A Distributed Implementation of Mobile Nets as Mobile Agents

Nadia Busi and Luca Padovani

University of Bologna, Department of Computer Science
Mura Anteo Zamboni 7, 40127 Bologna, Italy
{busi,lpadovan}@cs.unibo.it

Abstract. Mobile nets arise as a combination of the name managing techniques of the π-calculus with the representation of concurrency and locality of Petri nets. We propose MAGNETs, a variant of mobile nets that are suitable for an effective, distributed implementation. Such implementation extends an implementation of the Join calculus virtual machine with dynamic reconfiguration features.

1 Introduction

Service Oriented Computing is an emerging paradigm for developing autonomous computational elements in a wide-area, distributed setting. Web Services provide an important instantiation of such paradigm, and are supported by a number of standardized technologies for specifying service interfaces, interaction between services, service discovery, service composition and orchestration.

In this context, a variety of languages and infrastructures for Web Services have been recently proposed by leading consortia of industries and organizations. However, before the Service Oriented Computer paradigm becomes reality, there is a number of challenging issues that need to be addressed, including a formal framework for the rigorous specification of their semantics, as well as prototype implementations.

Petri nets [16] and the π-calculus [15] are the most prominent candidates for the definition of a formal semantics of Web Services infrastructures [1].

Petri nets are a widespread formalism for the representation of the behavior 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, they fall short in modelling the dynamic features arising in systems for Service Oriented Computing: these systems are characterized by an evolving structure, where both the number of the involved components and the links between them may change during the computation. The π-calculus [15] has been specifically introduced to deal with dynamically evolving systems: dynamicity is achieved by means of transmission of channel names, in combination with a name scoping mechanism. A modeling weakness of the π-calculus is the lack of support for advanced synchronization patterns, for which Petri nets provide a direct, natural representation [1].

Mobile nets [2, 7] arise as an attempt to combine Petri nets with the name managing techniques of the π-calculus, yielding a formalism that is suitable to modelling systems with an evolving structure, while retaining the natural representation of concurrency and locality typical of Petri nets.

M. Steffen and G. Zavattaro (Eds.): FMOODS 2005, LNCS 3535, pp. 259–274, 2005.

From a practical point of view, both the π-calculus and Petri nets (hence Mobile nets also) exhibit complex conflict and contact situations that hinder an effective distributed implementation. In the context of mobile calculi, the Join calculus [9] solves this problem by providing a local management of conflicting reaction rules.

The aim of this paper is to provide MAGNETs, a variant of Mobile nets that is suitable for a distributed implementation. The peculiar properties of MAGNETs are the following: we prevent the need to resolve distributed conflicts by enforcing locality of transition guards in the model, and we enable a constrained form of mobility that preserves the locality property. We support the effectiveness of MAGNETs by providing a distributed implementation, which is based on an enhancement of the Join calculus virtual machine developed in [17].

The paper is organized as follows: in the next section we present an informal overview of MAGNETs; the syntax and the semantics are presented in Section 3; the features of these nets are further explored by two examples in Section 4, while the distributed implementation is described in Section 5. We conclude with a comparison of related literature.

2 An Overview of MAGNETs

A system is modeled as a dynamic pool of net systems, each net system being essentially a mobile net. The pool is dynamic in the sense that new net systems may be created during the computation. Each net system is a triple with three components: a set of *places*, a set of *transitions*, and a *marking*. As for classical Petri nets, the marking, i.e., the distribution of tokens in the places, represents the current state of the system, and the system evolution is represented by firing, i.e., execution of the transitions.

Tokens coloring. As in Colored Petri Nets [11], tokens carry information; the "color" of a token is a (possibly empty) tuple of (place) names, either referring to local places of the nets or to remote places, belonging to another net of the configuration. The main differences between Mobile Nets and classical and Colored Petri Nets regard transitions, whose key features are listed below.

Local versus remote tokens. A transition can produce tokens for local places (places belonging to the net system where the transition resides) as well as for remote places (places belonging to a different net system). Consider the configuration $C_1 = N_a, N_b$, consisting of two mobile nets N_a and N_b, which are defined as follows:

$$N_a = [\{a_1, a_2\}, \{a_1 \mapsto a_2 b_1\}, a_1]$$
$$N_b = [\{b_1, b_2, b_3\}, \{b_1 \mapsto b_2, b_2 \mapsto b_3 a_1\}, b_3 b_3]$$

The net N_a consists of two places, a_1 and a_2, a transition $a_1 \mapsto a_2 b_1$ and an initial marking containing one uncolored token (or, equivalently, colored with the empty tuple) in place a_1. The net N_b consists of three places, two transitions, and an initial marking with two tokens in place b_3. The transition $t_a = a_1 \mapsto a_2 b_1$ in N_a is enabled in the current marking of N_a: when t fires, the token in a_1 is consumed, and two new tokens are produced: one is produced locally in place a_2 whereas the other is a token for the

remote place b_1, belonging to N_b. When the token b_1 reaches the net N_b, the transition $t_b = b_1 \mapsto b_2$ becomes enabled (whereas the other transition of N_b, $t'_b = b_2 \mapsto b_3 a_1$, is still disabled); transition t_b has only a local effect on the marking of N_b, whereas transition t'_b also produces a token for a remote place of net N_1.

Mobility. Besides being used to determine the color of the tokens produced by a transition – as in colored nets – the color of the tokens consumed by a transition can be used to determine the place where the transition puts the produced tokens. Consider the configuration $C_2 = N_a, N_b$, consisting of the following nets:

$$N_a = [\{a_1, a_2\}, \{a_1(x) \mapsto x\langle b_2\rangle\}, a_1\langle a_2\rangle a_1\langle b_1\rangle]$$
$$N_b = [\{b_1, b_2\}, \{b_1(x) \mapsto b_2\langle x\rangle\}, \emptyset]$$

The marking of N_a contains the tokens $a_1\langle a_2\rangle$ and $a_1\langle b_1\rangle$, hence the transition $t_a = a_1(x) \mapsto x\langle b_2\rangle$ can consume either the token $a_1\langle a_2\rangle$, thus producing a local token $a_2\langle b_2\rangle$ (i.e., a token colored with b_2 in the local place a_1), or it can consume the token $a_1\langle b_1\rangle$, thus producing a remote token $b_1\langle b_2\rangle$ for the net N_b.

Dynamicity. The last key feature of MAGNETs regards their dynamicity: not only the marking, but also the structure of the mobile nets may vary during the computation. A change in the structure of the nets is carried out by transitions: a transition can add a new transition (either to the net which it belongs to or to a remote net), or spawn a new (part of a) net. More precisely, the firing of a transition may produce either a new mobile net, or a set of places and transitions to be added to an existing net. Consider the configuration $C_3 = N_a$, where

$$N_a = [\{a_1, a_2\}, \{a_1 \mapsto \{\{b_1, b_2\}, \{b_1 \mapsto b_2 a_1\}, b_1\}, a_1]$$

When transition $t = a_1 \mapsto \{\{b_1, b_2\}, \{b_1 \mapsto b_2 a_1\}, b_1\}$ fires, the token in place a_1 is consumed and the new mobile net

$$N_b = [\{b_1, b_2\}, \{b_1 \mapsto b_2 a_1\}, b_1]$$

is added to the configuration. In other words, configuration C_3 evolves to configuration N'_a, N_b, where N'_a is the evolution of net N_a after the firing of transition t, that removes token a_1 from the marking of N_a:

$$N'_a = [\{a_1, a_2\}, \{a_1 \mapsto \{\{b_1, b_2\}, \{b_1 \mapsto b_2 a_1\}, b_1\}, \emptyset]$$

Consider now the configuration $C_4 = N_a, N_c$, where

$$N_a = [\{a_1, a_2\}, \{a_1 \mapsto \{\{b_1, b_2\}, \{b_1 c_1 \mapsto b_2 a_1\}, b_1\}, a_1]$$
$$N_c = [\{c_1\}, \{c_1 \mapsto c_1\}, c_1]$$

The firing of transition $t_a = a_1 \mapsto \{\{b_1, b_2\}, \{b_1 c_1 \mapsto b_2 a_1\}, b_1\}$ produces a so called pre–net $N = \{\{b_1, b_2\}, \{b_1 c_1 \mapsto b_2 a_1\}, b_1\}$; such a pre–net has a transition $t_b = b_1 c_1 \mapsto b_2 a_1$ whose pattern contains both a place of the pre–net and a place of the existing net N_c; hence, the places and the transitions of the pre–net N will be added

to the net N_c. The configuration C_4 evolves to the configuration N'_a, N'_c, where N'_a is obtained from N_a by removing the consumed token a_1, and N'_c is obtained from N_c by adding the places, transitions and tokens in the pre–net N:

$$N_a = [\{a_1, a_2\}, \{a_1 \mapsto \{\{b_1, b_2\}, \{b_1 c_1 \mapsto b_2 a_1\}, b_1\}, \emptyset]$$
$$N_c = [\{c_1, b_1, b_2\}, \{c_1 \mapsto c_1, b_1 c_1 \mapsto b_2 a_1\}, c_1 b_1]$$

Locality and linearity of patterns. One of the peculiar differences of MAGNETS w.r.t. the previous definitions of Mobile Nets [2, 7] consists in the locality and linearity of patterns. Patterns locality means that – given a transition $t = p \mapsto N'$ belonging to a net system $N = [S, T, m]$ – all the place names in the pattern p are local to N; in other words, each place from which transition t consumes tokens belongs to the set S of local places of the net system. This feature turns out to be crucial in the implementation of MAGNETS, as the conflict between transitions competing for the same resource can be resolved locally, with no need for an agreement between the distributed components of the system.

The behavior of a transition that produces a pre–net is driven by the locality requirement on patterns. If each transition in a pre–net $\{S', T', m'\}$ consumes tokens only from local places belonging to S', then the pre–net will evolve in a stand-alone net system (see configuration C_3 above). On the other hand, if some transition in the pre–net requires token consumption from a remote place (i.e., a place not in S'), and all the remote places occurring in patterns of transitions in T' belong to the same net system $N = [S, T, m]$, then the components of the pre–net are added to the net system N, which roughly evolves to the net system $[S \cup S', T \cup T', m \cup m']$ (see configuration C_4). If the remote places occurring in the patterns of transitions in T' belong to at least two different net systems, there is no way to add the components of the pre–net to an existing net system, while respecting the locality condition and avoiding to merge the two existing, distinct nets into a single one. In such a case, the pre–net $\{S', T', m'\}$ can neither evolve to a stand-alone net system, nor being added to an existing net system; hence, the transitions in T' will never fire and the tokens in m' will never become available. As an example, consider the configuration $C_5 = N_a, N_c$, where

$$N_a = [\{a_1, a_2\}, \{a_1 \mapsto \{\{b_1, b_2\}, \{a_1 c_1 \mapsto b_2\}, a_1\}, a_1]$$
$$N_c = [\{c_1\}, \{c_1 \mapsto c_1\}, c_1]$$

When transition $t = a_1 \mapsto \{\{b_1, b_2\}, \{a_1 c_1 \mapsto b_2\}, a_1\}$ fires, the following pre–net is produced:

$$\{\{b_1, b_2\}, \{a_1 c_1 \mapsto b_2\}, a_1\}$$

Note that, for transition $u = a_1 c_1 \mapsto b_2$ to fire, both a token from place a_1 of the net N_a and a token from place c_1 of net N_c are necessary. Hence, such a pre–net will never evolve to a (part of a) net system.

Pattern linearity requires that the bound names in a pattern are pairwise different. Linearity simplifies the implementation of the firability check for a transition; for linear patterns, this check can be carried out independently on each place in the pattern. Consider the transition $a(x)b(y) \mapsto c\langle x, y\rangle$: the firability check consists in looking for a token in place a and a token in place b (both colored with a single place name). On

the other hand, if we consider the nonlinear pattern of transition $a(x)b(x) \mapsto c\langle x, y\rangle$, the firability check becomes more involved, as it amounts to finding two tokens, one in place a and the other one in place b, with the *same* color.

3 Syntax and Semantics

We start with some auxiliary definitions. The names for the places in the nets are taken from a denumerable set. Markings and quasi–patterns are essentially multisets: a marking represents the distribution and the color of tokens in the places of a net, whereas quasi–patterns are an auxiliary notion useful for defining patterns: a pattern is a quasi–pattern for which the linearity condition holds.

Definition 1. *Let $\mathcal{S}$ be a denumerable set of* place names*; $s, s', \ldots, a, b, \ldots, x, y, \ldots$ range over $\mathcal{S}$, while S, S' range over sets of place names. Let $\mathcal{S}^*$ be the set of sequences over $\mathcal{S}$; $\bar{s}, \bar{s'} \ldots, \bar{a}, \bar{b}, \ldots, \bar{x}, \bar{y}, \ldots$ range over $\mathcal{S}^*$.*

A quasi–pattern *is generated by the following grammar:*

$$p \ ::= \ a(\bar{x}) \mid p \oplus p$$

A marking *is generated by the following grammar:*

$$m \ ::= \ \emptyset \mid a\langle \bar{b} \rangle \mid m \oplus m$$

The definition of linearity requires a definition of the *bound names* occurring in a quasi–pattern:

Definition 2. *The set of* bound names *in quasi–patterns is defined as follows:*

$$bn(a(x_1 \ldots x_n)) = \{x_1, \ldots, x_n\}$$
$$bn(p \oplus p') = bn(p) \cup bn(p')$$

The set of linear quasi–patterns *is the least set satisfying the following conditions:*

- *$a(x_1 \ldots x_n)$ is a linear quasi–pattern if $\forall 1 \leq i, j \leq n$: $x_i = x_j$ implies $i = j$;*
- *$p \oplus p'$ is a linear quasi–pattern if p and p' are linear quasi–patterns and $bn(p) \cap bn(p') = \emptyset$.*

We are now ready to introduce the main notions of MAGNETS: patterns, transitions, net expressions, net systems and configurations. A transition is a pair composed by a pattern – specifying the tokens to be consumed – and a net expression – specifying the tokens and the new components that are produced when the transition fires. A net expression is essentially a multiset, whose elements are tokens, transitions, and pre–nets. A net system represents a single component of the system; a net system is a triple composed by the set of local places, the set of transitions and the current marking of the local places. A (system) configuration represents the distributed system, and consists of a multiset of net systems.

Definition 3. *A* pattern *is a quasi–pattern that is linear. A* transition *has the form* $t ::= p \mapsto N$ *where* p *is a pattern and* N *is a net expression. We use* $T, T', \ldots$ *to range over sets of transitions. A* net expression *is generated by the following grammar:*

$$\begin{array}{rll} N ::= & \emptyset & \textit{the empty net} \\ \mid & a\langle \bar{b} \rangle & \textit{a token} \\ \mid & t & \textit{a transition} \\ \mid & \{S, T, m\} & \textit{a pre–net} \\ \mid & N \oplus N & \textit{a composition of nets} \end{array}$$

A net system *is a triple* $[S, T, m]$*, where* S *is the set of places of the net,* T *is the set of transitions and* m *is the current marking. A* system configuration *is generated by the following grammar:*

$$C ::= [S, T, m] \mid N \mid C \oplus C$$

The notions of bound (bn) and free (fn) names occurring in (components of) configurations are as usual and we omit them here for brevity. The set of *place names* in a pattern is the set of places from which the pattern consumes tokens. The set of place names in a marking is the set of places that are not empty in such a marking. Finally, the set of place names in the preset of a transition is the set of places from which the transition consumes tokens, i.e., is the set of place names in the pattern of the transition.

Definition 4. *The set of* place names *in patterns and markings is defined as follows:*

$$\begin{array}{rl} \mathit{places}(a(\bar{x})) = & \{a\} \\ \mathit{places}(p \oplus p') = & \mathit{places}(p) \\ & \cup\, \mathit{places}(p') \end{array} \qquad \begin{array}{rl} \mathit{places}(\emptyset) = & \emptyset \\ \mathit{places}(a\langle b \rangle) = & \{a\} \\ \mathit{places}(m \oplus m') = & \mathit{places}(m) \\ & \cup\, \mathit{places}(m') \end{array}$$

The set of place names in the preset of a transition *(of a set of transitions) is*

$$\mathit{pre}(p \mapsto N) = \mathit{places}(p) \qquad \mathit{pre}(T) = \cup_{t \in T} \mathit{pre}(t)$$

To lighten the definition of the semantics, we reason up to the structural congruence $\equiv$, defined as the least congruence satisfying the commutative monoidal laws for the (overloaded) composition operator $\oplus$.

Definition 5. *Let* $\equiv$ *be the least congruence over configurations (net expressions, patterns, markings) satisfying the following axioms:*

$$C \oplus \emptyset \equiv C \qquad C \oplus C' \equiv C' \oplus C \qquad C \oplus (C' \oplus C'') \equiv (C \oplus C') \oplus C''$$

To lighten the notation, we usually drop the composition operator. E.g., $a(x)\ b(y)$ denotes $a(x) \oplus b(y)$. We also drop the token color, if this is the empty tuple. E.g., $ab(\bar{y})$ denotes $a()b(\bar{y})$, and abb denotes $a\langle\rangle b\langle\rangle b\langle\rangle$.

We introduce a notion of *well–formedness* for net systems and configurations; this notion is needed for a correct definition of the semantics. The set of places of a net system consists of the places that are managed locally in the net, i.e., the tokens in such

places are consumed only by local transitions. Thus, a net system $[S, T, m]$ is well–formed if all the tokens in its marking and the place names in its transitions belong to the set of places S. A configuration is well–formed if its net system components are well–formed, the set of places in its components are pairwise disjoint, and all the free place names occurring in tokens, transitions or pre–nets that are moving towards their destination are names occurring in the set of places of some component. Well–formedness of configurations ensures that each place in the configuration is managed by a unique net system; hence, the destination of a remote token, transition or pre–net is uniquely determined.

Definition 6. *A net system* $[S, T, m]$ *is* well–formed *if* $places(m) \subseteq S$ *and* $pre(T) \subseteq S$. *A system configuration* C *is* well–formed *if* $C \equiv N \oplus \bigoplus_{i=1}^{k} [S_i, T_i, m_i]$, *and the following conditions are satisfied:*

- $\forall i : 1 \leq i \leq k \Rightarrow [S_i, T_i, m_i]$ *is well–formed, and*
- $S_i \cap S_j = \emptyset$ *for* $1 \leq i, j \leq k$, $i \neq j$, *and*
- $fn(N) \subseteq \bigoplus_{i=1}^{k} S_i$

In the following, we assume that the net systems and the configurations we deal with are well–formed.

We recall the standard notion of substitution, that will be used in the semantics:

Definition 7. *A* substitution ρ *on* $\mathcal{S}$ *is a partial function from* $\mathcal{S}$ *to* $\mathcal{S}$ *with finite domain (i.e., the set* $dom(\rho) = \{x \mid \exists y : (x, y) \in \rho\}$ *is finite).*

We say that a substitution ρ *is* applicable *to a net expression if, for all* $x \in dom(\rho)$, *each free occurrence of* x *in the expression does not lie within the scope of a binder* $\rho(x)$. *If* ρ *is applicable to a net expression* N, *then the application of* ρ *to the expression* N *– notation* $N\rho$ *– is obtained from* N *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 perform alpha conversion to satisfy the applicability condition.*

We use the notation $\{x/y\}$ *as a shorthand for the substitution* $\{(x, y)\}$.

To define the firing rule, we need the following definitions. A pattern instantiation for a pattern is a substitution that renames the bound names of the pattern (i.e., the placeholders for token colors). An instance of a pattern is a multiset satisfying the requirements specified by the patterns, i.e., containing the exact number of tokens, and such tokens are decorated with a tuple of the right length.

Definition 8. *A* pattern instantiation *for a pattern* p *is a substitution* ρ *on* $\mathcal{S}$ *such that* $dom(\rho) = bn(p)$. *The* instance of p via ρ, *denoted by* $p[\rho]$ *is the marking defined as*

$$a(x_1 \ldots x_n)[\rho] = a\langle \rho(x_1) \ldots \rho(x_n) \rangle$$
$$(p \oplus p')[\rho] = p[\rho] \oplus p'[\rho] \ .$$

For example, the substitution $\rho = \{a/x, b/y, b/z\}$ is a pattern instantiation for pattern $p = c(x, y)d(z)$, and $c\langle a, b\rangle d\langle b\rangle$ is the instance of p via ρ.

Now we are ready to define the semantics of configurations. The reduction relation is the least relation satisfying the axioms and rules reported in Table 1.

Table 1. Operational semantics of MAGNETS.

$$(1)\quad \frac{p \to N \in T}{[S,T,m \oplus p[\rho]] \to [S,T,m] \oplus N\rho} \qquad (2)\quad \frac{s \in S}{[S,T,m] \oplus s\langle \bar{s} \rangle \to [S,T,m \oplus s\langle s \rangle]}$$

$$(3)\quad \frac{places(p) \subseteq S}{[S,T,m] \oplus (p \mapsto N) \to [S, T \cup (p \mapsto N), m]}$$

$$(4)\quad \frac{S \text{ fresh} \wedge \forall t \in T \forall s \in S : s \in pre(t) \Rightarrow pre(t) \subseteq S}{\{S,T,m\} \to [S,\emptyset,\emptyset] \oplus T \oplus m}$$

$$(5)\quad \frac{S \text{ fresh} \wedge pre(T) \cap S' \neq \emptyset \wedge \forall t \in T \forall s \in S : s \in pre(t) \Rightarrow pre(t) \subseteq S \cup S'}{\{S,T,m\} \oplus [S',T',m'] \to [S \cup S', T', m'] \oplus T \oplus m}$$

$$(6)\quad \frac{C \to C'}{C \oplus C'' \to C' \oplus C''} \qquad (7)\quad \frac{C \equiv C' \quad C' \to C'' \quad C'' \equiv C'''}{C \to C'''}$$

Firing rule (1). If the marking of a net system contains the tokens required by the pattern of a transition $t = p \mapsto N$ (i.e., there exists a pattern instantiation such that the instance p is contained in the marking) then the transition is enabled. When t fires, the tokens required by the pattern are removed from the marking, and the (tokens, transitions and pre–nets contained in the) net expression $N\rho$, obtained by application of the pattern instantiation to N, is produced. All the components of the net expression are treated in the same way, regardless whether they are local (i.e., belong to the net system of transition t) or remote. The components will reach the net system to which they belong by axioms (2) and (3)[1].

Token and transition migration (2,3). The tokens, as well as the new transitions, produced by a firing of a transition reach the proper net system by application of this reduction.

Net creation (4). If the environment contains a pre–net whose transitions consume tokens only from places local to the pre–net, then such a prenet evolves in a new net system. The transitions and the marking of the pre–net are produced in the environment, as they may refer to another, already existing net system. Such components will reach the right net system by axioms (2) and (3). The freshness condition on the set of places of the pre–net, that can be enforced by performing alpha conversion, ensures the well–formedness of the reached configuration. The condition $\forall t \in T \forall s \in S : s \in pre(t) \Rightarrow pre(t) \subseteq S$ requires that all the transitions that will be local to the newly created net

[1] An alternative, equivalent semantics for the firing rule consists in partitioning the components of the net expression in two sets: the components that must be added to the net system of transition t, and those that must migrate to a remote net system. Then, the local components are directly added to the net system, while the remote components are released in the environment. This alternative semantics is closer to the implementation, while our choice provides a simpler rule.

do not consume tokens from remote places belonging to other existing nets. However, the pre–net may contain a transition u consuming all its token from places of another existing net; this is permitted, as the transition u will migrate to the proper net system after creation of the new net.

Net extension (5). If the transitions of a pre–net consume tokens from places of an already existing net system, then the components of the pre–net are added to the net system. The condition $pre(T) \cap S' \neq \emptyset$ requires that at least one transition in the pre–net consumes a token from a place of an existing net system (otherwise, the pre–net evolves in a new, independent net system by axiom (4)). The condition $\forall t \in T \forall s \in S : s \in pre(t) \Rightarrow pre(t) \subseteq S \cup S'$ requires that all the transitions, that will be local to the net obtained by adding the pre–net to the existing net system, do not consume tokens from remote places.

Composition (6). This rule permits the application of the axioms independently of the presence of other components in the configuration.

Structural congruence (7). Structurally congruent configurations behave equivalently.

The following proposition ensures that the well-formedness is preserved by the structural congruence relation and by the reduction semantics.

Proposition 1. *Let C be a well–formed configuration. If $C \equiv C'$, then C' is a well–formed configuration. If $C \rightarrow C'$, then C' is a well–formed configuration.*

4 Examples

4.1 Applet

In the first example we model the execution of some code *à la Java*, where an *applet* is downloaded from a server, and the computation occurs in the client:

$$Client = [\{runHere\}, \ldots, appletX\langle runHere\rangle runHere]$$
$$Server = [\{appletX\}, \{appletX(run) \mapsto \{\emptyset, \{run \mapsto \ldots\}, \emptyset\}\}, \emptyset]$$

The applet download is triggered by a token in place $appletX$. The token carries the client's location in its color ($runHere$). The applet code, which is contained in the spawned pre–net, is executed after the applet has migrated to the client.

4.2 Web Service Generation

The second example models a scenario in which some Web Services are generated from the corresponding factories. The idea in this case is that some services may be computationally too expensive to be carried over the server side. Instead, the Web Service is forked off its origin site and migrates to the client, where the computation occurs. A *service factory* is a net system with a transition expecting two place names in and out representing the place where the service expects its input and the place where the result

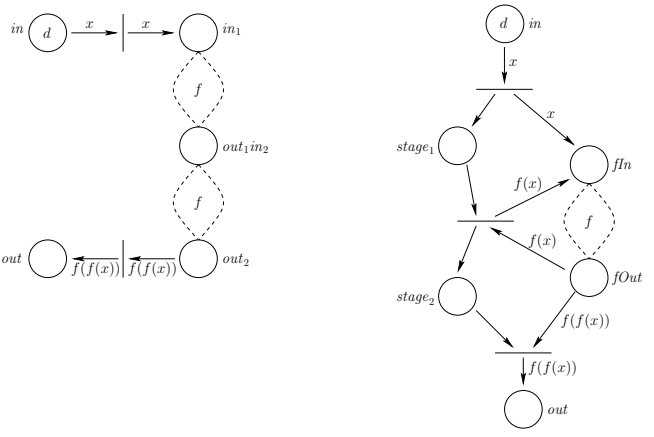

Fig. 1. Two different ways of combining services for computing $f \circ f$.

will be produced, once the computation is finished. These two places will usually be local to the client, thus forcing the generated service to move.

The following are service factories for the x^2 and $\sin x$ functions:

$$[\{squareFactory\}, \{squareFactory(in, out) \mapsto \\ \{\{tmp\}, \{in(x) \mapsto tmp\langle \ldots x \ldots \rangle, tmp(y) \mapsto out\langle \ldots y \ldots \rangle\}, \emptyset\}\}, \emptyset]$$

$$[\{sinFactory\}, \{sinFactory(in, out) \mapsto \{\emptyset, \{in(x) \mapsto out\langle \ldots x \ldots \rangle\}, \emptyset\}\}, \emptyset]$$

Next we show how to model a second-order Web Service generator $twiceFactory$ which, given a service factory for a generic function f, generates a service for the function $f \circ f$. In fact, there are at least two possible ways for modelling $twiceFactory$. In the first solution, the factory for f is invoked twice, and the two services are sequentially composed by means of a common place $out_1 in_2$ (left hand side of Figure 1):

$$\begin{array}{l} [\ \{twiceFactory_1\}, \\ \ \{twiceFactory_1(in, out, fFactory) \mapsto\ \{\ \{in_1, out_1 in_2, out_2\}, \\ \qquad\qquad\qquad\qquad\qquad\quad \{in(x) \mapsto in_1\langle x \rangle, out_2(x) \mapsto out\langle x \rangle\}, \\ \qquad\qquad\qquad\qquad\qquad\quad fFactory\langle in_1, out_1 in_2 \rangle \\ \qquad\qquad\qquad\qquad\qquad\quad fFactory\langle out_1 in_2, out_2 \rangle\}, \emptyset] \end{array}$$

In the second solution the factory for f is invoked only once, and the iterated application is enforced by means of an appropriate control flow in the generated service (right hand side of Figure 1):

$$\begin{array}{l} [\ \{twiceFactory_2\}, \\ \ \{twiceFactory_2(in, out, fFactory) \mapsto\ \{\ \{fIn, fOut, stage_1, stage_2\}, \\ \qquad\qquad\qquad\qquad\qquad\quad \{in(x) \mapsto stage_1\ fIn\langle x \rangle, \\ \qquad\qquad\qquad\qquad\qquad\quad\ \ stage_1\ fOut(r_1) \mapsto stage_2\ fIn\langle r_1 \rangle, \\ \qquad\qquad\qquad\qquad\qquad\quad\ \ stage_2\ fOut(r_2) \mapsto out\langle r_2 \rangle\}, \\ \qquad\qquad\qquad\qquad\qquad\quad fFactory\langle fIn, fOut \rangle\}, \emptyset] \end{array}$$

Roughly speaking, the two solutions resemble the differences between function inlining and function call.

The following is a possible invocation of $twiceFactory_1$ with the $squareFactory$ service, which yields a service for the computation of the x^4 function:

$$[\{in, out\}, \emptyset, twiceFactory_1\langle in, out, squareFactory\rangle]$$

5 Implementation

Entities in the formal model and their representation in the implementation have been purposefully given different names: not every entity in the implementation is paired with an entity in the abstract model, or the correspondence is only approximate.

5.1 Overall Organization of the Virtual Machine

The *virtual machine* that implements MAGNETs consists of the following kinds of objects.

Locations. Locations represent nets: the name "location" enforces the idea of a well-defined boundary that discriminates what is part of a net from what is not. Each location consists of a set of *ports*, representing places, a set of *rules*, representing transitions, and a set of running *threads*, representing ongoing computations. *Migration* of locations is controlled by the move instruction. Location handles have type lid.

Threads. Threads represent units of sequential computation within a location. As such, threads have no direct counterpart in the formal model. One can think of a thread as a sequence of actions implementing the semantics associated with a transition, once this has fired. A thread is made of

- a *code segment*, containing the list of virtual machine instructions to be executed;
- a *data segment*, containing the associations between names occurring in the thread and values;
- a *stack*, used for storing temporary data.

Thread handles have type tid.

Rules. Rules represent transitions. Each rule depends on a multiset of ports indicating the incoming transition arcs and has an associated blocked thread. Upon firing of the transition, the thread is spawned and its data segment enriched with the newly determined associations between names and values received from the ports.

Ports. Ports represent places. Each port has an associated queue of messages and has references to all the rules which depend upon it. Port handles have type pid and are made of the identifier of the location the port resides in, and a hidden, local port identifier.

Messages. Messages represent tokens. Messages can be sent to any known port, regardless of the location the port resides in.

5.2 Virtual Machine Instructions

The virtual machine is stack-based: every instruction takes its operands and produces results on a stack, just like the Java virtual machine does. Values of the virtual machine include port, thread, and location handles, as well as native data types (not shown for the sake of brevity). Table 2 lists the instructions of the virtual machine, as well as their signature specifying how the instruction manipulates the stack. The notation $x :: S$ represents a stack whose topmost value is x and where the stack underneath x is S.

Table 2. Instructions of the virtual machine.

Instruction	Stack before	Stack after
port	S	$pid :: S$
thread(n) c	$x_1 :: \cdots :: x_n :: S$	$tid :: S$
rule(n)	$tid :: pid_1 :: \cdots :: pid_n :: S$	S
location	S	$lid :: S$
send(n)	$pid :: x_1 :: \cdots :: x_n :: S$	S
spawn	$lid :: tid :: S$	S
load(n)	S	$x :: S$
store(n)	$x :: S$	S
whereis	$pid :: S$	$lid :: S$
move	$lid :: S$	S

What follows is an informal description of the semantics of the instructions. When talking about the values consumed or produced by instructions of the virtual machine, we will usually say "port" instead of "port handle". We will abuse the language similarly when referring to locations and threads. We will say "current thread" for the thread executing the instruction and "current location" for the location where the current thread is running:

port creates a new port in the current location. The port handle is pushed onto the stack;

thread(n) c creates a new thread, without executing it. The thread's code segment is c (a list of instructions), its environment is made of the x_i's vales on the stack ($i = 1, \ldots, n$). The thread handle is pushed onto the stack;

rule(n) creates a new rule in the current location involving the n ports on the stack. As soon as there is at least one message in the queue of each of the n ports, the specified thread is spawned;

location creates a new, empty location as a child of the current location. The new location is pushed onto the stack;

send(n) sends a tuple of n values to the port on the stack;

spawn causes the specified thread to be spawned in the specified location, which must be either an ancestor of the current location, or the current location itself;

$\mathsf{load}(x)$ pushes the value associated with the name x in the data segment onto the stack;
$\mathsf{store}(x)$ associates the name x with the topmost value on the stack;
whereis pops a port from the stack and pushes the port's location onto the stack;
move causes the current location to migrate inside location found on top of the stack.

5.3 Compiling MAGNETs

We now show the compilation rules for MAGNETs, that is how a net in the formal model is translated into instructions of the virtual machine. For the sake of brevity, only the most significant rules are presented here. We denote lists of instructions as $[i_1; i_2; \ldots; i_n]$ and we use @ for the usual append operation over lists.

A marking is compiled as a message communication. The names to be sent as well as the destination port are pushed onto the stack, and then the send instruction is executed:

$$[\![x\langle y_1, \ldots, y_n\rangle]\!] = [\mathsf{load}(y_n); \ldots; \mathsf{load}(y_1); \mathsf{load}(x); \mathsf{send}(n)]$$

Message communication is asynchronous, that is the send instruction is nonblocking.

A transition $p \mapsto N$ is compiled as a rule that associates the places in the pattern $places(p)$ with a thread that "implements" N. In order to create such thread, any free name occurring in N and that is not bound by p must be loaded onto the stack, so that an appropriate closure is created. Let $\{z_1, \ldots, z_k\} = fn(N) \setminus bn(p)$, we have

$$\begin{aligned}&[\![x_1(\overline{y_1}), \ldots, x_n(\overline{y_n}) \mapsto N]\!] = \\ &\quad [\mathsf{load}(x_n); \ldots; \mathsf{load}(x_1); \mathsf{load}(z_k); \ldots; \mathsf{load}(z_1); \mathsf{thread}(k)\ [\![N]\!]; \mathsf{rule}(n)]\end{aligned}$$

It is an error to create a rule for a pattern including ports that are not local to the current location. If this situation occurs, the current location "dies".

Consider now a prenet $\{S, T, m\}$ such that $pre(T) \subseteq S$. This net is unrelated to any other net since its set of transitions only involve ports that are locally defined. The net is represented as a fresh location within which an initialization thread is spawn. The thread creates the ports in S, stores them in its data segment, creates the rules for the transitions in T and finally sets the places with the initial marking. Let $\{z_1, \ldots, z_l\} = fn(\{S, T, m\})$, we have

$$\begin{aligned}&[\![\{\{x_1, \ldots, x_n\}, \{t_1, \ldots, t_k\}, m\}]\!] = \\ &\quad [\mathsf{load}(z_l); \ldots; \mathsf{load}(z_1); \\ &\quad\ \mathsf{thread}(l)\ [\mathsf{port}; \mathsf{store}(x_1); \ldots; \mathsf{port}; \mathsf{store}(x_n)]@[\![t_1]\!]@\cdots@[\![t_k]\!]@[\![m]\!]; \\ &\quad\ \mathsf{location}; \mathsf{spawn}]\end{aligned}$$

The last interesting case we examine is that of a net extension, that is a prenet $\{S, T, m\}$ connected with an existing net. This condition is formally specified as $S \subsetneq pre(T)$, or equivalently there exists a place $z \in pre(T) \setminus S$ that belongs to a different net. As in the case of a standalone net, a fresh location is created. However, this location must first migrate into the location owning z. Once it has reached z's location, a new locally spawned thread takes care of creating the new rules. Let $\{y_1, \ldots, y_l\} = fn(\{S, T, m\})$, we have

$$
\begin{array}{l}
[\![\{\{x_1,\ldots,x_n\},\{t_1,\ldots,t_k\},m\}]\!] = \\
\quad [\mathsf{load}(y_l);\ldots;\mathsf{load}(y_1);\mathsf{thread}(l) \\
\qquad [\mathsf{load}(z);\mathsf{whereis};\mathsf{move};\mathsf{load}(y_l);\ldots;\mathsf{load}(y_1);\mathsf{thread}(l) \\
\qquad\quad [\mathsf{port};\mathsf{store}(x_1);\ldots;\mathsf{port};\mathsf{store}(x_n)]@[\![t_1]\!]@\cdots@[\![t_k]\!]@[\![m]\!]; \\
\qquad\ \mathsf{load}(z);\mathsf{whereis};\mathsf{spawn}]; \\
\qquad \mathsf{location};\mathsf{spawn}]
\end{array}
$$

Finally, the composition of two nets N_1 and N_2 simply amounts at appending the code resulting from compiling N_1 and N_2 in isolation:

$$[\![N_1 \oplus N_2]\!] = [\![N_1]\!]@[\![N_2]\!]$$

5.4 Remarks

Synchronous communication. As it is stated, the formal model imposes a continuation-passing style arrangement of any sequential computation. As this imposes a considerable overhead in terms of both time and resource consumption (the creation of several temporary, short-living ports), we have enriched the actual implementation with primitive instructions for synchronous communication. Each thread is equipped with a *continuation port* that is used any time synchronization is required. The model syntax can be consequently enriched so as to provide a more comfortable (and familiar) programming framework for MAGNETs.

Rule matching. No mention has been made to the technicalities of pattern matching compilation but a number of optimizations can improve the naive algorithm that checks every rule upon reception of a message (see [14]). We briefly introduce two of them:
Simple port optimization. A *simple port* is a port occurring alone in one pattern only. Roughly speaking, such a place denotes a function as it is normally understood in conventional, sequential programming languages and the action of "firing" corresponds to function application. In this case, no pattern matching is actually necessary, and "firing" can be optimized to a real function call.
Static analysis of net configuration. Places whose names are not communicated outside the net where they reside can undergo a number of optimizations, since their interaction with other places is statically determined by the net configuration. In these cases, some places might be better represented as "net data", possibly protected by a locking mechanism that preserves atomic access.

Migration. The move instruction is by far the most unusual instruction in the virtual machine. One might argue that the compilation rules we have presented do not highlight sufficiently the complexity that underlies migration. Although the implementation of the move instruction indeed raises lots of technical issues, in particular the marshalling and unmarshalling of information, according to the experience gained from our development we can state that the main difficulty of mobile computation is not mobility *per se*. As soon as we have a neat definition of the boundaries of what we want to move, it does not make a great difference whether we move data, or code, or working environments.

Message routing. As locations migrate, the physical location of ports therein contained changes over time. In our implementation, each instance of the virtual machine keeps track of outgoing migrations. When a message is received for a port hosted in a location that has moved, the virtual machine does two things: first, it forwards the message to the site where the location has migrated to, thus ensuring that the message is eventually delivered; second, it notifies the sender with the new physical location of the destination port, so that subsequent communications can be direct and no routing is necessary.

6 Conclusion

MAGNETs are a formal model for the representation of systems with an evolving structure, suitable for a distributed implementation.

In the last years several extensions of Petri nets, dealing with mobility and/or dynamicity, have been proposed. Most of these extensions are devoted to adding object-oriented features on top of (colored) Petri nets (see e.g. [4, 6, 8, 12, 13, 18]): 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 differences w.r.t. our approach: first, our model lies at a lower level of abstraction, because the object-oriented features are not embedded in MAGNETs, but they need to be encoded; second, our aim is to embody dynamicity in our model, not to add it by an external structure.

Some recent extensions are devoted to the modeling of mobile agents in the so called *nets-within-nets* approach [5, 10, 20–22]: mobile agents are modeled as nets that move inside an environment, represented by a net. The two main differences w.r.t. MAGNETs are concerned with the technique adopted to obtain mobility and dynamicity. While MAGNETs borrows the name managing techniques of mobile process calculi, in the nets-within-nets approach mobility and dynamicity are achieved through an higher-order technique: namely, the tokens of the "environment" net are themselves nets. The extension of MAGNETs with higher-order features is a challenging endeavor, for which we plan further investigation.

Another extension, which is closer to the spirit of our model, is given by Self Modifying Nets [19]; 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 MAGNETs they depend only on the color of the consumed tokens.

Finally, we should remark the close relationship between MAGNETs and the Join calculus [9]. MAGNETs extend the Join calculus by allowing join definitions (reaction sites in the CHAM) to evolve at runtime, still preserving the locality constraints, and by having a weaker notion of pattern linearity. Another difference is given by the definition of the boundaries that mark a mobile agent. In the Join calculus with mobile agents there is an explicit notion of *named location*, and locations move by means of a primitive go operation of the formal model. In MAGNETs, there is no need to define such primitive operation, nor there is any need to give nets a name. Migration occurs transparently depending on the use of remote place names. To cope with these differences the implementation of the Join calculus with mobile agents presented in [17] has been suitably adapted.

References

1. W. M. P. van der Aalst, "Pi calculus versus Petri nets: Let us eat "humble pie" rather than further inflate the "Pi hype". http://tmitwww.tm.tue.nl/staff/wvdaalst/pi-hype.pdf
2. A. Asperti, N. Busi, "Mobile Petri Nets", Technical Report UBLCS-96-10, dept. of Computer Science, University of Bologna, Italy, 1996.
3. C. Fournet, G. Gonthier, J.-J. Levy, L. Maranget, D. Remy, "A Calculus of Mobile Agents", Proceedings of the 7th International Conference on Concurrency Theory (CONCUR'96)
4. E. Battiston, A. Chizzoni, F. De Cindio, "Inheritance and Concurrency in CLOWN", First Workshop on Object-Oriented Programming and Models of Concurrency, June 1995.
5. M. A. Bednarczyk, L. Bernardinello, W. Pawłowski, L. Pomello, "Modelling mobility with Petri hypernets", in Proc. WADT 2004, LNCS, Springer, to appear.
6. 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.
7. N. Busi. "Mobile Nets", in Proc. FMOODS'99, Kluwer, 1999.
8. 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.
9. C. Fournet, G. Gonthier, "The reflexive CHAM and the Join calculus", POPL'96, 1996.
10. M. Köhler, D. Moldt, H. Rölke, "Modelling mobility and mobile agents using nets within nets", in Proc. ICATPN 2003, LNCS 2679, Springer, 2003.
11. K. Jensen. "Coloured Petri Nets", EATCS Monographs in Computer Science, Springer, 1992.
12. C. A. Lakos, "Object Petri Nets – Definition and Relationship to Coloured Nets", Technical Report TR94-3, Computer Science Department, University of Tasmania, 1994.
13. 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.
14. L. Maranget, F. le Fessant, "Compiling Join Patterns", HLCL '98, volume 16(3) of Electronic Notes in Theoretical Computer Science, Nice, France, Sept. 1998.
15. R. Milner, J. Parrow, D. Walker, "A Calculus of Mobile Processes", in Information and Computation 100, pp.1-77, 1992.
16. C. A. Petri, "Kommunikation mit Automaten", PhD Thesis, Institut für Instrumentelle Mathematik, Bonn, Germany, 1962.
17. L. Padovani, "A Distributed Language with Mobile Agents: Design and Implementation", Master Thesis Dissertation, March 1998. http://www.cs.unibo.it/~lpadovan/master_thesis/main.pdf
18. 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.
19. R. Valk, "Generalizations of Petri Nets", in Proc. MFCS'81, LNCS 118, 1981.
20. R. Valk, "On Processes of Object Petri Nets", Bericht Nr. 185, Fachbereich Informatik, Universitat Hamburg, 1996.
21. R. Valk, "Petri nets as token objects: An introduction to elementary object nets", in Proc. ICATPN 1998, LNCS 1420, Springer, 1998.
22. R. Valk, "Concurrency in communicating object Petri nets", in *Concurrent Object-Oriented Programming and Petri Nets: Advances in Petri Nets*, LNCS 2001, Springer, 2001.