{X}}) = \mathcal{X}\}\\ &\uplus \{(t_{\mathcal{X}}.t, t') \mid t_{\mathcal{X}} \prec_{\rho_1} t' \text{ and } \lambda_1(t_{\mathcal{X}}) = \mathcal{X} \text{ and } t \in T_2\};\end{aligned}$$

- $P_1 \,\mathbf{sy}\, A = (N_1 \,\mathbf{sy}\, A, \rho)$ where $N_1 \,\mathbf{sy}\, A = (S, T, \iota)$ and ρ is the smallest set including ρ_1 such that for each $t' \in T$ resulting from a basic synchronisation of t_1 with t_2, and for each $t'' \in T$,

  - if $t_1 \prec_\rho t''$ or $t_2 \prec_\rho t''$, then $t' \prec_\rho t''$,
  - if $t'' \prec_\rho t_1$ or $t'' \prec_\rho t_2$, then $t'' \prec_\rho t'$;

- $P_1 \,\mathbf{rs}\, A = (N_1 \,\mathbf{rs}\, A, \rho)$, where $N_1 \,\mathbf{rs}\, A = (S, T, \iota)$ and $\rho = \rho_1 \cap (T \times T)$.

Control flow operators (sequential composition, iteration, parallel composition and choice) are based on refinement [9] and so we do not need a special definition for them. Scoping is defined as a synchronisation followed by a restriction: $P \,\mathbf{sc}\, A = (P \,\mathbf{sy}\, A) \,\mathbf{rs}\, A$. An example of synchronisation with priorities is given in figure 4, it shows how priorities are inherited by synchronisation ($\emptyset$ represents an internal invisible action). In the above definition, one may notice that synchronisation allows non well-formed priority relations to be produced. For instance, if we add $t_1 \prec t_2$ to the priority relation of the net given on the left of figure 4, we obtain $t_1 \prec t_4$ after synchronisation. In section 6.2, we will see that, thanks to some structural constraints; this kind of situation will never appear in PM-nets.

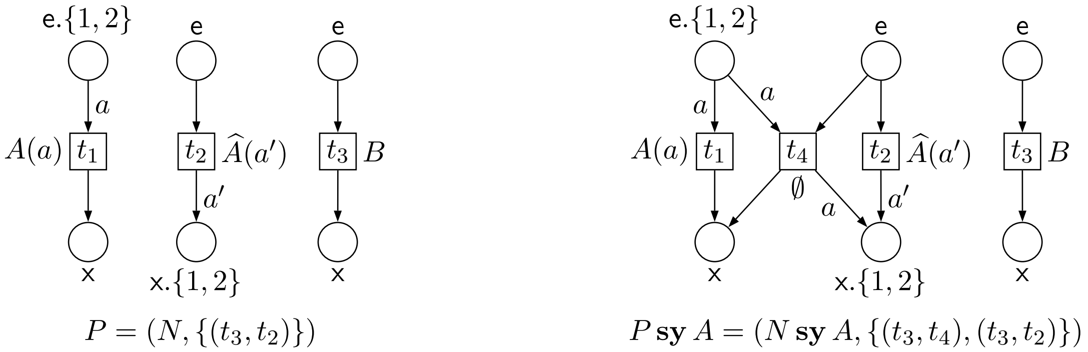


Figure 4. Example of synchronisation of an M-net with priorities. Restricting w.r.t. A would remove t_1 and t_2 (with their surrounding arcs) from the net and (t_3, t_2) from its priority relation.

4. New operations for preemption

4.1. Suspension

Thanks to priorities, suspending an arbitrary net P is quite simple: intuitively, the idea is to embed P in a net which contains a transition with a higher priority w.r.t. all the transitions in P. Then, enabling this transition suspends the execution of P while disabling it resumes the execution of P. As long as this transition stays enabled, P cannot evolve because of the priorities. This can be done compositionally with the M-net with priorities P_s represented in figure 5.

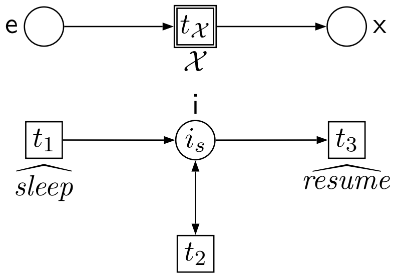


Figure 5. Net P_s whose priority relation is $\rho_s = \{(t_\mathcal{X}, t_2)\}$.

Definition 4.1. For any M-net with priorities P, we have:

$$\pi_s(P) = P_s[\mathcal{X} \leftarrow (P\,\mathbf{sc}\,\mathit{resume})]\,\mathbf{sc}\,\mathit{sleep},$$

where P_s is the M-net with priorities depicted in figure 5.

Operator π_s works as follows:

- P is first refined in P_s; because $t_\mathcal{X} \prec_{\rho_s} t_2$ and thanks to the definition of refinement in PM-nets, we have $t_\mathcal{X}.t \prec t_2$ for any transition t from $P\,\mathbf{sc}\,\mathit{resume}$;

- a scoping w.r.t. *sleep* allows P to suspend its execution by firing a transition labelled with an action *sleep*. This results in putting a token in place i_s and so in enabling transition t_2; thanks to priorities, P is then suspended as long as t_2 remains enabled;

- P can be resumed by removing the token from i_s thanks to transition t_3; this can be made by any transition, external to $\pi_s(P)$, with an action *resume* in its label which would synchronise with t_3.

Notice that the above definition states that P, before being refined into P_s, is scoped w.r.t. *resume* in order to realize the communications w.r.t. the actions *resume* which may come from an application of π_s nested in P. The reason is that a transition external to $\pi_s(P)$, which would synchronise w.r.t. action *resume*, should synchronise with transition t_3 and not with another one coming from a possible π_s nested in P.

In order to illustrate suspension and to show how it can be initiated from the outside of a process, we give here a simple example of a printer. A process P is responsible for the actual printing of the pages. We want the printer to be able to suspend printing (*i.e.*, to suspend P) when the paper tray is open. Printing is resumed when the tray is closed. We embed P into an environment which allows it to be suspended thanks to a synchronisation w.r.t. an action

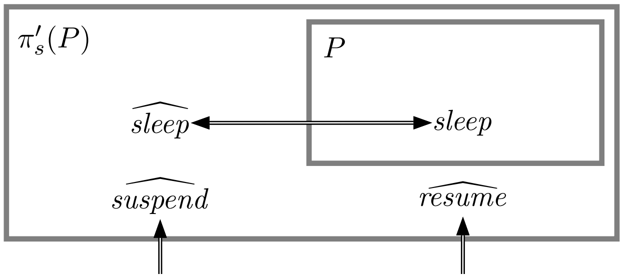


Figure 6. A simplified view of $\pi'_s(P)$.

suspend. This is performed by an operation π'_s defined as follows and sketched in figure 6, where we emphasise the communication interface with respect to suspension:

$$\pi'_s(P) = \pi_s\Big(\Big(\Big(P\,\mathbf{sc}\,suspend\ ;\ (\{term\}.\emptyset\,,\,\emptyset)\Big)\ \Big\|\ P_c\Big)\,\mathbf{sc}\,term\Big)$$

where $(\{term\}.\emptyset, \emptyset)$ is a PM-net with an empty priority relation, its net part being similar to the net depicted in figure 1, and net P_c is responsible for converting repeatedly each incoming *suspend* into a *sleep* in order to suspend the whole net $\pi'_s(P)$:

$$P_c = (\{\widehat{suspend}, sleep\}.\emptyset\,,\,\emptyset)\ *\ (\{\widehat{term}\}.\emptyset\,,\,\emptyset).$$

When P terminates, thanks to the scoping w.r.t. *term*, P_c is terminated too and all $\pi'_s(P)$ can terminate. Notice that operation π'_s can be used for *any* M-net with priorities P, making it externally suspendable. This shows how π_s can be considered as a low-level primitive on the top of which one can build more complex operations. Such an usage of π_s is not mandatory and this operator can be used without any restriction, as it is the case for all the other operators.

Having this suspendable printing process, we model the paper tray by M-net with priorities $(Tray, \emptyset)$ where *Tray* is shown in figure 7. Places s_1 and s_3 correspond to the state "tray closed" and place s_2 to the state "tray open". Transitions t_1 and t_3 correspond to the opening of the tray while transition t_2 corresponds to its closing. The complete system is then modelled by:

$$P_1 = \Big((\pi'_s(P); (\{term\}.\emptyset, \emptyset))\ \Big\|\ (Tray, \emptyset)\Big)\,\mathbf{sc}\,\{suspend, resume, term\}.$$

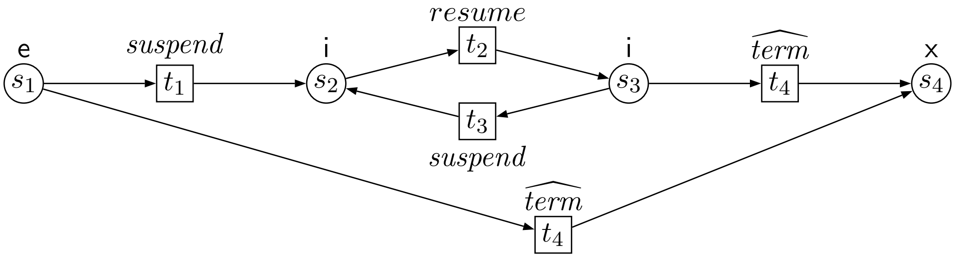


Figure 7. The net part *Tray* of the M-net with priorities which models the paper tray.

4.2. Abortion

In order to make a net P abortable, we proceed similarly as for suspension. The main difference is that, it is necessary to remove all the tokens from P and to produce the exit marking. Figure 8 shows net P_a used in the definition of the operation of abortion. Before refining P in P_a, each place s of P is attached a *clearing transition* t_s with an arc $s \xrightarrow{y} t_s$ where y is a variable in *Var* (in order to match any token from s). Each such clearing transition has a higher priority than all the other transitions outgoing from s. It is labelled with action *clear* and synchronised with t_2 in P_a. In order to remove the tokens from P, we exploit the side loop on transition t_2 (synchronised with the clearing transitions) in such a way that each time it fires, it removes one token from P. The clearing of P is enforced to complete before the whole abortable net can reach its exit marking thanks to the priority $t_3 \prec_{\rho_a} t_2$ which states that each firing of t_2 (actually, of a clearing transition synchronised with t_2) must occur before that of t_3. The firing of t_3 terminates the process of abortion. Another difference with suspension is that, when abortion is decided, a value e is transmitted thanks to action $\widehat{abort}(e)$. When the abortion completes, this value is again available through the label of transition t_3. This transmission of value $e \in Val$ can be seen as a reason given by a net for its abortion, it can be taken into account by its environment or discarded. In [20], it was used for the transmission of the names of the thrown exceptions.

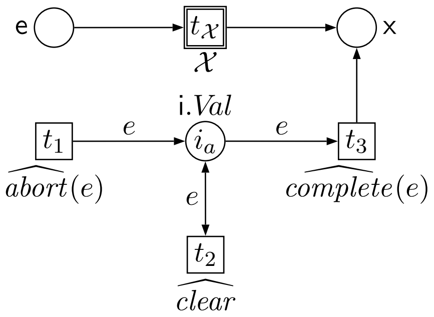


Figure 8. Net P_a, its priority relation being $\rho_a = \{(t_3, t_2)\}$.

We first define an auxiliary operation, **del** A which removes all the occurrences of A-based actions in a net. Assuming that A has the arity $k \geq 0$, if $P = ((S, T, \iota), \rho)$ is a M-net with priorities, we define $P\,\mathbf{del}\,A$ as $((S, T, \iota'), \rho)$ where ι' is such that, for all $s \in S$ and for all $t \in T$: $\iota'(s) = \iota(s)$, $\iota'(s, t) = \iota(s, t)$ and $\iota'(t, s) = \iota(t, s)$, and $\iota'(t)$ is $\iota(t)$ with all the occurrences of actions $A(x_1, \ldots, x_k)$ removed, for all $x_i \in Var \cup Val$, for $i \leq k$.

Definition 4.2. For any M-net with priorities P, we have:

$$\pi_a(P) = P_a[\mathcal{X} \leftarrow \langle\!\langle P\,\mathbf{del}\,complete \rangle\!\rangle; (\emptyset.\emptyset, \emptyset)]\,\mathbf{sc}\,\{clear, abort\},$$

where P_a is the M-net with priorities depicted in figure 8 and $\langle\!\langle \cdots \rangle\!\rangle$ is an auxiliary operation defined as follows. If $P' = P\,\mathbf{del}\,complete = ((S', T', \iota'), \rho')$ then $\langle\!\langle P' \rangle\!\rangle = ((S'', T'', \iota''), \rho'')$ with:

- $S'' = S'$;
- $T'' = T' \uplus \{t_s \mid s \in S\}$;

- $\forall s \in S''$ and $\forall t \in T''$:

$$\iota''(s) = \iota'(s), \qquad \iota''(t) = \begin{cases} \iota'(t) & \text{if } t \in T', \\ \{\mathit{clear}\}.\emptyset & \text{otherwise,} \end{cases}$$

$$\iota''(s,t) = \begin{cases} \iota'(s,t) & \text{if } t \in T', \\ \{y\} \subset \mathit{Var} & \text{if } t = t_s, \\ \emptyset & \text{otherwise,} \end{cases} \qquad \iota''(t,s) = \begin{cases} \iota'(t,s) & \text{if } t \in T', \\ \emptyset & \text{otherwise,} \end{cases}$$

- $\rho'' = \rho' \uplus \{(t, t_s) \mid s \in S' \text{ and } t \in s^\bullet\}$.

The reason for using operation **del** *complete* is similar to that for using **sc** *resume* in the definition of π_s. The difference here is that scoping would remove transition t_3 which should be preserved in order to produce the exit markings of aborted sub-nets.

Notice that this definition states that a net $(\emptyset.\emptyset, \emptyset)$ (this is the net of figure 1 with empty label, guard, and priority relation) is added in sequence to $\langle\!\langle P \,\mathbf{del}\, \mathit{complete} \rangle\!\rangle$ before the refinement. The reason is that $\langle\!\langle \cdots \rangle\!\rangle$ adds clearing transitions to the exit places of P and so the result would not be ex-directed. This sequential composition with a silent action restores ex-directedness[6].

As an illustration, we show how the abortion of an M-net with priorities can be initiated by its environment. The goal with respect to the previous example is to add a "cancel" button to our printer in order to be able to stop the current printing job. For this purpose, we define the following operation, sketched in figure 9:

$$\pi'_a(P) = \pi_a\Big(\Big(\Big(P \;;\; (\{\mathit{term}\}.\emptyset, \emptyset)\Big) \;\Big\|\; P_k\Big) \,\mathbf{sc}\, \mathit{term}\Big)$$

where P_k is responsible for converting an incoming $\mathit{kill}(e)$ into an $\mathit{abort}(e)$. Since only one abortion is possible, no iteration is necessary and P_k is simply defined as:

$$P_k = (\{\widehat{\mathit{kill}}(e), \mathit{abort}(e)\}.\emptyset, \emptyset) \;\square\; (\{\widehat{\mathit{term}}\}.\emptyset, \emptyset).$$

If the job completes when the cancel button has not been pressed, the synchronisation w.r.t. *term* ensures the termination of P_2.

The cancel button is modelled by M-net with priorities $(\mathit{Cancel}, \emptyset)$ where *Cancel* is shown in figure 10. Transition t_1 corresponds to the pressing of the button, the printer could show a message such as "cancelling job"; transition t_2 corresponds to the end of the cancelling and the printer could say something like "job cancelled". Then, the complete system is obtained from the previous one as follows:

$$P_2 = \Big(\Big(\pi'_a(P_1) \;;\; (\{\mathit{term}\}.\emptyset, \emptyset)\Big) \;\Big\|\; \Big((\mathit{Cancel}, \emptyset) \,\square\, (\{\widehat{\mathit{term}}\}.\emptyset, \emptyset)\Big)\Big) \,\mathbf{sc}\, \{\mathit{kill}, \mathit{complete}, \mathit{term}\}.$$

[6] In general, the introduction of silent transitions is avoided in M-nets and PBC because it could allow a net to enter a branch which is deadlocked. However, in our case, this is not a problem since we do not introduce this silent transition before the net being made abortable but only at its end.

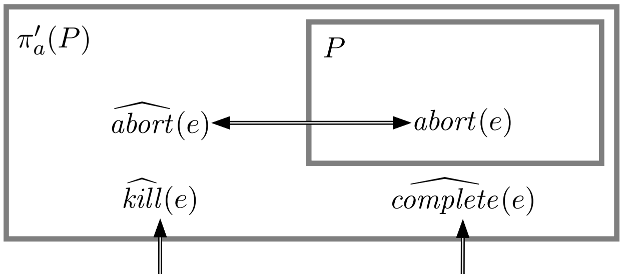


Figure 9. A simplified view of $\pi'_a(P)$.

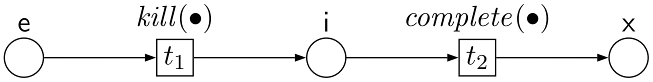


Figure 10. Net part *Cancel* of the PM-net which models the cancel button.

5. Preemptible M-nets: PM-nets

We are now in position to define *preemptible M-nets* (*PM-nets*). They are defined as a sub-class of M-nets with priorities with some structural constraints in order to ensure their priority relation is always well-formed. This sub-class is reasonably wide (see section 6.2) and sound with respect to the semantics of preemption (see section 6.1).

Definition 5.1. Let $P = (N, \rho)$ be an M-net with priorities. P is a PM-net iff:

- N is a safe ex-good M-net and $\rho = \emptyset$, or;
- P is defined as $\pi_s(P_1)$, $\pi_a(P_1)$, $P_1[\mathcal{X} \leftarrow P_2]$, $P_1 \| P_2$, $P_1;P_2$, $P_1 \,\square\, P_2$, $[P_1 * P_2 * P_3]$, $P_1\,\mathbf{sy}\,A$, $P_1\,\mathbf{rs}\,A$ or $P_1\,\mathbf{sc}\,A$ where P_1, P_2 and P_3 are PM-nets, $\mathcal{X}$ is a hierarchical symbol and A is an action symbol.

Proposition 5.1. Let $P = (N, \rho)$ be a PM-net. Then, ρ is well-formed.

Proof:
We proceed by induction on the structure of PM-nets and show that no cycle is ever introduced in the graphs of their priority relations. The nodes of such graphs are PM-net transitions and arcs are defined by the priority relation, i.e., if $t_1 \prec t_2$, then there is an arc from t_2 to t_1 in the graph.

The property is trivial for $P = (N, \emptyset)$. We assume that, the property holds for each PM-net $P_i = ((S_i, T_i, \iota_i), \rho_i)$, for $i \in \{1, 2\}$. Moreover, we assume that $P = ((S, T, \iota), \rho)$, A is an action symbol and $\mathcal{X}$ is a hierarchical symbol. Since control flow operators are based on refinement and scoping on synchronisation and restriction, the only interesting cases are the following.

(1) $P = P_1\,\mathbf{rs}\,A$: Obvious because $\rho \subseteq \rho_1$ since restriction may only remove transitions from T_1.

(2) $P = P_1[\mathcal{X} \leftarrow P_2]$. By the induction hypothesis and by the definition 3.3, the resulting priority is as sketched on the left in figure 11. The graph of ρ_1 and ρ_2 are merged and the only new arcs may be from some transitions in T_1 to some in T_2. Thus, no cycle can be introduced.

(3) $P = \pi_s(P_1)$. By the induction hypothesis and by the definition 4.1, the resulting graph is as sketched in the middle of figure 11, where t_2 comes from net P_s (see figure 5). No cycle can be introduced.

(4) $P = \pi_a(P_1)$. By the induction hypothesis and by definition 4.2, the resulting graph is as sketched on the right in figure 11. In this graph, the t_{s_i}'s are the emptying transitions attached to places $s_i \in S_1$, and t_3 is the $\{\widehat{complete}(e)\}$-labelled transition coming from net P_a (see figure 8). No cycle can be introduced.

(5) $P = P_1 \,\mathbf{sy}\, A$. By the induction hypothesis and by the definition 3.3, the only way to introduce a cycle in the graph of priority is through a synchronisation of transitions t_1 and t_k, for $1 \leq k$, such that $t_1 \prec \cdots \prec t_k$. Such a synchronisation may lead to a transition t with $t \prec \cdots \prec t_{k-1} \prec t$ (if $1 < k$) or $t \prec t$ (if $k = 1$), introducing a cycle. In PM-nets, transitions with higher priorities (like t_k here) are introduced only by π_s or π_a treated in cases (3) and (4). So, no cycle can be added here. □

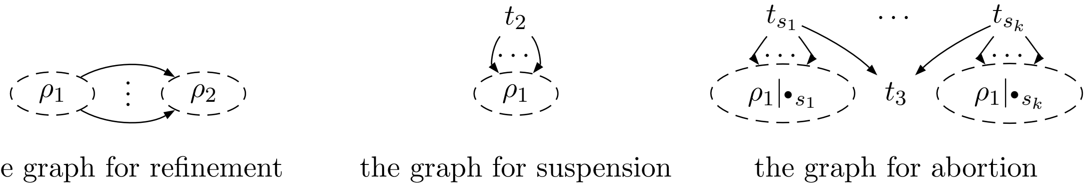


Figure 11. The graphs of the priorities relations produced by the main PM-net operators. Dashed ovals represent the graph of the priority relation indicated inside the oval ($\rho|_{\bullet s}$ being ρ restricted to the transitions from ${}^\bullet s$).

Definition 5.2. A PM-net (N, ρ) is ex-*good* if N is an ex-good M-net, and *safe* if N is safe.

6. Properties of PM-nets and links with existing work

6.1. Soundness of preemption

Let P be an arbitrary PM-net. Intuitively, the behaviour of $\pi_s(P)$ is called *sound* if, between a suspend and a resume, all the transitions in $\pi_s(P)$ which come from P are disabled. More precisely, let t be a transition created in $\pi_s(P)$ by the synchronisation w.r.t. *sleep*. If t fires, then the only enabled transitions in $\pi_s(P)$ are t_2 and t_3 and this remains true until t_3 fires. Similarly, the behaviour of $\pi_a(P)$ is *sound* if, after the firing of a transition created by the synchronisation w.r.t. *abort*, $\pi_a(P)$ reaches its exit marking without firing any transition coming from P.

In our framework, the behaviour of $\pi_s(P)$ is sound for any PM-net P. Indeed, $\pi_s(P)$ starts from its entry marking in which place i_s is not marked. It becomes marked only after the firing of a transition coming from the synchronisation w.r.t. *sleep* of t_1 and a *sleep*-labelled transition in P. Then, as long as t_3 does not fire, transition t_2 and t_3 are enabled. Priority $t_{\mathcal{X}} \prec_{\rho_s} t_2$ ensures that all the transitions from P are disabled since P is refined in $t_{\mathcal{X}}$. The firing of t_2 does not change the enabled transitions; moreover, no transition is allowed to fire concurrently to t_2 since the resulting step would be inconsistent. Finally, the firing of t_3 removes the token from i_s and the execution of $\pi_s(P)$ is resumed.

Similarly, the behaviour of $\pi_a(P)$ is sound for any PM-net P. Indeed, $\pi_a(P)$ starts from its entry marking in which place i_a is not marked and becomes marked only after the firing of a transition coming from the synchronisation w.r.t. *abort* of t_1 and an *abort*-labelled transition in P. Then, the transitions coming from the synchronisation w.r.t. *clear* (we call T_2 the set of these transitions) can be enabled (each marked place s enables emptying transition t_s) and so, thanks to the priorities, t_3 and all the transitions coming from P are disabled. Transition t_3 can fire, producing the exit marking, only when no transition in T_2 is enabled, which means that there is no more tokens in $\pi_a(P)$ except one in i_a. For the same reasons as in the previous case, there is no possibility for a concurrent firing of transitions during all this emptying stage.

6.2. Other properties

Let us observe first that a PM-net which has been constructed without the preemption operations is always ex-good and has an empty priority relation. This property states that the introduced extension is conservative: it does not change the existing model if one makes no usage of the operations introduced for preemption. This is the reason why we consider PM-nets as a "reasonably wide" model: it contains M-nets which have already proved to be useful [3, 13].

However, in general, a PM-net $P = (N, \rho)$ can have some crippled transitions (*i.e.*, transitions with no output place). They can easily be identified by their label $\{\widehat{resume}\}$. These transitions (belonging to the communication interface of P) are crucial in order to be able to resume a suspended sub-net. Therefore, these crippled transitions should be removed when not used. Actually, in that case, the PM-net of interest is $P\,\mathbf{rs}\,resume$ which is ex-good. This property is important because it shows that even if our modelling needs to relax the T-restrictness for some transitions, the final result can always be made T-restricted.

Proposition 6.1. Let P be a PM-net. Then, P is safe and ex-directed. Moreover, $P\,\mathbf{rs}\,resume$ is also T-restricted (and thus is ex-good).

Proof:
The algebra of M-nets has the property to preserve safeness: the composition of safe M-nets is always safe. As far as markings are concerned, the only difference between M-nets an PM-nets is the introduction of operators π_s and π_a. In the first case, the only problem could come from transition t_1 in P_s (see figure 5) which has no input place. But the scoping w.r.t. *sleep* creates only transitions with input places from P and removes t_1. Moreover, these transitions are not allowed to be concurrent since it would lead to inconsistent steps because of priority $t_1 \prec_{\rho_s} t_2$. In the second case, the same remark applies to transition t_1 in P_a (see figure 8) labelled $\widehat{abort}(e)$. Finally, the soundness ensures that no token is left in P after abortion, and so, that no accumulation of tokens is possible.

Concerning ex-directedness, it is enough to say that nets P_s and P_a used in the definition of π_s and π_a respectively, are ex-directedness and thus, so are the results of these operations.

The T-restrictedness of $P\,\mathbf{rs}\,resume$ comes from the fact that in P, the only non T-restricted transitions are those labelled by $\{\widehat{resume}\}$, coming from an application of π_s. The other non T-restricted transitions that one can found in nets P_s and P_a always give, by synchronisation, T-restricted transitions. Then these crippled transitions are removed by restriction as specified in the definition of preemption operators. □

The model of ex-good PM-nets may still appear somehow unsatisfactory, because of the use of priorities. It turns out that some results in the field of semantics of priority systems may be applied for PM-nets. In [7], the authors define a transformation of a finite safe Petri net Σ, equipped with a priority relation ρ, into a bounded Petri net which retains as much as possible of the concurrency of (Σ, ρ). In this context, *as much as possible* means that only semantics composed of *consistent steps* is considered (see [7, section 3]). This result can be directly applied to the unfolding of an ex-good finite PM-net. Then, applying the result from [8], the obtained bounded Petri net can be transformed into a safe Petri net which has the same *pomset* (partially ordered multi-sets) semantics, and we can state:

Proposition 6.2. Let P be a ex-good finite PM-net. Then, P can be transformed into a low-level safe Petri net having the same consistent step sequence semantics. □

Even if PM-nets can be transformed into safe Petri nets having an equivalent concurrent semantics, the construction given in [8] leads to really huge nets, and so, is not intended to be used in practice. Nevertheless, the above proposition is important: it means that safe Petri nets are expressive enough to model preemption, with a concurrent semantics. Notice that this transformation implies the loss of compositionality; this appears to be a reason for extending the existing tools to priorities rather than using this transformation. In practice, it should be possible to modify the existing model checker of PEP [10, 14] in order to have it dealing with priorities. Other tools may be adapted for model-checking PM-nets. For instance, in MARIA [23], coloured Petri nets are checked directly on their marking graphs [21], taking priorities into account could simply consist in changing the transition rule used by the tool.

7. Application to tasks

In this section we present an application of PM-nets to the expression of the concurrent semantics of tasks in a high-level programming language. The starting point of our approach is *B(PN)*^{2} (*Basic Petri Net Programming Notation*) [6, 17] which comprises, in a simple syntax, most traditional concepts of parallel programming. Thanks to its simplicity, it is 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. The most interesting aspect of B(PN)^2 is that it has an original formal semantics in terms of *boxes* and *M-nets*. We propose here to introduce tasks in B(PN)^2, with a simple and intuitive syntax. The presented approach allows one to define tasks (possibly nested, suspendable and abortable), which are able to interact with other tasks by sending signals and reacting to them.

7.1. 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. Figure 12 presents the fragment of the syntax of B(PN)^2 which is relevant to the application presented in this section.

An atomic action is a B(PN)^2 expression "$\langle$expr$\rangle$", *i.e.*, a term constructed over logical and arithmetical operators, constants (as for M-nets, *Val* is the set of the possible values) and

program	::=	**program** block
block	::=	**begin** scope **end**
scope	::=	com | decl ; scope
com	::=	$\langle$expr$\rangle$ | proc-call | com $\|$ com | com ; com | **do** alt-*set* **od**
	|	block | (com)
decl	::=	**var** *name* : *set* | **var** *name* : **chan** *k* **of** *set*
	|	**procedure** *name* (fpl) block | decl , decl
proc-call	::=	*name* (epl)

Figure 12. A fragment of the syntax of B(PN)^2. Keywords are typeset in bold face, non-terminal in roman face and italic denotes the values supplied by the program.

identifiers of *program variables* or *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", consisting in the name of the procedure followed by the effective parameter list "epl"), one of a number of command compositions operator or a block comprising some declarations for a command. Parentheses allow one 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, new declarations may result in the masking of the existing identifiers by the new ones. A procedure can be declared with or without parameters (in which case its formal parameter list "fpl" 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 and a type specification which can be "*set*", or "**chan** *k* **of** *set*" where *set* is a set of values in *Val*. 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 one 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.

7.2. PM-net Semantics of B(PN)^2

The definition of the M-net semantics of B(PN)^2 programs (having no preemptible constructs) is given in [6] through a semantical function Mnet. A PM-net semantics of such programs is easy to obtain through the canonical transformation from M-nets to PM-nets which simply adds an empty priority relation to the M-nets. We show in the next section how to extend B(PN)^2 with tasks and give to them a formal semantics through a semantical function PM. Tasks appear in B(PN)^2 as a new kind of resource, on the same level as the declarations of variables, channels and procedures.

The semantics of a program is defined *via* the semantics of its constituting parts. The main idea in describing a block is (1) to juxtapose the nets for its local resources declarations with the net for its command followed by a termination net for the declared variables, (2) to synchronise all matching data/command transitions and (3) 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 an 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 PM-net* of the corresponding type, *e.g.*, $N_{\mathit{Var}}(v, \mathit{set})$ for a variable v of type *set* or $N_{\mathit{Chan},k}(c, \mathit{set})$ 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.*, $\mathsf{PM}(\mathit{com}_1; \mathit{com}_2) = \mathsf{PM}(\mathit{com}_1); \mathsf{PM}(\mathit{com}_2)$. The semantics of the "**do** ... **od**" construct involves the PM-net iteration and choice operators. The semantics of an atomic action "⟨expr⟩" is PM-net $(\lambda.\gamma, \emptyset)$ (see figure 1) where λ is a set of actions corresponding to program resources involved in "expr", and γ is the guard obtained from "expr" with the program variables appropriately replaced by the corresponding net variables, *e.g.*, v^i for $'v$ and v^o for v'. For instance, we have:

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

The unique transition of the above PM-net performs a communication with the resource nets 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 retrieved on the channel.

7.3. Modelling Tasks

We introduce in the syntax of B(PN)^2 a new declaration and new commands:

decl ::= $\cdots$ | **task** *name* block signals
com ::= $\cdots$ | **start** *name* | **signal** $(\mathit{id}, \mathit{sig})$ | **abort** | **sleep**

A clause "**task** *name* block signals" declares a task called *name* whose body is given in "block" and where part "signals" declares the reactions to the signals received by the task. This declaration is very similar to that of a procedure. An instance of a task *name* can be started with a command "**start** *name*" which creates a new instance of the task which starts to execute its body. The task can communicate with the rest of the program through the usual devices provided by B(PN)^2, namely channels and shared variables. When it starts, each task instance is allocated a unique identifier which is modelled by a read-only resource *id*, available from the body of the task. Sending a signal to a task instance may be realized with command "**signal** (id, sig)" where *id* is the identifier of the target task instance and *sig* is the signal to send (which can be any value in *Val*). Commands "**abort**" and "**sleep**" are used by a task in order to abort or suspend itself, respectively. Part *signals* of the declaration of a task is a (possibly empty) list of clauses "**await** *sig* **then** *command*" which specify the command to run in reaction to a signal. During this reaction, the body of the task is suspended. We assume that there exist signals KILL, SUSPEND and RESUME, having the intuitively expected meaning, which cannot be used in a clause **await**. These signals are automatically handled by the semantics.

In order to manage all the instances identifiers for all the tasks, we define a Task Identifier Manager $P_{TIM} = (N_{TIM}, \emptyset)$ which is a global resource for the program, its net part is depicted in figure 13. This net is initialised when its transition t_1 fires and is terminated with transition t_2. Transitions t_a allocates a new identifier: it picks a token *id* from place i_1 (which holds the free identifiers) and put it into i_2 (used identifiers). Conversely, transition t_f frees an identifier. Notice that the program cannot terminate until all the tasks instances are terminated, *i.e.*, when all the identifiers are returned to i_1. By limiting the size of set *Ident*, one may control globally the maximum number of active tasks in a program.

Given this net, the semantics of a program is:

$$\mathsf{PM}(\textbf{program}\ \mathit{block}) = \Big(\Big((\{\widehat{\mathit{TIM}_{\mathit{init}}}\}.\emptyset, \emptyset)\ ;\ \mathsf{PM}(\mathit{block})\ ;\ (\{\widehat{\mathit{TIM}_{\mathit{term}}}\}.\emptyset, \emptyset)\Big) \Big\| P_{TIM}\Big)$$
$$\textbf{sc}\ \{\mathit{signal}, \mathit{alloc}, \mathit{free}, \mathit{TIM}_{\mathit{init}}, \mathit{TIM}_{\mathit{term}}\}$$

Where the scoping w.r.t. *signal* enforces all the communications related to sending and reacting to signals inside the body of the program.

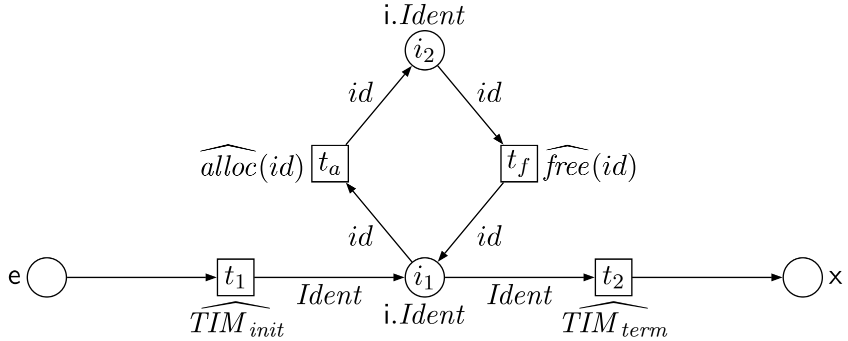


Figure 13. The net part N_{TIM} of the task instance identifier manager, where *Ident* is the set of identifiers.

Each task instance has to store locally its allocated identifier. This is made thanks to net $P_{id} = (N_{id}, \emptyset)$ whose net part is depicted in figure 14. The value of the identifier can be referred to in expressions as id, which leads in the semantics to an action $ID(id)$, analogously to what happens for ordinary program variables.

In order to be able to execute several concurrent instances of a task *name*, we use PM-net $P_{name} = (N_{name}, \emptyset)$ whose net part is depicted in figure 15. The principle is to refine the net which defines the task into transition $t_{\mathcal{X}}$. This way, each time an instance of *name* is started, transition t_{s1} or t_{s2} fires, putting a token in i_2. This enables the execution of a new instance of the task. When an instance terminates, the corresponding token is returned to i_1. In order to control the maximum number of instances for a given task, one may limit the size of set I_{name}. The semantics for starting an instance of a task is simply:

$$\mathsf{PM}(\textbf{start}\ \mathit{name}) = \big(\{\mathit{name}\}.\emptyset, \emptyset\big).$$

The scoping w.r.t. *name* is made at the level of the block which declares the corresponding task so an instance can be started from everywhere in this block. The termination of the block has also to terminate the tasks declarations thanks to a scoping w.r.t. name_{term}, as it is usual for any resource in B(PN)^2. Notice that a block cannot terminate until all the task instances it started have completed, *i.e.*, until all the tokens in N_{name} are returned to i_1.

The semantics of a task declaration is as follows:

$$\mathsf{PM}(\textbf{task}\ \mathit{name}\ \mathit{block}\ \mathit{signals}) = P_{name}[\mathcal{X} \leftarrow P_{task} \| P_{id}]\ \textbf{sc}\ \{\mathit{save}, \mathit{term}, \mathit{ID}\},$$

where

$$\begin{aligned} P_{task} = {} & (\{\mathit{alloc}(id), \mathit{save}(id)\}.\emptyset, \emptyset)\ ; \\ & \pi_a\Big(\Big(\Big(\pi'_s(\mathsf{PM}(\mathit{block}))\ ;\ (\{SH_{term}\}.\emptyset, \emptyset)\Big)\ \Big\|\ \mathsf{PM}(\mathit{signals})\Big) \\ & \qquad \textbf{sc}\ \{\mathit{suspend}, \mathit{resume}, SH_{term}\}\Big)\ \textbf{del}\ \mathit{complete}\ ; \\ & (\{\mathit{free}(id), \mathit{term}(id)\}.\emptyset, \emptyset) \end{aligned}$$

and where operator **del** is the same as in definition 4.2 and SH_{term} is used to terminate the Signal Handler $\mathsf{PM}(\mathit{signals})$ as shown below.

The semantics of a part *signals* given as a list of **await** clauses "$aw_1 \cdots aw_k$" is defined as follows: repeatedly, each signal is awaited and the reaction is executed when it is received; signals KILL, SUSPEND and RESUME are automatically added, with their expected reactions. Formally,

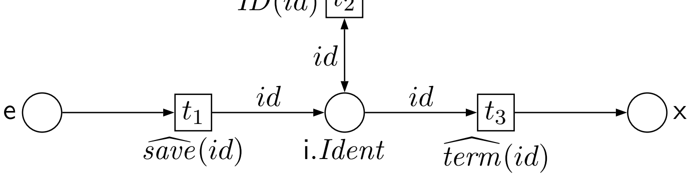


Figure 14. Net part N_{id} of the local identifier store, where $Ident$ is the set of instance identifiers of a task.

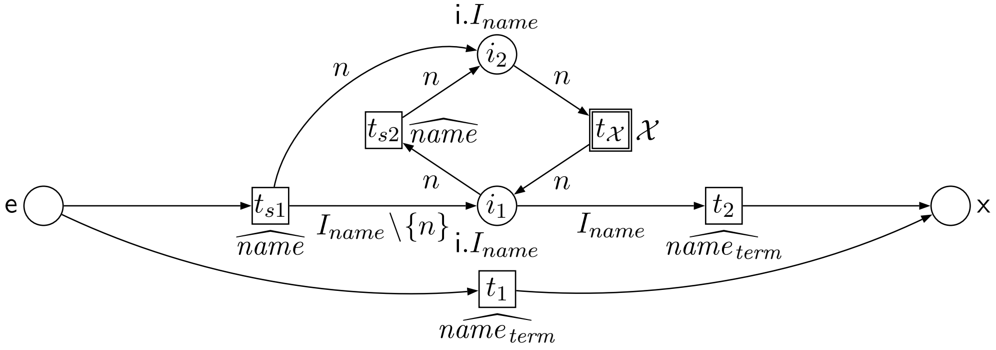


Figure 15. Net part N_{name} of the instance manager of a task where I_{name} is the set of instances identifiers.

we have:

$$\begin{aligned}
\mathsf{PM}(aw_1 \cdots aw_k) = \Big(&\mathsf{PM}(aw_1) \,\square \cdots \square\, \mathsf{PM}(aw_k) \\
&\square \left(\{\widehat{signal}(id, \textsc{suspend}), ID(id), suspend\}.\emptyset \;,\; \emptyset\right) \\
&\square \left(\{\widehat{signal}(id, \textsc{resume}), ID(id), resume\}.\emptyset \;,\; \emptyset\right) \\
&\square \left(\{\widehat{signal}(id, \textsc{kill}), ID(id), abort(id)\}.\emptyset \;,\; \emptyset\right)\Big) \\
&* \left(\{SH_{term}\}.\emptyset, .\emptyset\right) \\
\mathsf{PM}(\mathbf{await}\ sig\ \mathbf{then}\ command) = &\left(\{\widehat{signal}(id, sig), ID(id), suspend\}.\emptyset \;,\; \emptyset\right) \;; \\
&\mathsf{PM}(command) \;;\; (\{resume\}.\emptyset, \emptyset)
\end{aligned}$$

A signal KILL is reacted by action *abort* since not only the body of the task has to be terminated but also the signal handler. This is possible thanks to operation π_a applied to both body and signal handler of the tasks. One may observe that the other signals are also properly handled in P_{task}. In particular, suspension and resuming are realised through operations π'_s applied to the semantics of the body of the task.

Finally, for sending signals, the semantics is simply:

$$\begin{aligned}
&\mathsf{PM}(\mathbf{signal}\ (id, sig)) = (\{signal(id, sig)\}.\emptyset, \emptyset) \\
&\mathsf{PM}(\mathbf{abort}) = (\{abort(\bullet)\}.\emptyset, \emptyset) \\
&\mathsf{PM}(\mathbf{sleep}) = (\{sleep\}.\emptyset, \emptyset)
\end{aligned}$$

8. Conclusion

We presented what is, to the best of our knowledge, a first attempt to provide an algebraic model of Petri nets with abortion and suspension, giving them a concurrent semantics. We defined an algebra of *PM-nets*, as an extension of the algebra of M-nets, with priority relations between their transitions and with additional operations π_s and π_a. These new operators allow one to transform any PM-net into preemptible equivalents. We advocated that these operations can be considered as low-level primitive on the top of which one can build more complex operations. This point of view was illustrated with an example (allowing preemption to be initiated from the outside of a net) and with an application to the modelling of tasks in a parallel programming language. We showed that PM-nets can be considered as a high-level version of so called *priority systems* by defining an unfolding operation which transforms a PM-net into a low-level Petri net with a priority relation on transitions. Then, using results obtained in related areas, we showed that such nets can be transformed into safe Petri nets (without priorities) which retain as much as possible of the concurrency. This transformation leads to very large nets which may probably be intractable in practice, but it shows that safe Petri nets are powerful enough to model preemption with a concurrent semantics. As an illustration, we presented an application of the PM-net model to a treatment of tasks in the parallel programming language, B(PN)^2, which is provided with a concurrent semantics based on Petri nets.

Future works may emphasise the links with real-time, for instance by introducing *causal time*, already defined in [18] for M-nets, at the level of B(PN)^2. This would allow one to express timed systems using statements like delays and deadlines. Thus, this would turn B(PN)^2 into a full featured real-time programming language. It is already planned to extend existing tools (PEP and MARIA) in order to have them dealing with the changes defined in this paper. Another interesting work will be to apply this kind of semantics to other languages. We believe that, in the present state of the development, these ideas can be used to give a semantics for a reasonably rich part of the Ada programming language. In particular, the typing system and tasking features should be the first defined, taking benefits from our experience with B(PN)^2. Access and tagged types may be more difficult to obtain and would need further research on the subject.

References

[1] Berry, G.: Preemption in Concurrent Systems, *FSTTCS'93*, LNCS 761, Springer, 1993.

[2] Berry, G.: The Foundations of Esterel, *Language and Interaction: Essays in Honour of Robin Milner* (G. Plotkin, C. Stirling, M. Tofte, Eds.), MIT Press, 1998.

[3] Best, E.: A Memory Module Specification Using Composable High-level Nets. *Formal Systems Specification*, LNCS 1169, Springer, 1996.

[4] Best, E., Devillers, R., Koutny, M.: *Petri Net Algebra*, Monographs in Theoretical Computer Science, EATCS Series, Springer, 2001.

[5] Best, E., Fraczak, W., Hopkins, R., Klaudel, H., Pelz, E.: M-nets: An algebra of high-level Petri nets, with an application to the semantics of concurrent programming languages, *Acta Informatica*, **35**, 1998.

[6] Best, E., Hopkins, R. P.: B(PN)^2 — A basic Petri net programming notation, *PARLE'93*, LNCS 694, Springer, 1993.

[7] Best, E., Koutny, M.: Petri net semantics of priority systems, *Theoretical Computer Science*, **96**(1), 1992.

[8] Best, E., Wimmel, H.: Reducing k-safe Petri nets to pomset-equivalent 1-safe Petri nets, *ICATPN'2000*, LNCS 1825, Springer, 2000.

[9] Devillers, R., Klaudel, H., Riemann, R.-C.: General parameterised refinement and recursion for the M-net calculus, *Theoretical Computer Science*, to appear.

[10] Esparza, J.: Model checking using net unfoldings, *Science of Computer Programming*, **23**(2–3), 1994.

[11] Esparza, J., Römer, S., Vogler, W.: An improvement of McMillan's unfolding algorithm, *TACAS'96*, LNCS 1055, Springer, 1996.

[12] Genrich, H. J., Lautenbach, K., Thiagarajan, P. S.: Elements of General Net Theory, *Net Theory and Applications*, **84**, 1980.

[13] Grahlmann, B.: Verifying telecommunication protocols with PEP, *RELECTRONIC'95*, Scientific Society for Telecommunications, 1995.

[14] Grahlmann, B., Best, E.: PEP — more than a Petri net tool, *TACAS'96*, LNCS 1055, Springer, 1996.

[15] Halbwachs, N.: *Synchronous Programming of Reactive Systems*, Kluwer Academic Publishers, 1993.

[16] Kishinevsky, M., Cortadella, J., Kondratyev, A., Lavagno, L., Taubin, A., Yakovlev, A.: Coupling Asynchrony and Interrupts: Place Chart Nets, *ICATPN'97*, LNCS 1248, Springer, 1997.

[17] Klaudel, H.: Compositional high-level Petri net semantics of a parallel programming language with procedures, *Science of Computer Programming*, **41**, 2001.

[18] Klaudel, H., Pommereau, F.: Asynchronous links in the PBC and M-nets, *ASIAN'99*, LNCS 1742, Springer, 1999.

[19] Klaudel, H., Pommereau, F.: A concurrent and compositional Petri net semantics of preemption, *IFM'2000*, LNCS 1945, Springer, 2000.

[20] Klaudel, H., Pommereau, F.: A concurrent semantics of static exceptions in a parallel programming language, *ICATPN'01*, LNCS 2075, Springer, 2001.

[21] Latvala, T.: Model-checking LTL properties of high-level Petri nets with fairness constraints, *ICATPN'01*, LNCS 2075, Springer, 2001.

[22] Milner, R.: A calculus of communicating systems, 92, 1980.

[23] Mäkelä, M.: *MARIA: Modular reachability analyser for algebraic system nets*, `http://www.tcs.hut.fi/maria`, 1999.

[24] The PEP Team: PEP Homepage, `http://theoretica.informatik.uni-oldenburg.de/~pep`.