Exploiting Petri Nets in the Analysis of the Expressiveness of Publish/Subscribe Coordination

Nadia Busi and Gianluigi Zavattaro

Dipartimento di Scienze dell'Informazione, Università di Bologna,
Mura A.Zamboni 7, I-40127 Bologna, Italy.
busi,zavattar@cs.unibo.it

Abstract. Two main architectural styles emerged in the panorama of coordination infrastructures: the publish/subscribe model, based on the raising and catch of events, and the shared-dataspace – Linda-like – approach, in which coordination is achieved by introducing, reading and consuming data from a common repository. In [BZ01] we have initiated a rigorous comparison among the two approaches introducing two minimal process calculi, called $\mathcal{DS}$ and $\mathcal{PS}$, that respectively incorporate the main traits characterizing the dataspace and the publish/subscribe approaches, respectively. In this paper we discuss the expressive power of the publish/subscribe calculus $\mathcal{PS}$ and we show that even if (i) the calculus is expressive enough for modeling Turing powerful formalisms (in a nondeterministic manner) (ii) interesting properties such as termination (ie. all computations are finite) or the possibility for an event to be raised are still decidable. To prove (ii) we use decidability results that hold for Petri Nets extended with transfer arcs [DFS98].

1 Introduction

The rapid evolution of networking technologies calls for the creation of suitable architectural styles and methodologies for the development of distributed applications. In this context, several technologies have emerged. However, there is not yet a common agreement on which is the most suitable inter-agent communication and coordination platform. Two main architectures have been proposed in the literature:

1. *Publish/Subscribe*: inter-agent communication takes place through the raising of events performed by agents via a *publish* operation. Events are multicast to those agents which have previously registered their interest via a *subscribe* operation.
2. *Shared dataspace*: agents communicate by producing, consuming, and testing for the presence of objects in a shared, common repository.

The two approaches have been adopted in many proposals for inter-agent communication infrastructures: see, e.g., JEDI [CNF98] for the publish/subscribe approach and Linda [Gel85], WCL [Row98], MARS [CLZ98], and Lime [PMR99] for the shared dataspace one. In [BZ01] we have initiated a rigorous comparison among the two approaches. In particular, we have studied the interchangeability, namely the possibility to substitute one mechanism in place of the other without affecting the internal behaviour of the agents.

We have shown that the shared dataspace approach can be reduced to the publish/subscribe model simply by adding agents which have the responsibility to represent the state of the simulated common repository. On the other hand, a reduction of the publish/subscribe into the shared dataspace model cannot be provided in the case only the basic output, input, and read dataspace operations are considered. Interestingly, we have shown that this negative result no longer holds as soon as a global dataspace operation, such as the *copy-collect* primitive proposed in [RW98], is added. This coordination operator, introduced in order to solve the well-known *multiple read* problem of Linda, has the ability to collect all the data satisfying a certain pattern available in the dataspace.

In this paper we continue the investigation initiated in [BZ01] focusing on the expressive power of the publish/subscribe coordination approach. More precisely, we consider the process calculus $\mathcal{PS}$, introduced in [BZ01] in order to model the peculiar traits of publish/subscribe, and we investigate the expressiveness of the calculus, as well as the decidability of some relevant properties. To this aim, we make use of two standard tools of computability and concurrency theory. Regarding the expressive power, we investigate the ability of the calculus to model Random Access Machines (RAMs) [SS63], a well-knowm register based Turing powerful formalism. Such an expressiveness result can be used to prove the undecidability of some properties, such as, e.g., the existence of a deadlock. On the other hand, for the study on the decidability of properties we make use of (extended classes of) Petri nets, that are classes of concurrent models for which many interesting properties turn out to be decidable.

Concerning the expressiveness of the language, we recall a modeling of Random Access Machines in $\mathcal{PS}$, that we already informally discussed in [BZ01]. This modeling is nondeterministic because it introduces undesidered computations; nevertheless, all these computations are infinite thus the encoding of a RAM has a finite computation if and only if the corresponding RAM terminates. This proves that the existence of a finite computation is undecidable in $\mathcal{PS}$.

The existence of a deterministic encoding of RAMs was left as an open question in [BZ01]. In this paper, we answer negatively to this question showing that the existence of an infinite computation is a decidable property in $\mathcal{PS}$; thus, it is not possible to model RAMs deterministically because a deterministic encoding has only one possible computation, which is infinite if and only if the corresponding RAM does not terminate. In order to prove this result we resort to Petri Nets extended with transfer arcs. Transfer arcs permit to move all the tokens in a source place to a target place, on the firing of a given transition. Dufourd, Finkel and Schnoebelen have proved that termination (all computations are finite) and coverability are decidable for this class of Petri Nets [DFS98].

$$(\mathbf{sub}(\mathbf{e},\mathbf{A}').\mathbf{A} \oplus Ag, LS) \longrightarrow (\mathbf{A} \oplus Ag, LS \oplus \langle \mathbf{e},\mathbf{A}'\rangle)$$
$$(\mathbf{unsub}(\mathbf{e},\mathbf{A}').\mathbf{A} \oplus Ag, LS \oplus \langle \mathbf{e},\mathbf{A}'\rangle) \longrightarrow (\mathbf{A} \oplus Ag, LS)$$
$$(\mathbf{pub}(\mathbf{e}).\mathbf{A} \oplus Ag, LS) \longrightarrow (\mathbf{A} \oplus Ag \oplus \{\mathbf{A_i} \text{ for each occurrence of } \langle \mathbf{e},\mathbf{A_i}\rangle \text{ in } LS\}, LS)$$
$$(\mathbf{new}(\mathbf{A}').\mathbf{A} \oplus Ag, LS) \longrightarrow (\mathbf{A}' \oplus \mathbf{A} \oplus Ag, LS)$$
$$\frac{(\mathbf{A},\mathbf{LS}) \longrightarrow (\mathbf{A}',\mathbf{LS}')}{(\mathbf{K} \oplus Ag,\mathbf{LS}) \longrightarrow (\mathbf{A}' \oplus Ag,\mathbf{LS}')} \quad \text{if } K = A$$

Table 1. The semantics of the publish/subscribe calculus $\mathcal{PS}$.

We use this result to prove that termination is decidable in $\mathcal{PS}$. Moreover, the Petri Net modeling of the $\mathcal{PS}$ calculus permits us also to prove the decidability of another interesting property: the existence of a computation comprising the raising of a given event. The decidability of this property can be very useful in the design of systems, as it permits, e.g., to verify if a process could signal a failure.

The paper is organized as follows: Section 2 reports the syntax and semantics of the $\mathcal{PS}$ calculus, Section 3 discusses the nondeterministic modeling of RAM, Section 4 reports the decidability results, while Section 5 reports some conclusive remarks.

2 The Publish/Subscribe Calculus

The $\mathcal{PS}$ calculus borrows typical techniques from the tradition of process calculi for concurrency (e.g., Milner's CCS [Mil89]); an agent is described as a term of an algebra where the basic actions are typical coordination primitives. We consider primitives for event publication $pub(e)$, event subscription $sub(e, A)$, event unsubscription $unsub(e, A)$, plus an operator for agent creation $new(A)$. The result of a subscription $sub(e, A)$ consists of the production of a listener responsible to catch the event e, and then activate the corresponding reaction A. More precisely, whenever an instance of the event e is raised as result of the execution of $pub(e)$, the listener should create a new agent A. The listener can be removed exploiting the event unsubscribe operation $unsub(e, A)$.

Formally, let *Event*, ranged over by e, f, ..., be a denumerable set of possible events that can be exchanged among agents.

The set *Agent* of agents, ranged over by A, A', ..., is the set of terms generated by the following grammar:

$$\begin{array}{lcl} A & ::= & \mathbf{0} \mid \mu.A \mid K \\ \mu & ::= & pub(e) \mid sub(e,P) \mid unsub(e,P) \mid new(A) \end{array}$$

where K stands for a generic element of a set $Name$ of agent names; we assume that all agent name occurrences are equipped with a corresponding (guarded) defining equation of the form $K = A$. Agent names are used to support recursive definitions such as, for example, in the term $Repeat_{ef} = pub(e).pub(f).Repeat_{ef}$, which represents an agent able to repeatedly raise the events e and f. Agent $\mathbf{0}$ is an agent that can do nothing. Agent $\mu.A$ is an agent that can do the action μ and then behaves like A.

A publish/subscribe system configuration is composed by a multiset of active agents and by a multiset of active listeners. Formally, a system configuration is a pair $(Ag, LS) \in \mathcal{M}(Agent) \times \mathcal{M}(Event \times Agent)$. A pair $\langle e, A\rangle$ represents a listener which is responsible for the activation of agent A each time an event e is raised.

The rules for $\mathcal{PS}$ are listed in Table 1. The execution of the subscription operation adds a new listener, while the execution of an unsubscription removes one instance (if available) of the specified listener. The publish operation activates a new multiset of agents representing the reactions to the raised event: this set of agent, denoted by $\{\mathbf{A}'$ for each occurrence of $\langle \mathbf{e}, \mathbf{A}'\rangle$ in $LS\}$ corresponds to the multiset obtained by taking an agent A' for each occurrence of active listener $\langle e, A'\rangle$. The new operation simply produces a new agent, while the last rule indicates that an agent name K can do what its defining agent A can do.

3 A Nondeterministic Modeling of RAMs

A RAM is composed of a finite set of registers, that can hold arbitrary large natural numbers, and by a program, that is a sequence of simple numbered instructions, like arithmetical operations (on the contents of registers) or conditional jumps.

The execution of the program begins with the first instruction and continues by executing the other instructions in sequence, unless a jump instruction is encountered. The execution stops when an instruction number higher than the length of the program is reached.

In [Min67] it is shown that the following two instructions are sufficient to model every recursive function:

- $Succ(r_j)$: adds 1 to the content of register r_j;
- $DecJump(r_j, s)$: if the content of r_j is not zero, then decreases it by 1 and goes to the next instruction, otherwise jumps to instruction s.

We restrict to RAMs that start and finish their computation (in the case they terminate) with all the registers empty; it is a well known result that this restriction does not decrease the expressiveness of the formalism.

Now we present the encoding of RAM in $\mathcal{PS}$. The encoding is *nondeterministic*, as it introduces some extra infinite computation; nevertheless, it is ensured that a RAM terminates if and only if the corresponding encoding has a terminating computation. As termination cannot be decided in Turing equivalent formalisms, the same holds also for $\mathcal{PS}$.

The encoding implements nondeterministically *DecJump* operations: two possible behaviours can be chosen, the first is valid if the tested register is not zero, the second otherwise. If the wrong choice is made, the computation is ensured to be infinite; in this case, we cannot say anything about the corresponding RAM. Nevertheless, if the computation terminates, it is ensured that it corresponds to the computation of the corresponding RAM. Conversely, any computation of the RAM is simulated by the computation of the corresponding encoding in which no wrong choice is performed.

Given the RAM program R composed by the instructions $I_1 \ldots I_k$, the corresponding encoding is the agent below:

$$\begin{array}{l} sub(loop_1, pub(loop_1)).sub(loop_2, pub(loop_2)). \\ sub(p_1, P_1). \ldots sub(p_k, P_k).pub(p_1) \end{array}$$

If the i-th instruction is an increment of register r_j, the constant P_i is defined as

$$P_i = sub(isz_j, pub(loop_2)).pub(p_{i+1})$$

If the i-th instruction is a decrement of register r_j, or jump to instruction s, the constant P_i is defined as

$$\begin{array}{l} P_i = sub(c_i, \mathbf{0}). \\ \qquad new(unsub(c_i, \mathbf{0}).pub(loop_1). \\ \qquad\qquad unsub(isz_j, pub(loop_2)). \\ \qquad\qquad unsub(loop_1, pub(loop_1)). \\ \qquad\qquad sub(loop_1, pub(loop_1)).pub(p_{i+1})). \\ \qquad new(unsub(c_i, \mathbf{0}).pub(isz_j).pub(p_s)) \end{array}$$

It is worth noting that two listeners, $\langle loop_1, pub(loop_1)\rangle$ and $\langle loop_2, pub(loop_2)\rangle$, are produced before starting the simulation of the execution of instructions. If, during the execution of the encoding of instructions, an event $loop_1$ (resp. $loop_2$) is published, the listener will produce an agent $pub(loop_1)$ (resp. $pub(loop_2)$), thus obtaining an infinite loop that can be broken only by an unsubscribe operation.

The basic idea of this encoding is to represent the actual content of each register r_j with a corresponding number of listeners $\langle isz_j, pub(loop_2)\rangle$. Moreover, every time an increment (or a decrement) on the register r_j is performed, a new listener $\langle isz_j, pub(loop_2)\rangle$ is spawned (resp. unsubscribed). In this way it is possible to check if the actual content of a register r_j is zero by verifying if there are no occurrences of listener $\langle isz_j, pub(loop_2)\rangle$.

There are two possible wrong choices that can be performed during the computation: (i) a decrement on a register containing zero or (ii) a jump for zero on a non-zero register.

In the case (i), the agent

$$\begin{array}{l} pub(loop_1).unsub(isz_j, pub(loop_2)). \\ unsub(loop_1, pub(loop_1)). \\ sub(loop_1, pub(loop_1)).pub(p_{i+1}) \end{array}$$

is activated with no listener $\langle isz_j, pub(loop_2)\rangle$ available. Thus, the agent publishes event $loop_1$ and blocks trying to execute $unsub(loop_1, pub(loop_1))$. However, event $loop_1$ wakes up the listener $\langle loop_1, pub(loop_1)\rangle$, that produces an agent $pub(loop_1)$, thus starting an infinite loop.

In the case (ii), the agent $pub(isz_j).pub(p_s)$ is activated when there is at least one occurrence of listener $\langle isz_j, pub(loop_2)\rangle$. When event isz_j is published, the above listener(s) produces one (or more) occurrences of event $loop_2$, thus starting an infinite loop with listener $\langle loop_2, pub(loop_2)\rangle$.

Finally, it may happen that, even if we execute a decrement on a non-zero register, the agent takes a divergent computation. In fact, consider the agent

$$\begin{array}{l} pub(loop_1).unsub(isz_j, pub(loop_2)). \\ unsub(loop_1, pub(loop_1)). \\ sub(loop_1, pub(loop_1)).pub(p_{i+1}) \end{array}$$

First, it starts an infinite loop with listener $\langle loop_1, pub(loop_1)\rangle$, then, after removing a listener on event isz_j, it breaks the loop by removing the listener on $loop_1$; at this point, a new listener on $loop_1$ is created. If the agent $pub(loop_1)$ is not executed before creating the new listener, the infinite loop will restart with the new listener. However, this is not problematic because we are interested only in preserving terminating computations.

4 Decidability results

In this section we show that the existence of an infinite computation, as well as the possibility to reach a configuration where a given event can be raised, are decidable properties in $\mathcal{PS}$. In order to prove such results, we recall, using a notation convenient for our purposes, the definition of simple P/T nets extended with transfer arcs (see, e.g., [DFS98]).

Definition 1. *Given a set S, a* finite multiset *over S is a function $m : S \to I\!N$ such that the set $dom(m) = \{s \in S \mid m(s) \neq 0\}$ is finite. The* multiplicity *of an element s in m is given by the natural number $m(s)$. The set of all finite multisets over S, denoted by $\mathcal{M}_{fin}(S)$, is ranged over by m. A multiset m such that $dom(m) = \emptyset$ is called* empty *. The set of all finite sets over S is denoted by $\wp_{fin}(S)$. The set of all partial functions on S is denoted by $\mathcal{F}_p(S, S)$.*

Given the multiset m and m', we write $m \subseteq m'$ if $m(s) \leq m'(s)$ for all $s \in S$ while $\oplus$ denotes their multiset union*: $m \oplus m'(s) = m(s) + m'(s)$. The operator $\setminus$ denotes* multiset difference*: $(m \setminus m')(s) =$ if $m(s) \geq m'(s)$ then $m(s) - m'(s)$ else 0. The* scalar product*, $j \cdot m$, of a number j with m is $(j \cdot m)(s) = j \cdot (m(s))$.*

To lighten the notation, we sometimes use the following abbreviation. If m is a multiset containing only one occurrence of an element s (i.e., $dom(m) = \{s\}$ and $m(s) = 1$) we denote m by only s. Let m be a multiset over S and m' a multiset over $S' \supseteq S$, such that $(m'(s') = 0)$ for each $s' \in S' \setminus S$; with abuse of notation, we sometimes use m in place of m', and vice versa.

Let $f : A \to B$ be a function. It is extended to a function from $\mathcal{M}_{fin}(A)$ to $\mathcal{M}_{fin}(B)$ in the following way: if $m \in \mathcal{M}_{fin}(A)$ then, for all $b \in B$, $(f(m))(b) = \sum_{f(a)=b} m(a)$.

Definition 2. *A P/T net with transfer arcs is a triple $N = (S, T, m_0)$ where S is the set of* places, *T is the set of* transitions *(which are triples $(c, p, f) \in \mathcal{M}_{fin}(S) \times \mathcal{M}_{fin}(S) \times \mathcal{F}_p(S, S)$ such that the domain of the partial function f has no intersection with c and p), and m_0 is a finite multiset of places. Finite multisets over the set S of places are called* markings*; m_0 is called* initial marking. *Given a marking m and a place s, $m(s)$ denotes the number of occurrences of s inside m and we say that the place s contains $m(s)$* tokens. *A P/T net with transfer arcs is finite if both S and T are finite.*

A transitions $t = (c, p, f)$ is usually written in the form $c \overset{f}{\rightarrowtail} p$ and f is omitted when empty. The marking c is called the preset *of t and represents the tokens to be* consumed. *The marking p is called the* postset *of t and represents the tokens to be* produced. *The partial function f denotes the* transfer arcs *of the transition which connect each place s in the domain of f to its image $f(s)$. The meaning of f is the following: when the transition fires all the tokens inside a place s in the domain of f are transferred to the connected place $f(s)$.*

A transition $t = (c, p, f)$ is enabled *at m if $c \subseteq m$. The execution of the transition produces the new marking m' such that $m'(s) = m(s) - c(s) + p(s) + \sum_{s':f(s')=s} m(s')$, if s is not in the domain of f, $m'(s) = 0$, otherwise. This is written as $m \xrightarrow{t} m'$ or simply $m \longrightarrow m'$ when the transition t is not relevant. We use σ, σ' to range over sequences of transitions; the empty sequence is denoted by ε; let $\sigma = t_1, \ldots, t_n$, we write $m \xrightarrow{\sigma} m'$ to mean the* firing sequence *$m \xrightarrow{t_1} \cdots \xrightarrow{t_n} m'$. The net $N = (S, T, m_0)$ has an* infinite *computation if it has a legal infinite firing sequence.*

The basic idea underlying the definition of an operational net semantics for a process calculus is to decompose a system configuration (Ag, LS) into a multiset of sequential components, which can be thought of as running in parallel. Each sequential component has a corresponding place in the net, and will be represented by a token in that place. Reductions are represented by transitions which consume and produce multisets of tokens.

In our particular case we deal with two different kinds of sequential components: agents of the form $\mu.P$ and listeners of the form $\langle e, A\rangle$.

Besides these classes of components corresponding directly to terms of the language, we need to introduce a new kind of components $raised(e, A)$ used to model the effect of the raising of an event.

The way we represent subscription and unsubscription operations in our net semantics is standard. More interesting is the mechanism used to model event raising represented in Figure 1. Whenever an event e is raised by an agent $pub(e).A$, each token in a place $\langle e, A'\rangle$ is transferred to the corresponding place $raised(e, A')$. In order to realize this, we use a *transfer* arc that moves all the tokens inside the source place to the target one. Each token introduced

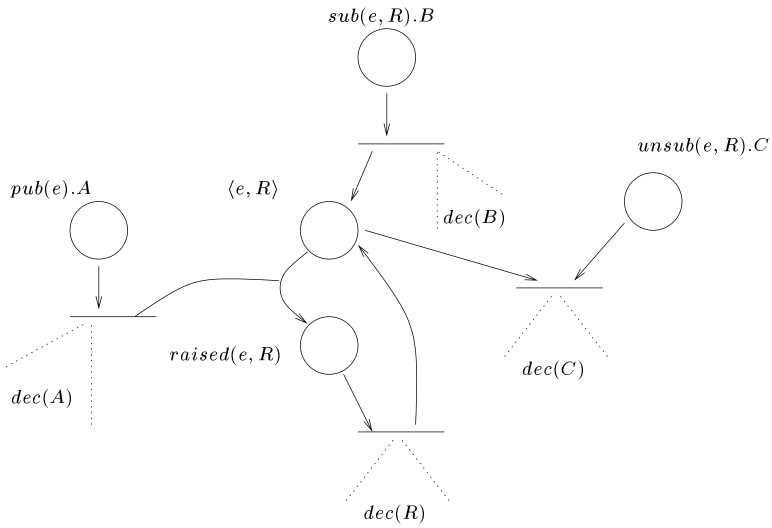

Fig. 1. Modeling the raising of an event.

in $raised(e, A')$ will be responsible for the activation of the new instance of A'. Moreover, when the activation happens, also a token in $\langle e, A'\rangle$ is introduced in order to register interest in the next raising of event e.

The main drawback of this procedure used to model event raising is that it is not executed atomically. For instance, another instance of event e can be raised before it terminates. In this case, the listeners whose corresponding token is still in the place $raised(e, A')$ will not be activated by the occurrence of this event. However, as we will prove in the following, even in the presence of this drawback the net semantics respects interesting properties, such as, e.g., the existence of infinite computations.

After the informal description of the net semantics we introduce its formal definition. Given the configuration (A, LS), we define the corresponding P/T system $Net((A, LS))$. In order to do this, we need the following notations.

- Let $\mathcal{S}$ be the set
  $\{A \mid\ A$ sequential agent$\} \cup \{\langle e, A\rangle, raised(e, A) \mid\ e$ event and A agent$\}$.
- Let the function $dec : \mathcal{M}(Agent) \times \mathcal{M}(Event \times Agent) \to \mathcal{M}_{fin}(\mathcal{S})$ be the decomposition of configurations into markings, reported in Table 2.
- Let $\mathcal{T}$ contain the transitions obtained as instances of the axiom schemata presented in Table 3.

The axioms in Table 2 define the decomposition of system configurations, agents and listeners.

The decomposition of a system configuration is obtained by an elementwise application of the decomposition functions to all the agents and the listeners belonging to the system configuration.

$$dec(Ag, LS) = dec(Ag) \oplus dec(LS)$$

$$dec(\mathbf{0}) = \emptyset$$

$$dec(\mu.A) = \{\mu.A\}$$

$$dec(K) = dec(A) \qquad \text{if } K = A$$

$$dec(\langle e, A\rangle) = \{\langle e, A\rangle\}$$

Table 2. Decomposition function.

The decomposition of the agent $\mathbf{0}$ generates no tokens, the decomposition of a sequential agent $\mu.A$ produces one token in the corresponding place, while the decomposition of an agent constant K corresponds to the decomposition of the agent on the right hand side of its defining equation.

The decomposition of a listener $\langle e, A\rangle$ generates one token in the corresponding place.

`sub(e,A').A`	$sub(e, A').A \rightarrowtail \langle e, A'\rangle \oplus dec(A)$
`unsub(e,A').A`	$unsub(e, A').A \oplus \langle e, A'\rangle \rightarrowtail dec(A)$
`new(A').A`	$new(A').A \rightarrowtail dec(A) \oplus dec(A')$
`pub(e).A`	$pub(e).A \overset{f}{\rightarrowtail} dec(A)$
	where $f = \{(\langle e, A'\rangle, raised(e, A')) \mid A'$ is an agent$\}$
`raised(e,A)`	$raised(e, A) \rightarrowtail dec(A) \oplus \langle e, A\rangle$

Table 3. Transition specification.

The axioms in Table 3 define the possible transitions.

Axiom `sub(e,A').A` deals with the execution of the subscritption primitive: a token in the place $\langle e, A'\rangle$ is produced, in order to listen to future raisings of event e.

Axiom `unsub(e,A').A` deals with the execution of an unsubscription: a token from place $\langle e, A'\rangle$ is consumed.

Axiom `new(A').A`, corresponding to the creation of a new agent A', produces the tokens corresponding to the decomposition of the newly created agent.

Axiom `pub(e).A` describes the raising of an event e: the transfer arcs move all the tokens from the places $\langle e, A' \rangle$, corresponding to listeners on e, in the corresponding places $raised(e, A')$. In this way, all the listeners on e are notified of the raising of the event. The activation of the corresponding processes A' requires a further step, described by the axiom `raised(e,A)`: an instance of agent A is activated (by producing tokens in the places in the decomposition of the agent) and a token is reintroduced in the place $\langle e, A \rangle$, in order to register interest in the next raising of event e.

Definition 3. *Let (Ag, LS) be a system configuration. We define the triple $Net(Ag, LS) = (S, T, m_0)$ where:*

$$\begin{aligned} S &= \{A \mid A \textit{ sequential subagent of some agent in } Ag \textit{ or in } LS\} \cup \\ &\quad \{\langle e, A\rangle, raised(e, A) \mid e \textit{ event in } Ag \textit{ or in } LS \textit{ and} \\ &\quad\ A \textit{ subagent of some agent in } Ag \textit{ or in } LS\} \\ T &= \{c \xrightarrow{f|_S} p \mid c \xrightarrow{f} p \in \mathcal{T} \textit{ and } dom(c) \subseteq S\} \\ m_0 &= dec(Ag, LS) \end{aligned}$$

where by $f|_S$ we mean the restriction of function f to its subdomain S.

It is not difficult to see that $Net(Ag, LS)$ is well defined, in the sense it is a correct P/T net with transfer arcs; moreover, it is finite. Moreover the net semantics is complete as it simulates all the possible computations allowed by the operational semantics.

Theorem 1. *Let $Net(Ag, LS) = (S, T, m_0)$ and (Ag', LS') be a configuration s.t. $dom(dec(Ag', LS')) \subseteq S$. If $(Ag', LS') \longrightarrow (Ag'', LS'')$ then there exists a transition sequence σ s.t. $dec((Ag', LS')) \xrightarrow{\sigma} dec((Ag'', LS''))$.*

The above theorem proves the completeness of the net semantics which, on the other hand, is not sound. Indeed, as we have already discussed, the encoding introduces some slightly different computations due to the non atomicity of the way we model the event notification mechanism. However, the introduction of these computations does not alterate the possibility to have an infinite computation. This is proved by the following Theorem.

Theorem 2. *Let $Net(Ag, LS) = (S, T, m_0)$. There exists an infinite computation of the configuration (Ag, LS) iff there exists an infinite firing sequence starting from m_0.*

Another interesting property that can be verified by means of the net semantics is the fact that an event will eventually be raised by a given system configuration. Indeed, this property can be reduced to the existence of a marking that covers one of the markings $\{pub(e).A\}$, where $pub(e).A$ is any sequential subagent of the system configuration.

Theorem 3. *Let $Net(Ag, LS) = (S, T, m_0)$. The event e can be raised by the system configuration (Ag, LS) iff one of the markings in the (finite) set $\{\{pub(e).A\} \mid pub(e).A \in S\}$ can be covered in $Net(Ag, LS)$.*

5 Conclusion

In this paper we have continued the investigation of the publish/subscribe coordination infrastructure initiated in [BZ01]. In particular, we have proved some decidability results for the publish/subscribe calculus introduced in that paper: the existence of an infinite computation as well as the existence of a computation in which a specific event is raised are decidable. These decidability results are proved resorting to Petri Nets extended with transfer arcs, a class of Petri Nets in which termination, as well as coverability, are decidable properties [DFS98].

References

[BZ01] N. Busi and G. Zavattaro. Publish/Subscribe vs. Shared Dataspace Coordination Infrastructures. In *Proc. of IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE'01)*, volume 1782, pages 328–333. IEEE press, 2001.

[CLZ98] G. Cabri, L. Leonardi, and F. Zambonelli. Reactive Tuple Spaces for Mobile Agent Coordination. In *Proc. 2nd Int. Workshop on Mobile Agents*, volume 1477 of *Lecture Notes in Computer Science*, pages 237–248. Springer-Verlag, Berlin, 1998.

[CNF98] G. Cugola, E. Di Nitto, and A. Fuggetta. Exploiting an event-based infrastructure to develop complex distributed systems. In *Proc. of the 20th ICSE*, 1998.

[DFS98] C. Dufourd, A. Finkel, and P. Schnoebelen. Reset nets between decidability and undecidability. In *Proc. of ICALP'98*, volume 1061 of *Lecture Notes in Computer Science*, pages 103–115. Springer-Verlag, Berlin, 1998.

[Gel85] D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[Min67] M.L. Minsky. *Computation: finite and infinite machines*. Prentice-Hall, 1967.

[PMR99] G.P. Picco, A.L. Murphy, and G.-C. Roman. Lime: Linda meets mobility. In *Proc. of the 21th ICSE*, 1999.

[Row98] A. Rowstron. Wcl: A co-ordination language for geographically distributed agents. *World Wide Web Journal*, 1(3):167–179, 1998.

[RW98] A. Rowstron and A. Wood. Solving the Linda multiple `rd` problem using the `copy-collect` primitive. *Science of Computer Programming*, 31(2-3):335–358, 1998.

[SS63] J.C. Shepherdson and J.E. Sturgis. Computability of recursive functions. *Journal of the ACM*, 10:217–255, 1963.