MOBILE NETS

Nadia Busi

Dipartimento di Scienze dell'Informazione, Università di Bologna,
Mura A. Zamboni 7, 40127 Bologna, Italy.

busi@cs.unibo.it

Abstract: A model, arising as a combination of the name managing techniques of the π-calculus with the natural representation of concurrency and locality of Petri nets, is presented.
We exploit the suitability of our model to represent concurrent systems with an evolving structure by means of the Hurried Russian Philosophers case study, an extension of the Dining Philosophers with dynamic reconfiguration features and multiple levels of activity.

INTRODUCTION

Petri nets [8] are widely accepted as the most diffused formalism to represent, in a natural way, the behaviour of concurrent systems. Since their introduction, they have been deeply studied, providing a variety of analysis techniques and making them suitable for industry applications. However, some problems arise when trying to model dynamic, concurrent systems, found e.g. in Computer Supported Cooperative Work; these systems are characterized by an evolving structure: both the number of involved components (due to the dynamic creation of new elements) and the links between components may change during the computation. The difficulties are due to the rigidity intrinsic in Petri nets: they have a fixed structure, that cannot be modified during the computation; hence, when describing a system, we need to specify at the beginning all the components that could be involved in the computation; moreover, their structure and connections cannot be changed. On the other hand, π-calculus [7] has been introduced to deal with dynamically evolving systems: dynamicity is achieved by means of transmission of channel names, combined with a name scoping mechanism. Mobile nets arise as an attempt to combine Petri nets with the name managing techniques of the π-calculus, yielding a formalism suitable to model systems with evolving structure, while retaining the natural representation of concurrency and locality typical of Petri nets.

P. Ciancarini et al. (eds.), *Formal Methods for Open Object-Based Distributed Systems*

A preliminary version of mobile nets was introduced in [1]; since then, their definition has evolved towards a flexible formalism, combining notational elegance with expressive comfort.
The main features of mobile nets can be summarized as follows:

- tokens are (tuples of) place names and transition labels;
- the postset of a transition is not fixed but depends on the colour of consumed tokens; the preset is specified through a list, and the place of an item in the preset may be dynamically specified by the colour of the token consumed by a preceding item.
- the structure of the net can dynamically vary due to the firing of a transition: new subnets can be added and their places can be fused with existing places; some transitions can be removed.

The introduction of output mobility (i.e. the postset is not fixed a priori) is guided by the following intuition: transitions in P/T nets can be regarded as instantaneous (i.e. they can produce tokens in the postset while they consume tokens from the preset), because there is no functional dependence of the postset on the preset. Moving to coloured nets, the colour of tokens in the postset depends on the colour of tokens in the preset; hence, a transition can be seen as composed by three phases: consume the tokens from the preset, compute the colour of the tokens to be produced, produce the tokens in the postset. As tokens in the postset can be produced only after a processing of the colour of the consumed tokens, why not to make also the places in the postset depending on the colour of consumed tokens? In this way we obtain mobile nets. Actually, the processing mechanism is simply substitution of place names, as tokens are coloured with (tuples of) place names. For example, consider the transition $a(x), b(y) \rightarrow x\langle y \rangle$: a token from place a and a token from place b are consumed, and a token – with the same colour as the token consumed from b – is produced in the place specified by the colour of the token consumed from a.
The extension to input mobility (no fixed preset) is motivated by the following reasoning: when considering nets with name unification in the preset (i.e. the transition $a(x), b(x) \rightarrow x$ is enabled only if both places a and b contain a token of the same colour), we have a dependency among the colours of tokens to be consumed; in mobile nets with input mobility we extend this dependency to the places in the preset: the preset is specified through a list (instead of a multiset), and the place of an item in the preset can be specified by the colour of a token of a preceding item. For example, the transition $a(x), x() \rightarrow b$ consumes a token from place a and an (uncoloured) token from the place specified by the colour of the token consumed from place a (and produces an uncoloured token in place b).
To deal with dynamically growing nets, instead of a simple marking, we allow the postset of a transition to be a subnet. When such a transition is fired, the new places and transitions of the postset-net are added to the current net structure; moreover, some of the places of the subnet may be required to be fused with preexisting places.
Regarding the possibility for a net to shrink, it is quite difficult to permit a direct elimination of places, because of the generation of dangling references, corresponding

to the occurrence of the eliminated places names in other parts of the net. For this reason, we allow direct elimination of transitions only; concerning places, the ones no longer used are removed by an implicit form of garbage collection.
The transition erasing mechanism is obtained in the following way: each transition can be equipped with a (not necessarily unique) label; besides the preset and the postset, a transition also has an associated erasing set, that is a set of labels corresponding to the transitions we want to remove in correspondence with its firing.
The implicit garbage collection mechanism relies on the following facts: places that become isolated, i.e., whose name does not occur anywhere in the net, implying that no transition can produce/consume tokens from it, can be safely removed without changing the behaviour of the net; the same holds for empty places, whose name occurs at most in the preset of some transition (in this case, also the transitions having that place in the preset can be removed).
Mobile nets seem promising as a formal basis for distributed object oriented systems: they have the dynamic instantiation and name binding mechanisms demanded for objects, and they have at the same time the locality and mobility mechanisms required for the most general distributed systems.
The paper is organized as follows: after recalling some basic definitions in Section 2, we give a formal definition of mobile nets in Section 3; in Section 4 we sketch the modeling of dynamic reconfiguration of object oriented distributed systems by mobile nets; finally, a solution to the Hurried Russian Philosophers case study is presented.

BASIC DEFINITIONS

In this section we recall the basic definitions of multiset, list and substitution on generic terms.
We denote with ω the set of natural numbers.

Definition 1 Given a set X, a *multiset* over X is a function $m : X \to \omega$.
Let $dom(m) = \{x \in X \mid m(x) > 0\}$.
A multiset is said to be *empty* if $m(x) = 0$ for all $x \in X$.
We write $m \subseteq m'$ if $m(x) \leq m'(x)$ for all $x \in X$.
We write $x \in m$ if $m(x) > 0$.
The *multiset union* is defined as $(m \oplus m')(x) = m(x) + m'(x)$.
The *multiset difference* is defined as

$$(m \setminus m')(x) = \begin{cases} m(x) - m'(x) & \text{if } m(x) > m'(x) \\ 0 & \text{otherwise} \end{cases}$$

A set $A \subseteq X$ can be regarded as multiset over X defined as follows:

$$A(x) = \begin{cases} 1 & \text{if } x \in A \\ 0 & \text{otherwise} \end{cases}$$

Definition 2 Given a set X, a *list* over X is a finite, possibly empty sequence $\bar{a} = a_1, \ldots, a_n$.
The set of lists over X is denoted as X^*; the set of nonempty lists over X by X^+.
The empty list is denoted by ε.
Let $\bar{a} = a_1, \ldots, a_n$ and $\bar{b} = b_1, \ldots, b_k$; list concatenation is defined as
$\bar{a}\bar{b} = a_1, \ldots, a_n, b_1, \ldots, b_k$.

A list $\bar{a}$ can be regarded as a multiset over X defined as follows:
$\bar{a}(x) = |\{i \mid a_i = x \wedge 1 \leq i \leq n\}|$.
The definitions of $dom(\bar{a})$ and $x \in \bar{a}$ can be derived by the corresponding definitions on multiset.

Given a function f on X, we extend it to subsets of X, multisets and lists over X by elementwise application:

Definition 3 Let f be a function on X.
Let $A \subseteq X$; $f(A)$ is the set $\{f(x) \mid x \in A\}$.
Let m be a multiset over X; $f(m)$ is the multiset defined as $f(m)(x) = \sum_{f(y)=x} m(y)$.
Let $a_1, \ldots, a_n$ be a list over X; $f(a_1, \ldots, a_n)$ is the list $f(a_1), \ldots, f(a_n)$.

We recall some basic concepts regarding substitution: they can be applied to any term for which the notions of variable and binder are given. An occurrence of x is said to be free if it does not lie within the scope of a binding occurrence of x; it is said to be bound if it is a binding occurrence or it is not free.

Definition 4 A *substitution* ρ on X is a partial function from X to X with finite domain. We say that ρ is *applicable* to a term if, for all $x \in dom(\rho)$, each free occurrence of x in the term does not lie within the scope of a binder $\rho(x)$. If ρ is applicable to a term t, then the term $t\rho$ is obtained from t by simultaneous substitution of each free occurrence of x with $\rho(x)$, for all $x \in dom(\rho)$. Before performing a substitution, it may be necessary to rename bound names to satisfy the applicability condition.
Let $X_1, \ldots, X_n$ be pairwise disjoint sets; we say that ρ is a *sort preserving substitution* on $X_1, \ldots, X_n$ if

- ρ is a substitution on $X_1 \cup \ldots \cup X_n$
- if $x \in dom(\rho) \cap X_i$ then $\rho(x) \in X_i$, for $i = 1, \ldots, n$

We use the notation $\{x/y\}$ as a shorthand for the substitution $\{(x, y)\}$.
The application of substitution ρ to term t is denoted by $t\rho$.

MOBILE NETS

In the definition of mobile nets we need three different sets of names: for places, transition labels and nets. Due to the binding mechanism and the dynamic generation of new subnets, we need these sets to be denumerable.
Let $\mathcal{S}$ be the set of *place names*; $s, s', \ldots$ range over it;
let $\mathcal{L}$ be the set of *transition labels*; $l, l', \ldots$ range over it; with E, E' we denote finite subsets of $\mathcal{L}$.
Sometimes we need to refer to an element of $\mathcal{S} \cup \mathcal{L}$ without worrying about which set it really belongs to: we will use lowercase letters, such as $a, b, \ldots, x, y, \ldots$, to range over $\mathcal{S} \cup \mathcal{L}$.
Let $\mathcal{N}$ is the set of *net names*; $X, Y, \ldots$ range over $\mathcal{N}$.
We require that $\mathcal{S}$, $\mathcal{L}$ and $\mathcal{N}$ are pairwise disjoint.

Besides the current net structure, the local state is represented by the *marking*, that is the distribution of tokens in places. Tokens are colored with tuples of place names and transition labels, i.e. elements of $(\mathcal{S} \cup \mathcal{L})^*$. The marking is represented by a multiset that, given a place and a token color, tells us the number of tokens of that color occurring in the place. Thus a marking is multiset over $\mathcal{S} \times (\mathcal{S} \cup \mathcal{L})^*$; $m, m', \ldots$ range over markings.
We use the following concrete notation for markings: $a\langle b, c\rangle, b\langle\rangle$ means a token with colour (b, c) in place a and an uncoloured token in place b. We omit the brackets when the colour tuple is empty, thus the marking above can be written as $a\langle b, c\rangle, b$.
The preset of a transition, specifying the tokens to be consumed for the transition to fire, is represented through a list of pairs of place names and token colours, called a *pattern*. A pattern is a non empty list over $\mathcal{S} \times (\mathcal{S} \cup \mathcal{L})^*$; the set of patterns will be denoted by $\mathcal{P}$, with $p, p', \ldots$ ranging over it.
We adopt the following concrete notation for patterns: $a(x, y), x(z), b()$ is a pattern requiring a token coloured with a pair in place a, a token coloured with a single name in the place given by the name of the first element of the colour of the token consumed from a, and an uncoloured token from place b. We can omit the braces from $b()$. $a(x), b(x)$ is a pattern requiring a token in a (coloured with a single element) and a token in b, such that both tokens have the same colour. To avoid confusion, we will not use the same name as both a free and bound name in a pattern, i.e. we avoid to write $x(y), a(x)$.
A *net expression* is a term generated by the following grammar:

N	$::=$	m	a marking
	$\mid$	$l : p, E \rightarrow N$	a transition
	$\mid$	X	a net name
	$\mid$	$(\nu\ x)N$	a place name/transition label abstraction
	$\mid$	Nx	a place name/transition label concretion
	$\mid$	$N \otimes N$	a composition of nets

The set of net expressions is denoted by $\mathcal{N}exp$, with $N, N', \ldots$ ranging over it.
We call *transition* a tuple $t = (l, p, E, N)$ (usually written as $l : p, E \rightarrow N$), where

- $l \in \mathcal{L}$ is the transition label;
- $p \in \mathcal{P}$ is the *preset*, specifying the tokens to be consumed for the transition to fire;
- $E \subseteq \mathcal{L}$ is the set of *erased transitions*: when t fires, all the transitions whose label belongs to E are removed from the net;
- $N \in \mathcal{N}exp$ is the *postset*, specifying the subnet that will be generated by the firing of t.

In the following, we omit the set E when empty and the label l when unnecessary (i.e. no transition will erase it).
We use $t, t', \ldots$ and $T, T', \ldots$ to denote single transitions and sets of transitions respectively.

The binders are name abstractions and tuple colours in patterns: in the name abstraction $(\nu\ x)N$, x is bound and the scope of the binder is N; in transition $a(x), p\rightarrow N$, x is bound and the scope of the binder is $p\rightarrow N$, i.e. the postset and the remaining part of the pattern.
Let $a(\bar{x}), p$ be a pattern; the free and bound names in the pattern are defined as follows:

$$\begin{array}{lcl} fn(a(\bar{x}), p) & = & \{a\} \cup (fn(p) \setminus dom(\bar{x})) \\ fn(\varepsilon) & = & \emptyset \end{array}$$

$$\begin{array}{lcl} bn(a(\bar{x}), p) & = & dom(\bar{x}) \cup bn(p) \\ bn(\varepsilon) & = & \emptyset \end{array}$$

The free and bound names in a net expression are defined as follows:

$$\begin{array}{lcl} fn(m) & = & \{a, dom(\bar{b}) \mid a\langle\bar{b}\rangle \in dom(m)\} \\ fn(l : p, E\rightarrow N) & = & \{l\} \cup E \cup fn(p) \cup (fn(N) \setminus bn(p) \\ fn(X) & = & \emptyset \\ fn((\nu\ x)N) & = & fn(N) \setminus \{x\} \\ fn(Nx) & = & fn(N) \cup \{x\} \\ fn(N \otimes N') & = & fn(N) \cup fn(N') \end{array}$$

$$\begin{array}{lcl} bn(m) & = & \emptyset \\ bn(l : p, E\rightarrow N) & = & bn(p) \cup bn(N) \\ bn(X) & = & \emptyset \\ bn((\nu\ x)N) & = & \{x\} \cup bn(N) \\ bn(Nx) & = & bn(N) \\ bn(N \otimes N') & = & bn(N) \cup bn(N') \end{array}$$

A net N is *closed* if $fn(N) = \emptyset$. A net name X is *guarded* in a net expression N if each occurrence of X in N is inside the postset of a transition.
A *net description* is composed by a principal net and a set of net defining equations:

$$\begin{array}{l} N \\ X_1 = N_1 \\ \ldots \\ X_k = N_k \end{array}$$

where X_i are distinct net names, N_i are closed nets and there are no sequences $X_{i_1} X_{i_2} \ldots X_{i_n}$ such that $1 \leq i_h \leq k$ for $1 \leq h \leq n$, $i_1 = i_n$ and X_{i_h} is unguarded in $N_{i_{h+1}}$ for $1 \leq h < n$. Moreover, each net name occurring in the net description appears in the left hand side of an equation.
A net is in *canonical form* if it is written as $(\nu\ \bar{s})(\nu\ \bar{l})((\otimes_{\ t \in T} t) \otimes m)$, where $\bar{s}$ and $\bar{l}$ are respectively sequences of place names and transition labels, T is a set of transitions and m is a marking.
We will adopt the simpler notation $(\bar{s}, \bar{l}, T, m)$, reminiscent of standard P/T nets, to denote nets in canonical form.
The principal net of a net description can always be transformed into a net in canonical form in the following way:

m	$=$	$(\varepsilon, \varepsilon, \emptyset, m)$	
t	$=$	$(\varepsilon, \varepsilon, \{t\}, \emptyset)$	
X_i	$=$	N_i	$1 \leq i \leq k$
$(\nu\ s_1)(\bar{s}, \bar{l}, T, m)$	$=$	$(s_1 \bar{s}, \bar{l}, T, m)$	if $s_1 \in \mathcal{S} \setminus \bar{s}$
$(\nu\ l_1)(\bar{s}, \bar{l}, T, m)$	$=$	$(\bar{s}, l_1 \bar{l}, T, m)$	if $l_1 \in \mathcal{L} \setminus \bar{l}$
$(s_1 \bar{s}, \bar{l}, T, m)s_2$	$=$	$(\bar{s}, \bar{l}, T\{s_2/s_1\}, m\{s_2/s_1\})$	if $s_1, s_2 \in \mathcal{S} \setminus \bar{s}$
$(\varepsilon, \bar{l}, T, m)s_2$	$=$	$(\varepsilon, \bar{l}, T, m)$	if $s_2 \in \mathcal{S}$
$(\bar{s}, l_1 \bar{l}, T, m)l_2$	$=$	$(\bar{s}, \bar{l}, T\{l_2/l_1\}, m\{l_2/l_1\})$	if $l_1, l_2 \in \mathcal{L} \setminus \bar{l}$
$(\bar{s}, \varepsilon, T, m)l_2$	$=$	$(\bar{s}, \varepsilon, T, m)$	if $l_1 \in \mathcal{L}$
$(\bar{s_1}, \bar{l_1}, T_1, m_1) \otimes$ $(\bar{s_2}, \bar{l_2}, T_2, m_2)$	$=$	$(\bar{s_1}\bar{s_2}, \bar{l_1}\bar{l_2}, T_1 \cup T_2, m_1 \oplus m_2)$	if $\bar{s_1} \cap \bar{s_2} = \emptyset$ and $\bar{l_1} \cap \bar{l_2} = \emptyset$

Since the structure of the net may change during the execution, the current state of the net is no longer represented by the marking only, but by a whole net. To simplify the task, we will use nets in canonical form to represent the current state of the system.
We now introduce some preliminary definitions for the notion of firing of a transition.
Let p be a pattern such that $bn(p) \cap fn(p) = \emptyset$.
A *pattern instantiation* for p is a sort preserving substitution ρ on $\mathcal{S}, \mathcal{L}$ such that $dom(\rho) = bn(p)$.
Let $(a(\bar{x}))[\rho] = a\rho\langle\bar{x}\rho\rangle$.
The *instance of p via* ρ is the multiset corresponding to the list $p[\rho]$, which is obtained by elementwise application of $[\rho]$.
Besides the consumption and production of tokens, the firing of a transition may remove the transitions whose label occurs in the erase set and produce new places and transitions. Given a set of transitions T and a set of transition labels E, let

$$T_{\neg E} = \{t \in T \mid t = l : p, K \rightarrow N \wedge l \notin E\}$$

Let $N_1 = (\bar{x}, \bar{l}, T, m)$ and $t = p, E \rightarrow N$, $t \in T$.
The transition t is *enabled* at N_1 if there exists a pattern instantiation ρ for p such that $p[\rho] \subseteq m$.
The firing of t at N_1 produces the net

$$N_2 = (\nu\ \bar{x}\bar{l})((T_{\neg E}, m \setminus p[\rho]) \otimes N\rho)$$

This is written ad $N_1[t_\rho\rangle N_2$.

OBJECT ORIENTED AND DYNAMIC RECONFIGURATION FEATURES

The representation of object oriented features in mobile nets can be accomplished following the guidelines below. Each class is specified by a subnet; each object occurring in the system is represented by an instance of the class subnet, added at creation time. The object may communicate to other objects a subset of its places, corresponding to public methods. Method invocation is achieved by putting a token in the place corresponding to a method of the server object; that tokens specifies a place where the client expects to obtain the result. Object deletion is modelled by transition erasing and by the garbage collection mechanism.

Now we illustrate how to deal with dynamic reconfiguration of the connections of system components. Take an object that makes use of a service; suppose that the service provider can change during the execution. We envisage two solutions for accessing the current server's methods:

- we put a tuple containing the references in a place:

  $service\langle method_1, \ldots, method_n \rangle$

  a service request is made by putting a token, coloured with data and a place where we expect to obtain the result, in the place corresponding to the required method invocation:

  $\ldots, service(\overline{methods}) \rightarrow method_i\langle data, returnplace\rangle, service\langle \overline{methods} \rangle$

  When we need to update the provider, we store in the service place the references to the methods of the new provider:

  $update(\overline{newmethods}), service(\overline{methods}) \rightarrow service\langle \overline{newmethods} \rangle$

- we isolate the transitions making use of the service and collect them in a subnet; we label each of these transitions with the same name; when an update is requested, we delete these transitions and generate new ones with the correct links to the new provider;

  $SERVICE =$
  $(\nu\ \overline{methods})(\nu\ l)($
  $l : \ldots \rightarrow method_i\langle data, returnplace\rangle \otimes$
  $l : update(\overline{newmethods}), \{l\} \rightarrow SERVICE\overline{newmethods})$

  at the beginning, we create a SERVICE subnet whose transitions are connected to the methods of the initial service provider:

  $SERVICE\overline{initialmethods}$

The first solution requires to get information about the current server each time a service is accessed; on the other hand, the change of the current server is a cheap operation. In the second solution, the update of the server causes a reconfiguration of the involved part of the net; after the reconfiguration, the client is directly connected to the service provider. Hence, the first solution is more suitable in situations where changes of the server are very frequent, whereas the second one can be employed when the server is more stable.

EXAMPLE: HURRIED RUSSIAN PHILOSOPHERS

Hurried Russian Philosophers arise as a combination of two case studies: the Hurried Philosophers [9] and the Russian Philosophers [5]. A Hurried Philosopher can leave the table or invite new philosophers, thus introducing object creation/destruction and

dynamic reconfiguration. When a Russian Philosopher ends eating, he starts dreaming of a Russian Philosopher's dinner, thus introducing multiple levels of activity.
The behaviour of Hurried Russian Philosophers can be described as follows: The number of philosophers in the table may change during the dinner: a philosopher may leave the table or invite new philosophers to join the dinner. When a philosopher remains alone, he can choose to continue the dinner, by inviting new philosophers, or to leave the table, ending the dinner. When a philosopher stops eating, he leaves the two forks and starts thinking to a philosopher's dinner; when this dinner ends, he wakes up and realizes to be hungry, hence tries to get the forks. Each fork can be on the table or used by one of the two philosophers; when a philosopher is asked the fork by his neighbour, he must yield it.
Now we present a description of the Hurried Russian Philosophers using mobile nets.

Behaviour of a philosopher:
A philosopher may ask a new philosopher to sit at his left hand side only if he has his left fork. He can decide to leave the table only if he has both his forks. A philosopher A comes with his right fork; that fork will always be his right fork during the dinner, and he will take it with him when he will leave the table. When inviting a new philosopher B, A tells him to sit at his left side; as B has his right fork, the current left fork of A becomes the left fork of B, and the right fork of B becomes the new left fork of A. So, the philosopher's left fork only may change when other philosophers join or leave the dinner.

Description of the net corresponding to a fork:
a fork is composed of four places, and at most one of them is not empty.

free a token in this place means that the fork is on the table

usedLP⟨LPreq⟩ the fork is used by the philosopher on his left side; a fork request has to be produced in place **LPreq**

usedRP⟨RPreq⟩ the fork is used by the philosopher on his right side

update⟨newfree, newLPreq, newRPreq, newupd, ack⟩ place used by the philosopher on the left when leaving, to communicate the new left fork places for his right neighbour.

Description of the net corresponding to a philosopher:
the states of a philosopher are represented by corresponding places: when a philosopher reaches the table, he is **hungry**; he realises that for eating he needs both his left and right forks and moves to the state **wantLF**, **wantRF**. Once he has **obtainedLF** and **obtainedRF**, he can start **eating**. When satiated, he wants to get rid of the forks, and reaches the state **leaningLF**, **leaningRF**. When he has **leanedLF** and **leanedRF**, he starts thinking to a dinner of philosophers; when that dinner ends, the philosopher wakes up and realizes to be **hungry**.
To manage the right fork, we use the following places:

freeRF fused with place **free** of the fork on his right. A token in that place means that the right fork is on the table.

hasRF fused with place **usedLP** of his right fork. A token means that the philosopher holds his right fork.

busyRF fused with place **usedRP** of his right fork. A token means that the right fork is held by the philosopher on his right.

updatingRF fused with place **update** of his right fork. When leaving, the philosopher uses this place to communicate to his right neighbour the information on his new left fork.

RFreq the place where the philosopher on the right asks for the common fork, putting a token with colour ⟨**who, where**⟩, meaning that he expects to receive a token in place **where**

and the following transitions:

wantRF, freeRF → **hasRF**⟨**RFreq**⟩ if the philosopher wants the fork, and it is free, he takes the fork.

wantRF, busyRF(neighReq) → **neighReq**⟨**RFreq, RFobtained**⟩ if the philosopher wants the fork and it is used by his right neighbour, he produces a fork request.

RFobtained, RFreq(who, where) → **wantRF, where, busyRF**⟨**who**⟩ if the philosopher has the fork, is not eating, and receives a request for the fork, he yields the fork to the requiring philosopher.

leaningRF, hasRF(_) → **leanedRF, freeRF** if the philosopher wants to lean the fork and there is non fork request, he leans it on the table.

leaningRF, RFreq(who, where) → **leanedRF, where, busyRF**⟨**who**⟩ if the philosopher wants to lean the fork and his neighbour required it, then he yields it to the neighbour.

The net corresponding to a philosopher is reported below (the subnet LFMANAGER is devoted to manage the left fork, invite a new philosopher, leave the table and finish the dinner).

```
PHIL =
 (ν   endDinner,
      freeLF, hasLF, busyLF, updatingLF,
      freeRF, hasRF, busyRF, updatingRF
      wantLF, obtainedLF, leaningLF, leanedLF, LFreq,
      wantRF, obtainedRF, leaningRF, leanedRF, RFreq,
      hungry, eating)(
```

% phil behaviour

```
hungry → wantLF, wantRF ⊗
obtainedLF, obtainedRF → eating ⊗
eating → leaningLF, leaningRF ⊗
leanedLF, leanedRF → TABLE hungry ⊗
```

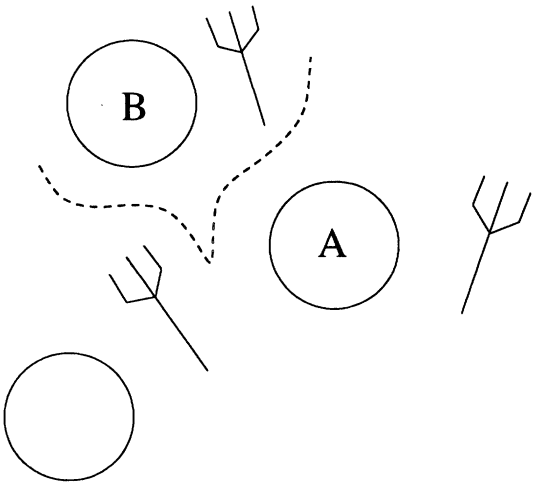

Figure 1.1 Inviting a new philosopher.

% right fork managing
wantRF, freeRF → hasRF⟨RFreq⟩ ⊗
wantRF, busyRF(neighReq) → neighReq⟨RFreq, RFobtained⟩ ⊗
RFobtained, RFreq(who, where) →
wantRF, where, busyRF⟨who⟩ ⊗
leaningRF, hasRF(_) → leanedRF, freeRF ⊗
leaningRF, RFreq(who, where) →
leanedRF, where, busyRF⟨who⟩ ⊗

% left fork managing, invite new phil, leave table
LFMANAGER freeLF hasLF busyLF updatingLF ⊗

% initial state
hungry)

The net LFMANAGER manages all the philosopher activities dealing with the left fork; as the fork on the left of a philosopher may change dynamically during the dinner, due to the addition/leaving of neighbouring philosophers, all transitions in LFMANAGER are labelled with label **l**; the transitions dealing with an update of the left fork (namely, the ones dealing with the invitation of a new philosopher and with a request of updating from a leaving left neighbour) remove all transitions labelled with **l** and create a new instance of LFMANAGER with the right links to the new left fork.
The transitions for left fork managing are the same as for right fork in the net PHIL.

Invite a new philosopher (see Figure 1.1):
The philosopher (A) may invite a new philosopher (B) on his left hand side after ending eating and getting rid of his left fork (oldF). If the left fork has been leaned, then:

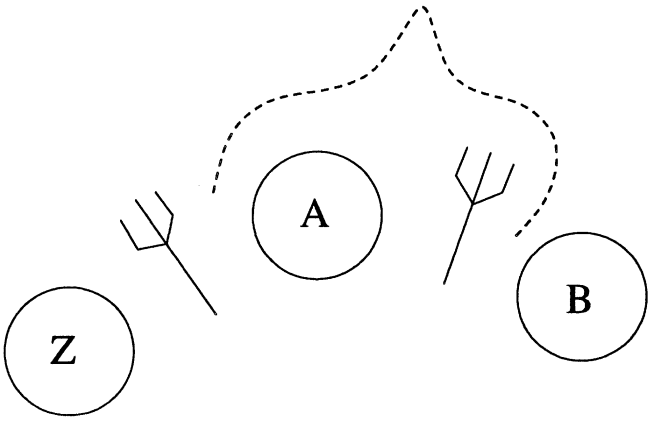

Figure 1.2 Leaving the table.

- all transitions in LFMANAGER are erased;
- four new places corresponding to a new fork (newF) are created;
- a token is produced in the place of newF representing the fact that the fork is on the table;
- a PHIL net, corresponding to B, is created: his left fork is oldF and his right fork is newF;
- an updated instance of LFMANAGER is created, so that the new left fork of A becomes newF.

Leave the table (Figure 1.2):
The philosopher (A) may leave the table after ending eating and when still holding both forks. If there are no requests for the left fork, then:

- a new place **ack** is created;
- an updating is required to his right neighbour (B): B can no longer use the right fork of A as left fork, because it will be removed, and the new left fork for B becomes the left fork of A;
- when B confirms he has performed the update, by putting a token in **ack**, two cases can happen:
  - if there are no requests for the right fork, then the left fork is leaned on the table
  - otherwise, the left fork (turning out to be the new left fork for B) is given to B.

If there is a request for the left fork, the behaviour is the same, but when B puts a token in **ack**:

- if there are non requests for the right fork, then the left fork is given to the left neighbour (Z)

- if both B and Z want the left fork, the fork is given to Z, and the fork request by B is forwarded to Z

Update left fork:
if a request to update the left fork is received, then all transitions in LFMANAGER are erased, a new instance of LFMANAGER with the new pointers to the left fork is created and an ack is sent to the philosopher requiring the updating.

End of the dinner:
the place **equal** is used to store the names of the places **free** of the left and right fork respectively. If these names are equal, meaning that the philosopher is left alone at the dinner, he can decide to leave the table, thus ending the whole dinner; this is signaled by producing a token in the place **endDinner**

```
LFMANAGER =
 (ν   freeLF, hasLF, busyLF, updatingLF,
      equal)
 (ν   l)(
% left fork managing
 l:  wantLF, freeLF → hasLF⟨LFreq⟩ ⊗
 l:  wantLF, busyLF(neighReq) →
     neighReq⟨LFreq, LFobtained⟩
 ⊗
 l:  LFobtained, LFreq(who, where) →
     wantLF, where, busyLF⟨who⟩
 ⊗
 l:  leaningLF, hasLF(_ ) → leanedLF, freeLF ⊗
 l:  leaningLF, LFreq(who, where) →
     leanedLF, where, busyLF⟨who⟩
 ⊗
% invite new phil
l:  leanedLF, {l} →
       (ν newfree, newusedLP, newusedRP, newupd)(
       newfree ⊗
       PHIL endDinner freeLF hasLF busyLF updatingLF
            newfree newusedLP newusedRP newupd⊗
       LFMANAGER newfree newusedRP newusedLP newupd) ⊗
% leaving table
l:  leaningLF, leaningRF, hasLF(_ ) →
```

```
        (ν ack)(
        updatingRF⟨freeLF,hasLF,busyLF,updatingLF, ack⟩ ⊗
        ack, hasRF(_ )→ freeLF ⊗
        ack, RFreq(who, where) →where, busyLF⟨who⟩ ) ⊗
1:   leaningLF, leaningRF, LFreq(whoL, whereL) →
        (ν ack)(
        updatingRF⟨freeLF,hasLF,busyLF,updatingLF, ack⟩ ⊗
        ack, hasRF(_ )→whereL, busyLF⟨whoL⟩ ⊗
        ack, RFreq(whoR, whereR) →
        whereL, whoL⟨whoR, whereR⟩ ) ⊗
```

% updating left fork

```
1:   updatingLF(free, has, busy, upd, ack), {1} →
        (LFMANAGER free has busy upd ⊗
        ack) ⊗
```

% end dinner

```
1:   equal(free, free) →endDinner ⊗
equal⟨freeLF, freeRF⟩ )
```

Description of a table of n philosophers:
a token in place **endDinner** means that the last philosopher left the table and the dinner is finished. Note that, when a TABLE is created by a philosopher that is thinking to it, the place **endDinner** is fused with the place **hungry** of the philosopher; hence, when the dinner he was thinking finishes, the philosopher wakes up and realizes to be hungry. The definition of the net TABLE consists in creating the places corresponding to n new forks and n PHIL nets, whose endDinner place is fused with the endDinner place of TABLE, and with the appropriate left and right forks.

```
TABLE =
 (ν   endDinner,
      % fork 1
      free1, usedLP1, usedRP1, upd1,
      % fork 2
      free2, usedLP2, usedRP2, upd2,
      ...,
      % fork n
      freen, usedLPn, usedRPn, updn)(
% philosopher 1
PHIL endDinner free1 usedLP1 usedRP1 upd1 free2 usedLP2
usedRP2 upd2
⊗ ...⊗
% philosopher n
PHIL endDinner freen usedLPn usedRPn updn free1 usedLP1
usedRP1 updn)
```

CONCLUSION

In this paper we presented a model suitable to represent systems with an evolving structure. In the last years several extensions of Petri nets, dealing with dynamicity, have been proposed.
Most of these extensions are devoted to add object oriented features on top of (coloured) Petri nets (see e.g. [2, 3, 4, 5, 6, 9, 11]): the object structure is represented through a net, and dynamic reconfiguration features are obtained by some mechanism external to the net. There are two main differencies w.r.t our approach: we think that our model lies at a lower level of abstraction, because the object oriented features are not embedded in mobile nets, but they need to be encoded; secondly, our aim is to embody dynamicity in our model, and not to add it by an external structure.
Another extension, more similar in spirit to our model, is represented by Self Modifying Nets [10]; in this case, the main difference regards the "locality" of transitions: while in Self Modifying nets the pre and post sets of a transition depend on the whole marking of the net, in mobile nets they depend only on the colour of the consumed tokens.
A formal comparison of the expressive power of these models deserves further investigation.
Another interesting topic consists in the development of analysis techniques for (subclasses of) mobile nets.

Acknowledgments

The author thanks Lorenzo Capra and Philippe Darondeau for their useful comments and Alfredo Chizzoni for suggesting the Hurried Russian Philosophers example.

References

[1] A. Asperti, N. Busi, "Mobile Petri Nets", Technical Report UBLCS-96-10, dept. of Computer Science, University of Bologna, Italy, 1996.

[2] E. Battiston, A. Chizzoni, F. De Cindio, 'Inheritance and Concurrency in CLOWN", in Proc. First Workshop on Object-Oriented Programming and Models of Concurrency, G. Agha and F. De Cindio eds., Turin, June 1995.

[3] D. Buchs, N. Guelfi, "CO-OPN: a concurrent object-oriented Petri nets approach" , in Proc. 12th Int. Conf. on Appl. and Theory of Petri Nets, Gjern, June 1991.

[4] J. Engelfriet, G. Leih, G. Rozenberg, "Net based description of parallel object-based systems, or POTs and POPs", LNCS 489, pp.229-273, Springer , 1991.

[5] C. A. Lakos, "Object Petri Nets – Definition and Relationship to Coloured Nets", Technical Report TR94-3, Computer Science Department, University of Tasmania, 1994.

[6] C. A. Lakos, "From Coloured Petri Nets to Object Petri Nets", in Proc. 16th Int. Conf. on Appl. and Theory of Petri Nets, LNCS 935, pp. 278-297, Springer Verlag, Turin, Italy, 1995.

[7] R. Milner, J. Parrow, D. Walker, "A Calculus of Mobile Processes", in Information and Computation 100, pp.1-77, 1992.

[8] C. A. Petri, *Kommunikation mit Automaten*, PhD Thesis, Institut für Instrumentelle Mathematik, Bonn, Germany, 1962.

[9] C. Sibertin-Blanc, "Cooperative Nets", in Proc. 15th Int. Conf. on Appl. and Theory of Petri Nets, LNCS 815, pp.471-490, Springer Verlag, Zaragoza, Spain, 1994.

[10] R. Valk, "Generalizations of Petri Nets", in Proc. MFCS'81, LNCS 118, pp. 140-155, Springer Verlag, 1981.

[11] R. Valk, "On Processes of Object Petri Nets", Bericht Nr. 185, Fachbereich Informatik, Universitat Hamburg, 1996.