Temporary Data in Shared Dataspace Coordination Languages*

Nadia Busi, Roberto Gorrieri, and Gianluigi Zavattaro

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

Abstract. The shared dataspace metaphor is historically the most prominent representative of the family of coordination models. According to this approach, concurrent processes interact via the production, consumption, and test for presence/absence of data in a common repository. Recently, the problem of the accumulation of outdated and unwanted information in the shared repository has been addressed and it has been solved by introducing non-permanent data, obtained by associating an expiration time to data. In this paper we investigate the impact of the adoption of different notions of non-permanent data on the expressiveness of a Linda based process calculus.

1 Introduction

The rapid evolution of computers and networks is calling for the development of middleware platforms responsible for the management of dynamically reconfigurable federations of devices, where processes cooperate and compete for the use of shared resources. In this scenario one of the most challenging topics is concerned with the coordination of the activities performed by the federated components.

The shared dataspace model has recently received a renewed interest; for example, Sun and IBM have respectively proposed JavaSpaces [8] and TSpaces [9] as middlewares for distributed Java programming. Both these products exploit generative communication: processes communicate through production, consumption and test for presence or absence of data in a shared dataspace; after its insertion in the dataspace, a datum has an independent existence, until it is explicitly withdrawn by a consumer.

The adoption of generative communication to manage the interaction of distributed components causes the following features to come into play: data are shared by a large, unpredictable number of processes, and the entity producing an information does not necessarily coincide with its user. The conjunction of these aspects leads to the unwanted effect of accumulations of outdated information. Such useless information can grow indefinitely, hence compromising

* Work partially supported by Italian Ministry of University - MURST 40% - Progetto TOSCA.

the performance of the whole system. A first solution could consists in leaving to each process the responsibility to take care of the data it produces, i.e., to explicitly free the resources when they are no longer needed. A first drawback of this approach is that a failure of the producer may cause the resources used by the data to be never freed. Moreover, this approach clashes with the basic principle of generative communication, i.e., the independence of a datum from any process, once it has been produced. In this setting, it is not unusual that the lifetime of the datum is longer than the lifetime of its producer, thus making a management of the datum by its producer impossible.

A commonly adopted solution to this problem (see, e.g., the leasing mechanism of JavaSpaces) is based on a notion of temporary data: rather than maintaining a datum until it has been explicitly consumed, the lifetime of the datum is decided by the producer. After this time has been expired, the existence of the datum is no longer granted.

The removal of outdated information can be carried out by a sort of expired-data collector, which is invoked when the workload of the system is low or when storage space needs to be freed for incoming data. We consider two possible implementations of the collector. The first one, called *unordered collection*, removes one of the expired data, whereas the second one, called *ordered collection*, removes one of the data which expired first.

The introduction of an upper bound to the guaranteed lifetime of datum, as well as different implementation policies, have an impact on the expressive power of the coordination primitives of Linda-like languages, which is aim of this paper to investigate.

We start by introducing a traditional Linda-like process calculus (a slight variation of the calculus in [1]) with permanent data, i.e., where data remain available in the dataspace until they are not explicitly consumed by some process.

Then, we move to temporary data by enriching each datum with a lifetime information. When an output operation is performed, the process specifies the time the datum is required to remain available in the dataspace. After the specified time has expired, the datum can be removed from the dataspace. Regarding the technique of removal of expired data, we consider both unordered and ordered collection.

We compare the three approaches described above from an expressiveness viewpoint. More precisely, we investigate decidability of properties, such as the existence of a terminating or a divergent computation, and the ability to provide an encoding of a Random Access Machine [7] that preserves some of these properties.

Regarding permanent data, we recall a deterministic encoding of RAMs, proposed in [1], which preserves the existence of terminating and divergent computations.

The removal policy influences the expressive power of the calculus with temporary data. The unordered collection policy gives rise to the weaker model. We show that it is possible to decide the existence of a divergent computation, by reduction to the same (decidable) property on place/transition nets with reset arcs [3]. On the contrary, the existence of terminating computations remains undecidable, because it is still possible to provide a (nondeterministic) encoding of RAMs which preserves the existence of terminating computations.

The ordered collection policy increases the expressive power of temporary data. In fact, we provide a nondeterministic encoding of RAMs that preserves the existence of divergent computations. Moreover, the termination-preserving encoding presented for unordered collection works also in this case.

As both termination and divergence are undecidable for temporary data with ordered collection, one might wonder if the calculus with ordered collection is equivalent to the one with permanent data. We prove that the former is weaker than the latter by showing the impossibility of providing a deterministic encoding of any RAM which preserves divergence (and termination) in the calculus with ordered collection. This result is achieved by reducing the divergence problem for a superclass of deterministic processes (namely, processes whose computations are either all finite or all infinite) to the divergence problem for place/transition nets with reset arcs.

The paper is organized as follows. In Section 2 we define the calculus with permanent data and we show a deterministic encoding of RAMs, preserving both termination and divergence. Section 3 is devoted to the presentation of the calculus with temporary data and of the two aforementioned collection policies. In Sections 4 and 5 we investigate the expressiveness of temporary data with unordered and ordered collection, respectively. Section 5 reports on related work.

2 Permanent Data

In this section we introduce the calculus with permanent data, and we show that it is Turing equivalent.

Let *Name* be a denumerable set of data ranged over by a, b, ..., and *Const* be a set of program constants ranged over by K, K', ...

Let *Prog*, ranged over by P, Q, ..., be the set of the programs defined by the following grammar:

$$P ::= \mathbf{0} \mid in(a).P \mid out(a).P \mid inp(a)?P\_P \mid P|P \mid K$$

The program $\mathbf{0}$ is the empty process. The program $in(a).P$ requires the consumption of an instance of datum a from the dataspace; after the consumption the continuation P is activated. The program $out(a).P$ produces a new instance of datum a and then behaves like P. The program $inp(a)?P\_Q$ represents a non-blocking version of input; an instance of datum a is required to be consumed, if it is present the datum is removed and the first continuation P is activated, otherwise the second continuation Q is activated.

Programs can be composed in parallel by using the operator $|$. We adopt program constants in order to permit recursive program definition: we assume that each constant is equipped with a definition $K = P$ and, as usual, we admit guarded recursion only [5].

The state of the dataspace is modeled by a multiset of data. An instance of datum a is denoted by $\langle a \rangle$. Formally, we define *DataSpace*, ranged over by DS, DS', ..., as $\mathit{DataSpace} = \mathcal{M}(\{\langle a \rangle \mid a \in \mathit{Name}\})$, where $\mathcal{M}(S)$ denotes the set of all the multisets on S. In the following we use $\oplus$ to denote multiset union, and we usually omit the brackets in the denotation of singletons (e.g., we use $\langle a \rangle$ instead of $\{\langle a \rangle\}$).

Let *Conf*, ranged over by C, C', ..., be the set of the configurations; a configuration is a pair composed of the active processes and the dataspace, i.e., $\mathit{Conf} = \{[P, DS] \mid P \in \mathit{Prog}, DS \in \mathit{DataSpace}\}$.

(1)	$[in(a).P, DS \oplus \langle a \rangle] \longrightarrow [P, DS]$	
(2)	$[out(a).P, DS] \longrightarrow [P, DS \oplus \langle a \rangle]$	
(3)	$[inp(a)?P\_Q, DS \oplus \langle a \rangle] \longrightarrow [P, DS]$	
(4)	$[inp(a)?P\_Q, DS] \longrightarrow [Q, DS]$	if $\langle a \rangle \notin DS$
(5)	$\dfrac{[P, DS] \longrightarrow [P', DS']}{[P \mid Q, DS] \longrightarrow [P' \mid Q, DS']}$	
(6)	$\dfrac{[P, DS] \longrightarrow [P', DS']}{[K, DS] \longrightarrow [P', DS']}$	if $K = P$

Table 1. The operational semantics for permanent data (symmetric rule of (5) omitted).

The semantics is described by a transition system $(\mathit{Conf}, \longrightarrow)$ defined as the minimal relation satisfying the axioms and rules in Table 1. Axiom (1) and (2) describe the execution of $in(a)$ and $out(a)$: in the first case one datum $\langle a \rangle$ is removed from the dataspace, in the second one it is added. The $inp(a)$ operation is described by the axioms (3) and (4); if the required datum is available it is consumed and the first continuation is activated (axiom (3)), otherwise the second continuation is chosen and the space is left unchanged (axiom (4)). Rules (5) and (6) describe the behaviour of local computation and of program constants, respectively (the symmetric rule of (5) is omitted).

A configuration C is *deterministic* if for each D such that $C \longrightarrow^* D$, the reached configuration D has at most one outgoing transition, i.e., for all D', D'', if $D \longrightarrow D'$ and $D \longrightarrow D''$ then $D' = D''$. A configuration C is *terminated* (denoted by $C \not\longrightarrow$) if it has no outgoing transition, i.e., if and only if there exists no C' such that $C \longrightarrow C'$. A configuration C has a terminating computation (denoted by $C\!\downarrow$) if C can block after a finite amount of computation steps, i.e., there exists C' such that $C \longrightarrow^* C$ and $C' \not\longrightarrow$. A configuration C has an infinite computation (denoted by $C\!\uparrow$) if there exists an infinite computation starting from C, i.e., there exist an infinite sequence $C_0, C_1, C_2, \ldots$ such that $C = C_0$ and, for each $i \geq 0$, $C_i \longrightarrow C_{i+1}$. Observe that, because of nondeterminism, the two above conditions are not in general mutually exclusive, i.e., given a program C both $C\downarrow$ and $C\uparrow$ may hold. A configuration C is *uniform w.r.t. termination* (uniform for short) if it satisfies the following: if C has a terminating computation, then all its computations are finite. Formally, C is uniform if and only if $C\downarrow$ implies $C\not\uparrow$.

We define a *structural congruence* for programs (denoted by $\equiv$) as the minimal congruence relation satisfying the usual laws for the parallel composition operator: $P \equiv P|\mathbf{0}, P|Q \equiv Q|P, P|(Q|R) \equiv (P|Q)|R$. Two structurally congruent programs are observationally indistinguishable (they act on the dataspace exactly in the same way); for this reason, in the remainder of the paper, we will make no distinction between $[P, DS]$ and $[P', DS]$ in the case $P \equiv P'$.

Random Access Machines

A Random Access Machine [7], simply RAM in the following, is a computational model composed of a finite set of registers $r_1 \dots r_n$, that can hold arbitrary large natural numbers, and a program $I_1 \dots I_k$, that is a sequence of simple numbered instructions.

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

In [6] it is shown that the following two instructions are sufficient to model every recursive function:

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

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

The (computation) state is represented by $(i, c_1, c_2, \dots, c_n)$, where i indicates the next instruction to execute and c_l is the content of the register r_l for each $l \in \{1, \dots, n\}$. Let R be a program $I_1 \dots I_k$, and $(i, c_1, c_2, \dots, c_n)$ be the corresponding state; we use the notation $(i, c_1, c_2, \dots, c_n) \longrightarrow_R (i', c'_1, c'_2, \dots, c'_n)$ to state that after the execution of the instruction I_i with contents of the registers $c_1, \dots, c_n$, the program counter points to the instruction $I_{i'}$, and the registers contain $c'_1, \dots, c'_n$. Moreover, we use $(i, c_1, c_2, \dots, c_n) \not\longrightarrow_R$ to indicate that $(i, c_1, c_2, \dots, c_n)$ is a terminal state, i.e., $i > k$.

Observe that the computation proceeds deterministically; that is, given a state reached during the computation, the subsequent state, if it exists, is unique. Thus, given a program R and its initial state $(1, 0, 0, \dots, 0)$ the computation either terminates (denoted by $R\downarrow$) or proceeds indefinitely (denoted by $R\uparrow$).

We compare the expressiveness of our calculi by studying the ability to provide encodings that satisfy some basic properties, outlined below. As RAMs are deterministic, the best encoding one may provide consists of a deterministic configuration, which terminates if and only if the RAM terminates. There are however weaker forms of encodings that, though adding some nondeterministic computation to the original behaviour, still permit to observe relevant properties of the encoded RAM.

Let $[\![(i, c_1, c_2, \ldots, c_n)]\!]_R$ be the encoding of the RAM with program R and corresponding state $(i, c_1, c_2, \ldots, c_n)$. We say that the encoding *preserves termination* if, for any RAM program R, $R\!\downarrow$ iff $([\![(1, 0, 0, \ldots, 0)]\!]_R)\!\downarrow$. The encoding *preserves divergence* if, for any RAM program R, $R\!\uparrow$ iff $([\![(1, 0, 0, \ldots, 0)]\!]_R)\!\uparrow$.

A deterministic encoding of RAMs

In this section we recall an encoding of RAMs [1] in the deterministic fragment of the calculus with permanent data.

Consider the state $(i, c_1, c_2, \ldots, c_n)$ with corresponding RAM program R. We represent the content of each register r_l by putting c_l occurrences of $\langle r_l \rangle$ in the dataspace. Suppose that the program R is composed of the sequence of instructions $I_1 \ldots I_k$; we consider k programs $P_1 \ldots P_k$, one for each instruction. The program P_i behaves as follows: if I_i is a *Succ* instruction on register r_j, it simply emits an instance of datum $\langle r_j \rangle$ and then activates the program P_{i+1}; if it is an instruction $DecJump(r_j, s)$, it performs an $inp(r_j)$ operation, if this operation succeeds (i.e., an instance of $\langle r_j \rangle$ has been withdrawn) then the subsequent program is P_{i+1}; otherwise it is P_s. According to this approach we consider the following definitions for each $i \in \{1, \ldots, k\}$:

$$\begin{array}{ll} P_i = out(r_j).P_{i+1} & \text{if } I_i = Succ(r_j) \\ P_i = inp(r_j)?P_{i+1}\_P_s & \text{if } I_i = DecJump(r_j, s) \end{array}$$

We also consider a definition $P_i = \mathbf{0}$ for each $i \notin \{1, \ldots, k\}$ which appears in one of the previous definitions. This is necessary in order to model the termination of the computation occurring when the next instruction to execute has an index outside the range $1, \ldots, k$.

The encoding is then defined as follows:

$$[\![(i, c_1, c_2, \ldots, c_n)]\!]_R \quad = \quad [P_i, \bigoplus_{1 \leq l \leq n} \{\underbrace{\langle r_l \rangle, \ldots, \langle r_l \rangle}_{c_l \text{ times}}\}]$$

The correctness of the encoding is stated by the following theorem.

Theorem 1. *Given a RAM program R and a state $(i, c_1, c_2, \ldots, c_n)$, we have $(i, c_1, c_2, \ldots, c_n) \longrightarrow_R (i', c'_1, c'_2, \ldots, c'_n)$ if and only if $[\![(i, c_1, c_2, \ldots, c_n)]\!]_R \longrightarrow [\![(i', c'_1, c'_2, \ldots, c'_n)]\!]_R$.*

Hence, the encoding preserves both termination and divergence. A consequence of this theorem is that, for the language with permanent data, both termination and divergence are undecidable.

3 Temporary Data

In this section we study temporary data, i.e., data whose presence in the dataspace is granted at least for a minimal period specified by the producer. At the end of this period the data may be removed in order to free space needed, e.g., to store incoming data. We consider two possible policies for the collection of the data to be removed: the first, called *unordered collection*, removes one datum chosen among those with granted lifetime expired, while the second, called *ordered collection*, removes one of the data which expired first, i.e., one of those with oldest expiration time.

In order to model temporary data we need to represent the passing of time. To be as general as possible, we do not fix any specific model of time. We only assume what follows: *Time*, ranged over by t, t', ..., is a generic set of time instants; *Inter*, ranged over by Δt, $\Delta t'$, ..., is a set of time intervals; $\leq$ is a total order on *Time* such that $t \leq t'$ means that the time instant t' follows the instant t; $+ : \mathit{Time} \times \mathit{Inter} \to \mathit{Time}$ is an addition operation such that $t + \Delta t$ is the time instant in which a time interval Δt, starting at time instant t, will finish. We make the minimal reasonable assumption that, for any time instant t and time interval Δt, $t \leq t + \Delta t$; this means that the time instant in which a time interval finishes follows the instant in which it starts.

The syntax of programs under temporary data is defined as for the calculus with permanent data with the unique difference that the output operation requires a further parameter indicating the minimal lifetime of the produced datum; namely, $out(a, \Delta t)$ is substitute for the $out(a)$ prefix. When a datum is produced with minimal lifetime Δt, its expiration time is computed as $t + \Delta t$ where t is the current time.

In the following we use $\langle a \rangle_t$ to denote an instance of datum a with expiration time t; the index t is sometimes omitted when not relevant. Formally, we redefine $\mathit{DataSpace} = \mathcal{M}(\{\langle a \rangle_t \mid a \in \mathit{Name}, t \in \mathit{Time}\})$. The current time is added as a third parameter to configurations, thus we redefine $\mathit{Conf} = \{[P, DS, t] \mid P \in \mathit{Prog}, DS \in \mathit{DataSpace}, t \in \mathit{Time}\}$. In the following, we sometimes omit also the third component t from the denotation of configurations in the case the current time has no relevance.

The operational semantics for the new calculus is defined by the axioms and rules in Table 2: $(1) \ldots (6)$ are simple adaptations of the corresponding axioms and rules in Table 1. The most signicant difference is that whenever a a coordination operation is performed the current time is updated according to the relation $t \leq t'$ specifying the passing of time. Observe that axiom (2) computes the expiration time of the produced datum considering the current time at the moment the datum is effectively introduced in the dataspace, i.e., the time instant t' of the target configuration.

The two new axioms (7_u) and (7_o) define the unordered and ordered collection policies, respectively: (7_u) removes one of the expired data (simply by checking whether its expiration time preceeds the current time) while (7_o) removes one of the objects which expired first (requiring also that the expiration

(1)	$[in(a).P, \langle a\rangle \oplus DS, t] \longrightarrow [P, DS, t']$	if $t \leq t'$
(2)	$[out(a, \Delta t).P, DS, t] \longrightarrow [P, \langle a\rangle_{t'+\Delta t} \oplus DS, t']$	if $t \leq t'$
(3)	$[inp(a)?P\_Q, \langle a\rangle \oplus DS, t] \longrightarrow [P, DS, t']$	if $t \leq t'$
(4)	$[inp(a)?P\_Q, DS, t] \longrightarrow [Q, DS, t']$	if $t \leq t'$ and $\langle a\rangle_t \notin DS$ for any t
(5)	$\dfrac{[P, DS, t] \longrightarrow [P', DS', t']}{[P \mid Q, DS, t] \longrightarrow [P' \mid Q, DS', t']}$	
(6)	$\dfrac{[P, DS, t] \longrightarrow [P', DS', t']}{[K, DS, t] \longrightarrow [P', DS', t']}$	$K = P$
(7_u)	$[P, \langle a\rangle_{t''} \oplus DS, t] \longrightarrow [P, DS, t']$	if $t \leq t'$ and $t'' \leq t'$
(7_o)	$[P, \langle a\rangle_{t''} \oplus DS, t] \longrightarrow [P, DS, t']$	if $t \leq t'$ and $t'' \leq t'$ and $t'' = min\{t \mid \langle a\rangle_t \in DS\}$

Table 2. The operational semantics for temporary data (symmetric rule of (5) omitted).

time is the minimum among those associated to the data currently in the dataspace).

4 Unordered Collection

In this section we consider the new calculus with temporary data under the assumption that the collection policy is unordered. In particular, we prove an expressiveness gap with respect to the calculus with permanent data: first of all we show that the RAM encoding presented in Section 2 no longer works; then we show the existence of an alternative encoding, which preserves termination only; finally, we show that it is not possible to define any divergence preserving encoding because $C\uparrow$ is decidable.

A termination preserving encoding of RAMs

The encoding of RAMs presented in Section 2 is not valid under temporary data, because the tuples representing the content of the registers may disappear; in this way, an undesired decrement of the registers may occurs. For example, the encoding of the terminating RAM program:

$$\begin{array}{l} 1 : inc(r_1) \\ 2 : decJump(r_1, 1) \end{array}$$

may never terminate in the case the datum $\langle r_1\rangle$ always expires and is removed before the second instruction is executed.

We now provide a nondeterministic encoding of RAMs which preserves termination. This encoding checks whether an undesired decrement of a register occurred during a completed finite computation; if this happens, we force the computation to diverge. In this way, if the computation terminates it is a correct computation.

In order to check the occurrence of undesired decrements of registers, we use the fact that the computation of the RAM we consider start and finish with all the registers empty; for this reason a completed RAM computation contains the same number of increments and decrements.

The encoding is modified by producing two particular programs $LOOP$ and $KILL$ every time an increment or a decrement is performed, respectively. $LOOP$ has the ability to perform an infinite computation until it communicates with an instance of the program $KILL$. In this way, if the total number of increments is strictly greater than the total number of decrements then the computation cannot terminate, due to the existence of some $LOOP$ programs unable to synchronize with a corresponding $KILL$ program. For this reason, if the computation terminates we can conclude that no undesired decrements of register occurred thus it is a correct computation. On the other hand, if the computation does not terminate we cannot conclude anything about the corresponding RAM because it could be the case that the computation diverges due to the presence of some $LOOP$ programs. $LOOP$ and $KILL$ are defined as follows:

$$\begin{array}{ll} LOOP &= inp(a)?\mathbf{0}\_LOOP \\ KILL &= out(a, \Delta t).\mathbf{0} \end{array}$$

where Δt can be any time interval. Observe that it could happen that a datum $\langle a \rangle_t$, produced by some $KILL$ program, expires and is removed before being consumed by any corresponding $LOOP$ program. If this happens, the corresponding $LOOP$ program will loop forever, implying that the computation will never terminate. As we are interested in terminating computations only, this does not introduce any undesired behaviours.

The new encoding redefines the program constants P_i by adding the spawning of the $KILL$ and $LOOP$ programs:

$$\begin{array}{ll} P_i = out(r_j, \Delta t).(LOOP|P_{i+1}) & \text{if } I_i = Succ(r_j) \\ P_i = inp(r_j)?(KILL|P_{i+1})\_P_s & \text{if } I_i = DecJump(r_j, s) \end{array}$$

Finally, the encoding of the state $(i, c_1, c_2, \ldots, c_n)$ is defined as follows:

$$[\![(i, c_1, c_2, \ldots, c_n)]\!]_R \quad = \quad [P_i | \prod_{c_1} LOOP | \ldots | \prod_{c_n} LOOP, \bigoplus_{1 \le l \le n} \{\underbrace{\langle r_l \rangle, \ldots, \langle r_l \rangle}_{c_l \text{ times}}\}]$$

where by $\prod_j P$ we denote the parallel composition of j instances of the program P. We do not indicate the expiration time of the data in the dataspace because it is not relevant in order to evaluate the content of the registers of the considered state. Observe that each instance of $\langle r_j \rangle$ is equipped with a corresponding program $LOOP$.

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

Theorem 2. *Given a state* $(i, c_1, c_2, \ldots, c_n)$ *and a RAM program* R*, we have that* $(i, c_1, c_2, \ldots, c_n) \longrightarrow_R (i', c_1', c_2', \ldots, c_n')$ *implies* $[\![(i, c_1, c_2, \ldots, c_n)]\!]_R \longrightarrow^* [\![(i', c_1', c_2', \ldots, c_n')]\!]_R$.

As we are assuming that terminal RAM states have all the registers empty, we have also that the corresponding encoding contains no $LOOP$ programs, thus it is terminated. Due to this observation and the above theorem, we can conclude that a RAM computation leading to a terminal state has a corresponding finite computation of the encoding.

The second theorem states that any partial computation of an encoding can be either extended to reach a correct configuration or it can only admit infinite computations.

Theorem 3. *Given an initial state* $(1, 0, 0, \ldots, 0)$ *and a RAM program* R*, we have that if* $[\![(1, 0, 0, \ldots, 0)]\!]_R \longrightarrow^* C$ *then one of the following holds:*

1. *there exists* C' *such that* $C \longrightarrow^* C'$ *and* $C' = [\![(i, c_1, c_2, \ldots, c_n)]\!]$ *where* $(1, 0, 0, \ldots, 0) \longrightarrow_R^* (i, c_1, c_2, \ldots, c_n)$*;*
2. C *has only infinite computations.*

A consequence of this theorem is that any computation of an encoding leading to a terminated configuration corresponds to a correct RAM computation.

Finally, we can conclude that as the RAM encoding preserves the existence of terminating computations, $C\!\downarrow$ is undecidable also for the calculus with temporary data.

Divergence is decidable

In order to show the impossibility to define a divergence preserving encoding, we prove that $C\!\uparrow$ is decidable. In order to prove this result we resort to a semantics in terms of Place/Transition nets extended with *reset* arcs, a formalism for which the existence of an infinite firing sequence is decidable (see [3]). Here, we report a definition of this formalism adapted to our purposes.

Definition 1. *Given a set* S*, we denote by* $\mathcal{P}_{fin}(S)$ *and* $\mathcal{M}_{fin}(S)$ *the set of the finite sets and multisets on* S*, respectively. 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* $(c, p, r) \in \mathcal{M}_{fin}(S) \times \mathcal{M}_{fin}(S) \times \mathcal{P}_{fin}(S)$ *such that* r *has no intersection with both* c *and* p*), and* m_0 *is a finite multiset of places. Finite multisets over the set* S *of places are called* markings*;* m_0 *is called* initial marking*. Given a marking* m *and a place* s*,* $m(s)$ *denotes the number of occurrences of* s *inside* m *and we say that the place* s *contains* $m(s)$ tokens*. A P/T net with reset arcs is finite if both* S *and* T *are finite.*

$$dec(\mathbf{0}) = \emptyset \qquad dec(in(a).P) = \{in(a).P\}$$
$$dec(out(a, \Delta t).P) = \{out(a).P\} \qquad dec(inp(a)?P\_Q) = \{inp(a)?P\_Q\}$$
$$dec(K) = dec(P) \quad \text{if } K = P \qquad dec(P|Q) = dec(P) \oplus dec(Q)$$
$$dec(\langle a\rangle_t \oplus DS) = \langle a\rangle \oplus dec(DS) \qquad dec(\emptyset) = \emptyset$$

`in(a,Q)`	$in(a).Q \oplus \langle a\rangle \longrightarrow dec(Q)$
`out(a,Q)`	$out(a).Q \longrightarrow \langle a\rangle \oplus dec(Q)$
`dis(a)`	$\langle a\rangle \longrightarrow \emptyset$
`inp+(a,Q,R)`	$inp(a)?Q\_R \oplus \langle a\rangle \longrightarrow dec(Q)$
`inp-(a,Q,R)`	$inp(a)?Q\_R \xrightarrow{\langle a\rangle} dec(R)$

Table 3. Definition of the decomposition function dec and net transitions $\mathcal{T}$.

A transition $t = (c, p, r)$ is usually written in the form $c \xrightarrow{r} p$ 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*. 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 $m \longrightarrow m'$ when the transition t is not relevant. A marking m is* dead *if no transition is enabled at m. The net has a* deadlock *if it has a legal firing sequence leading to a dead marking.*

The basic idea underlying the definition of an operational net semantics for a process calculus is to decompose a term 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* represenfing the programs $in(a).P$, $out(a, \Delta t).P$, or $inp(a)?P\_Q$.

Any datum is represented by a token in a particular place $\langle a\rangle$. The way we represent input and output operations is standard; $in(a)$ removes a token from the place $\langle a\rangle$, while $out(a)$ produces a new token in the same place. In order to model ephemeral data, we connect to each place $\langle a\rangle$ a transition which simply removes a token from the place.

More interesting is the mechanism we adopt to model the execution of an $inp(a)$ operation. The idea is that whenever an $inp(a)?Q\_R$ process moves, the corresponding net may have two possible behaviours: either (i) a token is consumed from place $\langle a \rangle$ and the continuation Q is activated, or (ii) the continuation R is activated and all the tokens in the place $\langle a \rangle$ are removed. This global consumption is achieved by using a reset arc.

The behaviour (i) corresponds to the successful execution of the $inp(a)$ operation, while (ii) corresponds to a sequence of moves: first each available datum $\langle a \rangle$ expires, then they are all removed, and finally the $inp(a)$ operation fails because no $\langle a \rangle$ is available any more.

The axioms in the first part of Table 3, describing the decomposition of programs and dataspaces in corresponding markings, state that the agent $\mathbf{0}$ generates no tokens; a sequential component produces one token in the corresponding place; a program constant is treated as its corresponding program definition; and the parallel composition is interpreted as multiset union, i.e, the decomposition of $P|Q$ is $dec(P) \oplus dec(Q)$. On the other hand, the decomposition of dataspaces is obtained simply by removing the time index from each single datum. Given a configuration $C = [P, DS, t]$ we define $dec(C) = dec(P) \oplus dec(DS)$ the marking containing the representation of both the active processes and the available data. Observe that the current time t of the configuration does not play any role in the definition of the corresponding marking.

The axioms in the second part of Table 3 define the possible transitions denoted by $\mathcal{T}$. Axioms `in(a,Q)` and `out(a,Q)` deal with the execution of the primitives $in(a)$ and $out(a)$, respectively: in the first case a token from place $\langle a \rangle$ is consumed, in the second one it is introduced. Axiom `dis(a)` removes one token from $\langle a \rangle$ in order to model one ephemeral datum $\langle a \rangle$ which disappears. Finally, axioms `inp+(a,Q,R)` and `inp-(a,Q,R)` describes the two possible behaviours for the $inp(a)$ 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 2. *Let $C = [P, DS, t]$ be a configuration such that P has the following related program constant definitions: $K_1 = P_1, \ldots, K_n = P_n$. We define the triple $Net(C) = (S, T, m_0)$, where:*

$$
\begin{aligned}
S &= \{Q \mid Q \textit{ is a sequential component of either } P,\, P_1,\, \ldots,\, P_n\} \cup \\
&\quad\ \{\langle a \rangle \mid a \textit{ is a message name in either } P,\, P_1,\, \ldots,\, P_n, \textit{ or } DS\} \\
T &= \{c \xrightarrow{r} p \in \mathcal{T} \mid \textit{the components and the data in } c \textit{ are also in } S\} \\
m_0 &= dec(C)
\end{aligned}
$$

Given a configuration C the corresponding $Net(C)$ is a finite P/T net with reset arcs.

In order to prove that $C\uparrow$ is decidable, we show that for any configuration C, $Net(C)$ has an infinite computation if and only if $C\uparrow$. This is a consequence of the following theorem based on two sentences. The first states that each computation step of the configuration C is matched by a transition in the corresponding net. The second states a similar symmetric result: each transition firable in the net can be mimicked by a sequence of computation steps of the corresponding configuration. The proof of the second sentence is by cases on the possible transitions; the unique non trivial case is `inp-(a,Q,R)` where we assume that all the data $\langle a \rangle$ expire and are removed in the configuration before the corresponding program $inp(a)?Q\_R$ performs its $inp(a)$ operation.

Theorem 4. *Consider the $Net(D)$ for some configuration D. Let m be a marking of the net such that $m = dec(C)$ for some configuration C.*

1. *If $C \longrightarrow C'$ then $m \longrightarrow dec(C')$ in $Net(D)$.*
2. *If $m \longrightarrow m'$ then $C \longrightarrow^+ C'$ with $dec(C') = m'$.*

5 Ordered Collection

Here we prove the main results regarding the ordered collection policy: (i) $C\uparrow$ is no more decidable because a divergence preserving encoding of RAM exists, and (ii) there exists no encoding of RAM which preserves both termination and divergence.

A divergence preserving encoding of RAMs

In the previous section we have discussed that it is not possible to define a divergence preserving encoding of RAMs when using temporary data under the unordered collection policy. The reason is that it is not possible, in the case of infinite computation, to check whether an undesired decrement of register occurs during the computation. Here we show that moving to ordered collection this check becomes possible.

The encoding presented in this section is based on the following idea. Each datum $\langle r_j \rangle$ is produced with an associated granted lifetime Δt. The granted lifetime is renewed before the execution of each instruction. The renewal is realized by removing and reproducing each datum. Between two subsequent renewals we check whether some of the data expire and are removed by expoiting the followign technique: before the first renewal we produce a special datum with a granted lifetime $\Delta t'$ shorter than Δt, then we check if this datum is still present at the end of the second one. In the case this special datum is still available, this means that the data emitted during the first renewal surely did not disappear before the second one (otherwise also the special datum should be removed as it should expire first) On the other hand, if the special datum expires and is removed, we cannot conclude any more that the computation is valid; in this case we block the computation. This forced termination is not a limitation of the encoding as we are interested in preserving the infinite computations only.

Concerning the renewal procedure, we have to divide it in two phases: first we rename each $\langle r_j \rangle$ in $\langle s_j \rangle$, and then each $\langle s_j \rangle$ in $\langle r_j \rangle$. In order to do this we need also two different special data $\langle a \rangle$ and $\langle b \rangle$.

The refreshing procedure is embedded in each program constant definition P_i. The new definitions are parametric in a program constant Q_i representing the effective part of the computation:

$$\begin{array}{ll} Q_i = out(r_j, \Delta t).P_{i+1} & \text{if } I_i = Succ(r_j) \\ Q_i = inp(r_j)?P_{i+1}\_P_s & \text{if } I_i = DecJump(r_j, s) \end{array}$$

We are now able to define the program constant P_i:

$$\begin{array}{lll}
P_i & = out(b, \Delta t').R_1\,toS_1 & \\
R_k\,toS_k & = inp(r_k)?(out(s_k, \Delta t).R_k\,toS_k)_R_{k+1}\,toS_{k+1} & \text{for } k \in \{1, \ldots, n-1\} \\
R_n\,toS_n & = inp(r_n)?(out(s_n, \Delta t).R_n\,toS_n)_(in(a).out(a, \Delta t').S_1\,toR_1) & \\
S_k\,toR_k & = inp(s_k)?(out(r_k, \Delta t).S_k\,toR_k)_S_{k+1}\,toR_{k+1} & \text{for } k \in \{1, \ldots, n-1\} \\
S_n\,toR_n & = inp(s_n)?(out(r_n, \Delta t).S_n\,toR_n)_(in(b).Q_i) &
\end{array}$$

where Δt and $\Delta t'$ are two generic time intervals such that for any time instant t we have $t + \Delta t' \leq t + \Delta t$.

We assume that before the execution of the program P_i the special datum $\langle a \rangle$ has been already emitted. In order to ensure this, we start the computation with the program $out(a, \Delta t').P_1$ instead of P_1 only.

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

$$[\![(i, c_1, c_2, \ldots, c_n)]\!]_R \quad = \quad [P_i, \langle a \rangle \oplus \bigoplus_{1 \leq l \leq n} \{\underbrace{\langle r_l \rangle, \ldots, \langle r_l \rangle}_{c_l \text{ times}}\}]$$

The proof that the encoding preserves the existence of infinite computations is based on two separated theorems. Also in this case, the first shows that each computation step of the RAM may be simulated by a sequence of steps of the encoding.

Theorem 5. *Given a state $(i, c_1, c_2, \ldots, c_n)$ and a RAM program R, we have that $(i, c_1, c_2, \ldots, c_n) \longrightarrow_R (i', c_1', c_2', \ldots, c_n')$ implies $[\![(i, c_1, c_2, \ldots, c_n)]\!]_R \longrightarrow^+ [\![(i', c_1', c_2', \ldots, c_n')]\!]_R$.*

The second theorem has to deal with all those intermediary configurations related to the renewal procedure. Nevertheless, we show that given a configuration reachable during a computation of the encoding, we have that either it is possible to extend the computation in order to reach the encoding of a correct RAM configuration, or the computation cannot be infinite.

Theorem 6. *Given an initial configuration $(1, 0, 0, \ldots, 0)$ and a RAM program R, we have that if $[\![(1, 0, 0, \ldots, 0)]\!]_R \longrightarrow^* C$ then one of the following holds:*

1. *there exists a conf. C' such that $C \longrightarrow^* C'$ and $C' = [\![(i, c_1, c_2, \ldots, c_n)]\!]_R$ where $(1, 0, 0, \ldots, 0) \longrightarrow_R^* (i, c_1, c_2, \ldots, c_n)$;*
2. *the configuration C has no infinite computations.*

Divergence is decidable (for uniform configurations)

In order to prove the impossibility to define a RAM encoding preserving both termination and divergence, we proceed by contraposition. We first assume the existence of such an encoding; then we show that for each RAM the corresponding encoding should be a configuration uniform with respect to termination;

finally, we prove that divergence is decidable for uniform configurations. This implies that the divergence of RAM is decidable (contradiction).

Suppose there exists a divergence and termination preserving encoding of RAMs: if the RAM terminates then its encoding has only finite computations, while if the RAM diverges then its encoding has only infinite computations. Hence, such a RAM encoding is uniform.

We resort to P/T nets to prove the decidability of divergence for uniform configurations. The P/T net semantics defined for the unordered collection is not satisfactory under the ordered policy. This because when a transition `inp-(a,Q,R)` fires, the connected reset arc removes all the tokens from the place $\langle a \rangle$ only. However, it could be the case that also other data expire before the some of the data $\langle a \rangle$; due to the ordered collection policy, also these data should be removed.

In order to overcome this limitation we extend the net with new reset arcs connecting each transition `inp-(a,Q,R)` with all the places representing data. In this way, when an `inp-(a,Q,R)` transition fires, all the data are removed. This behaviour corresponds to a sequence of computation steps in which all the data initially expire, then they are globally removed, and finally the considered $inp(a)$ fails.

Formally, given a configuration C and the previously defined $Net(C) = (S', T', m_0')$, we define the new net semantics $Nnet = (S, T, m_0)$ as follows:

$$\begin{aligned} S &= S' \\ T &= T' \setminus \{t \mid t \text{ is one of the } \texttt{inp-(a,Q,R)} \text{ transitions}\} \cup \\ &\quad\ \{t = c \stackrel{\{\langle b \rangle | \langle b \rangle \in S\}}{\longrightarrow} p \mid c \stackrel{\{\langle a \rangle\}}{\longrightarrow} p \in T' \text{ for some } a\} \\ m_0 &= m_0' \end{aligned}$$

here $\setminus$ denotes set difference.

The new net semantics has the following properties.

Theorem 7. *Let C be a configuration. We have that:*

1. *if $Nnet(C)$ has an infinite firing sequence then $C\uparrow$;*
2. *if $Nnet(C)$ has a deadlock then $C\downarrow$.*

The first sentence is the consequence of the fact that each transition in the net can be mimicked by the corresponding configuration. The unique non-trivial case is the new kind of transitions related to the $inp(a)$ operation which removes all the tokens from the places representing data: as described above this transition is simulated by a sequence of computation steps of the configuration.

The second sentence states that a computation in the net which leads to a dead marking has a corresponding finite computation of the considered configuration. The intuition behind this result is that each dead marking in the net represents has a corresponding terminated configuration. Indeed, a dead marking in the net models a configuration composed by a set of programs which are either terminated or blocked, because they are trying to execute an *in* operation on one datum which is not actually available. This kind of configuration is trivially terminated.

We conclude by showing that given a uniform configuration C we have that the net $Nnet(C)$ has an infinite firing sequence *if and only if* $C\uparrow$. The *only if* part, i.e., if $Nnet(C)$ has an infinite firing sequence then $C\uparrow$, has been proved as first sentence of the Theorem. The *if* part, i.e., if $C\uparrow$ then $Nnet(C)$ has an infinite firing sequence, is proved by contraposition: suppose that $C\uparrow$ and $Nnet(C)$ has no infinite firing sequence; this implies that the net has a deadlock, thus (by the second sentence of the Theorem) also $C\downarrow$; by uniformity of C we have $C\not\uparrow$ (contradiction).

6 Related Work

Recently, two proposals dealing with time in Linda-like languages appeared in the literature. Although both of them deal with timeout primitives, and not with temporary data, we briefly compare their models of time with our one. In [2] the passing of time is explicitely represented by means of transitions, and each action is assumed to take a unit of time to be performed. We claim that our results continue to hold also in this approach to time modeling. In [4] the two-phase functioning approach, tipical of synchronous languages, is adopted: in the first phase all actions are performed; when no further action can be performed, the second phase, consisting in a progress of time, takes place. If we follow this approach, our RAM encoding for permanent data also works in the case of temporary data.

References

1. N. Busi, R. Gorrieri, and G. Zavattaro. On the Expressiveness of Linda Coordination Primitives. *Information and Computation*, 156(1/2):90–121, 2000.
2. F.S. de Boer, M. Gabbrielli, and M.C. Meo. A Timed Linda Language. In *Proc. of Coordination2000*, volume 1906 of *Lecture Notes in Computer Science*, pages 299–304. Springer-Verlag, Berlin, 2000.
3. C. Dufourd, A. Finkel, and P. Schnoebelen. Reset nets between decidability and undecidability. In *Proc. of ICALP'98*, volume 1061 of *Lecture Notes in Computer Science*, pages 103–115. Springer-Verlag, Berlin, 1998.
4. J.M. Jacquet, K. De Bosschere, and A. Brogi. On Timed Coordination Languages. In *Proc. of Coordination2000*, volume 1906 of *Lecture Notes in Computer Science*, pages 81–98. Springer-Verlag, Berlin, 2000.
5. R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.
6. M.L. Minsky. *Computation: finite and infinite machines.* Prentice-Hall, 1967.
7. J.C. Shepherdson and J.E. Sturgis. Computability of recursive functions. *Journal of the ACM*, 10:217–255, 1963.
8. Sun Microsystem, Inc. *JavaSpaces Specifications*, 1998.
9. P. Wyckoff, S.W. McLaughry, T.J. Lehman, and D.A. Ford. T Spaces. *IBM Systems Journal*, 37(3), 1998.