Expired Data Collection in Shared DataSpaces [1]

Nadia Busi and Gianluigi Zavattaro [2]

Dipartimento di Scienze dell'Informazione, Università di Bologna, Mura Anteo Zamboni 7, I-40127 Bologna, Italy

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 Recently, the problem of the accumulation of outdated and unwanted information in the shared repository has been addressed. Typical garbage collection algorithms cannot be adopted in this context because there is no notion of unaccessible data. The most promising solution to this problem consists of the introduction of the notion of temporary data, intended as data with an associated expiration time. In this paper, we investigate the impact of different mechanisms for expired data collection on the expressiveness of shared dataspace coordination systems with temporary data.

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 coordination model, introduced for the first time by the Linda coordination language [10], has recently received a renewed interest; for example, Sun and IBM have respectively proposed JavaSpaces [20] and

[1] Revised and extended version of [6]

[2] Correspoding author. E-mail: zavattar@cs.unibo.it.

TSpaces [21] 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a shared dataspace; after its insertion in the dataspace, a datum has an independent existence, until it is explicitly withdrawn by a consumer.

These coordination technologies, in particular JavaSpaces and its underlying architecture Jini [19], have been designed to manage the interaction of dynamically reconfigurable confederations of components in *open distributed systems*. The term *distributed* means that the components may be distributed on different nodes of a network; *open* means that new components (as also new nodes) may be added or removed from the network during the lifetime of the system in an unpredictable manner. In this scenario the following features 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 accumulation of outdated information. Such useless information can grow indefinitely, hence compromising the performance of the whole system.

A first solution could consist 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.

Typical garbage collection mechanisms cannot be applied in this context because there is no notion of unaccessible data. Indeed, in an open environment it is not possible to make any assumption on the processes that will engage the system in the future; thus, it is not possible to check whether a shared datum, available at a certain time of the computation, will be subsequently accessed.

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.

As an example of temporary data, consider a newsgroup application, where a dataspace is used to hold the posted messages. When posting a message, a user may know that after a period of time the message is no longer important.

For example, the relevance of a conference call for paper strongly decreases after the submission deadline, although it may remain of some interest. In this case, the user specifies the time left until the submission deadline as required duration for the message. When this time expires, the message can be cancelled, for example to provide space for new messages.

This paper investigates the impact of temporary data in shared dataspace coordination languages. The loss of data persistency clearly decreases the expressive power (as it happens, e.g., when moving from reliable to unreliable/lossy channels [1]). Besides this observation, we concentrate on a more subtle phenomenon, that is, the impact of the choice of the *expired data collection* policy.

The removal of outdated information, indeed, can be carried out by an 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. Different policies may be considered in order to select the data to be removed when the expired data collector is activated. Among many possible policies, we concentrate on two of them which simply differ in the way the order of expiration of the data is taken into account. The former, called *unordered collection*, removes one of the expired data (independently of when they expired), whereas the latter, called *ordered collection*, removes the datum which expired first (thus respecting the order of expiration).

Our thesis is that this subtle difference between the unordered and the ordered collection policy may have a strong impact on the expressiveness of the overall shared dataspace coordination language. The remainder of the Introduction is devoted to describe the framework we use in order to support our thesis, plus a description of the structure of the paper.

1.1 The Framework

Our aim is to investigate the impact of temporary data in shared dataspace coordination languages, and we would like to prove results which apply to the entire family of these languages. There exist several languages based on the notion of shared dataspace (see, e.g., Linda [10], JavaSpaces [20], and TSpaces [21]) which differ (i) in the computational language in which the coordination primitives are embedded (e.g., C, Fortran, or Java) and (ii) in the structure of the data introduced in the shared repository (e.g., tuples or Java objects). For this reason, we abstract away from both the computational language and the structure of the data.

To this aim, we take as a starting point a process calculus (borrowed from a previous work [4]) which comprises only the coordination primitives, plus a sequential and a parallel composition operator necessary in order to describe the sequentialization of the coordination operations and the parallel execution of concurrent processes, respectively. Moreover, the data that can be produced, consumed, and tested for presence/absence are simply names without any structure.

Data are permanent in this initial process calculus. Then, the calculus is modified in order to deal with temporary data: we simply (i) enrich the output operation with the indication of the lifetime of the datum and (ii) associate to each datum the corresponding expiration time. When a datum is produced, its expiration time is computed by adding its lifetime to the current time.

The introduction of temporary data requires to model the elapsing of time in the process calculus. In the literature, a huge variety of timed calculi can be found (see, e.g, [2] for a survey). These calculi have different features inspired by different classes of systems. Just to mention two typical approaches, *two-phase functioning* has been used in the setting of synchronous systems such as reactive systems or electronic circuits (see, e.g., [16]), whilst the so-called *lazy* approach is particularly suited to model distributed systems where processes interact remotely (see, e.g., [18,3]).

Our modeling of time has been developed having in mind the typical features of shared dataspaces in open distributed systems. In particular, we consider the fact that, in this class of systems, a process may run on a different node with respect to the dataspace server (or dataspace servers in the case of a distributed implementation of the dataspace). For this reason, no assumption can be made about the time required by a coordination operation to be performed.

As an example, consider a process which simply produces a datum a, with an associated lifetime δt, and subsequently performs an input operation on that datum. This process is denoted in our process calculus by $out(a, \delta t).in(a)$. Assume that the $out(a, \delta t)$ operation is performed at time t, while the request for the execution of the $in(a)$ operation is taken into account by the dataspace server only at a subsequent time instant t'. Due to the previous observation, no assumption can be made on the delay between t and t'. This delay may be even longer than δt. In this case, it could happen that the $in(a)$ operation is not performed because the datum a has been already expired and collected.

Inspired by this observation, we model time following the so-called *lazy* approach, according to which actions may be delayed indefinitely. This delay reflects the unpredictability of the latency in distributed systems. To be more precise, we model the functional and the temporal behaviour with two different kinds of transition, and we do not impose any pre-emption, priority, or urgency of one kind of transition with respect to the other one.

The lazy approach has been already adopted in the literature for different purposes. For example, in [14] the lazy approach is useful to provide the timed process calculus of [15] with a suitable behavioural semantics, in [18] it is adopted to model processes which interact remotely, whilst in [3] it is used to represent timers in distributed systems.

We present two variants of the process calculus with temporary data: one with the unordered collection policy and another one with the ordered collection policy. In this way we obtain three calculi: the first one with permanent data, the second one with temporary data and unordered collection, and the third one with temporary data and ordered collection.

In order to compare permanent with temporary data and, more subtly, to contrast different expired data collection policies, we exploit tools borrowed from the classical theory of computation. In particular, we consider the possibility to provide the three process calculi with an encoding of Random Access Machines (RAMs) [17], a well-known Turing powerful formalism.

RAMs are deterministic; for this reason, the best encoding one may provide consists of terms of the calculi, with a deterministic behaviour, which simulate exactly the corresponding RAMs. We refer to an encoding satisfying this property as a *deterministic encoding*. However, as the process calculi are nondeterministic (that is, there exist terms which may give rise to different computations) there are weaker forms of encodings that, though adding some nondeterministic computation to the original behaviour, still permit to preserve relevant properties of the encoded RAM. We refer to this weaker form of encodings as *nondeterministic encodings*.

The properties we consider for nondeterministic encodings are the *preservation of termination* and the *preservation of divergence*. We say that an encoding preserves termination (resp. divergence) if each target of the encoding has a terminating (resp. infinite) computation if and only if the corresponding RAM terminates (resp. diverges). Clearly, a deterministic encoding preserves both termination and divergence.

The results we prove can be summarized as follows:

- For the calculus with permanent data we provide a deterministic encoding of RAMs. [3] This allows us to conclude that the existence of a terminating computation, as well as the existence of an infinite computation, is an undecidable property.
- For the calculus with temporary data and unordered collection, it is impossible to provide a deterministic encoding of RAMs. This is proved by showing that the existence of an infinite computation turns to be a decidable property. On the other hand, the existence of a terminating computation is still

[3] This encoding is not a new contribution as it is a simple adaptation of a similar encoding presented in [4].

an undecidable property, and this is proved by providing a nondeterministic encoding of RAMs which preserves only termination.
- Also for the calculus with ordered collection it is impossible to provide a deterministic encoding of RAMs. However, differently from the calculus with unordered collection, the existence of an infinite computation turns to be an undecidable property. This is proved by providing a nondeterministic encoding of RAMs which preserves only divergence. In this case, the impossibility to provide a deterministic encoding of RAMs is proved by showing that divergence is decidable for the subset of configurations of the calculus whose computations are either all terminating or all infinite.

Summarizing, these results allows us to conclude that (i) temporary data are not expressive enough to provide a deterministic encoding of permanent data, independently of the adopted expired data collection policy, and that (ii) the ordered collection policy cannot be encoded in the unordered one in a divergence preserving way.

Despite the fact that our results are proved on minimal process calculi, they have also an impact on richer coordination languages, with primitives possibly embedded in Turing powerful computational languages. We can interpret result (i) (resp. (ii)) as the necessity for an encoding of permanent in temporary data (resp. of the ordered in the unordered collection policy) to exploit the specific computational features of the considered Turing powerful language. Thus, it is impossible to provide an encoding which is independent from the host computational language. This contrasts with our interest in results which apply to the entire family of shared dataspace coordination languages, as well as with the idea which inspired the introduction of coordination models and languages, i.e., the clear separation between coordination and computation concerns [9].

1.2 Structure of the Paper

The remainder of 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 some concluding remarks.

2 Permanent Data

In this section we introduce the calculus with permanent data, and we show that RAMs can be faithfully modeled in this calculus.

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

Let *Prog*, ranged over by P, Q, $\ldots$, 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 [12].

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', $\ldots$, as $DataSpace = \mathcal{M}(\{\langle a \rangle \mid a \in Name\})$, 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', $\ldots$, be the set of the configurations; a configuration is a pair composed of the active program and the dataspace, i.e., $Conf = \{[P, DS] \mid P \in Prog, DS \in DataSpace\}$.

The semantics is described by a transition system $(Conf, \longrightarrow)$ 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

(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)	$\dfrac{[P, DS] \longrightarrow [P', DS']}{[P\|Q, DS] \longrightarrow [P'\|Q, DS']}$		
(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).

(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 exists 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 configuration C both $C\!\downarrow$ and $C\!\uparrow$ may hold.

We define a *structural congruence* for programs (denoted by $\equiv$) as the minimal congruence relation satisfying the usual laws for the parallel composition operator: $P \equiv P|\mathbf{0}, P|Q \equiv Q|P, P|(Q|R) \equiv (P|Q)|R$; and for program constant definition: $P \equiv K$ if $K = P$. 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 $P \equiv P'$.

A Random Access Machine [17], 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 [13] 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 assume, without loss of generality, that RAMs start and finish their computation (in the case they terminate) with all the registers empty.

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 $R\!\uparrow$).

As stated in the Introduction, we compare the expressiveness of our calculi by studying the ability to provide encodings of RAMs.

Definition 2.1 *Let $[\![(i, c_1, c_2, \dots, c_n)]\!]_R$ be the encoding of the RAM with program R and corresponding state $(i, c_1, c_2, \dots, c_n)$. We say that the encoding* preserves termination *if, for any RAM program R, $R\!\downarrow$ iff $([\![(1, 0, 0, \dots, 0)]\!]_R)\!\downarrow$. We say that the encoding* preserves divergence *if, for any RAM program R, $R\!\uparrow$ iff $([\![(1, 0, 0, \dots, 0)]\!]_R)\!\uparrow$.*

As termination and divergence are undecidable properties for RAMs, we have that if it is possible to encode RAMs into a calculus such that termination (resp. divergence) is preserved, then termination (resp. divergence) is an undecidable property for the target calculus, too.

2.2 A deterministic encoding of RAMs

In this section we adapt an encoding of RAMs, formerly presented in [4], in the deterministic fragment of the calculus with permanent data.

Consider the state $(i, c_1, c_2, \dots, c_n)$ 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 $I_1 \dots I_k$; we consider k programs $P_1 \dots P_k$, 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 $i \in \{1, \dots, k\}$:

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

We also consider a definition $P_i = \mathbf{0}$ for each $i \notin \{1, \dots, k\}$ 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 $1, \dots, k$.

The encoding is then defined as follows:

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

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

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

As a corollary of this theorem, we have that the encoding preserves both termination and divergence.

Corollary 2.2 *Given a RAM program R, we have that $R\downarrow$ if and only if $[\![(1,0,0,\ldots,0)]\!]_R\downarrow$, and $R\uparrow$ if and only if $[\![(1,0,0,\ldots,0)]\!]_R\uparrow$.*

Finally, a consequence of this lemma is that, for the language with permanent data, both termination and divergence are undecidable.

3 Temporary Data

In this section we modify the process calculus in order to model temporary data under both the *unordered* and the *ordered* collection policies.

In order to model temporary data, we need to represent the elapsing of time. In the Introduction, we have justified the choice of the *lazy* approach. We have also described that we model the functional and the temporal behaviour independently. More precisely, transitions are decorated with 2 different labels: σ for transitions modeling the execution of functional steps, and $\surd$ for transitions representing temporal steps, that is, transition which do not alter the programs and the dataspace, but which increment the current time, only.

We use natural numbers $I\!N$ to represent time instants as also time intervals. [4] In the following, we use $I\!N_+$ to denote the set of natural numbers strictly greater then 0. We use t, t', ..., to denote the current time of a configuration, and we use δt, $\delta t'$, ..., to denote time intervals representing the minimum lifetime of a datum, or the time required for an operation to be performed.

To be as general as possible, we parameterize our calculus with respect to a set and a function which describe the time corresponding to the execution of a functional and a temporal step, respectively. These time intervals may depend on the current programs and the state of the dataspace. More precisely, we assume what follows:

- $\Delta_\sigma \subseteq I\!N$ is a set of time intervals indicating the time needed for executing a functional step. We only assume that at least one possible time interval is defined, i.e., $\Delta_\sigma \neq \emptyset$.
- $\Delta_\surd : \mathit{Prog} \times \mathit{DataSpace} \to \mathcal{P}(I\!N_+)$ is a function which associates to each pair, composed of a program P and a dataspace DS, a set of time intervals all different from 0. These intervals represent the amount of time corresponding

[4] As we will discuss in the following, the results we prove hold also if we consider dense or continuous time (instead of discrete time) at the price of taking the non-Zeno assumption, that is, it is impossible to put infinitely many time increments in a finite time interval.

to the execution of a $\surd$ transition. In this case, we simply assume that if $DS \neq \emptyset$ then $\Delta_{\surd}(P, DS) \neq \emptyset$ for any P. This means that, if there exists at least one term sensible to the passing of time (i.e., at least one temporary datum), time may elapse independently of the currently active programs.

Observe that Δ_σ is a fixed set and not a function depending on, e.g., (i) the coordination operation which is actually executed or (ii) the current state of the dataspace. In the case (i), for instance, it could be possible to model the fact that an $in(a)$ operation is more time consuming than a $out(a)$ operation because it requires to verify the presence of datum a in the current state of the dataspace, while an *out* operation can be always executed whatever is the current state; (ii) could permit to model the fact that a test for absence operation could be much more time consuming in a huge dataspace than in a small one. In spite of this observation, for the sake of simplicity we have decided to consider Δ_σ as a fixed set because, as it will be clear in the following, this simplification does not limit our capability to instantiate in our calculus significant models of time. Hence, this choice does not limit the relevance of our results.

We are now ready to present the syntax of programs for the new calculus. The unique difference with respect to the calculus with permanent data is that we add a parameter to the output operation which represents the minimal lifetime of the datum.

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

Also in this case we use *Prog*, ranged over by P, Q, ..., to denote the set of programs.

As far as the modeling of the dataspace is concerned, we add the indication of the expiration time to the data inside the repository: more precisely, 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

$$DataSpace \;=\; \mathcal{M}(\{\langle a \rangle_t \mid a \in Name, t \in I\!N\})$$

The configurations of the new calculus should comprise also the indication of the current time. To this aim, we add a third component to configurations:

$$Conf \;=\; \{[P, DS, t] \mid P \in Prog, DS \in DataSpace, t \in I\!N\}$$

In the following, we sometimes omit also this third component t in the case the current time has no relevance.

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

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

The operational semantics for the new calculus is defined by the labelled transition system $(Conf, Label, \longrightarrow)$, where $Label = \{\sigma, \surd\}$, and $\longrightarrow$ is the least labelled transition relation satisfying the axioms and rules (1)–(6) and (8) in Table 2, plus either axiom (7_u), for the variant of the calculus with unordered collection, or axiom (7_o), for the variant of the calculus with ordered collection.

As stated above, the two labels σ and $\surd$ are used to distinguish the functional from the temporal transitions.

The axioms and rules (1)–(6) are simple adaptations of the corresponding axioms and rules in Table 1. The unique difference is that, whenever a coordination operation is performed, the current time is incremented according to what indicated by Δ_σ.

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 $t + \delta t$.

The two new axioms (7_u) and (7_o) define the unordered and the ordered collection policies, respectively: (7_u) removes one of the expired data (simply by checking whether its expiration time precedes the current time) while (7_o) removes one of the data which expired first (requiring also that the expiration time is the minimum among those associated to the data currently in the dataspace).

The last axiom (8) models temporal transitions: given $[P, DS, t]$, the current time t can be incremented by one of the intervals δt belonging to the set $\Delta_{\surd}(P, DS)$. Note that the increment depends on the current program P and dataspace DS: this is an important feature which allows us to instantiate in our process calculus different models of time.

3.1 Instantiating models of time

Our model of time is parametric with respect to the set Δ_σ and the function $\Delta_{\surd}$. Here, we show that by instantiating these two parameters, it is possible to represent in our setting different time models presented in the literature. In particular, we consider the unique calculi which, to the best of our knowledge, consider time in the shared dataspace coordination model: (i) the formal modeling of JavaSpaces reported in [5], (ii) the timed Linda Language of [7], and (iii) the class of timed process calculi reported in [11].

The time model of [5] can be summarized as follows: the execution of a coordination action takes no time, there is no notion of current time, and the behaviour of the terms sensible to the passing of time is modeled by associating timeouts to these terms. These timeouts represent delays, after which the behaviour of the corresponding term change. The active timeouts in a configuration are all simultaneously reduced by 1, whenever a special time transition, labelled with $\surd$, is performed. Transitions labeled with $\surd$ can be always activated, independently of the current configuration. In our process calculus, the same temporal behaviour can be obtained simply by instantiating $\Delta_\sigma = \{0\}$ and $\Delta_{\surd}(P, DS) = \{1\}$ for any P and DS.

In the timed Linda Language of [7], there exists an explicit timeout operation which behaves similarly to the delays described above. The main difference is that all the actions (and not only the time actions) take exactly one time unit. Moreover, time can pass only when a coordination operation is performed or there exists at least one active timeout. In our setting, we can model the same approach by instantiating $\Delta_\sigma = \{1\}$, $\Delta_{\surd}(P, DS) = \{1\}$ for any P and $DS \neq \emptyset$, and $\Delta_{\surd}(P, \emptyset) = \emptyset$ for any P. The main difference with respect to our calculus is that in [7] the terms sensible to the passing of time are those comprising at least an active timeout, while in our calculus they are the temporary data.

Observe that these two instantiations of Δ_σ and $\Delta_{\surd}$ satisfy our assumptions. Thus, the results that we prove in the next two sections apply to both the model of time of [5] and [7].

A final remark is devoted to the model of time adopted in [11], that is, the two-phase functioning. According to their approach, the execution of operations take no time, and time passes only when no operation can be performed, that is, operations are urgent and cannot be delayed.

We can consider a class of instantiations which follows the same approach. It is enough to assume $\Delta_\sigma = \{0\}$, $\Delta_{\surd}(P, DS) = \emptyset$ if there exists C such that $[P, DS, t] \xrightarrow{\sigma} C$ for some t, and $\Delta_{\surd}(P, DS) \neq \emptyset$, otherwise. In this way, temporal actions can be taken if and only if no functional operation can be performed. It is important to observe that this instantiation does not satisfy the assumption we make about the function $\Delta_{\surd}$. As a counter-example, consider the configuration $C = [in(a), \langle a \rangle_{t'}, t]$ with $t < t'$. This configuration has clearly at least one outgoing transition labelled with σ; thus $\Delta_{\surd}(in(a), \langle a \rangle_{t'}) = \emptyset$. This contrasts with the assumption that the function $\Delta_{\surd}$ cannot be empty for dataspaces which contain at least one datum. Informally, this means that in the configuration C the execution of $in(a)$ is urgent and cannot be delayed; this contrasts with the lazy approach we adopt.

As we will describe in the following, the assumption we make on $\Delta_{\surd}$ is necessary to prove the discrimination results reported in Sections 4.2 and 5.2. Hence, the results proved in those sections do not hold under the two-phase functioning approach.

3.2 Notation

In the following, we use α to range over *Label*, i.e., α may be either σ or $\surd$. We use $\bar{\alpha}$ to denote a sequence of labels, namely, $\bar{\alpha} = \alpha_1\alpha_2 \ldots \alpha_n$ where each α_i is either σ or $\surd$. With $C \xrightarrow{\bar{\alpha}} C'$, where $\bar{\alpha} = \alpha_1\alpha_2 \ldots \alpha_n$ we denote the sequence of transitions $C \xrightarrow{\alpha_1} C_1 \ldots C_{n-1} \xrightarrow{\alpha_n} C'$.

In the process calculus with temporary data we need to redefine the notion of termination and divergence. A configuration C is *terminated* (denoted by $C \not\longrightarrow$) if it has no more the ability to perform any functional step, i.e., for any C' such that $C \xrightarrow{\bar{\alpha}} C'$ there exists no C'' s.t. $C' \xrightarrow{\sigma} C''$. A configuration C has a terminating computation (denoted by $C\downarrow$) if it has the ability to reach a terminated configuration, i.e., there exists $\bar{\alpha}$ such that $C \xrightarrow{\bar{\alpha}} C'$ and $C' \not\longrightarrow$. A configuration C has an infinite computation (denoted by $C\uparrow$) if there exists an infinite computation starting from C which contains an infinite amount of functional transitions, i.e., there exists an infinite sequence $C_0, C_1, C_2, \ldots$ such that $C = C_0$ and, for each $i \geq 0$, we have $C_i \xrightarrow{\alpha_i} C_{i+1}$ and, for any j, there exists $k \geq 0$ such that $\alpha_{j+k} = \sigma$.

For the new calculus, we introduce the notion of *uniform* configuration, useful in order to characterize an interesting superclass of deterministic processes. 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 $C\not\uparrow$.

Also in the calculus with temporary data, we consider the structural congruence $\equiv$ for programs defined exactly as for the calculus with permanent data. Following the approach described for permanent data we will make no distinction between $[P, DS, t]$ and $[P', DS, t]$ in the case $P \equiv P'$.

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 2.1 cannot be directly adapted to cope with temporary data. Then, we show the existence of an alternative, nondeterministic encoding, which preserves termination only; finally, we show that it is not possible to define any divergence preserving encoding because $C\uparrow$ is decidable.

4.1 A termination preserving encoding of RAMs

The approach followed in the encoding of RAMs presented in Section 2.1 cannot be directly adapted to work under temporary data. Indeed, it could happen that a tuple $\langle r_j \rangle$, which represents a unit inside the register r_j, expires and is removed. In this way, an undesired decrement of the register r_j occurs.

Nevertheless, it is possible to adopt a more sophisticated approach, permitting to define a nondeterministic encoding of RAMs which preserves at least 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 synchronise 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:

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

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:

$$
[\![(i, c_1, c_2, \ldots, c_n)]\!]_R \quad = \quad \{[P_i | \prod_{l=1}^{n} \prod_{c_l} LOOP, \bigoplus_{l=1}^{n} \bigoplus_{k=1}^{c_l} \langle r_l \rangle_{t_{l,k}}, t] \mid t, t_{l,k} \in I\!N\}
$$

where by $\prod_j P$ we denote the parallel composition of j instances of the program P. The encoding is defined as a set of configurations, because different expiration times can be associated to the data, as well as different current times can be considered. Observe that each instance of datum r_l 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 of the encoding.

Theorem 2 *Given a state* $(i, c_1, c_2, \ldots, c_n)$*, a RAM program* R *and a configuration* $C \in [\![(i, c_1, c_2, \ldots, c_n)]\!]_R$*, we have that if* $(i, c_1, c_2, \ldots, c_n) \longrightarrow_R (i', c'_1, c'_2, \ldots, c'_n)$ *then there exists* C' *such that* $C \xrightarrow{\sigma^+} C'$ *and* $C' \in [\![(i', c'_1, c'_2, \ldots, c'_n)]\!]_R$.

Proof By case analysis. ∎

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.

For the proof of this theorem we need a preliminary lemma and definition, characterizing the configurations reachable from the encoding of the initial state of the RAM. These configurations, called henceforth *quasi-encodings*, are formed by:

- the encoding of a state of the RAM,
- kl pairs of *LOOP* and *KILL* programs (and na pairs of their derivatives),
- ul possible occurrences of *LOOP* programs not matched by a corresponding *KILL* program:

$$E(i, c_1, \ldots, c_n; kl, ul, na) =$$
$$\{[P_i | \prod_{kl}(KILL|LOOP) | \prod_{na+ul} LOOP | \prod_{l=1}^{n} \prod_{c_l} LOOP,$$
$$\bigoplus_{j=1}^{na} \langle a \rangle_{t_j} \oplus \bigoplus_{l=1}^{n} \bigoplus_{m=1}^{c_l} \langle r_l \rangle_{t_{l,m}}, t] \mid t, t_{l,m}, t_j \in I\!N\}$$

The following lemma states that any configuration reachable from the encoding of the initial state of a RAM is a quasi-encoding, either corresponding to the encoding of a reachable state of the RAM or having at least one unmatched *LOOP* program. Moreover, if the configuration contains an unmatched *LOOP*, then at least one unmatched *LOOP* will remain present in any reachable configuration.

Lemma 4.1 *Given an initial configuration* $(1, 0, 0, \ldots, 0)$*, a RAM program* R *and a configuration* $C \in [\![(1, 0, 0, \ldots, 0)]\!]_R$*, we have that if* $C \xrightarrow{\bar{\alpha}} C'$ *then there exist* $i', c'_1, \ldots, c'_n, kl', ul', n'_a$ *such that* $C' \in E(i', c'_1, \ldots, c'_n; kl', ul', na')$ *and one of the following conditions holds:*

(1) $ul' = 0$ *and* $(1, 0, 0, \ldots, 0) \longrightarrow^*_R (i', c'_1, c'_2, \ldots, c'_n)$
(2) $ul' > 0$ *and if* $C' \xrightarrow{\bar{\alpha}'} C''$ *then there exist* $i'', c''_1, \ldots, c''_n, kl'', ul'', na''$ *such*

that $C'' \in E(i'', c''_1, \ldots, c''_n; kl'', ul'', na'')$ *and* $ul'' > 0$.

Proof By induction on the length of $C \xrightarrow{\tilde{\alpha}} C'$. ■

A further Lemma allows us to state that whenever a configuration containing an unmatched $LOOP$ is reached during the computation of a RAM encoding, the computation cannot terminate.

Lemma 4.2 *Given an initial state* $(1, 0, 0, \ldots, 0)$*, a RAM program* R *and a configuration* $C \in [\![(1, 0, 0, \ldots, 0)]\!]_R$*, we have that if* $C \xrightarrow{\tilde{\alpha}} C'$ *and* $C' = E(i, c_1, \ldots, c_n; kl, ul + 1, na)$*, then* C' *has no terminating computations.*

Proof By condition (2) of Lemma 4.1 and observing that each each configuration containing an unmatched $LOOP$ contains a program $inp(a)?P\_Q$, which can always perform one functional move. ■

Theorem 3 *Given an initial state* $(1, 0, 0, \ldots, 0)$*, a RAM program* R *and a configuration* $C \in [\![(1, 0, 0, \ldots, 0)]\!]_R$*, we have that if* $C \xrightarrow{\tilde{\alpha}} C'$ *then one of the following holds:*

(1) there exists C'' *such that* $C' \xrightarrow{\tilde{\alpha}'} C''$ *and* $C'' \in [\![(i, c_1, c_2, \ldots, c_n)]\!]_R$ *where* $(1, 0, 0, \ldots, 0) \longrightarrow^*_R (i, c_1, c_2, \ldots, c_n)$*;*
(2) C' *has only infinite computations.*

Proof A consequence of Lemma 4.1 and 4.2. ■

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 the encoding preserves termination.

Corollary 4.3 *Given a RAM program* R *and a configuration* $C \in [\![(1, 0, 0, \ldots, 0)]\!]_R$*, we have that* $R\!\downarrow$ *if and only if* $C\!\downarrow$*.*

Proof A consequence of Theorems 2 and 3. ■

Hence, we can conclude that termination is undecidable under the unordered collection policy.

4.2 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 [8]). Here, we report a definition of this formalism adapted to our purposes.

Definition 4 *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 Place/Transition net (or 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.*

A transition $tr = (c, p, r)$ is usually written in the form $c \stackrel{r}{\mapsto} p$ 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 $tr = (c, p, r)$ 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 $m \xrightarrow{tr} m'$ or simply $m \longrightarrow m'$ when the transition tr 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* representing 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 the collection of expired 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

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

`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 $\mathcal{T}$.

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 the execution of the else branch of the *inp*. As in P/T nets with reset arcs it is impossible to test the absence of tokens in place $\langle a \rangle$, the transition `inp-(a,Q,R)` compensates this impossibility by first removing all tokens in $\langle a \rangle$. The execution of this transition corresponds to a sequence of moves in the calculus: first each available datum a expires, then they are all removed, and finally the else branch of the $inp(a)$ operation is taken (this is now possible 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.

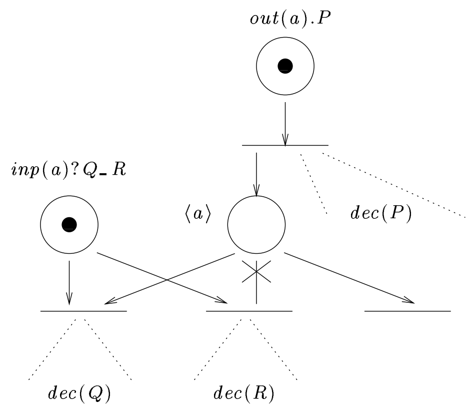

Fig. 1. The P/T net with reset arc $Net([out(a, \delta t).P|inp(a)?Q\_R, \emptyset, t])$.

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 datum $\langle a \rangle$, selected by the expired data collector, which is removed. Finally, axioms `inp+(a,Q,R)` and `inp-(a,Q,R)` describe 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 5 *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:*

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

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

As an example, we report $Net([out(a, \delta t).P|inp(a)?Q\_R, \emptyset, t])$ in Figure 1.

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 three sentences. The first states that each functional step of the configuration C is matched by a transition in the corresponding net, whereas the second states that each temporal step can be simulated in the net by performing no move (i.e., the source and target of a temporal step correspond to the same marking).

The third sentence states a similar symmetric result: each transition firable in the net can be mimicked by a sequence of steps, containing at least a functional one, of the corresponding configuration.

Theorem 6 *Consider the net $Net(D)$ for some configuration D. Let m be a marking of $Net(D)$ such that $m = dec(C)$ for some configuration C.*

(1) If $C \xrightarrow{\sigma} C'$ then $m \xrightarrow{r} dec(C')$ in $Net(D)$.
(2) If $C \xrightarrow{\surd} C'$ then $dec(C') = dec(C)$.
(3) If $m \xrightarrow{r} m'$ then $C \xrightarrow{\bar{\alpha}\sigma\bar{\alpha}} C'$, with $dec(C') = m'$.

Proof The proof of the first part is by induction on the derivation of the move $C \xrightarrow{\sigma} C'$, and then by case analysis on the last rule.

For the second part, observe that the decompositions of the source and the target of any instance of axiom (8) correspond to the same marking of the net.

The proof of the third sentence is by cases on the possible transitions.

The only nontrivial case is when the move is obtained by firing transition `inp-(a,P,Q)`. As this transition can fire even if place $\langle a \rangle$ contains tokens, its effect is reproduced in the calculus by a sequence of moves. First, a sequence of temporal steps is performed in order to reach an execution time greater than the expiration time of any instance of datum a; then, all the (now expired) occurrences of a are collected by repeated application of axiom (7_u); at this point, the dataspace does not contains data of kind a and the functional step obtained by axiom (4) is performed. Note that the existence of an initial sequence of temporal steps, leading to a sufficiently large execution time, is ensured by the assumption on function $\Delta_{\surd}$ (namely, it is always possible to let time pass if the dataspace is not empty), and because time is represented by natural numbers. ■

We are now ready to prove that the net semantics preserves divergence.

Corollary 4.4 *Given a configuration C, we have that $C \uparrow$ if and only if $Net(C)$ has an infinite firing sequence.*

As the existence of an infinite firing sequence is decidable for P/T nets with reset arcs [8], we can conclude that divergence is decidable also for the calculus with temporary data under the unordered collection policy.

Finally, we point out that the net semantics presented in this section ignores the information about time contained in a configuration, and the collection of expired data is simply represented by transitions that remove one token form places corresponding to data. This means that, under the lazy approach to time modeling, there is a tight relationship between temporary data with unordered collection and lossy/ephemeral data [1], i.e., data that can vanish at any moment after their creation. In fact, if we map a configuration of the calculus with temporary data in the configuration of a calculus with ephemeral data obtained by dropping all the time informations, the following properties are verified. Any computation of a configuration of the calculus with temporary data can be transformed in a computation of the corresponding configuration in the calculus with ephemeral data, by removing all time passing steps. On the other hand, a computation of the calculus with ephemeral data can be converted in a computation of the calculus with temporary data, by replacing each "disappearance" of a datum a by a sequence of time passing steps, sufficient to make an instance of datum a expire, followed by a collection of the expired datum.

5 Ordered Collection

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

5.1 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 when 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 exploiting the following 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 renewal. 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 can no longer conclude 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:

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

All the Q_i for indexes i outside the range of the instruction indexes are defined $Q_i = \mathbf{0}$.

We are now ready to define the program constant P_i:

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

where δt and $\delta t'$ are two time intervals such that $\delta t' < \delta t$.

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 $out(a, \delta t').Q_1$ instead of Q_1 only.

We define the encoding of a RAM program R in the initial state of the computation as

$$Init_R \ = \ [out(a, \delta t').Q_1, \emptyset, t]$$

where t can be any time.

A rigorous proof the correctness of the encoding w.r.t. diverging computations is rather technical and requires a quite heavy notation to represent all the configurations reachable from $Init_R$; for this reason, we omit the low level details of the proofs and provide an explanation of the underlying ideas.

We start observing that each configuration $[P, DS, t]$ reachable from the initial configuration $Init_R$ satisfies the following properties, concerning the shape of the program, the kind of data contained in the dataspace and the relationship among the expiration times of such data.

The program P is either equal to one of the constants Q_i, P_i, $R_k toS_k^i$ or $S_k toR_k^i$, for some i and k, or to a program obtained from one of the aforementioned constants by executing at most three functional steps.

The data contained in any reachable configuration are of the following kind: $\langle r_j \rangle_t$, $\langle s_j \rangle_t$, $\langle a \rangle_t$ and $\langle b \rangle_t$. Moreover, the following constraints are satisfied:

- at most one occurrence of datum of kind a is present;
- if $\langle a \rangle_t$ appears in the dataspace, then its expiration time is strictly lower than the expiration time of any datum of kind $\langle r_j \rangle_{t'}$ available in the dataspace;
- at most one occurrence of datum of kind b is present;
- if $\langle b \rangle_t$ appears in the dataspace, then its expiration time is strictly lower than the expiration time of any datum of kind $\langle s_j \rangle_{t'}$ available in the dataspace.

The following lemma states that whenever a datum is removed, because it has expired during a computation starting from the initial configuration $Init_R$, the computation can no more continue infinitely. This ensures that an infinite computation does not contain any undesired data withdrawal (due to data expiration).

Lemma 5.1 *Consider a RAM program R. Let C be a configuration such that $Init_R \xrightarrow{\bar{\alpha}} C$. If there exists C' such that $C \xrightarrow{\sigma} C'$ and the derivation of the transition makes use of axiom (7_o) then C' has no infinite computations.*

Proof The proof is by case analysis on the shape of configuration C.

Let $C \xrightarrow{\sigma} C'$, with $C = [P, DS, t]$. If derivation of this transition makes use of axiom (7_o), the removed datum is one of the following:

- the unique instance of the expired datum a is removed. In this case, it is possible to show that no other instance of a can be produced, and any possible execution leads, in a finite number of steps, to the intermediate point of execution of program $R_n toS_n^i$ where the only possible operation to be performed is $in(a)$; as the dataspace does not contain data of kind a, a terminated configuration has been reached.
- an instance of r_j is removed. As a consequence of the properties of the dataspace of any reachable configuration, we have that no datum a is present; moreover, the fact that at least one instance of r_j was contained in the dataspace ensures that P does not correspond to the intermediate point of the execution of $R_n toS_n^i$ where the next operation to be performed is

$out(a, \delta t')$. As in the previous case, after a finite number of steps the only possible operation to be performed is $in(a)$, i.e. a terminated configuration is reached.

- the unique instance of the expired datum b is removed. This case is similar to the first one.
- an instance of s_j is removed. This case is similar to the second one. ■

The next lemma is useful to prove that in the case no consumption of expired data is performed, then the computation of the initial $Init_R$ configuration proceeds deterministically.

Lemma 5.2 *Let C be a configuration such that $Init_R \xrightarrow{\tilde{\alpha}} C$. If there exist C', C'' such that $C \xrightarrow{\sigma} C'$ and $C \xrightarrow{\sigma} C''$, and the derivations of the two transitions do not make use of axiom (7_o), then there exist P, DS, t', and t'' such that $C' = [P, DS, t']$ and $C'' = [P, DS, t'']$.*

Proof By case analysis on the shape of configuration C. ■

The following notation is used to denote configurations reachable during a computation of the RAM program R:

$$[\![(i, c_1, \ldots, c_n)]\!]_R = \{[Q_i, \langle a \rangle_{t_a} \oplus \bigoplus_{j=1}^{n} \bigoplus_{k=1}^{c_j} \langle r_j \rangle_{t_{j,k}}, t] \mid t_{j,k}, t_a, t \in I\!N\}$$

The encoding is defined in terms of a set of configurations, because different expiration times could be associated to the currently available data, and also different current times could be taken into account.

We are now ready to prove that each computation step of the considered RAM program can be simulated by a sequence of steps in our encoding.

Theorem 5.3 *Consider a RAM program R. Let $(i, c_1, \ldots, c_n)$ be a configuration reachable during the computation of the program R, and take $C \in [\![(i, c_1, \ldots, c_n)]\!]_R$. If $(i, c_1, \ldots, c_n) \longrightarrow_R (i', c'_1, \ldots, c'_n)$ then there exists $C' \in [\![(i', c'_1, \ldots, c'_n)]\!]_R$ such that $C \xrightarrow{\sigma^+} C'$. Moreover, no one of the proofs of these transitions makes use of axiom (7_o).*

Proof By case analysis on the move of the RAM. ■

The next theorem and the subsequent corollary state that an infinite computation starting from the initial $Init_R$ configuration correctly simulates the infinite computation of the corresponding RAM program R.

Theorem 5.4 *Consider a RAM program R.*
Let $Init_R \xrightarrow{\alpha_1} C_1 \ldots \xrightarrow{\alpha_n} C_n \ldots$ be an infinite computation.
If $C_k \in [\![(i, c_1, \ldots, c_n)]\!]_R$ then there exist $i', c'_1, \ldots, c'_n$, and j such that $(i, c_1, \ldots, c_n) \longrightarrow_R (i', c'_1, \ldots, c'_n)$, $j > k$, and $C_j \in [\![(i', c'_1, \ldots, c'_n)]\!]_R$.

Proof A consequence of Lemma 5.1, Theorem 5.3 and (repeated application of) Lemma 5.2. ■

Corollary 5.5 *Consider a RAM program R. Let* $Init_R \xrightarrow{\alpha_1} C_1 \ldots \xrightarrow{\alpha_n} C_n \ldots$ *be an infinite computation. Then also R has an infinite computation.*

Proof By repeated application of Theorem 5.4. ■

Finally, we can conclude that the encoding preserves divergence.

Corollary 5.6 *Given a RAM program R, we have that* $R\uparrow$ *if and only if* $Init_R\uparrow$.

Proof The *only if* part is a direct consequence of Theorem 5.3, while the *if* part has been proved in Corollary 5.5. ■

Finally, we can conclude that divergence is undecidable under the ordered collection policy.

5.2 Divergence is decidable (for uniform configurations)

Here we prove that divergence is decidable for uniform configurations. We recall that a configuration is uniform (with respect to termination) if it satisfies the following condition: if the configuration has a terminating computation then all its computations are terminating (see Section 3.2 for the formal definition).

This decidability result is exploited to prove the impossibility to define a RAM encoding preserving both termination and divergence. The proof proceeds by contraposition. We first assume the existence of such an encoding; then we show that, for each RAM, the corresponding target w.r.t. the encoding should be a configuration uniform with respect to termination; finally, we prove that divergence is decidable for uniform configurations. As we have assumed that the encoding preserves divergence, this implies that also the divergence of RAMs is decidable (which is clearly false).

Let suppose, by contraposition, that there exists a divergence and termination preserving encoding of RAMs: if the RAM terminates then its target configuration has only finite computations, while if the RAM diverges then its target configuration has only infinite computations. Hence, such a target configuration is uniform.

We resort to P/T nets to prove that, for any uniform configuration, it is decidable whether it has a divergent computation. 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 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 define a new net semantics by applying the following two modifications to the net semantics presented in Subsection 4.2.

The first variation consists of replacing the transition `inp-(a,Q,R)` with an extended version, called henceforth `Einp-(a,Q,R)`, consisting in adding a reset arc between this transition and any place representing data. In this way, when a `Einp-(a,Q,R)` transition fires, all the data are removed. This behaviour corresponds to a sequence of computation steps in which first all the data expire, then they are globally removed, and finally the considered $inp(a)$ fails.

The second difference with respect to the previous net semantics is that we remove the transition `dis(a)`; indeed, this mechanism, which was used to simulate the withdrawal of expired data under the unordered collection policy, allows a datum to be withdrawn independently of the other data currently available. This approach does not satisfy the time constraints imposed by the ordered collection policy.

Formally, given a configuration C and the previously defined corresponding P/T net with reset arcs $Net(C) = (S', T', m'_0)$, we define the new net semantics $Nnet = (S, T, m_0)$ as follows:

$$
\begin{aligned}
S &= S' \\
T &= T' \setminus \{tr \mid tr \text{ is either one of the } \texttt{inp-(a,Q,R)} \text{ or } \texttt{dis(a)} \text{ transitions}\} \\
&\quad \cup \{tr = c \overset{\{\langle b \rangle \mid \langle b \rangle \in S\}}{\mapsto} p \mid c \overset{\{\langle a \rangle\}}{\mapsto} p \in T' \text{ for some } a\} \\
m_0 &= m'_0
\end{aligned}
$$

here $\setminus$ denotes set difference.

The new net semantics has the following properties.

The next lemma states that each transition in the net can be mimicked by one or more moves of the corresponding configuration.

Lemma 5.7 *Let $Nnet(D)$ be the net corresponding to some configuration D. Let C be a configuration such that $dec(C)$ is a marking of $Nnet(D)$. If $dec(C) \xrightarrow{r} m'$ then there exists a configuration C' such that $dec(C') = m'$ and $C \xrightarrow{\bar{\alpha}\sigma\bar{\alpha}} C'$.*

Proof The proof is by case analysis on the possible transitions.

The only nontrivial case is when the move is obtained by firing transition `Einp-(a,P,Q)`. As this transition can fire even if place $\langle a \rangle$ contains tokens, its effect is reproduced in the calculus by a sequence of moves. First, a sequence of temporal steps is performed in order to reach an execution time greater than the expiration time of any datum in the dataspace; then, all the (now expired) data are collected by repeated application of axiom (7_o); at this point, the dataspace does not contains data of kind a and the functional step obtained by axiom (4) is performed. Note that the existence of an initial sequence of temporal steps, leading to a sufficiently large execution time, is ensured by the assumption on relation $\Delta_{\surd}$ (namely, it is always possible to let time pass if the dataspace is not empty), and because time is represented by natural numbers. ∎

Corollary 5.8 *Let $Nnet(C)$ be the net corresponding to some configuration C. If $dec(C) \longrightarrow^* m'$ in $Nnet(C)$, then there exists a configuration C' such that $C \xrightarrow{\sigma^*} C'$ and $m' = dec(C')$.*

The following lemma 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 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 available. This kind of configuration is trivially terminated.

Lemma 5.9 *Let $Nnet(D)$ be the net corresponding to some configuration D. Let C be a configuration such that $dec(C)$ is a marking of $Nnet(D)$. If there exists no m s.t. $dec(C) \xrightarrow{r} m$ then there exists C' s.t. $C \xrightarrow{\bar{\alpha}} C' \not\xrightarrow{\sigma}$.*

Proof By a sequence of temporal steps, leading to an execution time greater than the expiration time of any datum, followed by repeated applications of axiom (7_o), from C we reach a configuration C' whose dataspace is empty. Then, supposing by contraposition the existence of a step $C' \xrightarrow{\sigma} C''$, it is possible to show the impossibility of the existence of such a step, by case analysis. ∎

We are now ready to prove that the new net semantics is divergence preserving at least for uniform configurations.

Theorem 7 *Let C be a uniform configuration. Then configuration C has a divergent computation if and only if $Nnet(C)$ has a divergent computation.*

Proof The *if* direction is a consequence of Lemma 5.7. The other direction proceeds by contraposition, making use of Corollary 5.8, Lemma 5.9 and the uniformity property. ■

As the existence of an infinite firing sequence is decidable for P/T nets with reset arcs, we can conclude that divergence is decidable also for uniform configurations of the calculus with temporary data under the ordered collection policy.

6 Conclusion

We have investigated the impact of the use of temporary data, intended as data with an associated expiration time, on the expressiveness of shared dataspace coordination languages. We have considered two approaches for the collection of expired data: the unordered collection removes one of the expired data, whereas the ordered collection removes the datum which expired first. In order to perform our investigation, we exploited timed process calculi: we adopted a discrete model of time, where time instants are represented by natural numbers, and parametric with respect to a set and a function, representing respectively the time necessary for an action to be executed and the increment of time performed by a time passing move.

Exploiting the introduced framework, we prove that the use of temporary data decreases the expressive power in the following sense: there is no way to provide an encoding of permanent data in temporary data that preserves both termination and divergence. Also the policy adopted for removal of expired data influences the expressiveness: in fact, it is not possible to provide a divergence preserving encoding of the ordered collection in the unordered one.

Recently, three proposals dealing with time in shared dataspace coordination languages appeared in the literature [5,7,11]. In Section 3.1 we showed that the models of time presented in [5] and [7] can be seen as instances of our model; hence, the results presented in this work continue to hold.

In [11] the two-phase functioning approach, typical 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. As we already discussed in Section 3.1, this approach does not fit our lazy modeling of time because of the condition on the function $\Delta_{\surd}$, according to which actions are not urgent, and can be delayed indefinitely.

The proofs of the positive results (i.e., the proofs concerning the existence of RAM encodings) do not make use of the above mentioned assumption. For this reason, these results continue to hold also under the two-phase functioning approach.

On the other hand, the negative results (i.e., the proofs about the impossibility to provide RAM encodings because divergence is decidable) do not hold for two-phase functioning. The reason is that, in the proofs of these results, the assumption on $\Delta_{\surd}$ is used in order to ensure that, starting from any configuration $[P, DS, t]$, and given a specific subset DS' of the dataspace DS, it is possible to reach a configuration $[P, DS \setminus DS', t']$, in which all the data in DS' have been expired and collected. This is clearly false, under two-phase functioning, because time cannot pass if there is at least an enabled functional step in the considered configuration $[P, DS, t]$.

A final remark is concerned with time models based on dense or continuous time instead of discrete. Even if our results are proved for discrete time, the only assumption we make is that it is guaranteed that a given time instant can be reached from any previous instant by a finite sequence of temporal steps. This assumption corresponds with the impossibility, for a dense or a continuos time model, to exhibit Zeno behaviours. Hence, it is reasonable to conjecture that our results hold also under this class of time models, provided that they can exhibit only non-Zeno behaviours.

Acknowledgements

We are grateful to the referees of the paper for their useful comments and suggestions.

References

[1] P. Aziz Abdulla and B. Jonsson. Verifying Programs with Unreliable Channels. *Information and Computation*, 127(2):91–101, 1996.

[2] J.C.M. Baeten and C.A. Middelburg. Process Algebra with Timing: Real Time and Discrete Time. *Handbook of Process Algebra*, J.A. Bergstra, A. Ponse and S.A. Smolka eds. Elsevier, 2001.

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

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

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

[6] N. Busi, R. Gorrieri, and G. Zavattaro. Temporary Data in Shared Dataspace Coordination Languages. In *Proc. of FOSSACS'01*, volume 2030 of *Lecture Notes in Computer Science*, pages 121–136. Springer-Verlag, Berlin, 2001.

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

[8] C. Dufourd, A. Finkel, and P. Schnoebelen. Reset nets between decidability and undecidability. In *Proc. of ICALP'98*, volume 1061 of *Lecture Notes in Computer Science*, pages 103–115. Springer-Verlag, Berlin, 1998.

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

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

[11] J.M. Jacquet, K. De Bosschere, and A. Brogi. On Timed Coordination Languages. In *Proc. of Coordination'00*, volume 1906 of *Lecture Notes in Computer Science*, pages 81–98. Springer-Verlag, Berlin, 2000.

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

[13] M.L. Minsky. *Computation: finite and infinite machines.* Prentice-Hall, 1967.

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

[15] F. Moller and C. Tofts. Relating Processes With Respect to Speed. In *Proc. of Concur'91*, volume 527 of *Lecture Notes in Computer Science*, pages 424–438. Springer-Verlag, Berlin, 1991.

[16] X. Nicollin and J. Sifakis. The Algebra of Timed Processes, ATP: Theory and Application. *Information and Computation*, 114(1):131–178, 1994.

[17] J.C. Shepherdson and J.E. Sturgis. Computability of recursive functions. *Journal of the ACM*, 10:217–255, 1963.

[18] I. Satoh and M. Tokoro. A Formalism for Remotely Interacting Processes. In *Proc of TPPP'94*, volume 907 of *Lecture Notes in Computer Science*, pages 216–228. Springer-Verlag, Berlin, 1994.

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

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

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