Process Calculi for Coordination: from Linda to JavaSpaces*

Nadia Busi, Roberto Gorrieri, and 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. We present a collection of process calculi featuring coordination primitives for the shared dataspace coordination model (inspired by Linda, JavaSpaces and TSpaces), some of which have never been formally defined before. The operational semantics of the calculi is used to clarify possible ambiguities of the informal definitions of these languages, to discuss possible implementation choices, to compare the expressive power of the new primitives and, finally, to support formal reasoning about programs written with these primitives.

1 Introduction

Coordination middlewares are emerging as suitable architectures for making easier the programming of distributed applications. JavaSpaces [12] and TSpaces [14], produced by Sun 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., [6]). 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:

* Work partially supported by Esprit working group n.24512 "Coordina"

- *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.

In this paper we initiate 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., [10, 8]) that seems flexible enough to be useful for our aims. We start our investigation by presenting a kernel process calculus, comprising the coordination primitives of Linda, inspired by a previous proposal studied in [1]. We then enrich it incrementally by adding in isolation each of the primitives sketched above: event notification, timeouts on blocking operations, leasing for timing service requests. For each primitive we discuss possible interpretations that may offer rather different expressive power to the language or may be suitable for different implementations. Some examples are reported to illustrate the flexibility of the newly introduced primitives. Future research regarding formal analysis of programs are addressed in the conclusive section.

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

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

Table 1. Operational semantics (symmetric rules omitted).

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* be the set of the possible configurations defined by the following grammar:

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

where:

$$
\begin{array}{l}
\mu ::= \mathit{write}(a) \mid \mathit{read}(a) \mid \mathit{take}(a) \\
\eta ::= \mathit{read}\exists(a) \mid \mathit{take}\exists(a)
\end{array}
$$

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 $\mathbf{0}$ (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 $read\exists(a)$ and $take\exists(a)$, which are the non-blocking variants of $read(a)$ and $take(a)$, respectively. The notation is inspired by the similar operations *readIfExists* and *takeIfExists* of JavaSpaces [12]. Their behaviour is represented by using terms having two possible continuations, e.g., $read\exists(a)?P\_Q$. 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 [10].

The semantics of the language is described via a labeled transition system $(Conf, Label, \longrightarrow)$ where $Label = \{\tau\} \cup \{a, \underline{a}, \overline{a}, \neg a \mid a \in Name\}$ (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 [7], condition that ensures (as proved in [7]) the existence of a unique transition system agreeing with it.

Alternative Semantics The semantics 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 [2], 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. The paper [3] shows the existence of an expressiveness gap between these two alternatives. We have chosen the ordered interpretation as it is the semantics adopted by the actual JavaSpaces specifications, as indicated in the sections 2.3 and 2.8 of [12], and also confirmed us by personal communications with John McClain of Sun Microsystems Inc. [9].

3 Event Notification

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

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

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

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

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

In this section we extend the previous calculus with an event notification mechanism inspired by the *notify* primitive of JavaSpaces [12].

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

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

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 $on(a, C)$ (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 $\mathit{write}(a)$ from τ to $\vec{a}$ (see the new rule (2′)). Axiom (14) indicates that the $\mathit{notify}(a, P)$ 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)).

Example of a Client/Server System As an example of use of the *notify* primitive, we consider a simple client/server system; clients ask for services producing objects $\langle \mathit{request} \rangle$, and the server consumes these objects and produces a reply $\langle \mathit{service} \rangle$. The complete system is defined as follows:

$$
\begin{array}{ll}
\mathit{Clients\&Server} & = \mathit{notify}(\mathit{request}, \mathit{Server}).(\mathit{Client}_1 \mid \ldots \mid \mathit{Client}_n) \\
\mathit{Client}_i & = \mathit{write}(\mathit{request}).\mathit{take}(\mathit{service}) \\
\mathit{Server} & = \mathit{take}(\mathit{request}).\mathit{write}(\mathit{service})
\end{array}
$$

As its first operation, the server is produced by means of the execution of a $\mathit{notify}(\mathit{request}, \mathit{Server})$ operation. Afterward, several clients are spawn. Each of these clients simply produces a request and then waits for a service. Each time a $\langle \mathit{request} \rangle$ is produced, this activates a new *Server* exploiting the event notification mechanism. The server consumes the object $\langle \mathit{request} \rangle$ and produces a corresponding object $\langle \mathit{service} \rangle$.

Observe that the notification mechanism allows the concurrent execution of more than one server; this happens if a request is produced before the previous one is served. According to this observation, we can say that we model a multithreaded server.

Alternative Semantics Observe that the synchronization between the action $\vec{a}$ and $\dot{a}$ is modeled as a synchronous multicast; when an action $\vec{a}$ occurs, it synchronizes with all the possible actions $\dot{a}$ that the environment may perform. The fact that all the interested listener are involved in the synchronization is ensured by the negative premises of the rules (17) and (19) and by the side condition $\alpha \neq \dot{a}$ of the locality rule (11). In this way, we model a reliable service. On the other hand, the event notification mechanism is sometimes a best-effort service such as in JavaSpaces [12], where some listeners may be not notified of the occurrence of an event in which they have registered their interest.

We can model a best-effort event notification service simply by removing the negative premise of the rules (17) and (19).

Expressiveness of Notify Regarding the expressiveness of the *notify* we first observe that it introduces an implicit form of recursion, then we address the problem of implementing the new operations using only primitives of the kernel language.

The *notify* primitive permits us to describe processes with an infinite behaviour without beeing forced to use recursive definitions. This because the listener $on(a, P)$, which is the result of a $notify(a, P)$ operation, is able to repeatedly produce a new program P every time the corresponding event occurs. For example, this mechanism can be exploited to simulate the behaviour of a term which recursively renames all objects $\langle a \rangle$ in $\langle b \rangle$ using the following non-recursive definition:

$$\begin{aligned} Trans &= notify(rec, Rename).write(rec) \\ Rename &= take(rec).take(a).write(b).write(rec) \end{aligned}$$

where the name *rec* is used to repeatedly activate an instance of a one-shot renaming program. In the kernel calculus introduced in the previous section, this kind of behaviour can be defined only by exploiting a recursive definition.

Another interesting question on the expressiveness of the *notify* primitive regards the possibility to encode it in terms of the coordination primitives of the kernel language, only. For example, one could think to simulate the above multi-threaded server without using the event notification mechanism. The answer to this question has been given in [5], where it is shown how to encode the *notify* primitive in a language similar to the kernel calculus of the previous section. Thus, we can conclude that a similar client/server system can be obtained also without using the event notification mechanism. However, the proposed encoding is rather complex; every time a new object is created, the sender must perform a protocol composed of three phases: first it is necessary to count the number of registered listener, then to communicate to each of them the happening of an interesting event, and finally to wait for the total amount of acknowledgements. Moreover, this protocol must be executed in mutual exclusion. For this reason, we can conclude that even if the *notify* primitive does not strictly increase the expressive power of the language, it is very useful for an easier, more direct, and more natural definition of interesting coordination patterns, such as the one adopted by the above multithreaded server. The encoding of *notify* presented in [5], strictly requires the test for absence incorporated in the $take\exists$ (or $read\exists$). This allows to conclude that *notify* strictly increases the expressive power of a language including only the *write*, *read*, and *take* basic operations.

In a related paper [4] we have investigated the interplay between the ordered and unordered interpretations of *write* (that we have described in the previous section) with the *notify* primitive. An interesting result is that *notify* permits us to encode the ordered approach on top of the unordered one.

4 Timeout

(3′)	$take(a,t)?P\_Q \xrightarrow{\tau} take(a)_t?P\_Q$	(4′)	$read(a,t)?P\_Q \xrightarrow{\tau} read(a)_t?P\_Q$
(20)	$take(a)_t?P\_Q \xrightarrow{a} P$	(21)	$read(a)_t?P\_Q \xrightarrow{a} P$
(22)	$\eta_t?P\_Q \xrightarrow{\surd} \eta_{t-1}?P\_Q \quad$ if $t \neq 0$	(23)	$\eta_0?P\_Q \xrightarrow{\surd} Q$
(24)	$\dfrac{P \xrightarrow{\surd} P' \quad Q \xrightarrow{\surd} Q'}{P\vert Q \xrightarrow{\surd} P'\vert Q'}$	(25)	$\dfrac{P \xrightarrow{\surd} P' \quad Q \not\xrightarrow{\surd}}{P\vert Q \xrightarrow{\surd} P'\vert Q}$

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

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 succeds, 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 our process calculus we represent two kinds of terms: 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, we define 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 the transition $take(a,t)?P\_Q \xrightarrow{\tau} take(a)_t?P\_Q$. 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 successfull 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 [12], 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.

Using Timeout The operations with timeout are particularly useful in a scenario where the test for absence, incorporated in the nonblocking operations *take*∃ and *read*∃, is difficult to implement. For example, in distributed implementations of the shared dataspace, a test for absence requires the snapshot of the whole distributed data repository. In order to avoid this, we may obtain nonblocking input operation by using timeouts. As an example, consider a simple program which has to consume one object which could be either $\langle a \rangle$ or $\langle b \rangle$ before terminating. In the kernel language this program could be defined as follows:

$$Consume \;=\; take\exists(a)?\mathbf{0}\_(take\exists(b)?\mathbf{0}\_\,Consume)$$

If the first $take\exists(a)$ operation requires a long time to be performed, the whole program could be negatively affected by this. For this reason, the following alternative program could be more efficient:

$$NewConsume \;=\; take(a,t)?\mathbf{0}\_(take(b,t)?\mathbf{0}\_\,NewConsume)$$

provided that the timeout t is defined appropriately.

The use of timeouts seems particularly useful also when the dataspace service is not reliable. Consider, as an example, a server implementing the dataspace which fails during the execution of a blocking *read* or *take* operation; the corresponding client is blocked until the server recovers. On the other hand, in the presence of timeout, the intended semantics allows the client to restart at least its local computation at the moment the indicated timeout period finishes.

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 in the transition system 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 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 (that we omit due to space limits) 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

$$(2'') \quad (\nu l)write(a,t).P \xrightarrow{\tau} \langle a\rangle_{l':t} | P\{l'/l\} \qquad \text{with } l' \text{ fresh}$$

$$(14') \quad (\nu l)notify(a,Q,t).P \xrightarrow{\tau} on(a,Q)_{l':t} | P\{l'/l\} \qquad \text{with } l' \text{ fresh}$$

$$(1') \quad \langle a\rangle_{l:t} \xrightarrow{\overline{a}} \mathbf{0} \qquad\qquad (15') \quad on(a,P)_{l:t} \xrightarrow{\dot{a}} P | on(a,P)_{l:t}$$

$$(26) \quad cancel(l)?P\_Q \xrightarrow{\dagger l} P \qquad\qquad (27) \quad cancel(l)?P\_Q \xrightarrow{\neg\dagger l} Q$$

$$(28) \quad renew(l,t)?P\_Q \xrightarrow{l:t} P \qquad\qquad (29) \quad renew(l,t)?P\_Q \xrightarrow{\neg l:t} Q$$

$$(30) \quad P_{l:t} \xrightarrow{\overline{\dagger l}} \mathbf{0} \qquad\qquad (31) \quad P_{l:t'} \xrightarrow{\overline{l:t}} P_{l:t}$$

$$(32) \quad P_{l:t} \xrightarrow{\surd} P_{l:t-1} \quad \text{if } t \neq 0 \qquad\qquad (33) \quad P_{l:0} \xrightarrow{\surd} \mathbf{0}$$

$$(34) \quad \frac{P \xrightarrow{\neg\lambda} P' \quad Q \not\xrightarrow{\overline{\lambda}}}{P|Q \xrightarrow{\neg\lambda} P'|Q} \qquad \lambda = \dagger l \quad \text{or} \quad \lambda = l:t$$

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

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. These are usually called leased services.

As it happens in JavaSpaces, we could introduce the notion of leasing for the *write* and *notify* operations. In our calculus we obtain this by adding to these operations a 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).

Inspired by JavaSpaces, we permit to cancel or renew previously leased services. In order to obtain this, each leased service is provided with a *unique identifier* which is used by two new operations called *cancel* and *renew* in order to indicate the service to be cancelled or renewed, respectively. Let *Lease*, ranged over by l, l', ..., be the set of lease identifiers.

The syntax of the calculus redefines the possible prefixes μ and extends the guards η:

$$\begin{aligned}\mu &::= (\nu l)\mathit{write}(a,t) \;\mid\; (\nu l)\mathit{notify}(a,C,t).P\\ \eta &::= \ldots \;\mid\; \mathit{cancel}(l) \;\mid\; \mathit{renew}(l,t)\end{aligned}$$

The notation and the mechanism we use to model the lease identifiers has been inspired by the π-calculus [11], where $(\nu l)P$ binds the name l in P. Similarly, we use $(\nu l)\mathit{write}(a,t).P$ and $(\nu l)\mathit{notify}(a,C,t).P$ to bind l in P. This reflects the fact that a leased object can be accessed (via the *renew* and *cancel* operations) only in the continuation of the *write* or *notify* operation which produced it. Operationally, we avoid name clashes by dynamically creating a fresh name (not previously used), to be substituted for the name l in the process P, whenever the *write* or *notify* operation is executed. For example, when a $(\nu l)\mathit{write}(a,t).P$ process performs its *write* operation, a new leased object is produced with a life length t and a fresh leasing identifier l'; this fresh name is substituted for the name l in P (see axiom (2")). The same mechanism is used also in axiom (14') for the new version of the *notify* primitive.

In order to associate to the leased resources the indication of the corresponding leasing identifier l and the remaining time t, we use a subscript notation $l:t$. Namely, we represent objects and listeners with $\langle a\rangle_{l:t}$ and $on(a,P)_{l:t}$, respectively. As a shorthand notation we use $P_{l:t}$ to denote a term which is either $\langle a\rangle_{l:t}$ or $on(a,P)_{l:t}$. Axioms (1') and (15') adapt the corresponding (1) and (15) to deal with leased resources.

The *cancel* and *renew* operations are used as guards of if-then-else forms because the operations could either succeed or fail, according to the presence or absence of the required leased resource. The execution of a cancel on the leased object l is denoted by the new label $\dagger l$, while its failure by $\neg\dagger l$ (see axioms (26) and (27)). On the other hand, the redefinition to t of the time granted to the leased object l is denoted by the label $l:t$, while its failure by $\neg l:t$ (see axioms (26) and (27)). The axioms (30) and (31) model the complementary operations of $\dagger l$ and $l:t$, respectively, which may be performed at any moment by the leased object l.

In order to execute a transition labelled with $\neg\dagger l$ (resp. $\neg l:t$) it is required that the environment does not contain any leased object with identifier l, formally, that it cannot perform any transition labeled with $\dagger l$ (resp. $l:t$). This is described by the rule (34).

The operational semantics is redefined by adding the new labels in $\{\dagger l\ ,\ l\!:\!t\ ,\ \overline{\dagger l}\ ,\ \overline{l\!:\!t}\ ,\ \neg\dagger l\ ,\ \neg l\!:\!t \mid l \in \mathit{Lease}, t \in I\!\!N \cup \{\infty\}\}$ and the new axioms in Table 4.

Example of a Seat Reservation Service As an example of use of leased resources we consider a seat reservation system for trains. The idea is to associate to each train a process responsible for seat booking. The process should expire at the leaving of the train.

$$\begin{array}{ll} \mathit{ResTrain} & = (\nu l)\mathit{notify}(\mathit{req}, \mathit{Reserve}, \mathit{time}).\mathit{CheckTrain}_l \\ \mathit{Reserve} & = \mathit{take}(\mathit{req}).\mathit{write}(\mathit{seat}, \infty) \end{array}$$

The process $\mathit{CheckTrain}_l$ is responsible to manage variations of the train scheduling. For example, if it is cancelled, the reserving process should be removed, while if the departure is delayed, the remaining lifetime should be changed. Thus, $\mathit{CheckTrain}_l$ (parametric in the name of the leasing identifier) can be defined as follows:

$$\begin{array}{rl} \mathit{CheckTrain}_l = & \mathit{take}(\mathit{delay}).\mathit{renew}(l, \mathit{newtime})?\mathit{CheckTrain}_l\_\mathbf{0} \mid \\ & \mathit{take}(\mathit{cancel}).\mathit{cancel}(l)?\mathbf{0}\_\mathbf{0} \end{array}$$

Alternative Semantics 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:

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

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

$$\langle a \rangle_{l:t} | \mathit{read}(b, t+1)?\mathbf{0}\_\mathit{take}(a)$$

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 $\mathit{take}(a)$. The object on the left has a lifetime shorter than $t+1$. According to the synchronous interpretation, $\langle a \rangle_{l:t}$ disappears before $\mathit{take}(a)$ can be performed, while this is not true under the asynchronous one. Thus, the $\mathit{take}(a)$ operation may succeed only under the asynchronous approach.

Another alternative interpretation concerns the granting policy. We model the leasing mechanism referring to an idel situation in which the leased services are granted exactly for the requested amount of time. This is not always possible in situations in which resources are limited. For example, in the Jini Distributed Leasing Specifications [13], the service provider may decide to grant the resource for a shorter period. We can easily adapt our semantics in order to take into account this feature by replacing rules (2") and (14') with the following:

$$(2'') \quad (\nu l)write(a,t).P \xrightarrow{\tau} \langle a\rangle_{l':t'}|P\{l'/l\} \qquad \text{with } l' \text{ fresh and } t' \leq t$$

$$(14') \quad (\nu l)notify(a,Q,t).P \xrightarrow{\tau} on(a,Q)_{l':t'}|P\{l'/l\} \qquad \text{with } l' \text{ fresh and } t' \leq t$$

The following example shows the differences between our approach and this alternative interpretation:

$$write(a,10).write(b,20).take\exists(b)?\mathbf{0}\_(take(a).write(c))$$

According to the ideal approach, we are sure that object $\langle b\rangle$ cannot expire before $\langle a\rangle$; thus object $\langle c\rangle$ will never be produced. On the other hand, if we move to the Jini-like interpretation, the object $\langle b\rangle$ may be granted for a shorter period; thus, $\langle c\rangle$ could be produced.

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.

As a first outcome, this analysis enlights some interesting unaddressed choices in the design of these languages which may influence the overall beaviour. For example, in Section 2 we discuss two alternative semantics for the *write* operation, and in Section 4 we present two possible variants for the representation of time.

We plan to use the formal framework developed here for the analysis and verifications of properties of systems modeled according to the coordination paradigm of JavaSpaces. This is particularly useful in this context as, when dealing with concurrent systems, it is quite difficult to intuitively gain a clear view of all possible computations, thus leaving undetected the presence of undesired behaviours. The adoption of a process algebraic approach permits us to address this kind of problems by adapting the large amount of formal techniques developed in this area.

For example, consider a master/worker system where the master produces some jobs that will be performed by the workers; only when all jobs have been processed, the whole system terminate. As a first attempt we may represent the system as follows:

$$\begin{aligned} System &= Master \mid Worker \mid \ldots \mid Worker \\ Master &= write(job).write(job).\ldots.write(job) \\ Worker &= take\exists(job)?(write(result).Worker)\_\mathbf{0} \end{aligned}$$

At a deeper analysis, we can find an undesired behaviour: if the workers perform their first test before the master starts producing the jobs, the final configuration will contain unprocessed jobs.

The problem addressed above can be fixed by blocking the workers until all the jobs are produced. The correct specifications is obtained by redefining the master and the worker as follows:

$$\begin{aligned} Master &= write(job).write(job).\ldots.write(job).write(start) \\ Worker &= read(start).take\exists(job)?(write(result).Worker)\_\mathbf{0} \end{aligned}$$

The correctness of this specification can be formally proved by showing that each computation terminates in a state consisting of the proper number of objects $\langle result \rangle$ (i.e. equal to the number of $write(job)$ operations performed by the master process).

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. Comparing Three Semantics for Linda-like Languages. *Theoretical Computer Science*, to appear, 2000. Extended abstract appeared in Proc. of *Coordination'97*.
3. N. Busi, R. Gorrieri, and G. Zavattaro. On the Expressiveness of Linda Coordination Primitives. *Information and Computation*, to appear, 2000. Extended abstract appeared in Proc. of *Express'97*.
4. N. Busi and G. Zavattaro. Event Notification in Data-driven Coordination Languages: Comparing the Ordered and Unordered Interpretations. In *Proc. of SAC2000*. 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 to appear of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 2000.
6. D. Gelernter and N. Carriero. Coordination Languages and their Significance. *Communications of the ACM*, 35(2):97–107, 1992.
7. J.F. Groote. Transition system specifications with negative premises. *Theoretical Computer Science*, 118:263–299, 1993.
8. C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
9. J. McClain. Personal communications. March 1999.
10. R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
11. R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes. *Information and Computation*, 100(1):1–77, 1992.
12. Sun Microsystem, Inc. *JavaSpaces Specifications*, 1998.
13. Sun Microsystem, Inc. *Jini Distributed Leasing Specifications*, 1998.
14. P. Wyckoff, S.W. McLaughry, T.J. Lehman, and D.A. Ford. T Spaces. *IBM Systems Journal*, 37(3), 1998.