On the Expressiveness of Distributed Leasing in Linda-like Coordination Languages

Nadia Busi  Roberto Gorrieri  Gianluigi Zavattaro

May 2000

Department of Computer Science
University of Bologna
Mura Anteo Zamboni 7
40127 Bologna (Italy)
Recent Titles from the UBLCS Technical Report Series

99-12 Simulative and Experimental Analysis of an Adaptive Playout Delay Adjustment Mechanism for packetized Voice across the Internet, M. Roccetti, V. Ghini e G. Pau, June 1999.
On the Expressiveness of Distributed Leasing in Linda-like Coordination Languages

Nadia Busi  Roberto Gorrieri  Gianluigi Zavattaro

May 2000

Abstract
Distributed leasing is an emerging paradigm adopted in applications based on the client-server architecture. The basic idea is that, at the moment a client asks for a service, it also declares the (maximal) duration that service should be granted. This paradigm has been recently adopted by SUN Microsystems in the definition of JavaSpaces, a coordination middleware inspired by the shared dataspace model of Linda. In this paper we investigate the interplay between distributed leasing and the Linda coordination model. To this aim, the output operation of a Linda-like calculus is adapted to enrich the tuple with expiration time; the semantics of the resulting calculus is studied under both assumptions that there is one unique global clock or many local clocks. We prove that the introduction of distributed leasing strictly decreases the expressiveness of the Linda-like calculus, and that global time is strictly more expressive than local time.

1. Work partially supported by Esprit working group n.24512 “Coordina”
1 Introduction

In the last decade we assisted to a dramatic evolution of computing systems, leading to a world-wide network connecting smaller, yet much more powerful processors. The next expected step in this direction is represented by the so-called ubiquitous computing, based on the idea of dynamically reconfigurable federations composed of users and resources required by those users. The Jini architecture [13] represents a first proposal of Sun Microsystems for a Java-based technology inspired by this new computing paradigm.

The scenario advocated by Jini is based on dynamically reconfigurable systems of distributed objects. One object may offer resources to other objects in the system by granting its use for a certain period of time; this period is determined, using a request/response form of negotiation, when access to the resource is first requested and then given. The grant is called lease, and this style of programming is referred to as distributed leasing.

The main motivation for the introduction of distributed leasing is that it provides a possible solution to the problem of partial failures. Consider a scenario in which resources are granted until they are explicitly freed. If a process which owns some resources crashes, those resources will be never freed. On the other hand, under distributed leasing, the resources will be automatically freed when the corresponding lease expires.

JavaSpaces [12] is the part of the Jini architecture devoted to the definition of a coordination infrastructure. It is inspired by the Linda [6] shared dataspace coordination model; it provides the abstraction of a shared space where the processes may introduce and retrieve the data and the information needed in order to communicate, cooperate, and coordinate. Distributed leasing plays an important role also in JavaSpaces, as it is adopted in order to negotiate the duration of the entities introduced inside the shared repository. At the moment a process emits a new datum, it also indicates its maximal expected duration. The manager of the space decides, at the moment the new object is introduced, whether to ensure the requested duration or to keep the object for a shorter period (because, e.g., the manager knows that, later on, the server will be down).

In this paper we investigate the interplay between distributed leasing and the Linda coordination paradigm. To this aim we start by recalling a Linda-like process calculus, introduced in [1], where we have investigated two variants for the output operation which have been called ordered and unordered, respectively, differing for the instant of time when the data become available in the shared dataspace. Under the ordered approach, according to which a datum becomes available immediately after its emission, the calculus is Turing complete. On the other hand, under the unordered approach, in which the datum becomes available only after an unpredictable delay, the expressive power strictly decreases as the existence of a terminating and an infinite computation become both decidable properties.

Here we embed distributed leasing in the Linda-like process calculus of [1] interpreted under the ordered approach: more precisely, we adapt the ordered output operation to include the expiration time into the emitted tuple. After that time, the tuple is removed. Note that the classic output operation that guarantees forever persistency of the emitted datum is no longer available, even if the expiration time is set to infinity; indeed, there is no guarantee that the manager of the shared dataspace will grant for the requested period of time.

The presence of distributed leasing moves our interest from the instant in which data become available (as in the two variations of the classic output operation discussed in [1]), to the instant in which they may disappear, not only as the effect of an explicit consumption operation, but also as the consequence of the termination of the leasing period. In other words, instead of considering that the processes have no knowledge of the instant in which the emitted data become available, we move to a scenario in which processes have no control on the instant in which the data disappear.

Another interesting aspect is related to the modeling of the passing of time. We consider two opposite approaches, global and local time. In the first case we suppose that there exists only one shared clock which is used to evaluate the instant in which all the leases expire. On the other hand, in the second approach each shared datum has its own (local) clock. The global time approach represents implementations of the dataspace where the data are stored all in the
same machine, while the second approach models distributed implementations in which it is not possible to make any assumption on the place where data actually reside.

In the following, we will denote with $L_o$ and $L_u$ the calculus of [1] interpreted under the ordered and unordered approaches, respectively; on the other hand, with $L_{gt}$ and $L_{lt}$ we denote the new calculi embedding distributed leasing under the global time and local time assumptions, respectively.

In [1] we have studied the expressiveness of $L_o$ and $L_u$ by investigating the possibility to encode Random Access Machines [11], a well known Turing powerful formalism.

We consider two basic properties that an encoding of RAMs may preserve: termination and divergence. RAMs are deterministic; thus, given a specific RAM, its computation either terminates (denoted with $\text{RAM} \downarrow$) or diverges (denoted with $\text{RAM} \uparrow$). On the other hand, a term $P$ taken from one of our process calculi may be nondeterministic; in other words, it could give rise to different computations, and these computations may be either finite or infinite. If $P$ has a finite computation (i.e., leading to a terminated/deadlocked state) we write $P \downarrow$, if $P$ has an infinite computation we write $P \uparrow$. Due to nondeterminism we may have both $P \downarrow$ and $P \uparrow$. Let $[]}$ be an encoding of RAMs; we say that the encoding preserves termination if $\text{RAM} \downarrow$ iff $][\text{RAM}] \downarrow$, and that it preserves divergence if $\text{RAM} \uparrow$ iff $][\text{RAM}] \uparrow$. Observe that due to nondeterminism, we may have that, given an encoding $][\,$ which preserves only termination (resp. divergence), and given a RAM such that $\text{RAM} \downarrow$ (resp. $\text{RAM} \uparrow$), also $][\text{RAM}] \downarrow$ (resp. $][\text{RAM}] \uparrow$ could hold.

In [1] we have presented an encoding of RAMs in $L_o$ which preserves both termination and divergence (this encoding is recalled in Section 3). We have proved also that, on the contrary, in $L_u$ it is impossible to define an encoding which satisfies at least one of the considered properties; this result is achieved by proving that $P \downarrow$ and $P \uparrow$ are decidable properties in $L_o$.

In this paper we extend our analysis of the ability to encode RAMs by moving to the new languages $L_{gt}$ and $L_{lt}$. Concerning $L_{lt}$, we present a RAM encoding which preserves termination, and we show the impossibility to define encodings which preserves divergence. This result is achieved by proving that $P \uparrow$ is a decidable property in this language. Even more interesting is the analysis of $L_{gt}$; we show two possible encodings, the first which preserves termination and the second which preserves divergence, and we prove the impossibility to define encodings which preserves both termination and divergence. In order to achieve this result we first observe that given an encoding $][\,$ which preserves both termination and divergence, we have that for any $\text{RAM}$ the term $][\text{RAM}]$ is uniform with respect to termination or divergence (i.e., its computations are either all finite or all infinite). After, we prove that $P \uparrow$ is decidable for uniform processes of $L_{gt}$.

The above results are summarized in the following table where we describe the different level of expressiveness of the various calculi by showing the properties that an encoding of RAM may preserve:

<table>
<thead>
<tr>
<th></th>
<th>Termination</th>
<th>Divergence</th>
<th>Termination and Divergence</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L_o$</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>$L_{gt}$</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td>$L_{lt}$</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>$L_u$</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
</tr>
</tbody>
</table>

The structure of the paper is as follows: in Section 2 we define the four process calculi, in Section 3 we present the RAM encodings, in Section 4 we prove the decidability of divergence in $L_{lt}$ and for uniform processes of $L_{gt}$, and Section 5 reports some conclusive remarks. Due to space limitations we do not report the proofs of the Theorems.

2 The Process Calculi

In this section we define the calculi described in the Introduction: we first present a complete syntax and semantics, and then we characterize the various calculi as subsets (or minimal variations) of this complete calculus.
using the composition operator UBLCS-2000-5

4

the parallel composition of subprograms, or a program constant

(i) it incorporates the indication of the expected maximal

Table 1. Operational semantics (symmetric rules omitted).

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>(\text{out}(a, t)?P \rightarrow \langle a \rangle t \mid P)</td>
</tr>
<tr>
<td>(2)</td>
<td>(\text{out}(a, t)?P \rightarrow Q)</td>
</tr>
<tr>
<td>(3)</td>
<td>(\langle a \rangle \rightarrow P \rightarrow 0)</td>
</tr>
<tr>
<td>(4)</td>
<td>(\text{in}(a).P \rightarrow P)</td>
</tr>
<tr>
<td>(5)</td>
<td>(\langle a \rangle \rightarrow \langle a \rangle_{t-1})</td>
</tr>
<tr>
<td>(6)</td>
<td>(\langle a \rangle_0 \rightarrow 0)</td>
</tr>
<tr>
<td>(7)</td>
<td>(\text{inp}(a)?P \rightarrow a \rightarrow P)</td>
</tr>
<tr>
<td>(8)</td>
<td>(\text{inp}(a)?P \rightarrow Q \rightarrow a \rightarrow Q)</td>
</tr>
<tr>
<td>(9)</td>
<td>(\frac{P \rightarrow a \rightarrow P'}{P</td>
</tr>
<tr>
<td>(10)</td>
<td>(\frac{P \rightarrow a \rightarrow P'}{P \rightarrow Q \rightarrow a \rightarrow Q'})</td>
</tr>
<tr>
<td>(11)</td>
<td>(\frac{P \rightarrow a \rightarrow P'}{P \rightarrow Q \rightarrow a \rightarrow Q'})</td>
</tr>
<tr>
<td>(12)</td>
<td>(\frac{P \rightarrow a \rightarrow P'}{P \rightarrow Q \rightarrow a \rightarrow Q'})</td>
</tr>
<tr>
<td>(13)</td>
<td>(\frac{P \rightarrow a \rightarrow P'}{P \rightarrow Q \rightarrow a \rightarrow Q'})</td>
</tr>
<tr>
<td>(14)</td>
<td>(\frac{P \rightarrow a \rightarrow P'}{P \rightarrow Q \rightarrow a \rightarrow Q'})</td>
</tr>
</tbody>
</table>

Let Name be a denumerable set of data types ranged over by \(a, b, \ldots\), Const be a set of program constants ranged over by \(K, K', \ldots\), and Time \(= \mathbb{IN} \cup \infty\) be a set of time intervals ranged over by \(t, t', \ldots\). We assume that \(t \leq \infty\) for any \(t\).

We have chosen natural numbers for the description of time intervals, thus assuming discrete time, because the same happens in our referring language JavaSpaces, where the current time is represented by an integer which is incremented each millisecond; in this way, a leasing period of length \(t\) (where \(t\) is an integer number) starting when the current time is \(c\), will finish at the end of the interval with current time \(c + t\). In our process calculus we do not use any value to represent the current time, but we only consider the passing of time, which is considered as divided into basic discrete intervals. As will be discussed in the following, we model the instant in which an interval finishes, and the subsequent starts, with a transition labelled with \(\sqrt{t}\). More precisely, we use \(P \rightarrow P'\) to state that the term \(P\) becomes \(P'\) due to the fact that the current interval has finished, and the subsequent just started.

Let Conf be the set of the possible configurations defined by the following grammar:

\[
P \rightarrow \langle a \rangle t \mid C \mid P|P
C \rightarrow 0 \mid \text{in}(a).C \mid \text{out}(a, t)?C \mid \text{inp}(a)?C \mid C|C \mid K
\]

We assume that the time interval \(t\) used in the output operation is strictly greater than \(0\). Moreover, we assume that an initial configuration is composed only of programs, i.e., the dataspace is empty. In this way we model the fact that, as happens in JavaSpaces, when a new space is created it is empty.

We assume also that each program constant \(K\) is equipped with a definition \(K = C\); this definition may be also recursive, but we assume only guarded forms of recursion [8].

Programs are represented by terms \(C\) containing the coordination primitives; the dataspace is modeled by representing each shared datum \(a\) with a remaining lifetime of \(t\), with the tuple \(\langle a \rangle t\). A configuration is composed of some programs and some shared data composed in parallel using the composition operator \(|\).

A program can be the terminated program \(0\) (which is usually omitted for the sake of simplicity), a prefix form \(\text{in}(a).C\), an if-then-else form \(\text{if}\eta C\text{then}\text{else}\text{D}\) (where \(\eta\) may be either \(\text{out}(a)\) or \(\text{inp}(a)\)), the parallel composition of subprograms, or a program constant \(K\).

The construct \(\text{out}(a, t)?C\text{\_}\text{D}\) represents, for two main reasons, the main novelty with respect to previous Linda based process calculi: (i) it incorporates the indication of the expected maximal
duration \( t \) for the emitted datum and (ii) it offers two possible continuations \( C \) and \( D \). The two continuations have been introduced in order to model the fact that in JavaSpaces the server of the dataspace may decide to grant no time to the new datum. In this case an exception is raised by the server which communicates to the process the failure of the output operation. We model this exception by activating the second continuation instead of the first one. Another novelty is that each term \( \langle \alpha \rangle_t \) is equipped with the remaining time \( t \).

The \( \text{in} \langle \alpha \rangle \) prefix is used to consume a shared datum with content \( \alpha \), and the guard \( \text{inp} \langle \alpha \rangle \) represents its non-blocking variant; if at least one datum with content \( \alpha \) is actually available for consumption, it is removed and the first continuation is activated; otherwise, the second continuation is chosen.

The semantics of the language is described via a labelled transition system (\( \text{Conf}, \text{Label}, \rightarrow \)) where \( \text{Label} = \{ \tau, \sqrt{\_}\} \cup \{ \alpha, \tau, \neg \alpha \mid \alpha \in \text{Name} \} \) (ranged over by \( \alpha, \beta, \ldots \)) is the set of the possible labels. The labelled transition relation \( \rightarrow \) is the smallest one satisfying the axioms and rules in Table 1. For the sake of simplicity we have omitted the symmetric rules of \((9) - (11)\) and \((14)\).

Axioms \((1)\) and \((2)\) describe the behaviour of the output operation. According to \((1)\), the emitted datum is produced with a remaining lifetime \( t' \) which is less or equal to the indicated maximal duration \( t \), but also strictly greater than 0; otherwise, see \((2)\), if the new datum is not introduced in the dataspace, the second continuation is activated. Axiom \((3)\) indicates that \( \langle \alpha \rangle_t \) is able to give its content to the environment by performing an action labelled with \( \tau \). The corresponding input operation, labelled with the complementary label \( \alpha \), may be performed by an input prefix \( \text{in} \langle \alpha \rangle \) as indicated by axiom \((4)\).

Axioms \((5)\) and \((6)\) describe the effect of the passing of time on the data. If the current lifetime \( t \) of a datum \( \langle \alpha \rangle_t \) is neither zero nor infinite (see \((5)\)), \( t \) is decremented by one; if it is zero (see \((6)\)), the corresponding datum is withdrawn.

Axioms \((7)\) and \((8)\) describe the semantics of \( \text{inp} \langle \alpha \rangle ? \text{P} \perp \text{Q} \): if the required \( \langle \alpha \rangle_t \) is present it can be consumed (see \((7)\)), otherwise, in the case no \( \langle \alpha \rangle_t \) is available, its absence is guessed by performing an action labelled with \( \neg \alpha \) (see \((8)\)).

Rule \((9)\) is the usual locality rule, which is valid only for labels \( \alpha \) different from \( \neg \alpha \) and \( \sqrt{\_} \), while rule \((10)\) is the usual synchronisation rule. Rule \((9)\) has two side conditions; \( \alpha \neq \neg \alpha \) is justified by the fact that an action labelled with \( \neg \alpha \) can be performed only if no \( \langle \alpha \rangle_t \) is available in the data repository, i.e., no action labelled with \( \neg \alpha \) can be performed by the terms in the environment (rule \((11)\)); on the other hand, the side condition \( \alpha \neq \sqrt{\_} \) is related to the way the passing of time is modeled, and it will be discussed in the following.

The rule \((12)\) allows a program constant \( K \), defined by \( K = C \), to perform exactly the same actions permitted to \( C \).

The remaining rules \((13)\) and \((14)\) describe how the structured term \( \text{P} \perp \text{Q} \) behaves according to the passing of time. Before discussing them, we observe that in the calculus we have both terms sensible to the passing of time (e.g., \( \langle \alpha \rangle_t \) with \( t \neq \infty \)) and processes not sensible (e.g., \( \langle \alpha \rangle_{\infty} \)). The sensibility to the passing of time of a process \( P \) may be detected simply by observing whether \( P \) has outgoing transitions labelled with \( \sqrt{\_} \) or not. If two processes sensible to the passing of time are composed in parallel, they synchronize on the execution of their \( \sqrt{\_} \) operation (see \((13)\)); on the other hand, if one process with outgoing \( \sqrt{\_} \) transitions is composed in parallel with a term not sensible to the passing of time, it can perform its transition locally (see \((14)\)).

Note that rules \((11)\) and \((14)\) use negative premises; however, the operational semantics is well defined, because the transition system specification is strictly stratifiable \([7]\), condition that ensures (as proved in \([7]\)) the existence of a unique transition system agreeing with it.

The operational semantics defined in Table 1 models configurations in which the passing of time is global, i.e., it is the same for all the components. This is ensured by the side condition \( \alpha \neq \sqrt{\_} \) of the locality rule \((9)\), which ensures that a process may perform locally its own \( \sqrt{\_} \) transitions only if all the other processes in the environment are not sensible to the passing of time. If we remove this side condition we obtain configurations in which the time may pass locally, as the components may or may not synchronize on the execution of their \( \sqrt{\_} \) transitions.

The calculus \( L_0 \) does not consider distributed leasing; it corresponds to the subcalculus in which the \( \sqrt{\_} \) operation is never used, and where the output operation always succeeds.
mally, it is obtained simply by removing the axioms and rules (2), (5), (6), (13), and (14). The
 calculus $L_o$ differs from $L_o$ only in the modeling of the output operation which is separated in
two independent steps: the emission and the subquent introduction of the new datum in the
dataset. This is achieved by substituting the rule (1) with two new rules:

\[(1') \text{out}(a, t) ? P \rightarrow \langle \langle a \rangle \rangle | P \quad (1'') \langle \langle a \rangle \rangle \rightarrow \langle a \rangle\]

where $\langle \langle a \rangle \rangle$ represents the fact that a new datum has been emitted but it is not yet available; it
will be available only when the transformation indicated by the axiom (1'') will be executed on
it. Note that we do not represent the expiration time of tuples; this because, as happens in $L_o$,
tuples are persistent.

Finally, the two calculi which embed distributed leasing are simply characterized as follows:
$L_{gl}$ models global time by considering the complete calculus, while $L_{lt}$ models local time simply
by removing the side condition $a \neq \sqrt{.}$ of the locality rule (9).

**Notation**

In the following we will only consider computations consisting of reduction steps, i.e., the
internal transitions that a stand-alone agent is able to perform independently of the context. In our
language, we consider as reductions not only the usual transitions labelled with $\tau$, but also the
non-standard labelled with $\neg a$. In fact, $P \Rightarrow_P P'$ indicates that $P$ may become $P'$ if no $\langle a \rangle_i$
is available in the external environment; hence, if $P$ is stand-alone (i.e. without external environ-
ment) it is able to become $P'$. Indeed, these labels have been used only for helping a SOS [10]
formulation of the semantics, but they correspond conceptually to internal steps. Formally, we
define reduction steps as follows:

\[P \rightarrow P' \text{ iff } P \rightarrow_P P' \text{ or } P \rightarrow_0 P' \text{ for some } a\]

We use $P \not\rightarrow$ to denote that $P$ is deadlocked, i.e., there exists no $P'$ such that $P \rightarrow P'$.

An agent $P$ has a terminating computation (denoted by $P \downarrow$) if it can block after a finite
amount of internal steps: $P \rightarrow^* P'$ with $P' \not\rightarrow$. On the other hand, an agent $P$ has an infinite
computation (denoted by $P \uparrow$) if there exists an infinite computation starting from $P$: for each
natural index $i$ there exists $P_i$ such that $P = P_0$ and $P_i \rightarrow P_{i+1}$. Observe that due to the nonde-
termination of our languages the two above conditions are not in general mutually exclusive, i.e.,
given a process $P$ both $P \downarrow$ and $P \uparrow$ may hold.

In the following we will sometime use $\langle a \rangle$ instead of $\langle a \rangle_i$ when $t$ has no importance, and
$\text{out}(a, t).P$ as a shorthand for $\text{out}(a, t)? P . 0$ and $\text{out}(a).P$ for $\text{out}(a, \infty).P$.

We define a structural congruence (denoted by $\equiv$) as the minimal congruence relation satisfying
the usual laws for the parallel composition operator:

\[P \equiv P|0 \quad P|Q \equiv Q|P \quad P|(Q|R) \equiv (P|Q)|R\]

As two structural congruent agents are observationally indistinguishable, in the remainder of the
paper we will reason up to structural congruence.

**3 Encoding Random Access Machines**

In this section we consider the problem of the encoding of Random Access Machines [11], a well
known Turing equivalent formalism, in our calculi.

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

To perform a computation, the inputs are provided in registers $r_1, \ldots, r_m$; if other registers
$r_{m+1}, \ldots, r_n$ are used in the program, they are supposed to contain the value 0 at the beginning
of the computation. The execution of the program begins with the first instruction and continues
by executing the other instructions in sequence, unless a jump instruction is encountered. The execution stops when an instruction number higher than the length of the program is reached; this happens if the program was executing the last instruction of the program and this instruction does not require a jump, or if the current instruction requires a jump to an instruction number not appearing in the program. If the program terminates, the result of the computation is the content of the registers specified as outputs.

In [9] it is shown that the following two instructions are sufficient to model every recursive function:

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

The state of the computation is represented by configurations $(i, c_1, c_2, \ldots, c_n)$ where $i$ indicates that the next instruction to execute is the $i^{th}$ and $c_l$ is the content of the register $r_l$ for $l \in \{1, \ldots, n\}$. Given a program $R$ and a configuration $(i, c_1, c_2, \ldots, c_n)$ we use the notation:

$$(i, c_1, c_2, \ldots, c_n) \rightarrow_R (i', c'_1, c'_2, \ldots, c'_n)$$

to state that after the execution of the $i^{th}$ instruction of the program $R$ with contents of the registers $c_1, \ldots, c_n$, the program counter points to the $i^{th}$ instruction, and the registers will contain $c'_1, \ldots, c'_n$. Moreover, we use:

$$(i, c_1, c_2, \ldots, c_n) \not\rightarrow_R$$

to state that the considered is a terminal configuration, i.e., instruction $i$ is not present in $R$.

Finally, we use $(i, c_1, c_2, \ldots, c_n) \uparrow_R$ to indicate that the configuration $(i, c_1, c_2, \ldots, c_n)$ gives rise to an infinite computation if the intended RAM program is $R$.

The section is now divided in three parts. In the first part, we recall two results that we have proved in [1]: (i) in $L$, it is possible to define an encoding which preserves both termination and divergence, while (ii) in $L_\ell$, it is not possible to define encodings which preserve at least one of the two properties. In the second part, we present an encoding of RAM which preserves only termination, valid for both $L_g$ and $L_{\ell t}$. Finally, in the third part, we show that in $L_{\ell t}$ it is possible to define another encoding which preserves only divergence.

### 3.1 A Termination and Divergence Preserving Encoding

Consider the configuration $(i, c_1, c_2, \ldots, c_n)$ and a corresponding RAM program $R$. We represent the content of each register $r_i$ by $c_i$ occurrences of $\langle r_i \rangle$. To model the program $R$ composed of the sequence of instructions $I_1 \ldots I_k$ with actual instruction $I_i$, we use a program constant $P_i$. If we consider a $\text{Succ}$ instruction on register $r_j$, we simply emit the corresponding datum $\langle r_j \rangle$ and then activate the subsequent program constant $P_{i+1}$. If we consider an instruction $\text{DecJump}(r_j, s)$, we perform an $\text{inp}(r_j)$ operation; if it succeeds, then an instance of $\langle r_j \rangle$ has been withdrawn from the datasource and the subsequent constant $P_{i+1}$ is activated; otherwise the constant $P_s$ is considered. According to this approach we consider the following program constant definitions for each $i \in \{1, \ldots, k\}$:

$$P_i = \text{out}(r_i).P_{i+1}$$

if $I_i = \text{Succ}(r_i)$

$$P_i = \text{inp}(r_i).P_{i+1}\cdot P_s$$

if $I_i = \text{DecJump}(r_j, s)$

We also assume a definition $P_i = 0$ for each $P_i$ appearing in the previous set of definitions with $i \not\in \{1, \ldots, k\}$. This is necessary in order to model the jumps outside the range of the considered instruction which represent the termination of the computation.

The encoding of a RAM program $R$ acting on a configuration $(i, c_1, c_2, \ldots, c_n)$ is then defined as follows:

$$\llbracket (i, c_1, c_2, \ldots, c_n) \rrbracket_R = \prod_{c_1 \text{ times}} P_i \cdot (\langle r_1 \rangle)^{\cdots} \cdot (\langle r_k \rangle)^{\cdots}$$

for $c_1$ times $c_2$ times
As we assume that initial configurations do not contain tuples, here we restrict our interest to the subset of RAMs which begins with all the registers empty. This is not a significant limitation as the registers may be explicitly initialized at the beginning of the computation.

In [1] we have proved that, given a RAM program $R$ starting from the configuration $(1, c_1, c_2, \ldots, c_n)$, it terminates (resp. diverges) if and only if the agent $\left[\{1, c_1, c_2, \ldots, c_n\}\right]_R$ has a terminating (resp. infinite) computation. This is a consequence of the following theorem.

**Theorem 3.1** Given a configuration $(i, c_1, c_2, \ldots, c_n)$ and a RAM program $R$, we have that

$$ (i, c_1, c_2, \ldots, c_n) \rightarrow_R (i', c'_1, c'_2, \ldots, c'_n) $$

if and only if

$$ \left[\{i, c_1, c_2, \ldots, c_n\}\right]_R \rightarrow \left[\{i', c'_1, c'_2, \ldots, c'_n\}\right]_R $$

Another result proved in [1] is that in $L_u$ the existence of a terminating or an infinite computation are decidable properties. This allows to conclude the impossibility to define in this language encodings of RAMs which preserves at least one of the two considered properties.

### 3.2 A Termination Preserving Encoding

The above encoding of RAMs does not work if we move to the calculi $L_{gt}$ and $L_{lt}$ which embed distributed leasing. This is a consequence of the fact that the persistency of data is no more ensured. The problem is that during the computation, one of the terms $(r_j)$ may disappear as effect of the expiration of its leasing period; in this way, the register $r_j$ is decremented even if no $DecJump(r_j, s)$ instruction has been executed.

Here we show that even in the presence of this problem, in $L_{gt}$ and $L_{lt}$ it is still possible to model a significant class of RAMs. However, the proposed encoding preserves only termination.

The subset that we consider comprises those RAMs which start and finish (in the case the computation terminates) with all the registers empty. The restriction to this subset is not significant; indeed, given a RAM program $\bar{R}$ starting from an initial configuration $c$, we can check whether a second configuration $c'$ may be its legal final result by moving to another RAM in the considered subset. This is achieved by extending the program $\bar{R}$ as follows: (i) add instructions at the beginning which initialize the registers; (ii) at the end of the computation check whether the registers correspond to the configuration $c'$ or not, if they correspond then empty them, otherwise start an infinite computation. In this way, the computation terminates (with all the registers empty) if and only if the considered RAM terminates its computation in the second configuration $c'$.

By forcing that RAM computations should start and finish with all the registers empty, we have that a correct complete computation should contain as many increments of registers, as many decrements. The idea inspiring the new encoding is to check whether this condition is satisfied or not, and ensure that the computation does not terminate in the case the condition is not satisfied.

This is achieved by producing a new term $LOOP$ every time an increment is performed; this term has the ability to perform an infinite computation until it synchronizes with another new term $KILL$ which is spawn when an increment is performed. The terms $LOOP$ and $KILL$ are program constants defined as follows:

$$ LOOP = inp(a)^0\_\_\_LOOP $$

$$ KILL = out(a, \infty)^0\_\_\_KILL $$

where $a$ is a name which is different from any other name used in the encoding. The term $LOOP$ tests repeatedly for the absence of data of the kind $a$, while $KILL$ is the unique term in the encoding which has the ability to produce one instance of this datum, thus blocking one corresponding instance of $LOOP$. 

UBLCS-2000-5
In the previous subsection we showed that both in the registers are decremented only as effect of a corresponding (due to the expiration of leasing periods) may happen. In the RAM encodings, this ensures that all the data during each basic interval of time, we ensure that no undesired consumption of data preserving at least termination. Here, we present a new encoding which preserves divergence

3.3 A Divergence Preserving Encoding

The new encoding requires a new definition of the program constants \( P_i \):

\[
P_i = \begin{cases} 
\text{out}(r_j).(\text{LOOP}|P_{i+1}) & \text{if } I_i = \text{ Succ}(r_j) \\
\text{inp}(r_j)'(\text{KILL}|P_{i+1})P_i & \text{if } I_i = \text{ DecJump}(r_j, s) 
\end{cases}
\]

In this new scenario, the encoding of a RAM program \( R \) acting on \((i, c_1, c_2, \ldots, c_n)\) is defined as follows:

\[
[(i, c_1, c_2, \ldots, c_n)]_R = P_i \prod_{c_1}(\langle r_1 \rangle|\text{LOOP}) \cdots \prod_{c_n}(\langle r_n \rangle|\text{LOOP})
\]

where by \( \prod_i P \) we denote the parallel composition of \( l \) instances of the term \( P \). Observe that each instance of \( \langle r_j \rangle \) is equipped with a corresponding term \( \text{LOOP} \). It is easy to see that an encoding of a final configuration with non-empty registers has only infinite computations due to the presence of at least one \( \text{LOOP} \) term. On the other hand, the encoding of a terminal configuration with empty registers is deadlocked.

The proof that the encoding preserves termination is based on two separated theorems. The first show that each computation of the RAM may be simulated by a computation in the encoding.

As we are assuming that terminal RAM configurations have all the registers empty, we have that their corresponding encoding is a terminated process; thus, each terminating RAM computation can be simulated by a finite computation of its encoding.

**Theorem 3.2** Given a configuration \((i, c_1, c_2, \ldots, c_n)\) and a RAM program \( R \), we have that

\[
\begin{align*}
&\text{if } (i, c_1, c_2, \ldots, c_n) \rightarrow_R (i', c'_1, c'_2, \ldots, c'_n) \\
&\text{then } [(i, c_1, c_2, \ldots, c_n)]_R \rightarrow_R [(i', c'_1, c'_2, \ldots, c'_n)]_R
\end{align*}
\]

The second theorem states that given a configuration which can be reached during a computation of an encoding, it may be either a correct configuration (plus the presence of an equal number of \( \text{LOOP} \) and \( \text{KILL} \) terms, or their derivatives) or a configuration which may only diverge. A consequence of this theorem is that if a wrong configuration is reached during the computation of the encoding, the computation may not terminate; in other words, if a RAM encoding has a finite computation, it corresponds to a correct terminating RAM behaviour.

**Theorem 3.3** Given a configuration \((i, c_1, c_2, \ldots, c_n)\) and a RAM program \( R \), we have that if \([(i, c_1, c_2, \ldots, c_n)]_R \rightarrow^* Q \) then there exist \( l, l', m \geq 0 \) such that the configuration \( R \) has one of the following forms:

1. \([(i', c'_1, c'_2, \ldots, c'_n)]_R \prod_i(\text{LOOP}|\text{KILL})\prod_i(\text{LOOP}|a)\) and \((i, c_1, c_2, \ldots, c_n) \rightarrow_R (i', c'_1, c'_2, \ldots, c'_n)\)
2. \(P_i|\prod_{c_1}(\langle r_1 \rangle|\text{LOOP})|\cdots|\prod_{c_n}(\langle r_n \rangle|\text{LOOP})|\prod_{i}(\text{LOOP}|\text{KILL})\prod_i(\text{LOOP}|a)L_{m+1}\text{LOOP}\)

3.3 A Divergence Preserving Encoding

In the previous subsection we showed that both in \( L_{gt} \) and \( L_{lt} \) it is possible to encode RAM preserving at least termination. Here, we present a new encoding which preserves divergence (but not termination) which is valid only under the global time assumption (i.e., only in \( L_{gt} \)).

Also in this case we assume that the initial and final configurations of RAM should all have the registers empty.

The new encoding considers the following property: when a new datum is produced, it has always a lifetime period greater or equal to 1 (see axiom (1)). If we refresh (remove and re-emit) all the data during each basic interval of time, we ensure that no undesired consumption of data (due to the expiration of leasing periods) may happen. In the RAM encodings, this ensures that the registers are decremented only as effect of a corresponding \( \text{DecJump} \) operation and not as a consequence of a lease expiration.
Theorem 3.4

Given a configuration \((i, c_1, c_2, \ldots, c_n)\) and a RAM program \(R\), we have that

\[
\text{if } (i, c_1, c_2, \ldots, c_n) \rightarrow_R (i', c_1', c_2', \ldots, c_n')
\]

then \([[(i, c_1, c_2, \ldots, c_n)]_R \rightarrow [[[i', c_1', c_2', \ldots, c_n']_R\]]_R\]
A consequence of this theorem is that an infinite computation of a RAM encoding simulates a correct diverging computation of the corresponding RAM.

**Theorem 3.5** Given a configuration \((i, c_1, c_2, \ldots, c_n)\) and a RAM program \(R\), we have that if \([i, c_1, c_2, \ldots, c_n]_R \rightarrow^* Q\) then one of the following holds:
1. there exists a process \(Q'\) such that \(Q \rightarrow^* Q'\) and \(Q' = [(i', c'_1, c'_2, \ldots, c'_n)]_R\) where \((i, c_1, c_2, \ldots, c_n) \rightarrow^*_R (i', c'_1, c'_2, \ldots, c'_n)\)
2. the process \(Q\) has no infinite computations.

### 4 Deciding Properties

In this section we prove that divergence (i.e., \(P \uparrow\)) is decidable in \(L_{it}\) and also for a particular subclass of processes in \(L_{it}\) which are uniform with respect to termination or divergence. Informally, a process is uniform if its computations are either all finite or all infinite; formally, \(P\) is uniform in the case \(P \downarrow\) if and only if \(P \not\uparrow\).

In order to prove that divergence is decidable we present two encodings, which preserves divergence, from the considered calculi to Place/Transition nets extended with reset arcs, a formalism in which the existence of an infinite computation is decidable (see, e.g., [5]). Here, we report a definition of this formalism adapted to our purposes.

**Definition 4.1** Given a set \(S\), we denote by \(P_{fin}(S)\) and \(M_{fin}(S)\) the set of the finite sets and multisets on \(S\), respectively; we use \(\oplus\) to denote multiset union. A P/T net with reset arcs is a triple \(N = (S, T, m_0)\) where \(S\) is the set of places, \(T\) is the set of transitions (which are triples \((c, p, r)\) of places \(c\), transitions \(p\), and ordered sets \(r\) of places), \(m_0\) is a finite multiset of places. Finite multisets over the set \(S\) of places are called markings; \(m_0\) is called initial marking. Given a marking \(m\) and a place \(s\), \(m(s)\) denotes the number of occurrences of \(s\) inside \(m\) and we say that the place \(s\) contains \(m(s)\) tokens. A P/T net with reset arcs is finite if both \(S\) and \(T\) are finite.

A transition \(t = (c, p, r)\) is usually written in the form \(c \xrightarrow{t} p\) and \(r\) is omitted when empty. The marking \(c\) is called the preset of \(t\) and represents the tokens to be consumed. The marking \(p\) is called the postset of \(t\) and represents the tokens to be produced. The set of places \(r\) denotes the reset places to empty. The meaning of \(r\) is the following: when the transition fires all the tokens inside a place in \(r\) are removed.

A transition \(t = (c, p, r)\) is enabled at \(m\) if \(c \subseteq m\). The execution of the transition produces the new marking \(m'\) such that \(m'(s) = m(s) - c(s) + p(s)\) if \(s\) is not in \(r\), and \(m'(s) = 0\) otherwise. This is written as \(m \xrightarrow{t} m'\) or simply \(m \rightarrow m'\) when the transition \(t\) is not relevant. We use \(\sigma, \sigma'\) to range over sequences of transitions; the empty sequence is denoted by \(\varepsilon\); let \(\sigma = t_1, \ldots, t_n\), we write \(m \xrightarrow{\sigma} m'\) to mean the firing sequence \(m \xrightarrow{t_1} \cdots \xrightarrow{t_n} m'\). The net \(N = (S, T, m_0)\) has an infinite computation if it has a legal infinite firing sequence. On the other hand, the net has a deadlock if it has a legal firing sequence leading to marking \(m\) such that no transition is enabled at \(m\).

#### 4.1 Deciding Divergence in \(L_{it}\)

In order to show that divergence is decidable we resort to a reset P/T net semantics of \(L_{it}\) which preserves at least divergence. The basic idea underlying the definition of an operational net semantics for a process calculus is to decompose a process \(P\) into a multiset of sequential components, which can be thought of as running in parallel. Each sequential component has a corresponding place in the net, and will be represented by a token in that place. Reductions are represented by transitions which consume and produce multisets of tokens.

In our particular case we deal with different kinds of sequential components: programs of the form \(in(a)\).\(P\) or \(\eta! P.Q\) (where \(\eta\) is either \(out(a, i)\) or \(inp(a)\)), and agents \(\langle a\rangle\).

The way we represent input operations is standard; output operations have two possible behaviours: they may either succeed (a new token is introduced in the place corresponding to the
Figure 1. Modeling the \textit{inp} operation (the line with the cross denotes a reset arc).

<table>
<thead>
<tr>
<th>\textit{inp}(a) &amp; Q \Rightarrow R</th>
<th>{a}</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{dec}(Q)</td>
<td>\textit{dec}(R)</td>
</tr>
</tbody>
</table>

Table 2. Decomposition function.

<table>
<thead>
<tr>
<th>Function</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{dec}(0)</td>
<td>\emptyset</td>
</tr>
<tr>
<td>\textit{dec}(\textit{out}+(a, Q, R))</td>
<td>\textit{out}+(a, t, Q, R) &amp; Q \Rightarrow R \Rightarrow {\textit{out}+(a, t, Q, R) &amp; {a} \Rightarrow \textit{dec}(Q)</td>
</tr>
<tr>
<td>\textit{out}-(a, t, Q, R)</td>
<td>\textit{out}-(a, t, Q, R) &amp; Q \Rightarrow R \Rightarrow \textit{dec}(R)</td>
</tr>
<tr>
<td>\textit{inp}+(a, Q, R)</td>
<td>\textit{inp}+(a, Q, R) &amp; Q \Rightarrow R \Rightarrow {\textit{inp}+(a, Q, R) &amp; {a} \Rightarrow \textit{dec}(Q)</td>
</tr>
<tr>
<td>\textit{inp}-(a, Q, R)</td>
<td>\textit{inp}-(a, Q, R) &amp; Q \Rightarrow R \Rightarrow {\textit{inp}-(a, Q, R) &amp; {a} \Rightarrow \textit{dec}(R)</td>
</tr>
</tbody>
</table>

Table 3. P/T net semantics that preserves divergence.
do not play any role; the decomposition of the other processes produces one token in the corresponding place; the decomposition of a program constant is defined in terms of its corresponding program; and the parallel composition is interpreted as multiset union, i.e, the decomposition of $P|Q$ is $\text{dec}(P) \oplus \text{dec}(Q)$.

The axioms in Table 3 define the possible transitions; let $\mathcal{T}$ contains all the transitions obtained as instances of the axiom schemata presented in Table 3. Axiom $\text{in}(a, Q)$ deals with the execution of the primitives $\text{in}(a)$: a token from place $\langle a \rangle$ is consumed. Axioms $\text{out}^+(a, t, Q, R)$ and $\text{out}^-(a, t, Q, R)$ describe the execution of an $\text{out}(a)$ operation: in the case the operation succeeds a new token in the place $\langle a \rangle$ is introduced and the first continuation is activated, otherwise the second continuation is activated and no token in $\langle a \rangle$ is introduced. Finally, axioms $\text{inp}^+(a, Q, R)$ and $\text{inp}^-(a, Q, R)$ describes the two possible behaviours for the $\text{inp}(a)$ operation; in the first case a token from place $\langle a \rangle$ is consumed and the first continuation is activated, in the second case the second continuation is activated and all the tokens in $\langle a \rangle$ are removed as effect of the presence of the reset arc.

**Definition 4.2** Let $P$ be an agent and $K_1 = P_1, \ldots, K_n = P_n$ the set of the definitions of the related constants. We define the triple $\text{Net}(P) = \langle S, T, m_0 \rangle$, where:

\[
S = \{ Q \mid Q \text{ is a subterm of } P, P_1, \ldots, P_n \text{ of the kind } \text{in}(a).Q, \eta?P.Q \} \cup \\
\{ \langle a \rangle \mid a \text{ is a message name in } P, P_1, \ldots, P_n \} \\
T = \{ c \xrightarrow{T} p \in \mathcal{T} \mid \exists Q \in S \text{ s.t. } Q \in \text{dom}(c) \} \\
m_0 = \text{dec}(P)
\]

Given an agent $P$ the corresponding $\text{Net}(P)$ is a finite P/T net with reset arcs.

The net semantics preserves divergence; this is a consequence of the two following theorems. The first states that an infinite firing sequence available in the net semantics corresponds to an infinite computation of the initial considered process. The computation of the process can be obtained simply by assuming that infinite leasing periods are never granted, and that every time an $\text{inp}(a)$ should fail, we simply wait (locally) for the expiration of the leasing periods of all the tuples $\langle a \rangle$.

**Theorem 4.3** Let $P$ be an agent. We have that if $\text{Net}(P)$ has an infinite computation then $P \parallel$.

The second theorems shows that each infinite computation of a process may be simulated in the corresponding net; the unique computation step which is not directly represented in the net semantics is the consumption of tuples as effect of lease expirations. Nevertheless, even if the tuple remains represented in the net, this does not prevent the subsequent computation steps. The unique critical step could be the failure of an $\text{inp}(a)$ operation; however, this step can be performed in the net as a transition corresponding to the failure of an $\text{inp}(a)$ operation can be fired even if tokens in the place $\langle a \rangle$ are actually available.

**Theorem 4.4** Let $P$ be an agent. We have that if $P \parallel$ then $\text{Net}(P)$ has an infinite computation.

### 4.2 Deciding Divergence of Uniform Processes of $L_{gt}$

The net semantics previously introduced does not work under the global time assumption. This because it assumes, when an $\text{inp}(a)$ operation fails, the execution of a sequence of moves during which all (and only) the leasing periods of the tuples $\langle a \rangle$ expire. Under global time, we may have that also other leasing periods expire; more precisely, also the tuples $\langle b \rangle$ with an associated expiration time shorter or equal to the maximal expiration time of tuples $\langle a \rangle$ should be removed.

In order to have a net semantics which works also under global time, the idea is to reset all the places corresponding to tuples; in other words, we wait for the global expiration of all the leasing periods. The new definition of $\text{Net}(P)$ can be achieved simply by taking the previous definition
and change the rule $\text{inp}-(a,Q,R)$ with the following new one:

$$
\begin{array}{c}
\text{inp}-(a,Q,R) \\
\text{inp}(a)^Q \cdot R \xrightarrow{(b) R \in n(P)} \text{dec}(R)
\end{array}
$$

where $n(P)$ represents the set of the message names appearing either in the process $P$ or in the related constant definitions.

The following theorem states that each infinite firing sequence available in the net semantics, simulate an infinite computation valid for the initial considered process. This computation can be obtained simply by assuming that infinite leasing periods are never granted, and that every time an $\text{inp}(a)$ operation fails we wait for a period longer than the maximal expiration time.

**Theorem 4.5** Let $P$ be an agent. We have that if $\text{Net}(P)$ has an infinite computation then $P \uparrow$.

In order to prove that $P \uparrow$ is decidable for uniform processes, we need a further result related to the presence of deadlocks in the net semantics. Indeed, we have that a computation in the net which leads to a deadlock marking corresponds to a finite computation of the initial process $P$. The intuition behind this result is that each deadlock configuration in the net, represents a deadlocked configuration of the corresponding process. Indeed, a deadlocked configuration in the net models a set of processes which are either terminated or blocked trying to consume an unavailable tuple; this because the unique blocking operation is $\text{in}$.

**Theorem 4.6** Let $P$ be an agent. We have that if $\text{Net}(P)$ has a deadlock then $P \downarrow$.

We are now able to state that for each uniform process $P$, we have that $P \uparrow$ if and only if $\text{Net}(P)$ has an infinite computation. By Theorem 4.5 we have that if $\text{Net}(P)$ has an infinite computation then $P \uparrow$. Consider now that $\text{Net}(P)$ has no infinite computation; this implies that the net has a deadlock, thus by Theorem 4.6 also $P \downarrow$. This implies, by uniformity, that also $P \not\uparrow$.

## 5 Conclusion

In this paper we have studied the expressiveness of an important aspect of coordination languages, such as JavaSpaces, namely distributed leasing. The critical point is that, as explicitly stated in the informal specification of the language [12], when an output operation requires to place in the space a new object with a certain duration, “if the requested time is longer than the space is willing to grant, you will get a lease with reduced time”. This implies that the classic, persistent, (ordered) output operation is no longer available. This suggests why the expressive power should decrease, and indeed we formally proved that this is the case. Moreover, if time is not global, then we loose also the possibility to observe the instant in which a lease expires. And this explains intuitively the second expressive gap formally proved in this paper. Finally, we prove that a language where there is control over the time a datum is introduced in the dataspace, but little control on the time of permanence, is more expressive than a language where there is no control over the time a datum is introduced (unordered output), even if then the datum is persistent.

This paper is one further step towards the proper understanding of the coordination primitives of JavaSpaces that, to the best of our knowledge, have been explored only in a few recent papers: relevant related results are reported in [4, 3], where two of the authors have studied the expressiveness of the notification mechanism in JavaSpaces (a sort of multicast) and in [2] where most of the basic features of JavaSpaces have been formally defined.

## References


