A Process Algebraic View of Shared Dataspace Coordination

Nadia Busi and Gianluigi Zavattaro

Dipartimento di Scienze dell'Informazione, Università di Bologna,
Mura Anteo Zamboni 7, I-40127 Bologna, Italy
e-mail:{busi,zavattar}@cs.unibo.it

Abstract

Coordination languages were introduced in the early 80's as programming notations to manage the interaction among concurrent collaborating software entities. Process algebras have been successfully exploited for the formal definition of the semantics of these languages and as a framework for the comparison of their expressive power. We provide an incremental and uniform presentation of a collection of process calculi featuring coordination primitives for the shared dataspace coordination model (inspired by Linda, JavaSpaces, TSpaces, and the like). On the one hand, the incremental presentation of the various calculi permits to reason about specific linguistic constructs of coordination languages. On the other hand, the uniform presentation of a family of related calculi allows us to obtain an overview of the main results achieved in the literature on different (and unrelated) calculi.

Key words: process calculi, coordination models and languages, tuple spaces, event notification, transactions

1 Introduction

In parallel and distributed computing systems, as also in the human community, the level of connectivity is dramatically increasing. In this new scenario, the components of structured systems surely need to define new ways to work, cooperate, and collaborate in their activities. The study of *coordination* investigates strategies in order to obtain benefits from the new connecting technologies.

The interdisciplinary study of coordination has been advocated for the first time by Malone and Crowston in [32]. Their thesis is that the study of coordination structures analogous to those used in bee hives or ant colonies may be

useful for certain aspects of human organizations, or that lessons learned about coordination in biological and human systems could illuminate in the analysis of the right tradeoffs between computing and communicating in distributed computer systems.

In the area of computer science the terms *coordination models* and *coordination languages* have been introduced by Gelernter and Carriero [26]. According to their approach, a complete (concurrent) programming model should be obtained as a combination of two separate pieces: the computation model and the coordination model. At the beginning of the 80's they introduced the coordination language Linda [25], and its shared dataspace coordination model, as a first proposal in this direction. Since then, dozens of coordination languages and models have been defined (see, e.g., [42] for references). More recently, Linda has received a renewed interest due to the introduction of coordination middlewares, such as JavaSpaces [38] and TSpaces [46] produced by Sun Microsystems and IBM, respectively, which are both based on the shared dataspace coordination model.

The original Linda model exploits a common communication medium which is a repository that can be accessed via a set of coordination primitives:

- *Input/Output operations*: asynchronous communication is realized by means of a (conceptually shared) communication medium (called dataspace) that is the actual place where all data exchanged by the coordinating processes are delivered to and extracted from. A sender may proceed just after performing an output operation which inserts the datum in the dataspace, while the receiver can remove the datum at any time after that datum is in the dataspace by performing an input operation. Hence, the asynchronous communication between the sender and the receiver is realized by means of two synchronous operations with the dataspace.
- *Read operation*: besides the input operation, also a non-consuming operation which does not remove the read datum is provided.
- *Conditional input/read predicates*: these are non-blocking variants of the remove and read operations; if the required datum is absent, the process is not blocked and continues with a different alternative.

Several proposals for extending the original single shared dataspace coordination model of Linda have been presented. In this paper, we analyse some of them which are particularly of interest when the shared dataspace model is considered for the development of applications in open, large, distributed systems.

- *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.* A datum inserted in the dataspace, as well as the interest in an event notification, need not 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.
- *Transactions.* A set of coordination operations can be grouped in a transaction, and executed in such a way that either all of them succeed or none of them is performed.

Most of these extensions have been first introduced and investigated in isolation in separated proposals and, more recently, they have been all combined in the JavaSpaces coordination service. For example, event notification has been introduced in the context of shared dataspace coordination by the so called reactive tuple spaces model supported by MARS [19] and TuCSoN [40] as well as the WCL coordination language [45], which comprises the coordination primitive *monitor*, used to allow a process to receive a copy of all the data (satisfying a certain pattern) which are already available in the dataspace, as also of those which will be subsequently produced. Another shared dataspace coordination language permitting to associate reactions to event is LIME [43] which comprises two kinds of reactive mechanisms: the local one that permits to execute and complete the program activated by the occurrence of an event (thus blocking the access to the dataspace to other concurrent processes) and the remote one according to which a reaction is activated after the occurrence of the event and executed concurrently with the other processes accessing the dataspace.

As far as timeouts are concerned, they have been investigated already in the context of Objective Linda [31], a proposal for an object based coordination infrastructure particularly suitable for open systems: they are the basic mechanism used to ensure that clients do not block forever waiting for servers which are no more available. Finally, transactions were already present in Persistent Linda [1], a proposal for making a Linda repository more similar to a database in such a way that, e.g., extended matching mechanisms can be exploited in order to perform more advanced queries on the actual state of the dataspace.

1.1 Paper contribution

In this paper we survey the main results achieved in more than a decade of research about the investigation, using a process algebraic approach, of the

foundations of shared dataspace coordination. In fact, before the exploitation of process calculi, very little was done to define formally the semantics of the proposed coordination languages. One may think that formalizing the intended semantics of the coordination 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 sometimes 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 for a given language. Moreover, in the absence of a formal semantics, awareness of the expressiveness capabilities of the various primitives is often lacking, as well as methods for reasoning about programs written with these primitives.

Process calculi have been exploited to solve the problems above permitting to give a formal semantics to the coordination languages of interest. Such a semantics fixes the actual interpretation of the primitives and can be a precise guide for the implementor as well as a tool for reasoning about language expressiveness and program properties. The main contribution of this paper is to provide an incremental and uniform presentation of a collection of process calculi featuring coordination primitives for the shared dataspace coordination model (inspired by Linda, JavaSpaces, TSpaces, and the like). On the one hand, the incremental presentation of the various calculi permits to reason about specific linguistic constructs of coordination languages. On the other hand, the uniform presentation of a family of related calculi allows us to obtain an overview of the main results achieved in the literature on different (and unrelated) calculi.

In order to concentrate on the coordination primitives, in the process calculi that we present we abstract away from the concrete programming language in which the coordination primitives are actually embedded (e.g., C or Fortran for Linda, and Java for JavaSpaces and TSpaces). Another difference which distinguishes the considered coordination languages is in the kind of data that can be introduced in the dataspace: for example, Linda uses tuples (ordered sequences of data) whilst JavaSpaces and TSpaces consider Java objects. In order to be general, the process calculi that we present abstract also from the structure of the data.

Figure 1 reports the collection of process calculi that we define and discuss in this paper. The lines in Figure 1 simply denote language inclusion. We start with a kernel calculus `L` comprising only *read*, *take*, and *write* operations used to test the presence, consume, and produce data, respectively; then we enrich the kernel calculus introducing in isolation test for absence operations `L[`$\exists$`]`, event notification `L[ntf]`, timeouts `L[`Δ`]`, leasing for timed services `L[lsn]`, and transactions `L[txn]`. We also define some calculi comprising more than

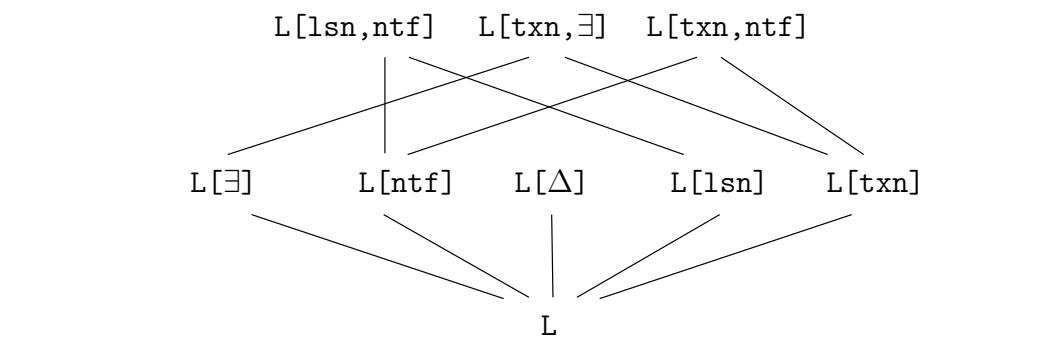

Fig. 1. The process calculi defined in this paper.

one of these features: `L[lsn,ntf]` that combines leased resources and event notification, `L[txn,∃]` that comprises transactions and test for absence operations, and `L[txn,ntf]` that includes both transactions and event notification.

Many other calculi can be easily obtained putting together the rules defining the syntax and the semantics of the corresponding subcalculi. For example, a calculus `L[∃,ntf]` (not formally defined in this paper) with test for absence operations and event notification can be defined simply by combining the syntax rules of `L[∃]` and those of `L[ntf]`. We anticipate that the unique calculi which cannot be obtained as simple combination of subcalculi are those comprising both transactions and timed primitives (timeout or leasing). The reason is that we model transactions as executed atomically in a single move. In presence of timed primitives, a less abstract approach should be taken, in which the time required for the execution of the single actions inside a transaction is counted. For this reason, the interplay between transactions and timing issues is left for a separate analysis.

The collection of process calculi that we present permits us to perform a comparison of the basic features of the several calculi thus obtaining insights on the differences and similarities among different sets of coordination primitives. Throughout the paper many results of this kind are reported; for example, in Section 4.3 you can find the description of an interesting hierarchy of expressiveness (see Section 1.2 for a description of the notion of expressiveness that we consider) according to which event notification strictly increases the expressive power of the basic *read*, *write*, and *take* coordination primitives, whilst it remains strictly less expressive than the test for absence coordination operations.

Another interesting outcome of our analysis is the clarification of some unaddressed choices in the design of these languages which may influence the overall behaviour. Throughout the paper many alternative design choices for the same coordination primitives are discussed. Among them, the most interesting is perhaps the comparison between two alternative semantics for the

write operation, called ordered and unordered respectively. In Section 2.1 we show that the two semantics are equivalent in the context of the kernel calculus L. On the other hand, in Section 3.2 we show that there exists a strong discrimination between these two semantics in the calculus L[∃] (which comprises also test for absence operators): namely, the calculus is Turing powerful under the ordered semantics while this is not the case under the unordered one.

1.2 Expressiveness in concurrency

As stated above, the various calculi that we present will be compared also on the basis of their expressive power. Several notions of expressiveness for concurrent languages are present in the literature, thus a clarification about which we are going to consider is needed.

In the context of concurrent process calculi, there exists no absolute notion of expressiveness equivalent to Turing completeness for sequential languages. The main tool for the comparison of the relative expressive power of two calculi, for instance L and L', is given by the investigation of relative encodings of L into L' (and vice versa). Different notions of expressiveness differ in the constraints imposed on properties of the encodings. Two kinds of constraints are usually considered, *syntactic* and *semantic* constraints.

For instance, the modular embeddings considered in [7] imposes the following syntactic constraint: the encoding should be a homomorphism with respect to the composition operators of the source calculus (this constraint can be considered only if the composition operators of L are present also in L'). As far as semantic constraints are concerned, an interesting example is given by the preservation of any reasonable semantics imposed in the encodings analysed in [41]: a semantics is reasonable if it discriminates two systems S and S' whenever there exists a computation of S in which the intended observable actions are different from the observables of any possible computation of S'.

In this paper we consider different notions of expressiveness that can be used to discriminate the considered calculi. The most relevant notion is based on the decidability of properties such as the existence of a terminating computation for the process P, denoted with $P\downarrow$, and the existence of an infinite computation for the process P, denoted with $P\uparrow$. We show that these properties are decidable in some of the calculi, while they are not in other ones. Consider, for instance, that $P\downarrow$ (resp. $P\uparrow$) is decidable in L while this is not the case for L'. We can conclude that there exists no encoding from L into L' that preserves the existence of an finite (resp. infinite) computation. In this case, we do not consider syntactic constraints but only semantic constraints: we distinguish

two systems S and S' if one has a finite (resp. infinite) computation, while the other one has not.

In some other minor cases, we consider different notions of expressiveness inspired by the modular embeddings of [6]; we will describe the actually considered notion in the corresponding sections.

1.3 Structure of the paper

Most of the results that are discussed in this paper have been proved in related work of the authors [10–15,17,18,48,47]: the main contribution of this paper is in the homogeneous presentation and overview of these results. In this paper, we usually report the results in an informal way giving more emphasis to their impact on the design of coordination languages than to their technical relevance. The reader interested in the details of the proofs of the results, may refer to the related literature of the authors. In the conclusive section, we report a detailed discussion about our previous work and its relation with the present paper.

The paper is structured as follows: in Section 2 we introduce the kernel calculus, while in the Sections 3–7 we consider conditional predicates, event notification, timeouts on blocking operations, leasing for timing service requests, and transactions, respectively. Finally, Section 8 reports some concluding remarks and a discussion about the related literature.

2 The Kernel Calculus: `L`

In this section we introduce the syntax and the operational semantics of a calculus called `L` comprising the basic Linda-like coordination primitives for producing, reading, and consuming data. This first calculus represents the common root of Linda-like languages. For instance, it corresponds to the fragment of JavaSpaces in which only the *read*, *take*, and *write* operations are considered, and no features related to time are taken into account. The calculus is a small variant of a previous calculus formerly presented in [10]. This calculus is essentially an asynchronous version of CCS [35] without the choice, the relabeling, and the restriction operators.

Let *Data* be a denumerable set of data ranged over by a, b, ...; the names in the set *Data* are used to denote the possible data that can be introduced in and retrieved from the dataspace. Let *Const* be a set of program constants ranged over by K, K', ...; as we will show in the following, program constants

(1) $\langle a \rangle \xrightarrow{\overline{a}} \mathbf{0}$

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

(3) $take(a).P \xrightarrow{a} P$

(4) $read(a).P \xrightarrow{\underline{a}} P$

(5) $$\frac{P \xrightarrow{\overline{a}} P' \quad Q \xrightarrow{a} Q'}{P|Q \xrightarrow{\tau} P'|Q'}$$

(6) $$\frac{P \xrightarrow{\overline{a}} P' \quad Q \xrightarrow{\underline{a}} Q'}{P|Q \xrightarrow{\tau} P|Q'}$$

(7) $$\frac{P \xrightarrow{\alpha} P'}{P|Q \xrightarrow{\alpha} P'|Q} \quad \alpha \neq \neg a, \vec{a}, \dot{a}, \surd$$

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

Table 1
Operational semantics of L (symmetric rules of (5)–(7) omitted).

- $\xrightarrow{\tau}$ internal/silent action
- $\xrightarrow{\overline{a}}$ offer an instance of datum a to the environment
- $\xrightarrow{a}$ consume an instance of datum a from the environment
- $\xrightarrow{\underline{a}}$ test the presence in the environment of datum a

Table 2
Labels used in the operational semantics of L.

are used for program definitions.

Let *Conf* ranged over by P, Q, ... be the set of the possible configurations defined by the following grammar:

$$P ::= \langle a \rangle \mid C \mid P|P$$

$$C ::= \mathbf{0} \mid \mu.C \mid C|C \mid K$$

where:

$$\mu ::= write(a) \mid read(a) \mid take(a)$$

Programs are represented by terms C containing the coordination primitives; the dataspace is modeled by representing each of its data a with a term $\langle a \rangle$.

The symbol C is used in the following to range over programs; it is worth stating here that sometimes we will also use P to denote programs; however, this is not an incorrect use of the notation because each program C is also a valid configuration, and P is used to range over configurations.

A configuration is composed of some programs and some available data 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 $\mu.P$, the parallel composition of subprograms, or a program constant K.

A prefix μ can be one of the primitives $write(a)$, which introduces a new datum $\langle a \rangle$ inside the data repository, $read(a)$, which tests for the presence of an instance of datum $\langle a \rangle$, and $take(a)$, which consumes an instance of datum $\langle a \rangle$. The last two primitives are blocking, in the sense that a program performing one of them cannot proceed until the operation is successfully completed. For example, a process $read(a).P$ can perform no move, because no instances of datum a are present. On the other hand, the process $write(a).P$ can always perform a τ move and produce a new datum $\langle a \rangle$, regardless of the contents of the dataspace.

Constants are used to permit the definition of programs with infinite behaviours. We assume that each constant K is equipped with exactly one definition $K = C$; as usual we assume also that only guarded recursion is used [35].

The semantics of the language is described via a labelled transition system $(Conf, Label, \longrightarrow)$ where $Label = \{\tau\} \cup \{a, \underline{a}, \overline{a} \mid a \in Data\}$ (ranged over by α, β, ...) is the set of the possible labels. The meaning of the labels is explained in Table 2. As usual, the labelled transition relation $\longrightarrow$ is the least one satisfying the axioms and rules in Table 1. For the sake of simplicity we have omitted the symmetric rules of $(5) - (7)$.

Axiom (1) indicates that $\langle a \rangle$ is able to give its contents to the environment by performing an action labelled with $\overline{a}$. Axiom (2) describes the output operation: in one step a new datum 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 $\underline{a}$.

Rule (5) is the usual synchronization rule; while rule (6) deals with the new label $\underline{a}$ representing a non-consuming operation: in this case the term performing the output operation (labelled with $\overline{a}$) is left unchanged as the *read* operation does not consume the datum tested for presence. Rule (7) is the usual local rule.[1] The last rule (8) allows a program constant K defined by $K = C$ to perform the same actions of its definition C.

We define a *reduction step*, denoted with $P \longrightarrow P'$, as a computation step corresponding to a transition that can be performed by a stand-alone program (i.e. a program without an actual context); for the kernel calculus a reduction step corresponds to a transition labelled with τ, i.e., $P \longrightarrow P'$ if $P \xrightarrow{\tau} P'$; by $\longrightarrow^*$ we mean the reflexive and transitive closure of $\longrightarrow$. A configuration

[1] Note that, for reuse of the rule, we have added to rule (7) a side condition which requires that label α is different from some kinds of label that will be introduced in the following sections.

P is *terminated*, denoted with $P \not\longrightarrow$, if it has no outgoing reductions; a configuration has a *terminating computation*, denoted with $P \downarrow$, if $P \longrightarrow^* P' \not\longrightarrow$. A configuration P has an *infinite computation*, denoted with $P \uparrow$ when it has an infinite sequence of reduction steps, formally, if for each natural number i there exists P_i such that $P_0 = P$ and $P_i \longrightarrow P_{i+1}$ for any index $i \geq 0$.

It is worth noting that, due to nondeterminism, a program P may have several computations, possibly both finite and infinite ones. Formally, it could be the case that both $P \downarrow$ and $P \uparrow$ hold.

2.1 Unordered Semantics

The semantics we have defined assumes that the programs are synchronous with the dataspace, in the sense that the emitted datum is available inside the data repository when the *write* operation terminates (see rule (2)). Other possible implementations have been considered in the literature (see, e.g., [11] for a discussion). Among them, one of the most interesting alternative interpretations assume that the processes are asynchronous with the dataspace, hence an emitted datum becomes available only after an unpredictable delay.

In [11] we have called *ordered* the former semantics (because the order of emission is respected in the introduction of the data in the repository) and *unordered* the latter (because the order is not respected).

The unordered semantics can be modeled simply by extending the language with a new prefix representing the asynchronous output operation $write_u$ having the following semantics:

$$(2'_u)\ \ write_u(a).P \xrightarrow{\tau} \langle\langle a \rangle\rangle | P \qquad (2''_u)\ \ \langle\langle a \rangle\rangle \xrightarrow{\tau} \langle a \rangle$$

where $\langle\langle a \rangle\rangle$ is an auxiliary term to be added to *Conf* in order to model data which have been already sent but which have not yet reached the shared repository.

The ordered and the unordered alternative semantics for the *write* operations have been analyzed in [12] in the presence of operations able to test the absence of data in the shared repository; in that paper we prove the existence of an expressiveness gap between the two semantics which will be discussed in the following section. On the other hand, it is interesting to observe that the expressiveness gap does not hold for the kernel calculus.

Namely, in Appendix A.1 the proof that for the kernel calculus `L` the ordered and unordered approaches are equivalent under the weak bisimulation equivalence [35] is reported, a typical equivalence relation which does not take into account unobservable steps labelled with τ.

The kernel calculus is not Turing powerful. This result is a consequence of a result we have proved in [12], where we have shown that under the unordered semantics for the *write* operation the existence of a terminating computation is decidable for a calculus which is essentially an extension of the kernel calculus. Hence, termination is decidable for the kernel calculus provided that the semantics of the *write* operation is the unordered one.

If we want to extend this result to the ordered semantics, the weak bisimilarity between the ordered and unordered semantics proved in Appendix A.1 is not enough because weak bisimulation does not preserve the termination of processes. [2] For this reason, in order to state that the ordered and the unordered semantics are equivalent in L also from the point of view of the termination of processes, we need a further result which is proved in Appendix A.2. Given this further result, we can conclude that the existence of a terminating computation is decidable for processes of the kernel calculus L also when the semantics for the *write* operation is that given by the ordered interpretation.

Note that this result – namely, the decidability of the existence of a terminating computation for the kernel calculus – is not in contrast with the result proved in [10]. In that paper we provided an encoding of Random Access Machines (RAMs) – a Turing powerful formalism – in a variant of the kernel calculus *extended* with the restriction operator. Indeed, in the RAMs encoding in [10] the presence of restriction (that is absent in the kernel calculus of this paper) turns out to be a key ingredient to obtain Turing equivalence.

3 The Calculus with Test for Absence: L[∃]

In this section we extend the kernel calculus with two further coordination primitives *read*∃ and *take*∃ which are non-blocking variants of the *read* and *take* operations, respectively. These operations behave like the corresponding *read* and *take* only if the required datum is present; otherwise, they terminate by indicating the impossibility to find the required data.

These two coordination primitives are inspired by the predicative versions of the input and read operations available in Linda: differently from the standard input and output operations, the predicative version are nonblocking

[2] As a counter-example, consider the two weakly bisimilar CCS processes, $rec\ X.\tau.X$, which performs an infinite sequence of unobservable τ actions, and the terminated process $\mathbf{0}$.

$$(1)\quad \langle a\rangle \xrightarrow{\overline{a}} \mathbf{0} \qquad (2)\quad write(a).P \xrightarrow{\tau} \langle a\rangle|P$$

$$(3)\quad take(a).P \xrightarrow{a} P \qquad (4)\quad read(a).P \xrightarrow{\underline{a}} P$$

$$(5)\quad \frac{P \xrightarrow{\overline{a}} P' \quad Q \xrightarrow{a} Q'}{P|Q \xrightarrow{\tau} P'|Q'} \qquad (6)\quad \frac{P \xrightarrow{\overline{a}} P' \quad Q \xrightarrow{\underline{a}} Q'}{P|Q \xrightarrow{\tau} P|Q'}$$

$$(7)\quad \frac{P \xrightarrow{\alpha} P'}{P|Q \xrightarrow{\alpha} P'|Q} \quad \alpha \neq \neg a, \vec{a}, \dot{a}, \surd \qquad (8)\quad \frac{P \xrightarrow{\alpha} P' \qquad K = P}{K \xrightarrow{\alpha} P'}$$

$$(9)\quad take\exists(a)?P\_\,Q \xrightarrow{a} P \qquad (10)\quad take\exists(a)?P\_\,Q \xrightarrow{\neg a} Q$$

$$(11)\quad read\exists(a)?P\_\,Q \xrightarrow{\underline{a}} P \qquad (12)\quad read\exists(a)?P\_\,Q \xrightarrow{\neg a} Q$$

$$(13)\quad \frac{P \xrightarrow{\neg a} P' \quad Q \xrightarrow{\overline{a}}\!\!\!\!\!/}{P|Q \xrightarrow{\neg a} P'|Q}$$

Table 3
Operational semantics of `L[∃]` (symmetric rules of (5)-(7) and (13) omitted).

and, in case there is no matching tuple available, returns the boolean value false. Moreover, the two primitives we add here are simplified versions of the *readIfExists* and *takeIfExists* operations of JavaSpaces where timeout is not considered. Timeouts can be associated to the *readIfExists* and *takeIfExists* operations of JavaSpaces in order to define a maximum amount of time for which the operation will wait for concurrent transactions to settle. Transactions are discussed in Section 7.

Formally, the two new operations are introduced as guards for a new type of programs with two possible continuations. The new syntax is defined as follows:

$$P ::= \langle a\rangle \;\mid\; C \;\mid\; P|P$$

$$C ::= \mathbf{0} \;\mid\; \mu.C \;\mid\; \eta?C\_\,C \;\mid\; C|C \;\mid\; K$$

where:

$$\mu ::= write(a) \;\mid\; read(a) \;\mid\; take(a)$$

$$\eta ::= read\exists(a) \;\mid\; take\exists(a)$$

The new set of terms is denoted by $Conf[\exists]$. The behaviour of the new kind of program $\eta?P\_\,Q$ is defined as follows: the first continuation P is chosen if the requested datum is available in the data repository; in this case the non-blocking operation behaves exactly as the corresponding blocking one. On the other hand, if no instance of the requested datum is currently available, the second continuation Q is chosen and the dataspace is left unchanged.

$\xrightarrow{\neg a}$	suppose that the environment contains no instances of datum a

Table 4
New labels used in the operational semantics of L[∃].

The operational semantics is defined by extending the set of labels as follows $Label[\exists] = Label \cup \{\neg a \mid a \in Data\}$. The meaning of the new label is explained in Table 4. The operational semantics of L[∃] is defined by the labelled transition system $(Conf[\exists], Label[\exists], \longrightarrow)$ where the labelled transition relation $\longrightarrow$ is the least one satisfying the axioms and rules in Table 3. For the sake of simplicity we have omitted the symmetric rule of (13). Observe that the side condition of the locality rule (7) now comes into play due to the presence of the label $\neg a$.

Axioms and rules (1)–(8) are exactly the same as those reported in the Table 1. The new axioms (9) and (10) describe the semantics of $take\exists(a)?P\_Q$: if the required $\langle a \rangle$ is present, it can be consumed (axiom (9)); otherwise, in the case $\langle a \rangle$ is not available, its absence is asserted by performing an action labelled with $\neg a$ (axiom (10)). Axioms (11) and (12) are the corresponding ones for the $read\exists(a)$ operator; the only difference is that $\underline{a}$ is used instead of a.

As observed above, the locality rule (7) is not valid for the label $\neg a$; indeed, an action of this kind can be performed only if no datum $\langle a \rangle$ is available in the dataspace, i.e., no actions labelled with $\overline{a}$ can be performed by the terms in the environment. This is exactly what is stated by the rule (13). [3]

The definition of the reduction relation should be revisited for the calculus L[∃] in order to take into account also the new label $\neg a$. This new label has been introduced in order to provide the test for absence operators with a structured operational semantics. However, a stand-alone process may perform transitions labelled with $\neg a$ as well as transitions labelled with τ. Indeed, if $P \xrightarrow{\neg a} P'$ this means that no $\langle a \rangle$ is available in P, thus the test for absence operation represented by the transition may be executed. Formally, $P \longrightarrow P'$ if $P \xrightarrow{\tau} P'$ or $P \xrightarrow{\neg a} P'$.

[3] Note that rule (13) uses a negative premise. However, the operational semantics is well defined, because the transition system specification is *strictly stratifiable* [27], a condition that ensures (as proved in [27]) the existence of a unique transition system agreeing with it. Also the calculi that will be presented in the following sections contain rules with negative premises: they are well defined because the transition system specifications are stratifiable as well.

3.1 Unordered Semantics

In Section 2.1 we have presented the alternative unordered semantics for the *write* operation, and we have proved that the ordered and unordered semantics are weakly bisimilar in the kernel calculus. It is interesting to observe that the weak bisimulation (which is a congruence for the kernel calculus, see the discussion in Appendix A.1) is no longer a congruence for the new calculus with test for absence `L[∃]`. As a counter-example consider the two following weakly bisimilar programs:

$$write(a).write(b) \qquad write_u(a).write_u(b)$$

If the two above programs are put in parallel with the program:

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

we obtain the two configurations:

$$write(a).write(b) \mid read(b).read\exists(a)?\mathbf{0}\_\, write(c)$$

$$write_u(a).write_u(b) \mid read(b).read\exists(a)?\mathbf{0}\_\, write(c)$$

These two configurations cannot be considered equivalent because the first cannot produce $\langle c \rangle$, while the second can: it is enough that $\langle b \rangle$ becomes available before $\langle a \rangle$.

This example shows that in the new calculus `L[∃]` no reasonable observational congruence could equate the ordered and the unordered semantics. More precisely, no observational congruence that permits to observe the fact that a datum will be produced in the dataspace – after performing some internal move – could equate the two semantics.

3.2 Expressive Power

From the point of view of the expressive power, it is interesting to observe that the new calculus `L[∃]` is Turing powerful. This result is proved in [12] by showing how to simulate Random Access Machines (a Turing powerful formalism) in a calculus corresponding to `L[∃]`. This result has the following consequence: there exists no computable encoding of `L[∃]` into `L` (i.e., of the test for absence operation in terms of the *read*, *take*, and *write* operations) that preserves at least the terminating behaviour of programs.

Another interesting result proved in [12] is the existence of a further discrimination between the ordered and unordered interpretations for the *write* operation besides that discussed in Section 3.1. While `L[∃]` is Turing powerful

according to the ordered semantics, it is not so according to the unordered interpretation. Namely, in [12] we prove that the existence of a terminating computation is not decidable under the ordered semantics, while this is the case under the unordered one.

4 The Calculus with Event Notification: `L[ntf]`

In this section we extend the kernel calculus with an event notification mechanism inspired by the *notify* primitive of JavaSpaces [38]. Event notification is achieved in JavaSpaces declaring a pair composed by a *template* and one *listener*. Once declared one of these pairs, each introduction in the space of new instances of tuples matching with the template are notified to the listener that reacts by activating a new process. We extend our calculus with the new term $on(a, C)$ that models a pair composed by a template represented by a and a listener represented by C.

The syntax of the kernel language is simply extended by permitting the new prefix $notify(a, C)$ plus the new term $on(a, C)$ which will be discussed in the following:

$$P ::= \langle a \rangle \mid on(a, C) \mid C \mid P|P$$

$$C ::= \mathbf{0} \mid \mu.C \mid C|C \mid K$$

where:

$$\mu ::= write(a) \mid read(a) \mid take(a) \mid notify(a, C)$$

The new set of configurations is denoted with $Conf[ntf]$. The new prefix operation $notify(a, C)$ is used by programs interested in the future incoming arrivals of the data of kind a; every time a new instance is produced, the reaction C should be activated. To model this behaviour we introduce the new term $on(a, C)$. This term represents a listener who spawns an instance of program C every time a new datum $\langle a \rangle$ is introduced into the dataspace. The listeners $on(a, C)$ cannot occur in initial configurations, as it is reasonable to assume that they can be produced only as effect of the execution of a $notify(a, C)$ operation. Note the difference between a listener $on(a, C)$ and a process $notify(a, C).P$: while the $notify(a, C)$ action of the latter process produces a new instance of a listener $on(a, C)$ – and does not react to any event – the listener $on(a, C)$ does not perform any action, but only reacts to event of production of a new instance of datum a.

The labels for the new calculus are given from the set $Label[ntf] = Label \cup \{\dot{a}, \vec{a} \mid a \in Data\}$. The meaning of the new label is explained in Table 6.

(1)	$\langle a \rangle \xrightarrow{\overline{a}} \mathbf{0}$	(2′)	$write(a).P \xrightarrow{\vec{a}} \langle a \rangle \mid P$
(3)	$take(a).P \xrightarrow{a} P$	(4)	$read(a).P \xrightarrow{a} P$
(5)	$\dfrac{P \xrightarrow{\overline{a}} P' \quad Q \xrightarrow{a} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}$	(6)	$\dfrac{P \xrightarrow{\overline{a}} P' \quad Q \xrightarrow{a} Q'}{P \mid Q \xrightarrow{\tau} P \mid Q'}$
(7)	$\dfrac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q} \quad \alpha \neq \neg a, \vec{a}, \dot{a}, \surd$	(8)	$\dfrac{P \xrightarrow{\alpha} P' \qquad K = P}{K \xrightarrow{\alpha} P'}$
(14)	$notify(a, Q).P \xrightarrow{\tau} on(a, Q) \mid P$	(15)	$on(a, P) \xrightarrow{\dot{a}} P \mid on(a, P)$
(16)	$\dfrac{P \xrightarrow{\dot{a}} P' \quad Q \xrightarrow{\dot{a}} Q'}{P \mid Q \xrightarrow{\dot{a}} P' \mid Q'}$	(17)	$\dfrac{P \xrightarrow{\dot{a}} P' \quad Q \not\xrightarrow{\dot{a}}}{P \mid Q \xrightarrow{\dot{a}} P' \mid Q}$
(18)	$\dfrac{P \xrightarrow{\vec{a}} P' \quad Q \xrightarrow{\dot{a}} Q'}{P \mid Q \xrightarrow{\vec{a}} P' \mid Q'}$	(19)	$\dfrac{P \xrightarrow{\vec{a}} P' \quad Q \not\xrightarrow{\dot{a}}}{P \mid Q \xrightarrow{\vec{a}} P' \mid Q}$

Table 5
Operational semantics of `L[ntf]` (symmetric rules of (5)–(7) and (17)–(19) omitted).

$\xrightarrow{\vec{a}}$	occurrence of the event "creation of a new datum a"
$\xrightarrow{\dot{a}}$	a listener on the event "creation of a new datum a" is woken

Table 6
New labels used in the operational semantics of `L[ntf]`.

The operational semantics is provided, similarly to the previous calculi, by the labelled transition system $(Conf[ntf], Label[ntf], \longrightarrow)$ where $\longrightarrow$ is defined by the axioms and rules in Table 5 (we omit the symmetric rules of (5)–(7) and (17)–(19)). Observe that a new rule (2′) is substituted for the rule (2). Observe also that the side condition of the locality rule (7) comes into play for the new calculus `L[ntf]` due to the presence of the labels $\vec{a}$ and $\dot{a}$.

The new labels $\vec{a}$ and $\dot{a}$ represent the occurrence and the observation respectively of the event "creation of a new datum a". This event occurs when a new datum is introduced into the shared dataspace. For this reason we change the label associated to the execution of the $write(a)$ operation from τ to $\vec{a}$ (see the new rule (2′)). The other axioms and rules from (1) to (8) are exactly the same as those for the kernel calculus `L`.

The new 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 program 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 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)).

The introduction of the new labels $\vec{a}$ and $\dot{a}$ requires the redefinition of the reduction relation. A step labelled with $\vec{a}$ corresponds to a *write* operation which can be performed by stand-alone programs independently of the environment. On the other hand, a step labelled with $\dot{a}$ requires that a particular event occurs in the environment. Thus, it is reasonable to use the following definition: $P \longrightarrow P'$ if $P \xrightarrow{\tau} P'$ or $P \xrightarrow{\vec{a}} P'$.

4.1 Example of a Client/Server System

As an example of use of the *notify* primitive, we consider a simple client/server system where clients ask for services producing data $\langle request \rangle$, and a server reacts to service requests by consuming these data and by producing a reply $\langle service \rangle$. The complete system is defined as follows:

$$
\begin{aligned}
Clients\&Server &= notify(request, Server).(Client_1 \mid \ldots \mid Client_n) \\
Client_i &= write(request).take(service) \\
Server &= take(request).write(service)
\end{aligned}
$$

As its first operation, the server is created by executing a $notify(request, Server)$ operation. More precisely, this operation generates the term $on(request, Server)$. At the same time, several clients, $Client_1 \ldots Client_n$, are spawned. Each one of these clients simply produces a request and then waits for a service. Each time a $\langle request \rangle$ is produced, the listener $on(request, Server)$ spawns an instance of program *Server*. This program consumes the datum $\langle request \rangle$ and produces a corresponding datum $\langle service \rangle$.

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 multi-threaded server.

4.2 Best-effort Semantics

The synchronisation 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 listeners 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 (7). In this way, we model a reliable notification service. On the other hand, an event notification mechanism may be a best-effort service, that is, it is not ensured that all the registered listeners are notified of the occurrence of an event. This happens, e.g., in JavaSpaces [38] where no guarantee is provided by the event notification mechanism.

We can model a best-effort event notification service simply by removing the negative premise of the rules (17) and (19). This modification permits, e.g., to the term $write(a)|on(a,P)|on(a,Q)$ to perform a synchronization between simply the first and the second program (without involving the third one), thus reaching the configuration $\langle a\rangle|on(a,P)|P|on(a,Q)$ where only one reaction (the program P) is activated.

4.3 Expressive Power

Regarding the expressiveness of the event notification mechanism introduced by the *notify* operation, we first observe that it embeds an implicit form of recursion; then, we address the problem of implementing the event notification mechanism in the process calculi introduced in the previous sections. Moreover, we report observations concerning the alternative unordered and best-effort semantics.

4.3.1 Recursion

The *notify* primitive permits us to describe processes with an infinite behaviour without being 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 instance of 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 data $\langle a\rangle$ 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 datum *rec* is used to repeatedly activate an instance of the renaming program *Rename*. More precisely, a renaming program produces a new datum *rec* before terminating in order to activate a new instance of *Rename*. Observe that the *Trans* program must produce the first *rec* datum in order to activate the first instance of the program *Rename*.

4.3.2 Comparing `L[ntf]` *with* `L[∃]` *and* `L`

Another interesting question on the expressiveness of the *notify* primitive concerns the possibility to encode it in the process calculi introduced in the previous sections. The answer to this question has been given in [18], where we showed an interesting hierarchy of expressiveness: the *notify* primitive strictly increases the expressiveness of a calculus which corresponds to the kernel calculus `L`, but it is strictly less expressive than the test for absence operations of the calculus `L[∃]`.

More precisely, we can summarize the results in [18] as follows (we recall that $P \downarrow$ means that P has a finite computation while $P \uparrow$ means that P has an infinite computation):

(i) Event notification cannot be encoded with only *take*, *write*, and *read* operations; this is proved by showing that the property $P \downarrow$ is decidable in a calculus corresponding to the kernel calculus `L`, while such a property is not decidable in `L[ntf]`.

(ii) Event notification can be encoded by using also the test for absence mechanism. This means that it is possible to map each system in `L[ntf]` to a system in `L[∃]` that has an equivalent behaviour (see [18] for more details). This is proved by presenting a rather complex encoding which implements the *write* operation with a protocol based on three separated phases: first it is necessary to count the number of registered listeners, then it is required to communicate to each of them the occurrence of an interesting event, and finally it is necessary to wait for an acknowledgement from each of the notified listeners. According to the terminology of [16], this encoding is *open*: an encoding $[\![P]\!]$ is open if $[\![P]\!] = [\![P]\!] | \prod_{a \in n(P)} R_a$ where $n(\cdot)$ is the set of names occurring in P and $[\![\cdot]\!]$ is a homomorphism with respect to parallel composition. Even if an open encoding is not a homomorphism with respect to parallel composition, it satisfies the following property: $[\![P|Q]\!] = [\![P]\!] | [\![Q]\!] | \prod_{a \in n(Q) \setminus n(P)} R_a$.

(iii) Test for absence cannot be encoded with *take*, *write*, and event notification: this is proved by showing that the property $P \uparrow$ is not decidable in a language corresponding to `L[∃]`, while such a property is decidable in `L[ntf]`.

For the above reasons, we can conclude that the *notify* primitive strictly in-

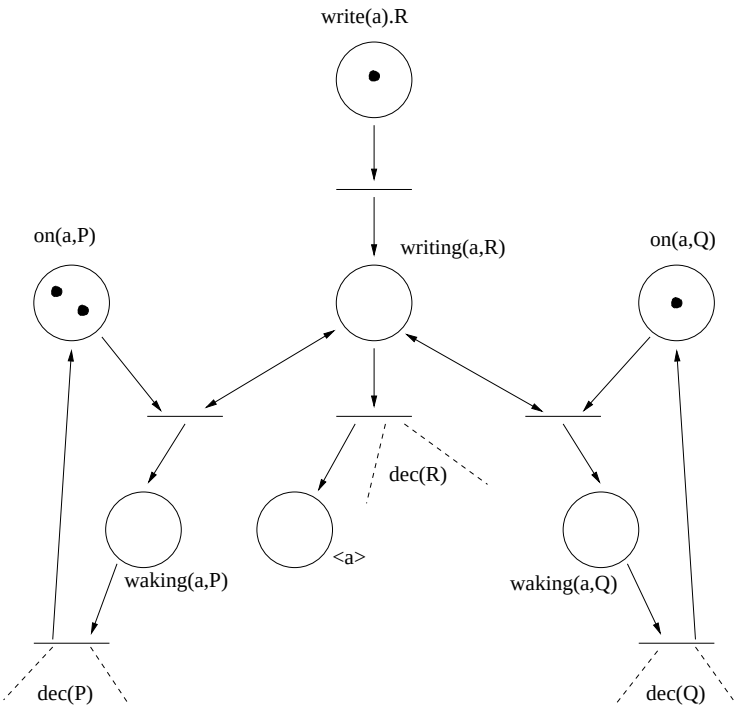

Fig. 2. The net corresponding to $on(a, P) \mid on(a, P) \mid on(a, Q) \mid write(a).R$.

creases the expressive power of a shared dataspace coordination language with only *write*, *read*, and *take* operations but it does not increase the expressive power of a language comprising also test for absence operations.

4.3.3 Unordered Semantics

In a related paper [17] we have investigated the interplay between the ordered and unordered interpretations of *write* in the presence of the *notify* primitive. The most interesting results proved there are the following: (i) under the ordered semantics, the calculus `L[∃]` is strictly more expressive than `L[ntf]`, whilst under the unordered semantics the vice versa holds (i.e., `L[ntf]` is strictly more expressive than `L[∃]`); (ii) in the presence of the *notify* primitive the ordered semantics can be encoded in terms of the unordered semantics.

4.3.4 Best-effort Semantics

Another interesting observation regarding the expressive power of the calculus `L[ntf]` concerns the alternative best-effort semantics discussed in the Section 4.2; here we discuss that under the best-effort semantics the termination property (i.e., $P \downarrow$) becomes a decidable property, thus the expressive power strictly decreases.

For the sake of space we simply sketch the idea we have followed in order to prove that under the best-effort semantics the termination property (i.e., $P \downarrow$) is decidable. Following an approach similar to the one used in [18], it is possible to define a function mapping terms of `L[ntf]` to Place/Transition Petri nets [44] in such a way that the termination problem for `L[ntf]` under the best-effort semantics is reduced to the deadlock problem for Place/Transition Petri nets [44]. As the existence of a deadlock is decidable for such a class of nets [9,20], we obtain the decidability of termination for `L[ntf]` with best-effort semantics. More precisely, given a process P, it is possible to construct a Petri net $Net(P)$, and to map each process Q reachable from P (i.e., such that $P \longrightarrow^* Q$) to a marking $dec(Q)$ of $Net(P)$, satisfying the following properties:

- each transition performed by Q can be mimicked by a nonempty sequence of transitions starting from the marking $dec(Q)$ in $Net(P)$;
- if a transition fires at marking $dec(Q)$ in $Net(P)$, leading to marking m', then there exist a process Q'' such that $Q \longrightarrow^+ Q''$ (i.e., Q'' is reached from Q by performing a nonempty sequence of steps) and $dec(Q'')$ is reachable from m' by performing a (possibly empty) sequence of transitions.

From the above properties it is easy to deduce that P terminates if and only if $Net(P)$ has a deadlock.

In Figure 2 we present a portion of net corresponding to the following process:

$$on(a, P) \mid on(a, P) \mid on(a, Q) \mid write(a).R$$

The picture shows the main difference between the Place/Transition Petri nets semantics in [18] – where all listeners are notified of the occurrence of the events of interest – and the nets used here for `L[ntf]` – where there is no guarantee of notification. The basic idea is to decompose a process in its basic components: data, listeners and prefixed terms. Each component of a process is represented as a token in the marking of the net. When the subprocess $write(a).R$ performs the write operation, a token in place $writing(a, R)$ – corresponding to an intermediate phase of the write operation – is produced. During this intermediate phase, a number of listeners (but not necessarily all of them) can be woken, by moving a token, e.g., from place $on(a, P)$ to the corresponding place $waking(a, P)$. After some (possibly no or all) listener has been woken up, the write operation is concluded: the token from place $writing(a, P)$ is removed, the new datum a is produced (by putting a token in place $\langle a \rangle$) and the tokens corresponding to process R (denoted by $dec(R)$) are produced. The presence of a token in place, e.g., $waiting(a, P)$ enables a transition that produces the tokens corresponding to process P, as well as a token in place $on(a, P)$, thus reactivating the corresponding listener.

5 The Calculus with Timeout: L[Δ]

In this section we model and analyse timeouts as a mechanism to obtain non-blocking versions of the *read* and *take* operations which do not require the ability to perform a test for absence of the required data in the dataspace. The idea is to associate with each execution of a *read* or *take* operation the indication of a time interval (which is used as a timeout). When the timeout expires, the operation fails and terminates. In the modeling of the timed out versions of the primitives, we have been inspired by the *read* and *take* operations of JavaSpaces. Indeed, they permit to include as an additional parameter a timeout that indicates the maximum amount of time to wait for matching entries to be read or consumed.

Syntactically, the new operations $read(a, t)$ and $take(a, t)$ are introduced where t denotes the time interval indicating the timeout. We will use these operations as guards of if-then-else terms (e.g., $read(a, t)?P\_Q$) where the first continuation is chosen when the operation succeeds, otherwise the second continuation is activated.

It is worth noting that a *read* or a *take* operation with timeout may fail due to two possible reasons: (i) no instance of the required datum is present in the data repository when the timeout expires, (ii) no instance of the required datum is found, even if present, in the data repository before the timeout expires. The condition (ii) may occur, e.g., in distributed implementation of the data repository where the test for presence of a datum may be delayed by the temporary disconnection of some servers.

As an example, consider the configuration

$$\langle a \rangle \mid read(a, 2)?\mathbf{0}\_write(b)$$

in which a datum $\langle a \rangle$ is available and a program requires to read it under a timeout interval equal to 2. It could happen that the datum cannot be accessed by the program before the timeout expiration; in this case the operation fails and the datum $\langle b \rangle$ is produced.

It is also interesting to note that when an operation with timeout fails, the program which performed the operation does not know for which of the two possible reasons, thus it cannot conclude anything about the presence or absence of the required datum.

The use of timeouts seems particularly useful when the dataspace service is not reliable. Consider, as an example, a dataspace server which crashes during the execution of a blocking *read* or *take* operation performed by a client. Usually,

this client is blocked until the server recovers. On the other hand, if timeouts are associated to the *read* or *take* operation the client may be resumed at the end of the timeout, independently of the state of the crashed dataspace server.

The way we represent time is inspired by the modeling of time adopted in standard implementations of shared dataspaces (e.g., in JavaSpaces): the current time is represented by an integer which is incremented periodically (e.g., each millisecond). A timeout is expressed in terms of a number of time periods (e.g., a number of milliseconds). If an operation with an associated timeout t starts its execution when the current time is c, then its timeout will expire at the end of the time period with current time $c + t$.

In our process calculus we obtain an equivalent representation of the passing of time without exploiting an explicit value for the current time: we simply use, for each of the currently active timeouts, a value representing the number of basic time periods which remain before the elapsing of the timeout. At the end of each basic time period, all these numbers are decremented. When the number corresponding to a timeout becomes 0, this means that the timeout will elapse at the end of the current basic time period.

Speaking formally, we use transitions labeled with $\surd$ in order to model the modification induced on a configuration by the elapsing of a basic time interval: $P \xrightarrow{\surd} P'$ states that the configuration P becomes P' due to the fact that the a basic time period has elapsed.

As an example, consider the following possible computation of the configuration discussed above.

$$\langle a \rangle \mid read(a, 2)?\mathbf{0}\_\, write(b) \xrightarrow{\tau} \langle a \rangle \mid read(a)_2?\mathbf{0}\_\, write(b) \xrightarrow{\surd}$$
$$\langle a \rangle \mid read(a)_1?\mathbf{0}\_\, write(b) \xrightarrow{\surd} \langle a \rangle \mid read(a)_0?\mathbf{0}\_\, write(b) \xrightarrow{\surd}$$
$$\langle a \rangle \mid write(b) \xrightarrow{\tau} \langle a \rangle \mid \langle b \rangle$$

The first transition corresponds to the beginning of the execution of the *read* operation. We use the notation $read(a)_2$ to indicate that a $read(a)$ operation is currently under execution with a remaining timeout period 2. After three transitions labelled with $\surd$ the *read* operation fails; thus the second continuation is activated, and finally the datum $\langle b \rangle$ is produced.

It is worth noting that we have to deal with two kinds of terms inside configurations: those sensitive to the passing of time and those which are not. For example, the datum $\langle a \rangle$ in the example above is not influenced by the transitions labeled with $\surd$.

A term sensitive to the passing of time can be discriminated by one term

which is not, simply by observing if it has outgoing transitions labelled with $\surd$.

As discussed above, we use the terms $read(a)_t?P\_Q$ (resp. $take(a)_t?P\_Q$) to represent an operation $read(a)$ (resp. $take(a)$) under execution with an associated remaining timeout t. These kind of terms cannot appear as an initial program of a configuration: they can be introduced only as the result of the beginning of the execution of a *read* (resp. *take*) operation. The term $read(a,t)?P\_Q$ is not sensitive to the passing of time until the execution of the *take* operation starts its execution: this is represented by the transition $take(a,t)?P\_Q \xrightarrow{\tau} take(a)_t?P\_Q$. The index t will be subsequently decreased each time a transition labelled with $\surd$ is performed.

We introduce the notation $\eta_t?P\_Q$ to denote either the term $take(a)_t?P\_Q$ or the term $read(a)_t?P\_Q$. The duration t is taken from the set of time intervals $Time$, ranged over by t, t', ..., which contains all the natural numbers plus a special symbol ∞ used to represent an infinite time interval. An infinite time interval never expires; we model this simply by assuming $\infty - 1 = \infty$.

To be as general as possible, we do not make any assumption on the number of operations that may be performed during a basic time period, i.e., we do not fix a maximum number of transitions with a label different from $\surd$ which may be executed between two subsequent $\surd$ transitions. Nevertheless, it is not difficult to adapt our approach to a context in which a maximum number n of operations per period is fixed: it is enough to consider all the computations we model, and then remove all those in which there are more than n transitions with a label different from $\surd$ between two subsequent $\surd$ transitions.

We are now ready to introduce the syntax and the operational semantics of the new calculus. The $read(a)$ and $take(a)$ prefixes are removed and the new operations with timeout are introduced as guards for if-then-else forms:

$$P ::= \langle a \rangle \mid C \mid P|P$$

$$C ::= \mathbf{0} \mid \mu.C \mid \eta?C\_C \mid C|C \mid K$$

where:

$$\mu ::= write(a)$$

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

The new set of configurations is denoted by $Conf[\Delta]$. The labels are extended by introducing also $\surd$, i.e., $Label[\Delta] = Label \cup \{\surd\}$. The meaning of the new label is explained in Table 8.

The semantics is defined by the transition system $(Conf[\Delta], Label[\Delta], \longrightarrow)$ where $\longrightarrow$ is defined as the least labelled transition relation satisfying the

(1)	$\langle a \rangle \xrightarrow{\overline{a}} \mathbf{0}$	(2)	$write(a).P \xrightarrow{\tau} \langle a \rangle \vert P$
(3′)	$take(a,t)?P\_Q \xrightarrow{\tau} take(a)_t?P\_Q$	(4′)	$read(a,t)?P\_Q \xrightarrow{\tau} read(a)_t?P\_Q$
(5)	$\dfrac{P \xrightarrow{\overline{a}} P' \quad Q \xrightarrow{a} Q'}{P \vert Q \xrightarrow{\tau} P' \vert Q'}$	(6)	$\dfrac{P \xrightarrow{\overline{a}} P' \quad Q \xrightarrow{\underline{a}} Q'}{P \vert Q \xrightarrow{\tau} P \vert Q'}$
(7)	$\dfrac{P \xrightarrow{\alpha} P'}{P \vert Q \xrightarrow{\alpha} P' \vert Q}\ \alpha \neq \neg a, \vec{a}, \dot{a}, \surd$	(8)	$\dfrac{P \xrightarrow{\alpha} P' \qquad K = P}{K \xrightarrow{\alpha} P'}$
(20)	$take(a)_t?P\_Q \xrightarrow{a} P$	(21)	$read(a)_t?P\_Q \xrightarrow{\underline{a}} P$
(22)	$\eta_t?P\_Q \xrightarrow{\surd} \eta_{t-1}?P\_Q$ 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 7
Operational semantics for timeouts (symmetric rules of (5)–(7) and (25) omitted).

$\xrightarrow{\surd}$	elapsing of a basic time interval

Table 8
New labels used in the operational semantics of `L[Δ]`.

axioms and rules in Table 7 (we omit the symmetric rules of (5)–(7) and (25)) where the axioms (3′) and (4′) are substituted for (3) and (4), respectively. The other axioms from (1) to (8) are exactly the same as those for the kernel calculus `L`.

Axioms (3′) and (4′) model the beginning 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 label in the rule (23) is $\surd$ because with $\eta_0?P\_Q$ we denote an input during its last time period of executability. At the end of this period (i.e. on the execution of the global synchronization on the $\surd$ action), the input actually fails and the alternative process Q is activated. 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 processes have an outgoing transition labelled with $\surd$, they synchronize on the execution of this operation. On the other hand, one of the two processes can perform its own transition $\surd$ without affecting the other term which is not sensitive to the passing of time (as discussed above, this is reflected by the fact that it has no outgoing transitions labelled with $\surd$).

The introduction of the new label $\surd$ requires the redefinition of the reduction relation indicating the computation step that may be performed by stand-alone configurations: $P \longrightarrow P'$ if $P \xrightarrow{\tau} P'$ or $P \xrightarrow{\surd} P'$. Indeed, terms performing transitions labelled with $\surd$, simply change according to the passing of time independently of their context.

5.1 Asynchronous Semantics

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 negative premise of rule (25) and the side condition $\alpha \neq \surd$ of the locality rule (7), according to which a process cannot perform locally its transitions labelled with $\surd$. If we remove the side condition $\alpha \neq \surd$ from rule (7), it may happen that given two programs P and Q sensitive to the passing of time (i.e., there exist P' and Q' such that $P \xrightarrow{\surd} P'$ and $Q \xrightarrow{\surd} Q'$), both $P|Q \xrightarrow{\surd} P'|Q$ and $P|Q \xrightarrow{\surd} P|Q'$ becomes valid transitions. In these transitions, the time passes only in one of the two programs, thus it passes *asynchronously*.

To distinguish between the two interpretations of time passing, we use $P \longrightarrow_s^* P'$ to denote that a configuration P' can be reached from P by performing a sequence of reduction steps 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 interpretation.

The synchronous approach models systems with a global clock (e.g., centralized systems), while the asynchronous approach reflects the behaviour of systems where a global clock does not exist (e.g., distributed systems). In many applications, the absence of the global clock becomes critical; consider, e.g., typical problems such as distributed consensus. It is interesting to observe that, on the other hand, in our context the synchronous and the asynchronous approaches can be considered equivalent, at least from the point of view of the configurations that can be reached during a computation. More precisely, the equivalence result is a consequence of a theorem reported in Appendix B, where we show that, given an initial configuration P (i.e., a configuration in which no timeout is counted yet), the configurations that can be reached from P under the synchronous time are exactly the same as those reachable under the asynchronous time. Formally, for each $P \longrightarrow_s^* P'$ then also $P \longrightarrow_a^* P'$ and vice versa.

It is worth to anticipate here that this equivalence result does not hold any longer in the calculi with leasing that will be introduced in the next section.

6 The Calculi with Leasing: `L[lsn]` and `L[lsn,ntf]`

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.

Following the JavaSpaces approach, we consider the leasing mechanism applied to two kinds of services: the storing of data in the shared repository and the managing of the listeners interested in the notification of the occurrence of events inside the dataspace. These two leased services are activated by means of the coordination primitives *write* and *notify*, respectively. For this reason, we add to these two 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*).

The two forms of leased resources are modeled separately: we first define the calculus `L[lsn]` as extension of the kernel calculus with leased data, then we extend `L[lsn]` in order to deal also with leased listeners, and we call the obtained calculus `L[lsn,ntf]`.

We also permit cancelling and renewing of 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 and renewed, respectively. These unique identifiers are taken from the set *Lease*, ranged over by l, l', ...

Leasing and timeouts are both features related to time; however, they are exploited for different purposes. Timeouts are particularly useful on the client-side: they can be used to avoid the blocking of clients waiting for services which are not provided by the servers in due time. On the other hand, leasing is useful on the server-side: it allows the servers to free resources allocated to clients which do not renew the corresponding leasing before the leasing period expires.

Given this observation, it is clear that a direct implementation of the leasing mechanism in terms of timeouts (and the vice versa) is not a trivial task. For example, one may think to model a leased datum by using a persistent datum plus a specialized client program which explicitly removes that datum when a timeout corresponding to the expected leasing period expires. This modeling does not represent correctly the leasing model for two main reasons: (i) it is not ensured that the specialized client program has the ability to immediately remove the datum at timeout expiration (e.g., in the case of a temporary

disconnection between the program and the repository); (ii) it could be the case that the persistent datum is removed by another program before the end of the corresponding leasing period, and the specialized program removes another instance of the datum which is not the expected one.

Conversely, one may think to model a timeout by exploiting a datum in the dataspace, thus on the server-side, with a leasing period corresponding to the timeout. When the datum expires, this indicates that the timeout has elapsed. This approach cannot be followed because it requires the ability to write client programs which observe immediately the instant in which an available datum becomes unavailable. None of the usual coordination primitives (not even the test for absence operations) has this ability.

Due to these differences between leasing and timeout, we have decided to model both the mechanisms independently (i.e., in two separated calculi) because it seems the best way to investigate in isolation the properties of these two time dependent paradigms. One interesting property that we are able to prove following this approach is that synchronous and asynchronous time are equivalent for timeouts (as stated in Section 5.1 and proved in Appendix B), but this is not the case for leasing (see Section 6.3).

As in the previous section, *Time* denotes the set of time intervals. In order to associate with 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 data and listeners no longer with the terms $\langle a \rangle$ and $on(a, P)$, respectively, rather with the new notations $\langle a \rangle_{l:t}$ and $on(a, P)_{l:t}$.

We are now ready to present the first calculus `L[lsn]`. The syntax is obtained by removing the term $\langle a \rangle$ and by adding the new $\langle a \rangle_{l:t}$; we also remove the prefix $write(a)$ and we introduce the new $(\nu l)write(a, t)$. Moreover, we introduce $cancel(l)$ and $renew(l, t)$ as guards for if-then-else terms. Formally:

$$P ::= \langle a \rangle_{l:t} \mid C \mid P|P$$

$$C ::= \mathbf{0} \mid \mu.C \mid \eta?C\_C \mid C|C \mid K$$

where:

$$\mu ::= (\nu l)write(a, t) \mid read(a) \mid take(a)$$

$$\eta ::= cancel(l) \mid renew(l, t)$$

The obtained configurations are denoted by $Conf[lsn]$.

In order to define new leasing identifiers, we use $(\nu l)write(a, t).P$ to bind the lease identifier l in P; we ensure that the name is bound in P by replacing each free occurrence of l in P with a fresh name l' (see rule $(2'')$), i.e., a name which has not been previously used. This avoids name clashes of two different

(1′) $\langle a\rangle_{l:t} \xrightarrow{\overline{a}} \mathbf{0}$

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

(3) $take(a).P \xrightarrow{a} P$ (4) $read(a).P \xrightarrow{\underline{a}} P$

(5) $\dfrac{P \xrightarrow{\overline{\alpha}} P' \quad Q \xrightarrow{\alpha} Q'}{P|Q \xrightarrow{\tau} P'|Q'}$ $\alpha = a$ or $\alpha = \dagger l$ or $\alpha = l:t$ (6) $\dfrac{P \xrightarrow{\overline{a}} P' \quad Q \xrightarrow{\underline{a}} Q'}{P|Q \xrightarrow{\tau} P|Q'}$

(7) $\dfrac{P \xrightarrow{\alpha} P'}{P|Q \xrightarrow{\alpha} P'|Q}$ $\alpha \neq \neg a, \vec{a}, \dot{a}, \surd$ (8) $\dfrac{P \xrightarrow{\alpha} P' \qquad K = P}{K \xrightarrow{\alpha} P'}$

(26) $cancel(l)?P\_\, Q \xrightarrow{\dagger l} P$ (27) $cancel(l)?P\_\, Q \xrightarrow{\neg\dagger l} Q$

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

(30) $\langle a\rangle_{l:t} \xrightarrow{\overline{\dagger l}} \mathbf{0}$ (31) $\langle a\rangle_{l:t'} \xrightarrow{\overline{l:t}} \langle a\rangle_{l:t}$

(32) $\langle a\rangle_{l:t} \xrightarrow{\surd} \langle a\rangle_{l:t-1}$ if $t \neq 0$ (33) $\langle a\rangle_{l:0} \xrightarrow{\surd} \mathbf{0}$

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

Table 9
Operational semantics for `L[lsn]` (symmetric rule of (5)–(7) and (34) omitted).

$\xrightarrow{\dagger l}$	cancel the leased object l in the environment
$\xrightarrow{\overline{\dagger l}}$	the leased object l is cancelled
$\xrightarrow{\neg\dagger l}$	no leased object l is cancelled
$\xrightarrow{l:t}$	renew the leasing period of object l in the environment to time t
$\xrightarrow{\overline{l:t}}$	the time of the leased object l is renewed to t
$\xrightarrow{\neg l:t}$	no leased object l is renewed

Table 10
New labels used in the operational semantics of `L[lsn]`.

leased objects. To keep the operational semantics clean and simple, in rule (2″) we simply require that the name l' does not appear yet in other parts of the system. To be more rigorous, we should add a new label denoting the creation of a new leased datum by rule (2″), then we put a side condition on rule (7) to ensure that the name l' does not occur in Q. This reflects the fact that a leased datum can be accessed (via the *renew* and *cancel* operations) only in the continuation of the *write* operation which produced it.

In the following, in the term $(\nu l)write(a,t).P$ we sometimes omit (νl) when

the leasing identifier l is not subsequently used and we omit t when it is ∞.

The operational semantics of the new calculus is defined by the labelled transition system $(Conf[lsn], Label[lsn], \longrightarrow)$ where

$$Label[lsn] = Label \cup \{\dagger l\ ,\ l\!:\!t\ ,\ \overline{\dagger l}\ ,\ \overline{l\!:\!t}\ ,\ \neg\dagger l\ ,\ \neg l\!:\!t \mid l \in Lease, t \in Time\}$$

(the meaning of the new labels is summarized in Table 10, will be also described below) and $\longrightarrow$ is the least labelled transition relation satisfying the axioms and rules in the Table 9 (we omit the symmetric rule of (5)–(7) and (34)). Observe that the new axioms $(1')$ and $(2'')$ are substituted for (1) and (2), respectively.

Axiom $(1')$ simply adapts the axiom (1) to the new notation for leased data $\langle a\rangle_{l:t}$. Axiom $(2'')$ describes the creation of a new leased datum as the result of the execution of a *write* operation. As discussed above, a fresh leasing identifier l' is substituted for the name l in the continuation (i.e., in the process P).

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 visible effect of the execution of a *cancel* on the leased object l is denoted by the new label $\dagger l$, and its failure by $\neg\dagger l$ (see axioms (26) and (27)). The other new label $l : t$ is used to denote an action which renews the leasing period, granted to the leased object identified by l, to the new time period t; the failure of a renewing action is denoted with another label $\neg l : t$ (see the axioms (28) and (29)). Axioms (30) and (31) model the complementary operations of $\dagger l$ and $l : t$, respectively, which may be performed at any time by the leased object with leasing identifier 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, that is, that the environment cannot perform any transition labelled with $\dagger l$ (resp. $l : t$). This is described by rule (34).

6.1 Adding Event Notification: `L[lsn,ntf]`

We now define the calculus `L[lsn,ntf]` with leasing and event notification. The syntax can be obtained by extending the syntax of `L[lsn]` with the $(\nu l)notify(a, C, t)$ primitive and the representation of the leased listeners $on(a, C)_{l:t}$:

$$P ::= \langle a\rangle_{l:t} \mid on(a, C)_{l:t} \mid C \mid P|P$$
$$C ::= \mathbf{0} \mid \mu.C \mid \eta?C\_C \mid C|C \mid K$$

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

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

$(15')$ $on(a,P)_{l:t} \xrightarrow{\dot{a}} P|on(a,P)_{l:t}$

(16) $$\frac{P \xrightarrow{\dot{a}} P' \quad Q \xrightarrow{\dot{a}} Q'}{P|Q \xrightarrow{\dot{a}} P'|Q'}$$

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

(18) $$\frac{P \xrightarrow{\vec{a}} P' \quad Q \xrightarrow{\dot{a}} Q'}{P|Q \xrightarrow{\vec{a}} P'|Q'}$$

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

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

$(32')$ $P_{l:t} \xrightarrow{\surd} P_{l:t-1}$ if $t \neq 0$ $\qquad$ $(33')$ $P_{l:0} \xrightarrow{\surd} \mathbf{0}$

Table 11
Additional axioms and rules for `L[lsn,ntf]` (symmetric rules of (17)–(19) omitted).

where:

$$\mu ::= (\nu l)write(a,t) \mid read(a) \mid take(a) \mid (\nu l)notify(a,C,t)$$

$$\eta ::= cancel(l) \mid renew(l,t)$$

The new set of configurations is denoted by $Conf[lsn,ntf]$. Term $on(a,C)_{l:t}$ represents a leased listener with leasing identifier l and remaining time t. Leased listeners are created as a result of the execution of the new prefix $(\nu l)notify(a,C,t)$, where (νl) is a binder for the leasing identifier l. Also for the *notify* primitive with leasing, we sometimes omit (νl) when l is not subsequently used and t when equal to ∞.

In this new calculus we deal with both the two kinds of leased objects: data (denoted with $\langle a\rangle_{l:t}$) and listeners (denoted with $on(a,C)_{l:t}$). We introduce $P_{l:t}$ as a uniform notation for both the two kinds of leased objects, where P may be either of the form $\langle a\rangle$ or of the form $on(a,C)$.

The set of labels should contain those needed to model the leasing metaphor, as also those for the notification mechanism: $Label[lsn,ntf] = Label[lsn] \cup Label[ntf]$.

The operational semantics of the calculus is defined by the labelled transition system $(Conf[lsn,ntf], Label[lsn,ntf], \longrightarrow)$ where $\longrightarrow$ is the least labelled transition system satisfying the axioms and rules in the Tables 9 and 11 (where the axiom $(2''')$ is substituted for $(2'')$, and axioms $(30')$–$(33')$ are substituted for (30)–(33), respectively). Observe that axioms $(30')$–$(33')$ are simple adaptations of the corresponding (30)–(33), where the new general notation $P_{l:t}$ is used instead of $\langle a\rangle_{l:t}$.

Observe that it is not necessary to modify the axioms $(26) - (29)$ and the rule (34) as they are valid also for the new kind of leased resource $on(a, C)_{l:t}$.

6.2 Example of a Seat Reservation Service

As an example of the 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 train departure. In order to implement this idea, we may use a program *ResTrain* which produces a listener with an associated lifetime *time* corresponding to the time which remains before the train leaves. This listener is responsible to manage each single reservation.

$$\begin{aligned} ResTrain &= (\nu l) notify(req, Reserve, time).CheckTrain_l \\ Reserve &= take(req).write(seat) \end{aligned}$$

According to this modeling, a seat request is represented by a datum $\langle req \rangle$, which, when produced, activates an instance of the *Reserve* program. This program is responsible for taking the request, and producing the corresponding seat reservation (modeled by a datum $\langle seat \rangle$).

The process $CheckTrain_l$, which remains active after the production of the listener, is responsible to manage variations of the train scheduling. For example, if the train departure is cancelled, the reserving process should be removed, while if the departure is delayed, the remaining lifetime should be changed accordingly. Thus, $CheckTrain_l$ (parametric in the name of the leasing identifier) can be defined as follows:

$$\begin{aligned} CheckTrain_l = {} & take(delay).renew(l, newtime)?CheckTrain_l \_ \mathbf{0} \mid \\ & take(cancel).cancel(l)?\mathbf{0} \_ \mathbf{0} \end{aligned}$$

where we assume that the datum $\langle delay \rangle$ represents a departure delay, while $\langle cancel \rangle$ represents the departure cancel. Observe that after a departure delay, the program $CheckTrain_l$ remains active, as it could be the case that the train departure may receive a further delay (or it could be even cancelled).

6.3 Asynchronous Semantics

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 equivalence result between the two interpretations of time passing which holds in the calculus with timeouts (see Section 5.1) no longer holds in the new calculi with leasing. 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:

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

The process on the right hand side requires the presence of datum $\langle b \rangle$ in order to continue its execution. As this datum will never be produced, its behaviour consists of waiting for a $t+1$ long period, and then becoming process $take(a)$. The datum on the left hand side has a lifetime shorter than $t + 1$. According to the synchronous interpretation, $\langle a \rangle_{l:t}$ 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.

6.4 Alternative Granting Policies

In the calculi with leasing we have introduced above we assume that the leased services have an associated leasing period which corresponds exactly to the one required by the clients. This approach can be followed in an ideal situation in which the leased services can be always granted according to the requirements of the clients. This is not always possible, e.g., in situations in which resources are limited. For example, in the Jini Distributed Leasing Specifications [39], the service provider may decide to grant the resource for a shorter period. As JavaSpaces is introduced as a specific Jini service, it also adopts this alternative granting policy. We can easily adapt our semantics in order to take into account this feature by replacing rules ($2'''$) and ($14'$) with the following:

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

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

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

Other granting policies may be considered according to which leased resources are not freed exactly on leasing expiration, but subsequently when, e.g., an expired resources collector is activated. According to this alternative approach leased services may be provided even after their leasing period expire. This approach looks particularly useful to implement a garbage collector mechanism for shared dataspaces which removes data, among those with an expired leasing, only when it is necessary to free space, e.g., in order to store incoming data. If it is not necessary to free space, data may remain available in the repository even after their leasing has expired.

It is interesting to observe that several policies could be chosen in order to select the datum, among those expired, which should be removed when the garbage collector is activated. For example, in [15] we discuss two possible policies: the *ordered collection* policy, which selects the datum which expired first, and the *unordered collection* policy, according to which one among all the expired data could be chosen. In the following subsection we discuss an interesting expressiveness gap between these two possible policies.

6.5 Expressive Power

Our first consideration on the expressive power of the calculi with leasing concerns the observation that persistence is lost when we consider the Jini-like granting policy (discussed in the previous subsection) according to which leased resources could be granted with a shorter period w.r.t. the one requested by the clients.

Consider, for example the creation of a permanent datum (i.e., a datum with an infinite lifetime): which is described by the transition

$$(\nu l)\mathit{write}(a, \infty).P \xrightarrow{\tau} \langle a \rangle_{l':\infty} | P\{l'/l\}$$

If we consider the Jini-like semantics (see rule $(2'')$) the new datum may be produced with a shorter leasing time, thus it is no longer permanent.

In order to better understand the impact on the expressiveness of the introduction of non-permanent data, in [14] we have compared (a calculus very similar to) $\mathtt{L[\exists]}$ with another calculus which is a slight modification obtained simply by considering data leased following the Jini-like approach. The interesting result is that the calculus with leased data is strictly less expressive than the initial calculus with permanent data. This is proved by showing that

the property $P\uparrow$ is undecidable for the calculus with permanent data while it becomes decidable when moving to leased data. Another interesting result is that, even if the calculus becomes less expressive when moving to the leased data, it remains strictly more expressive than the kernel calculus L (which does not contain any test for absence operations). This second result is proved by showing that the property $P\downarrow$, which is a decidable property in L, is no longer decidable in the calculus with leased data.

As discussed in the previous subsection, the leasing mechanism is a useful programming notation which could be used also in order to manage the overburdening of useless and outdated information stored inside the shared repository, simply by adding a garbage collector which removes expired data when it is necessary to free space in the repository. We have already hinted two possible policies which could be adopted in order to implement such a collector: the *ordered collection* policy, according to which the collector selects for deletion the datum which expired first, and the *unordered collection* policy, according to which one among all the expired data could be chosen.

In [15] we have investigated the two collection policies, and we have proved an interesting discrimination result using a calculus corresponding to L[$\exists$] modified with the introduction of this expired data collection mechanism. We have proved on that calculus that $P\uparrow$ is a decidable property under the unordered collection policy, while this is not the case under the ordered collection policy.

7 The Calculi with Transactions: L[txn], L[txn,$\exists$], and L[txn,ntf]

An important feature of languages for the coordination of distributed processes is represented by the transactional mechanism. Transactions permit to group a set of coordination operations into a bundle that acts as a single atomic operation. A set of coordination operations, grouped in a transaction, is executed according to the *all-or-nothing* principle, namely, in such a way that either all of them succeed or none of them is performed. The correctness of the state of the data repository is usually ensured by requiring transactions to satisfy the so called *ACID* (atomicity, consistency, isolation and durability) properties, traditionally supported by database management systems.

JavaSpaces includes transactional mechanisms in a very elegant way: transactions are *created* and *committed* (or aborted) using a transactional service. Once started, the transaction has an associated identifier. This identifier is added as an additional parameter to all coordination primitives composing the transaction. In order to have a simplified syntax, we assume that the coordination primitives of a transaction x are all those primitives executed

$$(1'')\quad \langle a\rangle \xrightarrow{\overline{a}} \bullet \qquad (2)\quad write(a).P \xrightarrow{\tau} \langle a\rangle|P$$

$$(3)\quad take(a).P \xrightarrow{a} P \qquad (4)\quad read(a).P \xrightarrow{\underline{a}} P$$

$$(5)\quad \frac{P \xrightarrow{\overline{a}} P' \quad Q \xrightarrow{a} Q'}{P|Q \xrightarrow{\tau} P'|Q'} \qquad (6)\quad \frac{P \xrightarrow{\overline{a}} P' \quad Q \xrightarrow{\underline{a}} Q'}{P|Q \xrightarrow{\tau} P|Q'}$$

$$(7)\quad \frac{P \xrightarrow{\alpha} P'}{P|Q \xrightarrow{\alpha} P'|Q} \quad \alpha \neq \neg a, \vec{a}, \dot{a}, \surd \qquad (8)\quad \frac{P \xrightarrow{\alpha} P' \qquad K = P}{K \xrightarrow{\alpha} P'}$$

$$(35)\quad \langle a\rangle \xrightarrow{\Downarrow x} \langle a\rangle \qquad (36)\quad \bullet \xrightarrow{\Downarrow x} \bullet \qquad (37)\quad commit(x).P \xrightarrow{\downarrow x} P$$

$$(38)\quad \frac{P \xrightarrow{\alpha} P' \quad Q \xrightarrow{\alpha} Q'}{P|Q \xrightarrow{\alpha} P'|Q'} \quad \alpha \in \{\downarrow x, \Downarrow x\} \qquad (39)\quad \frac{P \xrightarrow{\downarrow x} P' \quad Q \xrightarrow{\Downarrow x} Q'}{P|Q \xrightarrow{\downarrow x} P'|Q'}$$

$$(40)\quad \frac{P \xrightarrow{\sigma_1} P' \quad P' \xrightarrow{\sigma_2 \downarrow x} P''}{P \xrightarrow{\sigma_1\sigma_2\downarrow x} P''} \quad \sigma_1 \neq \overline{a} \qquad (41)\quad \frac{P \xrightarrow{\sigma\downarrow x} P'}{create(x).P \xrightarrow{\sigma} P'}$$

$$(42)\quad P \succ\!\!\xrightarrow{\epsilon} P \qquad (43)\quad \frac{P \succ\!\!\xrightarrow{\sigma} P' \quad P' \xrightarrow{\overline{a}} P''}{P \succ\!\!\xrightarrow{\sigma\overline{a}} P''}$$

$$(44)\quad \frac{P \succ\!\!\xrightarrow{\sigma} P' \quad P' \xrightarrow{\overline{a}}}{P \succ\!\!\xrightarrow{\sigma\overline{a}} P'} \qquad (45)\quad \frac{P \xrightarrow{\sigma_1} P' \quad Q \succ\!\!\xrightarrow{\sigma_2} Q'}{P|Q \xrightarrow{\sigma} P'|Q'} \quad \sigma \in \sigma_1 \gg \sigma_2$$

Table 12
Operational semantics for `L[txn]` (symmetric rules of (5)–(7), (39) and (45) omitted).

between the execution of a *create(x)* and a subsequent *commit(x)* operation. In this way, it is not necessary to add the transaction identifier as an additional parameter.

We incrementally model transactions starting from the kernel calculus; then we extend the mechanism to cope with test for absence and event notification.

The approach we follow in our modeling, is to consider a transaction as a sequence of actions which must be executed atomically, i.e., not interleaved with actions performed by programs not involved in the transaction. Formally, the execution of a transaction is represented by a transition labeled by a sequence of labels, one for each action occurring inside the transaction.

A transaction is started with an operation of creation and possibly terminated by a commitment operation, performed by all the processes involved in the transaction. When performed within a transaction, a read operation may test

$\xrightarrow{\downarrow x}$	active commitment of transaction x
$\xrightarrow{\Downarrow x}$	passive commitment of transaction x

Table 13
New labels used in the operational semantics of `L[txn]`.

for the presence of either a datum produced under that transaction or a datum in the external dataspace. A take operation behaves in a similar way, and the selected datum will be withdrawn from the dataspace only if the transaction succeeds. A datum written during a transaction will not be visible to processes external to the transaction until the transaction commits; before commitment, this datum can be consumed by a process inside the transaction; in that case, the datum will never become externally visible.

In order to identify transactions, we will use *transaction names* taken from the set Txn, ranged over by x, y, ... In order to define formally the syntax of the new calculus `L[txn]` we have to add the syntax of the kernel calculus the new term $\bullet$ plus two new prefixes $create(x)$ and $commit(x)$, the former allowing the creation of new transactions and the latter for the transaction commitment:

$$P ::= \langle a \rangle \mid C \mid P|P$$
$$C ::= \mathbf{0} \mid \bullet \mid \mu.C \mid C|C \mid K$$

where:

$$\mu ::= write(a) \mid read(a) \mid take(a) \mid create(x) \mid commit(x)$$

where x is a transaction name taken from Txn. The $create(x)$ operation is used in order to start a new transaction called x; $commit(x)$ indicates the interest of a participant to the transaction to commit the transaction x, i.e., to terminate the transaction with success. The new term $\bullet$ is used to distinguish between terminated programs $\mathbf{0}$ and consumed data (now denoted with $\bullet$). This is necessary because, in order to successfully complete a transaction, all the involved processes must execute a $commit(x)$ action. In case there is a terminated process $\mathbf{0}$, the transaction cannot commit. The new set of configurations is denoted by $Conf[txn]$.

The set of possible labels is extended with two new elements: $\downarrow x$ representing an active commitment operation performed by terms agreeing on the successful termination of the transaction x, and $\Downarrow x$ representing a passive commitment performed by placeholders $\bullet$ for consumed data. Formally, $Label[txn] = Label \cup \{\downarrow x, \Downarrow x \mid x \in Txn\}$. The meaning of the new labels is explained in Table 13.

In order to model the operational semantics of transactions we proceed as

follows:

- we combine the sequence of actions executed within the same transaction in a unique transition whose label is such a sequence of actions;
- as a transaction can consume or test for presence data that are not created within the transaction, it is necessary to model a form of synchronization between the transaction and the external environment; the actions of the external environment are modeled via an auxiliary transition system $\overset{\sigma}{\rightarrowtail}$ where σ is a sequence of actions $\overline{a}$ or $\underline{a}$, denoting the availability of a datum $\langle a \rangle$ for consumption or for test of presence, respectively.

The operational semantics of the calculus is defined by the labelled transition system $(\mathit{Conf}[\mathit{txn}], \mathit{Label}[\mathit{txn}]^*, \longrightarrow)$. Observe that we move from simple labels to sequences of labels. Indeed, as a transaction consists of a sequence of operations performed in a single atomic step, its behaviour is operationally represented as a single transition, labelled with a sequence of actions. In the following we use σ, σ_1, σ_2, to range over $\mathit{Label}[\mathit{txn}]^*$, and ϵ to denote the empty sequence. The transition relation $\longrightarrow$ is defined as the least one satisfying the axioms and rules in the Table 12, where $(1'')$ is substituted for (1) and we omit the symmetric rule of (5)–(7), (39) and (45). Observe that rules (2)–(8) correspond to those already used in the kernel calculus L.

To synchronize a transaction with its surrounding environment, we introduce an auxiliary transition system, whose transitions are denoted by $\rightarrowtail$ and labelled with sequences of labels $\overline{a}$ and $\underline{a}$, representing that a datum $\langle a \rangle$ is offered for consumption or for testing its presence, respectively. The sequences of labels of this auxiliary transition system will be matched with the sequences of labels performed by transitions. To represent the fact that, at some point of a sequence of labels, the environment contains a datum a that is tested for presence by the transition, we need the additional label $\underline{a}$.

A transaction can successfully terminate if and only if all the programs involved in that transaction can perform a commitment. To this aim, it is necessary to distinguish among terminated programs, which cannot perform any kind of commitment and are denoted by $\mathbf{0}$, and placeholders for consumed data, that can perform a passive commitment. As stated above we denote these placeholders by $\bullet$. The new axiom $(1'')$ must be substituted for (1) in order to introduce the new term $\bullet$ as the result of the consumption of data. Axioms (35) and (36) permit data and placeholders for consumed data to perform a passive commit operation. On the other hand, by axiom (37) a program willing to commit a transaction can start an (active) commit operation. Rules (38) and (39) model the synchronization of commitment operations: an active commit can be performed if and only if at least one process performs an active commit, and all the other components perform an (active or passive) commit.

Rule (40) permits to group all the actions performed in a transaction in a single move. To capture also sequences originating from nested transactions, we need to label the left-hand transition in the premise with a sequence of actions. The data produced during a transaction cannot be read from processes external to the transaction, until the transaction commits. The side condition of the rule prevents this unwanted behaviour. Finally, rule (41) exports the moves performed inside a transaction to the external environment. Note that σ_2 in rule (40) and σ in rule (41) could be the empty string.

The second group of rules in Table 12 (rules (42)–(45)) define the auxiliary transition system denoting the environment moves to synchronize with a transaction. The environment can offer nothing (rule (42)), or sequences containing actions $\overline{a}$ (rule (43)) to denote that a datum has been consumed by a transaction from the environment, and $\underline{\overline{a}}$ (rule (44)), to denote that the environment contains a datum $\langle a \rangle$, that is tested for presence by the transaction.

To represent the synchronization of a transaction with its surrounding environment, we need an auxiliary function $\gg$: $Label^* \times Label^* \to \wp(Label^*)$, where $\wp(Label^*)$ denotes the collection of the sets of sequences of labels, defined as the least relation satisfying the following rules:

- $\varepsilon \in (\varepsilon \gg \varepsilon)$
- if $\sigma \in (\sigma_1 \gg \sigma_2)$ then $\tau\sigma \in (a\sigma_1 \gg \overline{a}\sigma_2)$ and $\tau\sigma \in (\underline{a}\sigma_1 \gg \underline{\overline{a}}\sigma_2)$
- if $\sigma \in (\sigma_1 \gg \sigma_2)$ then $a\sigma \in (a\sigma_1 \gg \sigma_2)$ and $\underline{a}\sigma \in (\underline{a}\sigma_1 \gg \sigma_2)$

The second item models the cases where a datum consumed/tested for presence by the transaction is offered by the process that we are synchronizing with the transaction, whereas in the third item the datum consumption/test request is passed to the external environment.

Rule (45) models the synchronization of a transaction with the environment: according to the definition of $\gg$, the consumption of a datum in the transaction can synchronize with an offer for the consumption of the datum by the environment, and a test for presence of a datum in the transaction can synchronize with an offer for test for presence by the environment. Non-τ actions of the transaction that are not synchronized with processes in parallel to the one performing the transaction are propagated to the environment.

Due to the introduction of the new labels obtained as sequences of symbols, we need to redefine our notion of reduction relation indicating the computation step that may be performed by stand-alone configurations. It is not enough to consider simply steps labeled with τ, but we have to consider also those transitions labeled with a sequence of symbols τ. Formally, $P \longrightarrow P'$ if $P \xrightarrow{\tau^*} P'$.

We illustrate the behaviour of processes containing transactions by means of

some examples.

Consider the process $\langle a\rangle|create(x).take(a).write(c).read(b).commit(x)$. The term inside the transaction can perfom the following moves:
$take(a).write(c).read(b).commit(x) \xrightarrow{a}$
$write(c).read(b).commit(x) \xrightarrow{\tau}$
$\langle c\rangle|read(b).commit(x) \xrightarrow{\underline{b}}$
$\langle c\rangle|commit(x) \xrightarrow{\downarrow x}$
$\langle c\rangle|\mathbf{0}$.
These moves can be grouped into the sequence
$take(a).write(c).read(b).commit(x) \xrightarrow{a\tau\underline{b}\downarrow x} \langle c\rangle|\mathbf{0}$.
The synchronization with the remaining part of the process can happen in two different ways:

- the subprocess $\langle a\rangle$ offers datum a for consumption by performing the move $\langle a\rangle \overset{\bar{a}}{\rightarrowtail} \bullet$;
  the synchronization between the move performed by the transaction and this move gives rise to the move
  $\langle a\rangle|create(x).take(a).write(c).read(b).commit(x) \xrightarrow{\tau\tau\underline{b}} \bullet|\langle c\rangle$;
  this means that the transaction tests for presence of a datum b in the external environment;
- the subprocess $\langle a\rangle$ does not offer datum a for consumption; in this case, the synchronization gives rise to the move
  $\langle a\rangle|create(x).take(a).write(c).read(b).commit(x) \xrightarrow{a\tau\underline{b}} \langle a\rangle|\langle c\rangle$,
  meaning that the transaction both removes datum a and tests for presence of datum b from the external environment.

To illustrate what happens if a datum produced in a transaction is also consumed within that transaction, consider the following process:
$create(x).take(a).write(c).take(c).commit(x)$.
In this case, the term inside the transaction can perform the following moves:
$take(a).write(c).take(c).commit(x) \xrightarrow{a}$
$write(c).take(c).commit(x) \xrightarrow{\tau}$
$\langle c\rangle|take(c).commit(x) \xrightarrow{\tau}$
$\bullet|commit(x) \xrightarrow{\downarrow x}$
$\bullet|\mathbf{0}$.
The process performs the move
$create(x).take(a).write(c).take(c).commit(x) \xrightarrow{a\tau\tau} \bullet|\mathbf{0}$;
hence, datum c will never be available to external processes.

The coordination language Linda provides only elementary operations on the dataspace, because they permit to operate on a single datum at a time. Transactions can be exploited to implement more sophisticated coordination primitives, operating on many data at the same time.

As an example, the following transaction models a set consumption operation $take(a, b)$, that succeeds only if both one datum a and one datum b are withdrawn from the dataspace:

$$create(x).(take(a).commit(x)|take(b).commit(x))$$

In general, it is possible to implement multiset rewriting operations à la Gamma [2], where in an atomic step a multiset of data (a set with possibly multiple occurrences of data) is withdrawn from the dataspace and, subsequently, another multiset of new data is produced.

In the literature, the expressiveness of Linda and Gamma have been compared adopting *modular embeddings* [6], a criterion based on the following idea: a language is as expressive as another one is, if there exists an encoding of programs of the former language into programs of the latter. The encoding should preserve some composition operator for programs, usually, at least the parallel composition operator. In other words, we want to define encoding functions $[\![\,]\!]$ such that the encoding $[\![P|Q]\!]$ of the parallel composition of two programs corresponds to the parallel composition $[\![P]\!]|[\![Q]\!]$ of the encodings of the programs themselves.

In particular, in [47] modular embeddings which preserve only parallel composition are used to prove that two calculi, one similar to `L[∃]` and another one comprising multiset rewriting operations, cannot be embedded one into the other. Applying this result to our context, we can say that test for absence operations cannot be embedded using transactions, and also that the transaction mechanism cannot be embedded exploiting test for absence operations. In other terms, test for absence and the transaction mechanism are incomparable features using the modular embedding approach.

In a related paper [8], transactions in Linda-like languages have been modeled: several results on the expressiveness of transactions (as well as the incomparability result of [47]) are proved using a slightly different form of modular embeddings, in which not only the parallel composition operator should be preserved, but also other operators such as the sequential and the choice composition of programs.

$$(9)\ \ take\exists(a)?P\_Q \xrightarrow{a} P \quad (10)\ \ take\exists(a)?P\_Q \xrightarrow{\neg a} Q$$

$$(11)\ \ read\exists(a)?P\_Q \xrightarrow{a} P \quad (12)\ \ read\exists(a)?P\_Q \xrightarrow{\neg a} Q$$

$$(13)\ \ \frac{P \xrightarrow{\neg a} P' \quad Q \overset{\overline{a}}{\not\longrightarrow}}{P|Q \xrightarrow{\neg a} P'|Q}$$

$$(46)\ \ \frac{P \overset{\sigma}{\rightarrowtail} P' \quad P' \overset{\overline{a}}{\not\longrightarrow}}{P \overset{\sigma\neg\overline{a}}{\rightarrowtail} P'}$$

Table 14
Additional axioms and rules for `L[txn,∃]` (symmetric rule of (13) omitted).

7.2 Adding Test for Absence: `L[txn,∃]`

In order to extend the transaction mechanism to the calculus with test for absence, we have to handle synchronization of test for absence with the environment. If a test for absence for a is successfully performed inside a transaction, this means that the environment does not contain occurrences of $\langle a\rangle$.

The syntax of the new calculus is obtained by adding the if-then-else terms to syntax of the calculus `L[txn]`:

$$P ::= \langle a\rangle \mid C \mid P|P$$

$$C ::= \mathbf{0} \mid \bullet \mid \mu.C \mid \eta?C\_C \mid C|C \mid K$$

where:

$$\mu ::= write(a) \mid read(a) \mid take(a) \mid create(x) \mid commit(x)$$

$$\eta ::= read\exists(a) \mid take\exists(a)$$

The new set of terms is denoted by $Conf[txn,\exists]$. The label $\overline{\neg a}$ is introduced in order to indicate that an environment does not contain occurrences of $\langle a\rangle$. Thus, we define $Label[txn,\exists] = Label[txn] \cup \{\overline{\neg a} \mid a \in Data\}$. As usual, the operational semantics is given by the labeled transition system $(Conf[txn,\exists], Label[txn,\exists]^*, \longrightarrow)$ where $\longrightarrow$ is defined as the least labelled transition relation which satisfies the axioms and rules in the Table 12 plus the additional axioms and rules in the Table 14. Observe that axioms (9)–(12) and rule (13) correspond to those already used for the calculus with test for absence `L[∃]`. The unique new rule is (46) which ensures that an environment offers $\overline{\neg a}$ if and only if it contains no occurrence of $\langle a\rangle$.

Now a transaction can synchronize with the environment only if each test for

absence performed in the transaction is matched by a corresponding $\overline{\neg a}$ action by the environment. We extend the function $\gg$ to cope with this case:

- if $\sigma \in (\sigma_1 \gg \sigma_2)$ then $\neg a\sigma \in (\neg a\sigma_1 \gg \overline{\neg a}\sigma_2)$

7.2.1 Example of test-and-set Operator

We show that the combination of transactions and predicates permit to implement a test-and-set primitive, which is the main ingredient in consensus algorithms for networks of symmetric processes.

The test-and-set operator $t\&s(a)$ atomically verifies the presence of data $\langle a \rangle$ and, if no datum $\langle a \rangle$ is available, produces atomically a new occurrence of it; this operator is implemented by the following process:

$$create(x).read\exists(a)?commit(x)\_write(a).commit(x)$$

In [11] it is proved that the test-and-set operator cannot be satisfactorily implemented in the calculus `L[∃]`. Moreover, in [48] another result concerning the impact on the expressiveness of `L[∃]` of the introduction of the test-and-set operation. Indeed, it is proved that in a calculus similar to `L[∃]` the leader election problem in symmetric networks cannot be solved, while this problem becomes solvable when also the test-and-set operation is added.

Given these results, we can conclude that the addition of transactions to the calculus `L[∃]` strictly increases its expressive power.

7.3 Adding Event Notification: `L[txn,ntf]`

In this section we consider the event notification mechanism. We will follow an approach close to the one adopted in JavaSpaces: a *notify* performed within a transaction provides notification of *write* operations performed within that transaction. When the transaction commits, any request for event notification performed during the transaction is dropped. Moreover, a *write* operation performed within a transaction generates event notification to listeners external to the transaction only if the written datum has not been withdrawn during the transaction.

The syntax of `L[txn]` is extended by introducing the $notify(a, C)$ primitive and the representation $on(a, C)$ for the listeners:

$$P ::= \langle a \rangle \mid on(a, C) \mid C \mid P|P$$

$$C ::= \mathbf{0} \mid \bullet \mid \mu.C \mid C|C \mid K$$

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

(16) $\dfrac{P \xrightarrow{\dot{a}} P' \quad Q \xrightarrow{\dot{a}} Q'}{P|Q \xrightarrow{\dot{a}} P'|Q'}$ (17) $\dfrac{P \xrightarrow{\dot{a}} P' \quad Q \xrightarrow{\dot{a}}\!\!\!\!\!/\;\;}{P|Q \xrightarrow{\dot{a}} P'|Q}$

(18) $\dfrac{P \xrightarrow{\vec{a}} P' \quad Q \xrightarrow{\dot{a}} Q'}{P|Q \xrightarrow{\vec{a}} P'|Q'}$ (19) $\dfrac{P \xrightarrow{\vec{a}} P' \quad Q \xrightarrow{\dot{a}}\!\!\!\!\!/\;\;}{P|Q \xrightarrow{\vec{a}} P'|Q}$

(35′) $\langle a \rangle \xrightarrow{\Downarrow x\vec{a}} \langle a \rangle$ (47) $on(a, P) \xrightarrow{\Downarrow x} \bullet$

(38′) $\dfrac{P \xrightarrow{\alpha\rho_1} P' \quad Q \xrightarrow{\alpha\rho_2} Q'}{P|Q \xrightarrow{\alpha\rho_1\rho_2} P'|Q'}$ $\alpha \in \{\downarrow x, \Downarrow x\}$ (39′) $\dfrac{P \xrightarrow{\downarrow x\rho_1} P' \quad Q \xrightarrow{\Downarrow x\rho_2} Q'}{P|Q \xrightarrow{\downarrow x\rho_1\rho_2} P'|Q'}$

(40′) $\dfrac{P \xrightarrow{\sigma_1\rho_1} P' \quad P' \xrightarrow{\sigma_2\downarrow x\rho_2} P''}{P \xrightarrow{\sigma_1\tau\sigma_2} P''}$ $\sigma_1 \neq \dot{a}, \bar{a} \ \wedge \ \forall a : \vec{a} \notin \sigma_1$ (41′) $\dfrac{P \xrightarrow{\sigma\downarrow x\rho} P'}{create(x).P \xrightarrow{\sigma\rho} P'}$

(51) $\dfrac{P \succ\!\!\!\rightarrow^{\sigma} P' \quad P' \xrightarrow{\dot{a}} P''}{P \succ\!\!\!\rightarrow^{\sigma\dot{a}} P''}$ (52) $\dfrac{P \succ\!\!\!\rightarrow^{\sigma} P' \quad P' \xrightarrow{\dot{a}}\!\!\!\!\!/\;\;}{P \succ\!\!\!\rightarrow^{\sigma\bar{\dot{a}}} P'}$

Table 15
Additional axioms and rules for `L[txn,ntf]` (symmetric rules of (17)–(19) and (39′) omitted).

where:

$$\mu ::= write(a) \mid read(a) \mid take(a) \mid create(x) \mid commit(x) \mid notify(a, C)$$

The new configurations are denoted with $Conf[txn, ntf]$. The set of labels should include also the labels $\dot{a}$ and $\vec{a}$ used to model the event notification mechanism (already discussed in Section 4): $Label[txn, ntf] = Label[txn] \cup \{\dot{a}, \vec{a} \mid a \in Data\}$. In the following, we use $\rho, \rho_1, \rho_2, \ldots$ to range over sequences of notification labels, i.e., $\rho, \rho_1, \rho_2, \ldots$ ranges over $\{\vec{a} \mid a \in Data\}^*$. The labelled transition system $(Conf[txn, ntf], Label[txn, ntf]^*, \longrightarrow)$ is defined by considering the axioms and rules in the Tables 12, plus those in Table 15 where $(35'), (38') - (41')$ are substituted for $(35), (38) - (41)$. Observe that the axioms and rules (14)–(19) correspond to those already used for the calculus with event notification `L[ntf]`.

When a transaction commits, all data written and not withdrawn under the transaction must raise an event notification. To this aim, transitions performed by transactions are labelled with action sequences with format $\sigma\rho$, $\sigma \downarrow x\rho$ or $\sigma \Downarrow x\rho$, where ρ contains only notification labels and σ does not contain

notification labels; the sequence σ represents the set of actions performed inside the transaction, whereas ρ permits to notify the data produced by the transaction to the external listeners. Note that the actual order of notification labels in the ρ part of a sequence has no relevance, i.e., ρ can be considered as a multiset of notifications instead of a sequence. We chose to consider ρ as a sequence instead of as a multiset for the sake of uniformity and to simplify the notation.

On transaction commitment, each datum $\langle a \rangle$ within the transaction produces an event notification label $\vec{a}$ (axiom (35')). On the other hand, all listeners produced by notification requests performed during the transaction are dropped (axiom (47)).

Rules (38')-(41') are a slight adaptation of rules (38)-(41) to cope with these event notification labels. According to rule (40'), event notifications performed during a transaction (either by a *write* operation or at the end of a nested transaction) are hidden to the external environment. The notification of data written but not consumed during the transactions will be performed on commitment.

As we have added event notifications to the moves performed by a transaction, we have to cope with this extension to handle synchronization with the environment. If an event notification $\vec{a}$ is performed by a transaction, all listeners in the environment waiting for that event must be woken up. To model this fact, we introduce a new label $\overline{\dot{a}}$: the environment offers $\overline{\dot{a}}$ (see rule (52)) if and only if it contains no listener on production of datum $\langle a \rangle$. On the other hand, by rule (51) the environment offers $\dot{a}$ if there are listeners interested in production of data $\langle a \rangle$.

Now a transaction can synchronize with the environment only if each event notification performed in the transaction is matched by a corresponding $\dot{a}$ or $\overline{\dot{a}}$ action by the environment. We extend the function $\gg$ to cope with this case:

- if $\sigma \in \sigma_1 \gg \sigma_2$ then $\vec{a}\sigma \in \vec{a}\sigma_1 \gg \dot{a}\sigma_2$
- if $\sigma \in \sigma_1 \gg \sigma_2$ then $\vec{a}\sigma \in \vec{a}\sigma_1 \gg \overline{\dot{a}}\sigma_2$

7.3.1 Alternative Semantics

It is worth noting the existence of at least one possible alternative semantics for event notification under transactions. We have followed the JavaSpaces approach by removing listeners created inside a transaction when the transaction terminates (see axiom (47)). On the other hand, we could maintain these listeners active. This can be obtained simply by replacing axiom (47) with the following:

$(47')$ $on(a, P) \xrightarrow{\Downarrow x} on(a, P)$

This alternative semantics seems more appropriate in order to satisfy a typical assumption about transactions: if a transaction consists of a single operation we can drop the transaction surrounding that operation, obtaining a behaviourally equivalent program. Only under the alternative semantics $create(x).notify(a, P).commit(x).Q$ is equivalent to $notify(a, P).Q$.

8 Conclusion and Related Work

We have presented a collection of process calculi inspired by the shared dataspace coordination model and languages initiated by Linda [25]. More precisely, we have defined calculi of increasing complexity in order to concentrate, step-by-step, on several relevant extensions of the native Linda shared dataspace coordination model. The extensions that we have considered are concerned with primitives related to *event notification*, *timeouts*, *timed data*, *timed event listeners*, and *transactions*. Another interesting line of extension of the Linda coordination model deals with the distribution of the data belonging to the shared dataspace on different nodes. We have not modeled these extensions because we consider this issue orthogonal with respect to the analysis of the coordination primitives we have presented in this paper.

Nevertheless, it is worth to mention some of the main proposals of shared dataspace coordination languages that deal with the distribution of data on several nodes. KLAIM [23] permits to locate shared dataspaces on different locations and allows programs to introduce and retrieve data on local as well as remote locations. Moreover, the primitive *eval* permits to send code that will be executed on a remote location, thus supporting mobility. A more recent family of decentralized Linda-like coordination models[43,33,34] supports a finer grained form of decentralization. Data are not associated to a specific space, but each datum has its own location. Dataspaces are then obtained as overlay structures that group together data possibly located at different sites. For example, in Lime[43] data are associated to agents, which are software components running on hosts. A group of connected hosts form a confederation. All the agents running in the same confederation share the same dataspace (called Transiently Shared Dataspace); this dataspace is obtained at run-time grouping together the data stored in the agents currently running on the confederated hosts. Logical mobility is supported in the sense that agents can move from one host to another. Physical mobility, on the other hand, is supported in the sense that hosts can move by joining and leaving confederations. When a host/agent leaves (resp. joins) a confederation, the data stored in that host/agent leave (resp. join) the corresponding dataspace. In TOTA [33] and swarm-Linda [34] also data can autonomously move according to propagation

rules that are associated to each datum at creation time.

The remainder of this section is devoted to a discussion of related literature structured as follows: a discussion of the previous work of the authors, the presentation of other approaches used to model shared dataspace coordination in process calculi, and finally a conclusive subsection reporting the references to papers of the process algebra community in which process calculi are used to investigate features related to ours (namely, timeouts and non-persistent data).

8.1 Previous Work of the Authors

As already discussed in the Introduction, the main contribution of this paper is in the homogeneous presentation and overview of previous work of the authors [10–12,48,47,13–15,17,18]. The many results hinted here are fully detailed in the references above.

In [10] we have initiated our formal investigation of shared dataspace coordination languages focusing on Linda. In particular, we studied observational equivalences (based on the notion of bisimulation) for several subcalculi comprising different sets of Linda coordination primitives. In [11] this investigation is extended taking into account also different interpretations for the *write* operation among which the unordered interpretation (according to which the processes communicate asynchronously with the repository) and the ordered one (which assumes that the communication between the processes and the repository is synchronous). The main difference between these two semantics is shown in [12] where we prove that a Linda-based process calculus is Turing powerful under the ordered semantics while this is not the case under the unordered one.

In [48] coordination primitives able to test the absence of data alternative to the Linda predicates are modeled and a hierarchy of expressiveness among them is investigated. The basic coordination primitives of Linda are able to act on a single datum at a time (e.g., consume or withdrawn one single datum). In [47] we investigate operations able to consume and produce multisets of data in a single atomic action, and the expressiveness of this new coordination mechanism is compared to the basic Linda coordination primitives.

In [13] we have initiated the investigation of more recent coordination middlewares (such as JavaSpaces and TSpaces), which extend the traditional Linda coordination model with the features described above.

The introduction of event notification has been investigated in [18] and in [17]: in [18] we show that event notification strictly increases the expressiveness of

coordination languages with only *write*, *read*, and *take* operations (while this is not the case if also test for absence is considered); in [17] we investigate the difference between the ordered and the unordered semantics in presence of event notification, showing that the gap of expressiveness proved in [12] does not hold any more.

The notion of leased data, as introduced by the JavaSpaces specifications, has been investigated in [14]; in that paper we consider two interpretations of the passing of time, namely global (according to which a global clock exists) and local (several independent local clocks exist), and we prove a gap of expressiveness between the two interpretations. In [15] we move to a different notion of timed data, according to which a datum could remain available in the repository even after its expiration; we show that a gap of expressiveness, similar to the previous one, holds between two implementations of the expired data collector, the first which removes one datum selected among all those expired, and the second which selects the datum which expired first.

As far as the modeling of the new features is concerned, this is the first paper in which we consider timeouts and transactions. Moreover, we report in the Appendix the proof of two new equivalence results: Appendix A considers weak bisimilarity and the preservation of process termination between the ordered and the unordered semantics (in the calculus with only *write*, *read*, and *take* operations); on the other hand, in Appendix B we prove the equivalence between the global and the local interpretation of the passing of time in the calculus with timeouts.

8.2 Shared Dataspaces in Process Calculi

Traditional process calculi adopt channel-based communication (see, for instance, CSP [29] and CCS [35]). Besides the previous work of the authors, other process calculi which consider shared dataspace communication can be found in the literature.

In [21] a process calculus is defined which comprise the basic input, output, and read Linda primitives. Differently from our approach, the read operation is treated as an input with a subsequent emission of the consumed datum. This means that $rd(a).P$ is just a macro for $in(a).out(a).P$. This approach is not satisfactory in a context, such as our own, in which the notification of data production is considered. Indeed, in the presence of *notify*, the program $rd(a).P$ is different from $in(a).out(a).P$ because the former should not produce any reaction, while the latter should trigger all those reactions related to *notify* operations previously executed on the datum a.

A similar modeling of the Linda coordination mechanism has been adopted

in [24]: in addition, that paper reports the investigation of adequate observational equivalences based on testing [22]. Observational semantics for Linda-like coordination languages are studied also in [7]: however, in that paper a different approach is considered based on a denotational semantics which is fully abstract with respect to an observational criterion based on sequences of states of the shared dataspace that a computation may produce.

The same authors have investigated also features related to time in a more recent paper [30]. Differently from our approach, they assume *two phases functioning*: time passes only when no action can be taken, that is, only when all the processes are blocked waiting for time passing and cannot execute any other action. This approach looks adequate for the modeling of synchronous systems. In this paper we try to be more general, taking into account also systems in which time passes asynchronously.

An alternative modeling of time has been adopted in [5] where, besides the typical *read*, *write* and *take* primitives, also a read operation with timeout is considered. Differently from our modeling of time, all the actions take exactly one unit of time. Moreover, in the case no other action can be fired in the system, and a datum $\langle a \rangle$ is available, a read operation on that datum must succeed even if executed under timeout. More precisely, following the approach of [5] the *read* operation in the configuration

$$\langle a \rangle \mid read(a, 2)?\mathbf{0}\_ write(b)$$

can only succeed (i.e., $\langle b \rangle$ cannot be produced) while in our approach it can also fail (i.e., $\langle b \rangle$ can be produced as discussed in Section 5).

In all the above papers, only Linda-like coordination primitives are considered in order to access the shared dataspace coordination medium. Alternative proposals are based on multiset rewriting. In particular, it is worth to mention the Calculus of Gamma Programs in [28] (a calculus in which rewriting rules can be combined exploiting a sequential and a parallel composition operator) and the Chemical Abstract Machine [4] (in which dataspaces can contain also other dataspaces, thus obtaining a hierarchical structure of nested data repositories).

8.3 Related Process Calculi

We conclude reporting a comparison with related papers of the process algebra community in which similar aspects, such as timeouts or non-persistent data, are investigated.

We have modeled time following a typical approach, according to which actions are atomic and time passes in between them via explicit global synchronizations. This approach has been modeled, in the form we have adopted, for the first time in [37]: time is divided in discrete intervals, and transitions are used to model the influence of the passing of time on the terms of the calculus. More precisely, two new prefix operators are introduced in a CCS-like language in order to model delayed and idle processes: delayed processes are blocked for exactly a predefined number of basic time intervals; idle processes, on the other hand, are not influenced by the passing of time. Two different choice composition operators, a *strong* and a *weak* choice, are used to combine these two basic time mechanisms in order to model more complex behaviours. The main difference with our approach is that we have decided to avoid the introduction of new prefixes to model time features; indeed, in our process calculi, prefixes always correspond to coordination primitives. For example, we do not use any syntax to represent processes not influenced by the passing of time, but we consider a semantic property, that is, the absence of outgoing transitions denoting time passing (i.e., transitions labelled with $\surd$).

Finally, it is worth to mention [3]: in that paper the π–calculus [36] is extended in order to perform a formal analysis of the two-phase commit protocol. In particular, two extensions are of interest with respect to our process calculi: *lossy messages* and *timers*. Lossy messages are data which could be lost, e.g., during transmission. These messages are similar to the non-persistent data we have in our process calculus with leasing. The main difference is that lossy messages may disappear at any time, while leased data remain available at least until leasing expiration. Timers are used to model programs with timeout: more precisely $timer^t(P, Q)$ denotes a program which may behave like P, in the case it starts acting before t clock ticks, otherwise it becomes Q (exactly after t clock ticks). This approach is very close to our timeout notation: indeed, our term $take(a)_t?P\_Q$ corresponds (in the notation of [3]) to $timer^t(take(a).P, Q)$.

References

[1] B. Anderson and D. Shasha. Persistent Linda: Linda + Transactions + Query. In *Research Directions in High-Level Parallel Programming Languages*, volume 574 of *Lecture Notes in Computer Science*, pages 93–109. Springer-Verlag, Berlin, 1991.

[2] J.P. Banâtre and D. Le Métayer. Programming by Multiset Transformation. *Communications of the ACM*, 36(1): 98–111, 1993.

[3] M. Berger and K. Honda. The Two-Phase Commit Protocol in an Extended π-Calculus In *Proc. of EXPRESS'00*, volume 39 of *Electronic Notes in Theoretical Computer Science*, Elsevier, 2000.

[4] G. Berry, G. Boudol. The Chemical Abstract Machine. *Theoretical Computer Science*, 96:217–248, 1992.

[5] F.S. de Boer, M. Gabbrielli, and M.C. Meo. A Timed Linda Language. In *Proc. of Coordination'00*, volume 1906 of *Lecture Notes in Computer Science*, pages 299–304. Springer-Verlag, Berlin, 2000.

[6] F.S. de Boer and C. Palamidessi. Embedding as a Tool for Language Comparison: On the CSP Hierarchy. In *Proc. of CONCUR'91*, volume 527 of *Lecture Notes in Computer Science*, pages 127–141. Springer-Verlag, Berlin, 1991.

[7] A. Brogi and J.M. Jacquet. Modeling Coordination via Asynchronous Communication. In *Proc. of Coordination'97*, volume 1282 of *Lecture Notes in Computer Science*, pages 238–255. Springer-Verlag, Berlin, 1997.

[8] A. Brogi and J.M. Jacquet. On the Expressiveness of Coordination Models. In *Proc. of Coordination'99*, volume 1594 of *Lecture Notes in Computer Science*, pages 134–149. Springer-Verlag, Berlin, 1999.

[9] N. Busi. Petri Nets with Inhibitor and Read Arcs: Semantics, Analysis and Application to Process Calculi, Ph.D. Thesis, Department of Mathematics, University of Siena, Italy, 1998.

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

[11] N. Busi, R. Gorrieri, and G. Zavattaro. Comparing Three Semantics for Linda-like Languages. *Theoretical Computer Science*, 240(1):49–90, 2000.

[12] N. Busi, R. Gorrieri, and G. Zavattaro. On the Expressiveness of Linda Coordination Primitives. *Information and Computation*, 156(1/2):90–121, 2000.

[13] N. Busi, R. Gorrieri, and G. Zavattaro. Process Calculi for Coordination: from Linda to JavaSpaces. In *Proc. of AMAST2000*, volume 1816 of *Lecture Notes in Computer Science*, pages 198-212. Springer-Verlag, Berlin, 2000.

[14] 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.

[15] N. Busi, R. Gorrieri, and G. Zavattaro. Temporary Data in Shared-Dataspace Coordination. In *Proc. of FOSSACS2001*, volume 2030 of *Lecture Notes in Computer Science*, pages 121–136. Springer-Verlag, Berlin, 2001.

[16] N. Busi, A. Rowstron, and G. Zavattaro. State- and Event-Based Reactive Programming in Shared Dataspaces. In *Proc. of Coordination2002*, volume 2315 of *Lecture Notes in Computer Science*, pages 111–124. Springer-Verlag, Berlin, 2002.

[17] 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.

[18] 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.

[19] 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.

[20] A. Cheng, J. Esparza, J. Palsberg. Complexity results for 1-safe nets. In *Theoretical Computer Science*, 147:117–136, 1995.

[21] P. Ciancarini, K.K. Jensen, and D. Yankelevich. On the Operational Semantics of a Coordination Language. In *Object-Based Models and Languages for Concurrent Systems*, volume 924 of *Lecture Notes in Computer Science*, pages 77–106, Springer Verlag, 1995.

[22] R. De Nicola and M.C.B. Hennessy. Testing Equivalences for Processes. *Theoretical Computer Science*, 34:83-133, 1984.

[23] R. De Nicola, G. Ferrari, and R. Pugliese. KLAIM: A Kernel Language for Agents Interaction and Mobility. *IEEE Transactions on Software Engineering*, 24(5):315–330, May 1998. Special Issue: Mobility and Network Aware Computing.

[24] R. De Nicola and R. Pugliese. A Process Algebra based on Linda. In *Proc. of Coordination'96*, volume 1061 of *Lecture Notes in Computer Science*, pages 160–178, Springer Verlag, 1996.

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

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

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

[28] C. Hankin, D. Le Métayer, and D. Sands. A Calculus of Gamma Programs. In *Proc. of 5th International Workshop on Languages and Compilers for Parallel Computing*, volume 757 of *Lecture Notes in Computer Science*, pages 342–355. Springer-Verlag, Berlin, 1993.

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

[30] J.M. Jacquet, K. de Bosschere, and A. Brogi. On timed coordination languages. In *Proc. of Coordination'00*, volume 1906 of *Lecture Notes in Computer Science*, pages 81–99. Springer-Verlag, Berlin, 2000.

[31] T. Kielmann. Objective Linda: A Coordination Model for Object-Oriented Parallel Programming. PhD thesis, Dept. of Electrical Engineering and Computer Science, University of Siegen, Germany, 1997.

[32] T.W. Malone and K. Crowston. The Interdisciplinary Study of Coordination. *ACM Computing Surveys*, 26(1):87–119, 1994.

[33] M. Mamei, F. Zambonelli and L. Leonardi. Tuples On The Air: A Middleware for Context-Aware Computing in Dynamic Networks. In *Proc. of The International Workshop on Mobile Computing Middleware*, pages 342-347, 2003.

[34] R. Menezes and R. Tolksdorf. A New Approach to Scalable Linda-systems Based on Swarms. In *Proc. of ACM Symposium on Applied Computing*, SAC 2003, Melbourne, FL, USA.

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

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

[37] F. Moller and C. Tofts. A Temporal Calculus of Communicating Systems. In *Proc. of Concur'90*, volume 458 of *Lecture Notes in Computer Science*, pages 467–480. Springer-Verlag, Berlin, 1990.

[38] Sun Microsystem, Inc. *JavaSpaces Specifications 1.1*, 1999.

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

[40] A. Omicini and F. Zambonelli. Coordination of mobile information agents in TuCSoN. *Journal of Internet Research*, 8(5), 1998.

[41] C. Palamidessi. Comparing the Expressive Power of the Synchronous and the Asynchronous π-calculus. *Mathematical Structures in Computer Science*, 13(5): 685-719, 2003

[42] G.A. Papadopoulos and F. Arbab. Coordination Models and Languages. *Advances in Computers*, 46:329–400, 1998.

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

[44] W. Reisig. *Petri nets: An Introduction*. EATCS Monographs in Computer Science, Springer, 1985.

[45] A. Rowstron. WCL: A Co-ordination Language for Geographically Distributed Agents. *World Wide Web*, 1:167–179, 1998.

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

[47] G. Zavattaro. On the Incomparability of Gamma and Linda. Technical Report SEN-R9827, Centrum voor Wiskunde en Informatica-CWI, 1998.

[48] G. Zavattaro. Towards a Hierarchy of Negative Test Operators for Generative Communication. In *Proc. of EXPRESS'98*, volume 16(2) of *Electronic Notes in Theoretical Computer Science*, Elsevier, 1998.

A Ordered vs. Unordered Semantics in L

A.1 Weak bisimulation

Here we prove that for the kernel calculus L the two approaches for the interpretation of the *write* operation, namely the ordered and unordered semantics, are equivalent under the weak bisimulation equivalence [35], a typical equivalence relation which does not take into account unobservable steps labelled with τ.

Definition A.1 A binary, symmetric relation $\mathcal{R}$ on *Conf* is a *weak bisimulation* if $(P, Q) \in \mathcal{R}$ implies:

- if $P \xrightarrow{\alpha} P'$ with $\alpha \neq \tau$ then there exists Q', Q'', Q''' such that $Q \xrightarrow{\tau}{}^{*} Q'' \xrightarrow{\alpha} Q''' \xrightarrow{\tau}{}^{*} Q'$ and $(P', Q') \in \mathcal{R}$;
- if $P \xrightarrow{\tau} P'$ then there exists Q' such that $Q \xrightarrow{\tau}{}^{*} Q'$ and $(P', Q') \in \mathcal{R}$.

Two agents P and Q are *weak bisimilar*, written $P \approx Q$, if there exists a weak bisimulation $\mathcal{R}$ such that $(P, Q) \in \mathcal{R}$.

In the presence of the choice composition operator the weak bisimulation relation is not a congruence [35]; on the other hand, it is a congruence in the presence of only the standard prefixes and the parallel composition operator (as is the case for our kernel calculus L).

In order to prove the equivalence between the ordered and unordered semantics for the *write* operation we define: $Ord(P)$ the term obtained by replacing all the unordered output $write_u$ in P with their ordered version *write*; on the other hand let $Unord(P)$ be the term obtained by replacing all the ordered output *write* in P with their unordered version $write_u$.

Theorem A.2 For any configuration P, we have that $Ord(P) \approx Unord(P)$.

Proof. The thesis is a direct consequence of the fact that the weak bisimulation relation is a congruence and that $write(a).P \approx write_u(a).P$ for any program P.

The equivalence between $write(a).P$ and $write_u(a).P$ can be proved as follows; we first observe that both programs have only one outgoing transition labelled with τ leading to the configurations $\langle a \rangle | P$ and $\langle\langle a \rangle\rangle | P$, respectively; then it is enough to show that the two reached configurations are weak bisimilar, i.e., $\langle a \rangle | P \approx \langle\langle a \rangle\rangle | P$. To prove this it is enough to consider the trivial equivalence $\langle a \rangle \approx \langle\langle a \rangle\rangle$ and to use the fact that $\approx$ is a congruence. ∎

Here we prove that for the kernel calculus `L` the ordered and the unordered semantics are equivalent from the point of view of the termination of processes.

We recall some notation which is used in the following: a *reduction step*, denoted with $P \longrightarrow P'$, corresponds to a transition labelled with τ (i.e. $P \xrightarrow{\tau} P'$); a configuration P is *terminated*, denoted with $P \not\longrightarrow$, if it has no outgoing transitions labelled with τ; a configuration has a *terminating computation*, denoted with $P \downarrow$, if $P \longrightarrow^* P' \not\longrightarrow$ (where $P \longrightarrow^* P'$ is the reflexive and transitive closure of $P \longrightarrow P'$).

The following fact states that a configuration interpreted under the ordered semantics is terminated if and only if it is terminated also under the unordered one. This is a direct consequence of the following observation: a terminated configuration is the parallel composition of data available in the shared dataspace and programs which are either terminated or willing to read or consume unavailable data. Thus, in a terminated configuration there exists no program able to perform any *write* operation; thus the kind of interpretation for the *write* primitive has no influence.

Fact 1 *For any configuration P, we have that $Ord(P) \not\longrightarrow$ if and only if $Unord(P) \not\longrightarrow$.*

The following fact consists of two statements: the first is trivial to prove and indicates that a computation under the ordered semantics can be always simulated under the unordered one. The second sentence, which can be proved by induction on the length of the derivation $Unord(P) \longrightarrow^* P'$, states that each computation under the unordered semantics can be extended in order to obtain an equivalent computation valid under the ordered one.

Fact 2 *For any configuration P, we have that:*

- *if $Ord(P) \longrightarrow^* P'$ then $Unord(P) \longrightarrow^* Unord(P')$;*
- *if $Unord(P) \longrightarrow^* P'$ then there exists P'' such that $P' \longrightarrow^* P''$ and $Ord(P) \longrightarrow^* Ord(P'')$.*

We can now prove the new equivalence result between the ordered and unordered semantics.

Theorem A.3 For any configuration P, we have that $Ord(P) \downarrow$ if and only if $Unord(P) \downarrow$.

Proof. The thesis is a direct consequence of the Facts 1 and 2. ∎

B Synchronous vs. Asynchronous Time in L[Δ]

Synchronous and asynchronous time can be considered equivalent for the calculus with timeouts, as a consequence of the following theorem stating that, given an initial configuration P (i.e., a configuration in which no timeout is counted yet), the configurations that can be reached from P under the synchronous time are exactly the same as those reachable under the asynchronous time.

Theorem B.1 Let P be an initial configuration not including terms sensible to the passing of time, i.e., without subterms of the kind $\eta_t?P\_Q$. We have that $P \longrightarrow_s^* P'$ if and only if $P \longrightarrow_a^* P'$.

Proof. The two directions of the double implication are proved separately. The *only if* part (a computation under the synchronous time is valid also under the asynchronous one) follows directly from the fact that the components sensitive to the passing of time may synchronize on their $\surd$ transitions also under the asynchronous time.

The *if* part (a computation under the asynchronous time can be simulated also under the synchronous one) is proved by induction on the length of the computation $P \longrightarrow_a^* P'$. The base case (length equal to 0) is trivial. In the inductive case we can assume the existence of P'' such that $P \longrightarrow_a^* P'' \longrightarrow_a P'$.

Consider now $P'' \longrightarrow_a P'$: we have two cases to analyse.

As first case we consider that the reduction is due to the firing of a transition labelled with τ (i.e., $P'' \xrightarrow{\tau} P'$ under the asynchronous time) then the thesis is proved because the asynchronous and the synchronous reductions do not differ in their τ labelled transitions.

As second case we consider that the reduction is due to the firing of a transition labelled with $\surd$ (i.e., $P'' \xrightarrow{\surd} P'$ under the asynchronous time). The transition involves only the terms $\eta_t?P\_Q$ which synchronize on the $\surd$ transition: each of these terms is replaced by the corresponding successor indicated by the axiom (22) or (23).

The configuration P'' could contain other terms of the kind $\eta_t?Q\_R$ not involved in the $\surd$ transition. This set of terms is partitioned into two subsets: the *old* terms with a time index which has been decreased at least once, and the *new* terms which are still during their first time interval (i.e., their index is still the one defined at the moment of their creation).

Each old term $\eta_t?Q\_R$ has a predecessor of kind $\eta_{t+1}?Q\_R$; each new term $\eta_t?P\_Q$ has a predecessor of kind $take(a,t)?Q\_R$ or $read(a,t)?Q\_R$.

Let P''' be the term obtained from P'' by substituting all the old and new terms with their corresponding predecessors. As P''' differs from P'' only for these predecessors, we have that $P \longrightarrow_a^* P'''$ with a computation shorter than $P \longrightarrow_a^* P''$. By induction hypothesis also $P \longrightarrow_s^* P'''$ holds.

We finally prove the thesis showing that also $P''' \longrightarrow_s^* P'$ holds. Indeed, P' can be obtained by substituting in P''' each term $\eta_t?Q\_R$ with its successor, and by substituting each predecessor of a new term with the corresponding new term. For these reasons the computation $P''' \longrightarrow_s^* P'$ consists of a $\surd$ transition followed by one transition for each new term that should be produced from its corresponding predecessor occurring in P'''. ■