ON THE SEMANTICS OF JAVASPACES

Nadia Busi Roberto Gorrieri Gianluigi Zavattaro

Dipartimento di Scienze dell'Informazione, Università di Bologna,

Mura Anteo Zamboni 7, I-40127 Bologna, Italy.

busi,gorrieri,zavattar@cs.unibo.it

Abstract JavaSpaces is a coordination middleware for distributed Java programming recently proposed by Sun Microsystems. It is inspired by the Linda coordination model: processes interact via the output ($write$), consumption ($take$), test for presence ($read$) and the test for absence ($takeIfExists$ and $readIfExists$) of data inside a shared repository. Three are the most interesting new features introduced by JavaSpaces. The first one is an *event notification* mechanism ($notify$): a process can register interest in the incoming arrivals of a particular kind of data, and then receive communication of the occurrence of these events. The second feature is the so-called *distributed leasing*: at the moment a client outputs a tuple, it also declares its required lifetime (that the server may satisfy only partially). The third feature is a *timeout* on the blocking operation of input: if no instance is found before the timeout expires, then the operation fails and terminates. We present a structured operational semantics for a process algebra featuring these coordination primitives. This abstract semantics is used to clarify possible ambiguities of the informal definitions of JavaSpaces, to discuss possible implementation choices and to compare the expressive power of the new primitives. Interestingly enough, many subtle phenomena occur, some of which might lead to reconsider the actual choice of primitives.

1. INTRODUCTION

Coordination middlewares are emerging as suitable architectures for making easier the programming of distributed applications. JavaSpaces [15] and TSpaces [17], produced by Sun Microsystem and IBM respectively, are the most prominent proposals; they are both based on the shared dataspace coordination architecture, originally proposed in Linda (see, e.g., [9]). The basic idea behind Linda is the so-called *generative communication*; its main features are the following:

- *Asynchronous communication*: it is realized by means of a (conceptually shared) communication medium (called tuple space) that is the actual place where all messages/tuples are delivered and extracted by the processes. A sender may proceed just after performing the insertion of the tuple in the tuple space, while the receiver can remove the tuple at any time after that tuple is in the tuple space. Hence, the asynchronous communication between the sender and the receiver is realized by means of two synchronous operations with the tuple space.

- *Read operation*: a process can check the presence of some tuple, without removing it from the space.
- *Conditional input/read predicates*: these are non-blocking variants of the remove and read operations; if the required message is absent, the process is not blocked and continues with a different alternative.

More recent extensions of the Linda paradigm (e.g., JavaSpaces and TSpaces) include some new primitives useful for coordination of complex applications in open, large, distributed systems. In this paper we will focus on some of them:

- *Event notification*. Besides the data-driven coordination typical of Linda, it may be very useful to include in a language an event-driven mechanism of process activation. A process can register its interest in future arrivals of some objects and then receive communication of each occurrence of this event.
- *Blocking operations with timeouts*. The operations of removal or reading of an object can be weakened by expressing the deadline beyond which the operation fails.
- *Guaranteed duration of service*. An object inserted in the dataspace as well as the interest in an event notification need not to hold forever; in many cases it is useful that the language has the capability to declare time bounds on these operations, and even better to re-negotiate such bounds if needed.

Despite of the clear relevance of such primitives for coordination middlewares, very little has been done to define formally their semantics. One may think that formalizing the intended semantics of these primitives is superfluous, as their semantics could appear obvious. Unfortunately, this is not the case. In places, the informal definition of these primitives in the available documentation is quite ambiguous (e.g., variants of the same primitives exist in different languages and even in the same language); this may have the effect of giving too much freedom in the implementation choices, hence producing semantically different implementations. Moreover, awareness of the expressiveness capabilities of the various primitives is often lacking, as well as methods for reasoning about programs written with these primitives.

The standard approach to solve the problems above is to give a formal semantics to the coordination language of interest. Such a semantics would fix the actual interpretation of the primitives, can be a precise guide for the implementor as well as a tool for reasoning about language expressiveness and program properties. For example, in [2] we have presented two alternative formal semantics for a Linda-based process calculus related to two possible interpretations of the output operations. In the first interpretation, called *ordered*, the processes are synchronous with the tuple space, while in the second one, called *unordered*, they are considered asynchronous. We have proved an interesting gap of expressiveness between the two interpretations showing that the calculus is Turing powerful under the ordered interpretation while this is not the case under the unordered one. Another interesting phenomenon is that this discrimination result holds only in the presence of test for absence operators; indeed, the calculus with only output, input, and read operations is not Turing powerful neither under the ordered interpretation.

In [6] we have initiated an investigation about the semantics of languages like JavaSpaces and TSpaces, by abstracting the coordination primitives away from the concrete language. To this aim, we use the mathematical machinery developed by the process algebra community (see, e.g., [13, 11]) that seems flexible enough to be useful for our aims. The approach adopted is to the following: we have started from a Linda-based process calculus and the we extend it with the new primitives sketched above: event notification, timeouts on blocking operations, leasing for timing service requests.

In this paper we report (simplified versions of) the process calculi presented in [6]. In addition to [6], we discuss for each of the extensions interesting subtle phenomena which occur when alternative interpretations of the semantics are considered. The alternative interpretations are of two kinds: *ordered v.s. unordered* interpretation for the output operation, and *synchronous v.s. asynchronous* interpretation for the passing of time.

2. THE KERNEL LANGUAGE

In this section we introduce the syntax and the operational semantics of a calculus comprising the basic Linda-like coordination primitives. It is a small variant of a previous calculus formerly presented in [1].

Let $Name$ be a denumerable set of object types ranged over by a, b, ..., and $Const$ be a set of program constants ranged over by K, K', ...

Let $Conf$ (ranged over by P, Q, ..., possibly indexed) be the set of the possible configurations defined by the following grammar:

$$\begin{array}{lcl} P & ::= & \langle a \rangle \;\mid\; C \;\mid\; P|P \\ C & ::= & \mathbf{0} \;\mid\; \mu.C \;\mid\; \eta?C\_C \;\mid\; C|C \;\mid\; K \end{array}$$

where:

$$
\begin{array}{llll}
(1) & \langle a\rangle \xrightarrow{\overline{a}} \mathbf{0} & (2) & \mathit{write}(a).P \xrightarrow{\tau} \langle a\rangle | P \\[1ex]
(3) & \mathit{take}(a).P \xrightarrow{a} P & (4) & \mathit{read}(a).P \xrightarrow{\underline{a}} P \\[1ex]
(5) & \mathit{take}\exists(a)?P\_Q \xrightarrow{a} P & (6) & \mathit{take}\exists(a)?P\_Q \xrightarrow{\neg a} Q \\[1ex]
(7) & \mathit{read}\exists(a)?P\_Q \xrightarrow{\underline{a}} P & (8) & \mathit{read}\exists(a)?P\_Q \xrightarrow{\neg a} Q \\[1ex]
(9) & \dfrac{P \xrightarrow{\overline{a}} P' \quad Q \xrightarrow{a} Q'}{P|Q \xrightarrow{\tau} P'|Q'} & (10) & \dfrac{P \xrightarrow{\overline{a}} P' \quad Q \xrightarrow{\underline{a}} Q'}{P|Q \xrightarrow{\tau} P|Q'} \\[3ex]
(11) & \dfrac{P \xrightarrow{\alpha} P'}{P|Q \xrightarrow{\alpha} P'|Q} \quad \alpha \neq \begin{cases} \neg a, \vec{a}, \\ \dot{a}, \surd \end{cases} & (12) & \dfrac{P \xrightarrow{\neg a} P' \quad Q \not\xrightarrow{\overline{a}}}{P|Q \xrightarrow{\neg a} P'|Q}
\end{array}
$$

$$
(13) \quad \frac{P \xrightarrow{\alpha} P' \qquad K = P}{K \xrightarrow{\alpha} P'}
$$

Table 1 Operational semantics (symmetric rules omitted).

$$
\begin{array}{lcl}
\mu & ::= & \mathit{write}(a) \ | \ \mathit{read}(a) \ | \ \mathit{take}(a) \\
\eta & ::= & \mathit{read}\exists(a) \ | \ \mathit{take}\exists(a)
\end{array}
$$

Programs are represented by terms C containing the coordination primitives; the dataspace is modeled by representing each of its objects a with a term $\langle a\rangle$. A configuration is composed of some programs and some available objects composed in parallel using the composition operator $|$. A program can be a terminated program **0** (which is usually omitted for the sake of simplicity), a prefix form $\mu.P$, an *if-then-else* form $\eta?P\_Q$, the parallel composition of subprograms, or a program constant K.

A prefix μ can be one of the primitives $\mathit{write}(a)$, which introduces a new object $\langle a\rangle$ inside the data repository, $\mathit{read}(a)$, which tests for the presence of an instance of object $\langle a\rangle$, and $\mathit{take}(a)$, which consumes an instance of object $\langle a\rangle$. The last two primitives are blocking, in the sense that a program performing one of them cannot proceed until the operation is successfully completed.

The guards of the if-then-else forms are $\mathit{read}\exists(a)$ and $\mathit{take}\exists(a)$, which are the non-blocking variants of $\mathit{read}(a)$ and $\mathit{take}(a)$, respectively. The notation is inspired by the similar operations *readIfExists* and *takeIfExists* of JavaSpaces [15]. Their behaviour is represented by using terms having two possible continuations, e.g., $\mathit{read}\exists(a)?P\_Q$. The first continuation P is chosen if the requested object is available in the data repository; in this case the non-blocking operations behave exactly as the corresponding blocking ones. On the other hand, if no instance of the requested object is actually available, the second continuation Q is chosen and the data repository is left unchanged.

Constants are used to permit the definition of programs with infinite behaviours. We assume that each constant K is equipped with one and only one definition $K = C$; as usual we assume also that only guarded recursion is used [13].

The semantics of the language is described via a labeled transition system $(\mathit{Conf}, \mathit{Label}, \longrightarrow)$ where $\mathit{Label} = \{\tau\} \cup \{a, \underline{a}, \overline{a}, \neg a \mid a \in \mathit{Name}\}$ (ranged over by $\alpha, \beta, \ldots$) is the set of the possible labels. The labeled transition relation $\longrightarrow$ is the smallest one satisfying the axioms and rules in Table 1. For the sake of simplicity we have omitted the symmetric rules of $(9) - (12)$.

Axiom (1) indicates that $\langle a \rangle$ is able to give its contents to the environment by performing an action labeled with $\overline{a}$. Axiom (2) describes the output operation: in one step a new object is produced. Axiom (3) associates to the action performed by the prefix $in(a)$ a label a, the complementary of $\overline{a}$, while axiom (4) associates to a $read(a)$ prefix a label $\underline{a}$.

Axioms (5) and (6) describe the semantics of $take\exists(a)?P\_Q$: if the required $\langle a \rangle$ is present it can be consumed (axiom (5)), otherwise, in the case $\langle a \rangle$ is not available, its absence is guessed by performing an action labeled with $\neg a$ (axiom (6)). Axioms (7) and (8) are the corresponding axioms for the $read\exists(a)$ operator; the unique difference is that the label $\underline{a}$ is used instead of a.

Rule (9) is the usual synchronization rule; while rule (10) deals with the new label $\underline{a}$ representing a non-consuming operation: in this case the term performing the output operation (labeled with $\overline{a}$) is left unchanged as the $read$ operation does not consume the tested object.

Rule (11) is the usual local rule, which is valid only for labels different from $\neg a$; indeed, an action of this kind can be performed only if no object $\langle a \rangle$ is available in the data repository, i.e., no actions labeled with $\overline{a}$ can be performed by the terms in the environment (rule (12)). The side condition of rule (11) considers also labels that will be defined in the following sections.

The last rule (13) allows a program constant K defined by $K = C$ to perform the same actions permitted to C.

Note that rule (12) uses a negative premise; however, the operational semantics is well defined, because the transition system specification is strictly stratifiable [10], condition that ensures (as proved in [10]) the existence of a unique transition system agreeing with it.

Reduction steps are those transformations which a configuration may have when considered stand-alone, in other words, without environment. Operationally, we denote reductions with $P \longrightarrow P'$ defined as follows:

$$P \longrightarrow P' \quad \text{if and only if} \quad P \xrightarrow{\tau} P' \text{ or } P \xrightarrow{\neg a} P' \text{ or } P \xrightarrow{\vec{a}} P' \text{ or } P \xrightarrow{\surd} P'$$

where the last two labels will be discussed in the following.

We denote by $P \uparrow$ the fact that P may give rise to an infinite sequence of reduction steps, and by $P \downarrow$ the fact that P has a finite sequence of reduction steps leading to a configuration without outgoing reductions.

2.1. ORDERED V.S. UNORDERED INTERPRETATION OF THE OUTPUT OPERATION

The semantics that we have defined assumes that the programs are synchronous with the dataspace, in the sense that the emitted object is surely already available inside the data repository at the moment the $write$ operation terminates (see rule (2)). In a previous paper on the semantics of Linda [3], this kind of semantics is called *ordered* and alternative interpretations are presented, among which the *unordered* one which assumes that programs are asynchronous with the dataspace: the emitted object becomes available after an unpredictable delay. Operationally, the unordered semantics is modeled by substituting the rule (2) with the two following:

$$(2') \quad write(a).P \xrightarrow{\tau} \langle\langle a\rangle\rangle|P \qquad\qquad (2'') \quad \langle\langle a\rangle\rangle \xrightarrow{\vec{a}} \langle a\rangle$$

Let $L_o[wrt]$ and $L_u[wrt]$ the calculus without the $read\exists$ and $take\exists$ primitives interpreted under the ordered and unordered interpretations, respectively; let $L_o[wrt\exists]$ and $L_u[wrt\exists]$ be the calculi interpreted under the two interpretations.

The paper [2] shows the existence of an expressiveness gap between these two alternatives. More precisely, adopting the result to the actual setting, we have that $L_o[wrt]$, $L_u[wrt]$, and $L_u[wrt\exists]$ are not Turing powerful as $P \uparrow$ and $P \downarrow$ are decidable; on the other hand, $L_o[wrt\exists]$ is Turing powerful and $P \uparrow$ and $P \downarrow$ are both undecidable.

The decidability results are proved by resorting to a finite Place/Transition net semantics (a formalism in which termination and divergence are decidable), while the undecidability results are consequence of the fact that it is possible to encode Random Access Machines (RAM), which is a well known register based Turing powerful formalism, into the considered language.

3. EVENT NOTIFICATION

In this section we extend the previous calculus with an event notification mechanism inspired by the $notify$ primitive of JavaSpaces [15].

The syntax of the kernel language is simply extended by permitting a new prefix:

$$\mu \quad ::= \quad ... \quad | \quad notify(a, C)$$

A program $notify(a, C).P$ can register its interest in the future incoming arrivals of the data of kind a, and then receive communication of each occurrence of this event. This behaviour can be modeled by introducing a new term

$$(2') \quad write(a).P \xrightarrow{\vec{a}} \langle a\rangle | P$$

$$(14) \quad notify(a,Q).P \xrightarrow{\tau} on(a,Q)|P \qquad (15) \quad on(a,P) \xrightarrow{\dot{a}} P|on(a,P)$$

$$(16) \quad \frac{P \xrightarrow{\dot{a}} P' \quad Q \xrightarrow{\dot{a}} Q'}{P|Q \xrightarrow{\dot{a}} P'|Q'} \qquad (17) \quad \frac{P \xrightarrow{\dot{a}} P' \quad Q \not\xrightarrow{\dot{a}}}{P|Q \xrightarrow{\dot{a}} P'|Q}$$

$$(18) \quad \frac{P \xrightarrow{\vec{a}} P' \quad Q \xrightarrow{\dot{a}} Q'}{P|Q \xrightarrow{\vec{a}} P'|Q'} \qquad (19) \quad \frac{P \xrightarrow{\vec{a}} P' \quad Q \not\xrightarrow{\dot{a}}}{P|Q \xrightarrow{\vec{a}} P'|Q}$$

Table 2 Operational semantics of $notify$ (symmetric rules omitted).

$on(a, C)$ (that we add to the syntax as an auxiliary term) which is a listener that spawns an instance of program C every time a new object $\langle a\rangle$ is introduced in the dataspace. This is modeled by extending the possible labels with the set $\{\dot{a}, \vec{a} \mid a \in Name\}$, by adding the rules in Table 2 (we omit the symmetric rules of $(17)-(19)$) to the ones in Table 1, and by substituting the rule $(2')$ for the rule (2). Negative premises are used in the new rules (see rules (17) and (19)), but the transition system specification is still stratifiable; thus, the operational semantics is well defined.

The new labels $\vec{a}$ and $\dot{a}$ represent the occurrence and the observation of the event "creation of a new object of kind a", respectively. This event happens when an output operation is executed; for this reason we change the label associated to the prefix $write(a)$ from τ to $\vec{a}$ (see the new rule $(2')$). Axiom (14) indicates that the $notify(a, P)$ prefix produces a new instance of the term $on(a, P)$. This term has the ability to spawn a new instance of P every time a new $\langle a\rangle$ is produced; this behaviour is described in axiom (15) where the label $\dot{a}$ is used to describe this kind of computation step.

Rules (16) and (17) consider actions labelled with $\dot{a}$ indicating the interest in the incoming instances of $\langle a\rangle$. If one program able to perform this kind of action is composed in parallel with another one registered for the same event, then their local actions are combined in a global one (rule (16)); otherwise, the program performs its own action locally (rule (17)). Rules (18) and (19) deal with two different cases regarding the label $\vec{a}$ indicating the arrival of a new instance of $\langle a\rangle$: if there are terms waiting for the notification of this event are present in the environment, then they are woken-up (rule (18)); otherwise, the environment is left unchanged (rule (19)).

In the following we use $L_o[wrt\exists n]$ and $L_u[wrt\exists n]$ to denote the language extended with the $notify$ primitive interpreted under the ordered and unordered semantics, respectively; on the other hand, $L_o[wrtn]$ and $L_u[wrtn]$ denote the two sublanguages without the $read\exists$ and $take\exists$ primitives.

3.1. EXPRESSIVENESS OF NOTIFY

Two of the authors have investigated the expressiveness of the $notify$ primitive in two related papers.

In [5] an interesting intermediary level of expressiveness is proved by showing that $L_o[wrtn]$ permits to simulate RAMs, but only in a weak sense: this means that the representation we give of RAMs in our language have several possible behaviours among which one which corresponds to the one of the considered RAM. Moreover, the presented encoding has the property that all the other computations are ensured to be infinite; thus the encoding preserves termination. For this reason, the property $P \downarrow$ is not decidable. On the other hand, it is proved that $P \uparrow$ is decidable by providing a divergence preserving encoding of the language in terms of Place/Transition nets extended with transfer arcs [8], a formalism in which the existence of an infinite computation is decidable. Moreover, it is also provided an encoding of the $notify$ primitive in the language without event notification but with the test for absence operations. The idea is that when a process registers its interest in some particular event, it introduces a corresponding object into the shared repository. Every time a $write$ operation is performed, a protocol composed of three phases must be executed: first it is necessary to count the number of registered listeners (by counting the corresponding objects in the shared repository), then to communicate to each of them the occurrence of the event, and finally to wait for the total amount of acknowledgements. Moreover, this protocol must be executed in mutual exclusion. All the results proved in [5] for the language $L_o[wrtn]$ hold also for $L_u[wrtn]$.

In [4] the relation between the ordered and unordered interpretation is revisited in the presence of $notify$. The interesting result is that the calculus becomes Turing powerful also under the unordered interpretation, indeed we prove that RAMs can be encoded into $L_u[wrt\exists n]$; moreover, the $notify$ primitive allows a faithful encoding of the ordered semantics on top of the unordered one.

All these results are reported in the Figure 1. The three layers of expressiveness are described recalling, for each of them, whether termination or divergence are decidable properties. We call the intermediary layer *weakly* Turing powerful for the fact that Turing formalisms can be simulated only in a weak sense. The existence of these layers ensures that it is not possible to encode one language inside a layer in another language inside a lower layer. Regarding the languages in the top layer, the arrow represents the existence of encodings from the language interpreted under the ordered semantics to the other two languages.

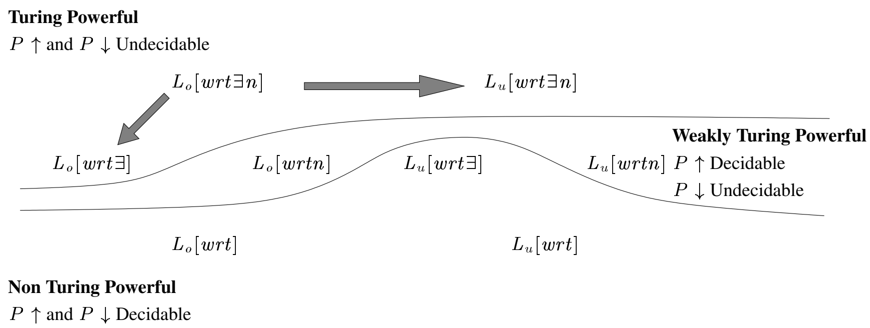

Figure 1 Overview of the results about $notify$.

$$(3') \quad take(a,t)?P\_Q \xrightarrow{\tau} take(a)_t?P\_Q$$

$$(4') \quad read(a,t)?P\_Q \xrightarrow{\tau} read(a)_t?P\_Q$$

$$(20) \quad take(a)_t?P\_Q \xrightarrow{a} P \qquad (21) \quad read(a)_t?P\_Q \xrightarrow{a} P$$

$$(22) \quad \eta_{t+1}?P\_Q \xrightarrow{\surd} \eta_t?P\_Q \qquad (23) \quad \eta_0?P\_Q \xrightarrow{\surd} Q$$

$$(24) \quad \frac{P \xrightarrow{\surd} P' \quad Q \xrightarrow{\surd} Q'}{P|Q \xrightarrow{\surd} P'|Q'} \qquad (25) \quad \frac{P \xrightarrow{\surd} P' \quad Q \not\xrightarrow{\surd}}{P|Q \xrightarrow{\surd} P'|Q}$$

Table 3 Operational semantics for timeouts (symmetric rule of (25) omitted).

4. TIMEOUT

In this section we consider the problem of modeling operations equipped with timeouts, which are used as extra parameters for the blocking operations $read$ and $take$ in order to indicate a maximum amount of time during which the presence of the requested object is investigated. If no instance is found before the end of this period, the operation fails and terminates.

In order to model this kind of behaviour, we change the primitives $read(a)$ and $take(a)$ with the new $read(a,t)$ and $take(a,t)$ operations (where t defines the timeout period). These operations are no more used as prefixes of terms of the kind $\mu.P$, but as guards of if-then-else forms $\eta?P\_Q$. In this way, we can describe both the program P, chosen if the operation succeeds, and the program Q, activated if, on the contrary, the operation fails.

The way we represent time is inspired by JavaSpaces, where the current time is represented by an integer which is incremented each millisecond. If the execution of an operation with timeout period t is scheduled when the current time is c, the operation fails at the end of the interval with current time $c + t$.

In our process calculus we do not use any value to represent the current time, but we only consider the passing of time, which is considered as divided into basic discrete intervals. Operationally, we model the instant in which an interval finishes, and the subsequent start, by means of a transition labelled with $\surd$. More precisely, we use $P \xrightarrow{\surd} P'$ to state that the term P becomes P' due to the fact that the current interval has finished, and the subsequent just started. In this way, transitions related to the passing of time (labeled with $\surd$) and standard transitions representing coordination operations (with label different from $\surd$) cohexist in our labelled transition system. Morever, we do not put any restriction on the way these two kinds of transitions are related one each other. For example, we do not fix any bound to the number of standard transitions which can be performed between two $\surd$ transitions. This choice reflects the fact that we do not make any assumption on the computation speed, thus neither on the number of coordination operations which may be performed during one basic time interval.

The terms of our process calculus are of two different kinds: those sensible to the passing of time and those which are not. In the first case, the terms have outgoing transitions labelled with $\surd$, while in the second they have not.

As an example, the term $\langle a \rangle$ representing an object of kind a inside the dataspace, has no outgoing transition labelled with $\surd$; on the other hand, the program $take(a)_t?P\_Q$, representing the execution of a $take$ operation with a remaining timeout period of t, has the transition $take(a)_t?P\_Q \xrightarrow{\surd} take(a)_{t-1}?P\_Q$.

Terms of the kind $take(a)_t?P\_Q$ cannot appear as initial programs of a configuration; they only arise as the result of the scheduling of a $take$ operation. The term $take(a,t)?P\_Q$ is not sensible to the passing of time until the execution of the $take$ operation is scheduled; this instant is represented by $take(a,t)?P\_Q \xrightarrow{\tau} take(a)_t?P\_Q$. The index t means that t complete intervals should pass before the operation fails; the failure is modeled by the transition $take(a)_0?P\_Q \xrightarrow{\surd} Q$ (in this way the failure happens after that $t-1$ transitions labelled with $\surd$ are executed).

We introduce the notation $\eta_t?P\_Q$ to denote either the term $take(a)_t?P\_Q$ or $read(a)_t?P\_Q$. The duration t is an integer number or a special symbol ∞ which denotes an infinite duration (we assume $\infty - 1 = \infty$).

The new syntax is obtained by removing the $read(a)$ and $take(a)$ prefixes and by extending the possible guards of the if-then-else forms:

$$\eta \ ::= \ \ldots \mid take(a,t) \mid read(a,t) \mid take(a)_t \mid read(a)_t$$

The new semantics informally described above is modeled by introducing the new label $\surd$ and the axioms and rules reported in Table 3 (the axioms $(3')$ and $(4')$ are substituted for (3) and (4), respectively).

Axioms (3') and (4') model the starting of the timeout periods. Axioms (20) and (21) represent a successful execution of these operations, while axioms (22) and (23) represent the passing of time; the subscript t in $\eta_t?P\_Q$ is decremented if it is not 0 (axiom (22)), otherwise the timeout period finishes and the second continuation is chosen (axiom (23)). The rules (24) and (25) describe how the structured term $P|Q$ behaves according to the passing of time. If both the processes have an outgoing transition labeled with $\surd$, they synchronize on the execution of this operation; on the other hand, one of the two processes can perform its own transition $\surd$ locally, only if the other one is not sensible to the passing of time, i.e., it has no outgoing transitions labeled with $\surd$. A negative premise is used in the rule (25), but the labelled transition system is still well defined.

Observe that a timeout period may terminate even if the sought object is available. Thus, when an operation fails we cannot conclude anything about the presence or absence of the requested object. We have adopted this weak semantics as it seems the intended interpretation of JavaSpaces; see [15], where in Section 2.5 we read "A `read` request ... will wait until a matching entry is found ... up to the timeout period". For example, in the configuration:

$$\langle a \rangle \mid read(a,t)?\mathbf{0}\_write(b)$$

object $\langle b \rangle$ may be produced if $\langle a \rangle$ is not found before the end of the timeout t.

4.1. SYNCHRONOUS V.S. ASYNCHRONOUS INTERPRETATION OF TIME PASSING

We have modeled configurations in which the passing of time is global, i.e., it is the same for all the components. According to this approach, the time passes synchronously. This is ensured by the side condition $\alpha \neq \surd$ of the locality rule (11), according to which a processes cannot perform locally its own transitions $\surd$. If we remove this side condition we obtain configurations in which the time passes asynchronously, as their components may or may not synchronize on the execution of their transitions labelled with $\surd$. In the following we use $P \longrightarrow_s^* P'$ to denote that a configuration P' can be reached from P via a sequence of reductions under the synchronous interpretation, while we use $P \longrightarrow_a^* P''$ to denote that P'' can be reached under the asynchronous one.

The synchronous approach is used to model centralized systems with a global clock, while the asynchronous approach corresponds to distributed systems where the global clock is not available. Distributed and centralized systems usually present strongly different behaviours; however, we prove that for the

$$(2'') \quad write(a,t)?P\_Q \stackrel{\tau}{\longrightarrow} \langle a\rangle_{t'}|P \quad \text{with } t' \le t$$
$$(2''') \quad write(a,t)?P\_Q \stackrel{\tau}{\longrightarrow} Q$$
$$(1') \;\; \langle a\rangle_t \stackrel{\overline{a}}{\longrightarrow} \mathbf{0} \qquad (26) \;\; \langle a\rangle_{t+1} \stackrel{\surd}{\longrightarrow} \langle a\rangle_t \qquad (27) \;\; \langle a\rangle_0 \stackrel{\surd}{\longrightarrow} \mathbf{0}$$

Table 4 Operational semantics for leasing (symmetric rules omitted).

calculus presented in this section this discrimination does not hold. On the other hand, we will see in the next section that the introduction of leasing in the calculus will permit to observe differences between the synchronous and the asynchronous interpretations.

The equivalence between the two approaches is a consequence of the following theorem, stating that, given an initial configuration P, the configurations that can be reached from P are exactly the same under the synchronous and the asynchronous approaches.

Theorem 1 *Let P be a configuration not including terms of the kind $\eta_t?P\_Q$. We have that $P \longrightarrow_s^* P'$ if and only if $P \longrightarrow_a^* P'$.*

The proof of the theorem is based on two observations: first, a computation under the synchronous approach is trivially valid also under the asynchronous one; second, the order of $\surd$ transitions in a computation under the asynchronous approach can be changed in order to obtain another valid computation leading to the same configuration such that the new computation can be mimicked under the synchronous approach. Intuitively, this can be done because there is no way to observe the instant in which a timeout period starts, instant which is modeled by transitions labelled with τ (see rules (3') and (4'))

5. LEASING

Leasing represents an emerging style of programming for distributed systems and applications. According to this style, a service offered by one object to another one is based on a notion of "granting the service for a certain period of time". In this way, objects which ask for services declare also the corresponding period of interest in that service. The server decides whether to grant the service for the complete period or for a shorter one. These are usually called leased services.

In JavaSpaces the notion of leasing is introduced in relation to the $write$ and $notify$ operations. In [6] we have modeled leased versions of these two operators by adding an extra parameter which represents the duration of the interval for which the emitted datum should be maintained inside the data repository (for write) or the amount of time after which the listener for the event should be removed (for notify). Moreover, two further primitives $renew$ and $cancel$ are introduced in order to extend the duration or to terminate the leasing period of a particular previously defined leased service.

For simplicity, in this paper we only present a modeling of leasing for the $write$ operation; moreover, we omit the $renew$ and $cancel$ operations (see [6] for the complete treatment).

The syntax of the calculus extended with leasing uses the $write$ operation no more as a prefix, but as a guard of if-then-else form:

$$\eta \quad ::= \quad \ldots \mid write(a,t)$$

In order to associate to the leased objects the indication of the remaining lifetime t, we use a subscript notation t. Namely, we represent objects with $\langle a \rangle_t$ to indicate that the represented object will be removed after t basic time intervals due to the expiration of the leasing period.

The operational semantics is redefined by adding the new axioms in Table 4 where $(1')$ is substituted for (1) while $(2'')$ and $(2''')$ are substituted for $(2')$.

Axiom $(2'')$ models a successful execution of an output operation; the side condition represents the fact that the new object may be leased with a lifetime shorter than the requested one. Axiom $(2''')$ indicates that the output operation may also fail; in this case the second possible continuation is activated. The other axioms describe the semantics for the shared objects $\langle a \rangle_t$ which may be read/consumed (axiom $(1')$), may reduce their remaining lifetime as effect of the termination of a basic time interval (axiom (26)), or may be removed as effect of lease expiration (axiom (27)).

5.1. SYNCHRONOUS V.S. ASYNCHRONOUS INTERPRETATION OF TIME PASSING

Also in the language extended with leasing, we could think to adopt either the synchronous or asynchronous interpretations of time passing described in the previous section. It is interesting to observe that the introduction of leased resources permits us to distinguish between them, formally, the Theorem 4.1 does not hold any more. As a counter example consider the following program:

$$write(a,t).read(b,t+1)?\mathbf{0}\_take(a)$$

After the execution of the write operation the following term is obtained:

$$\langle a \rangle_t | read(b,t+1)?\mathbf{0}\_take(a)$$

The right hand process requires the presence of object $\langle b \rangle$ in order to continue its execution. As this object will never be produced, its behaviour consists of waiting for a $t+1$ long period, and then becoming process $take(a)$. The object on the left has a lifetime shorter than $t+1$. According to the synchronous interpretation, $\langle a \rangle_t$ disappears before $take(a)$ can be performed, while this is not true under the asynchronous one. Thus, the $take(a)$ operation may succeed only under the asynchronous approach.

The above example shows that it is no more true that the configurations reachable from an initial configuration are exactly the same under the two interpretations. The example uses a read operation with timeout. An even more discriminating result between the two interpretations is proved in [7] for a calculus without event notification and timeouts; in other words, a calculus corresponding to the kernel language defined in Table 1 where the $write$ operation is substituted by the one defined in Table 4.

For this simple language we have proved a gap of expressiveness between the two interpretations for the passing of time by considering the ability to encode RAMs under synchronous and asynchronous time.

Concerning asynchronous time, we have presented a RAM encoding which preserves termination, and we show the impossibility to define encodings which preserves divergence. This result is achieved by proving that $P \uparrow$ is a decidable property in this language. Even more interesting is the analysis of synchronous time; we show two possible encodings, the first which preserves termination and the second which preserves divergence, and we prove the impossibility to define encodings which preserves both termination and divergence. In order to achieve this result we first observe that given an encoding which preserves both termination and divergence, we have that for any RAM, the corresponding encoding is *uniform* with respect to termination or divergence (i.e., its computations are either all finite or all infinite). After, we prove that $P \uparrow$ is decidable for uniform processes.

The above results are summarized in the following table where we report also the results proved regarding the relation between the ordered and unordered interpretations for the output operation proved in [2] and recalled in the Introduction. The table reports the properties that the encoding of RAMs may preserve:

	Termination	Divergence	Termination & Divergence
Ordered Output	YES	YES	YES
Leasing Sync. Time	YES	YES	NO
Leasing Async. Time	YES	NO	NO
Unordered Output	NO	NO	NO

6. CONCLUSION

We have proposed a formal semantics for primitives typically adopted by Linda-like coordination middlewares, with a particular emphasis on the innovations introduced in JavaSpaces specifications.

In particular we have focused on alternative interpretations of two kinds: *ordered v.s. unordered* interpretation for the output operation, and *synchronous v.s. asynchronous* interpretation for the passing of time. The most interesting results are:

1 The event notification mechanism introduces an intermediary level of expressiveness: the calculus with output, input, read, and notify is Turing powerful in the *weak* sense under both the ordered and unordered interpretations; the calculus extended with event notification becomes Turing powerful also under the unordered approach.

2 The synchronous and asynchronous interpretations for the passing of time are equivalent in the presence of time-outs only, while they are discriminated by an expressiveness gap in the presence of distributed leasing.

Moreover, further observations follows from the results reported in the table of Section 5.1. The critical point is that under leasing, when an output operation requires to place in the space a new object with a certain duration, "if the requested time is longer than the space is willing to grant, you will get a lease with reduced time". This implies that the classic, persistent, (ordered) output operation is no longer available. This suggests why the expressive power should decrease, and indeed we formally proved that this is the case. Moreover, if time is not global, then we loose also the possibility to observe the instant in which a lease expires. And this explains intuitively the second expressive gap. Finally, we have that a language where there is control over the time a datum is introduced in the dataspace, but little control on the time of permanence, is more expressive than a language where there is no control over the time a datum is introduced (unordered output), even if then the datum is persistent.

The results we have proved on our JavaSpaces based process calculi might lead to reconsider the choice of the coordination operators and their semantics. One idea, for example, could be to move from the ordered semantics to the unordered one; indeed, we have proved that the presence of $notify$ allows us to simulate the ordered semantics (and this simulation can be exploited only in those particular cases in which the order is strictly necessary). In other words, we are convinced that these results can be considered not only a first step in the direction of a better and formal understanding of the coordination model advocated by JavaSpaces, but they also represents insights useful in the design of new JavaSpaces-like languages.

References

[1] N. Busi, R. Gorrieri, and G. Zavattaro. A Process Algebraic View of Linda Coordination Primitives. *Theoretical Computer Science*, 192(2):167–199, 1998.

[2] N. Busi, R. Gorrieri, and G. Zavattaro. On the Expressiveness of Linda Coordination Primitives. *Information and Computation*, 156:90-121, 2000. Extended abstract appeared in Proc. of *Express'97*.

[3] N. Busi, R. Gorrieri, and G. Zavattaro. Comparing Three Semantics for Linda-like Languages. *Theoretical Computer Science*, to appear, 2000. Extended abstract appeared in Proc. of *Coordination'97*.

[4] N. Busi and G. Zavattaro. Event Notification in Data-driven Coordination Languages: Comparing the Ordered and Unordered Interpretations. In *Proc. of SAC2000*, pages 233-239. ACM Press, 2000.

[5] N. Busi and G. Zavattaro. On the Expressiveness of Event Notification in Data-driven Coordination Languages. In *Proc. of ESOP2000*, volume 1782 of *Lecture Notes in Computer Science*, pages 41-55. Springer-Verlag, Berlin, 2000.

[6] N. Busi, R. Gorrieri, and G. Zavattaro. Process Calculi for Coordination: from Linda to JavaSpaces. In *Proc. of AMAST2000*, volume to appear of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 2000.

[7] N. Busi, R. Gorrieri, and G. Zavattaro. On the Expressiveness of Distributed Leasing in Linda-like Coordination Languages. Technical report UBLCS-2000-5, Department of Computer Science, University of Bologna, Italy. May 2000.

[8] 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.

[9] D. Gelernter and N. Carriero. Coordination Languages and their Significance. *Communications of the ACM*, 35(2):97–107, 1992.

[10] J.F. Groote. Transition system specifications with negative premises. *Theoretical Computer Science*, 118:263–299, 1993.

[11] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[12] J. McClain. Personal communications. March 1999.

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

[14] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes. *Information and Computation*, 100(1):1–77, 1992.

[15] Sun Microsystem, Inc. *JavaSpaces Specifications*, 1998.

[16] Sun Microsystem, Inc. *Jini Distributed Leasing Specifications*, 1998.

[17] P. Wyckoff, S.W. McLaughry, T.J. Lehman, and D.A. Ford. T Spaces. *IBM Systems Journal*, 37(3):454-474, 1998.