Linda in Space-Time: An Adaptive Coordination Model for Mobile Ad-Hoc Environments

Mirko Viroli[1], Danilo Pianini[1], and Jacob Beal[2]

[1] Alma Mater Studiorum – Università di Bologna, Italy
{mirko.viroli,danilo.pianini}@unibo.it
[2] Raytheon BBN Technologies, USA
jakebeal@bbn.com

Abstract. We present a vision of distributed system coordination as a set of activities affecting the space-time fabric of interaction events. In the tuple space setting that we consider, coordination amounts to control of the spatial and temporal configuration of tuples spread across the network, which in turn drives the behaviour of situated agents. We therefore draw on prior work in spatial computing and distributed systems coordination, to define a new coordination language that adds to the basic Linda primitives a small set of space-time constructs for linking coordination processes with their environment. We show how this framework supports the global-level emergence of adaptive coordination policies, applying it to two example cases: crowd steering in a pervasive computing scenario and a gradient-based implementation of Linda primitives for mobile ad-hoc networks.

1 Introduction

A common viewpoint in developing coordination models on top of Linda [16] is that tuples are a mechanism to reify and exchange events/data/knowledge that are important to system coordination, and to synchronise the activities of coordinated components in a parallel and/or distributed system. A tuple is, however, a very point-wise abstraction, and applications often need to express relationships articulated across the physical environment through which the computational system is distributed. There is thus a need for coordination models and languages that raise the level of abstraction from the single tuple to a spatial structure of tuples, without forgetting the possibility that such a structure, as well as being distributed, may also be highly dynamic and mobile.

Such spatial coordination models have been developed in a number of previous papers [29,17,18,24]. These span multiple application contexts, such as pervasive computing, where mobility, large system scale, and/or situatedness invite the coordination space to be interpreted as a distributed substrate for spatial data structures. These models enhance the standard tuple-space settings with primitives for spreading tuples from nodes to their neighbors, and letting them

M. Sirjani (Ed.): COORDINATION 2012, LNCS 7274, pp. 212–229, 2012.

affect (or be affected by) the context of neighbors. Iterative spreading can then lead to the stabilisation of given tuple structures, which find applications such as retrieval of items of interest in mobile environments, as in the TOTA middleware [17] or the pervasive ecosystems model [30]. These prior models, however, tend to be either ad hoc (e.g., [24]) or tightly tied to a particular metaphor (e.g., [29]).

These approaches may also be viewed through the research lens of spatial or amorphous computing [6], which argues that computation in a dense and mobile system is best understood and designed in terms of spatial abstractions (e.g., regions, partitions, gradients, trails, paths) and temporal abstractions (e.g., fading, growth, movement, perturbation). The Proto language [4] is an archetype of this approach, presenting a model in which these abstraction—intrinsically discrete because of the nature of computing devices—would actually tend toward their continuum version as the density of the network, which we are continuing to experience with current ICT technologies. As this research program links geometric continuum abstractions and individual computing devices, it thus has the potential of addressing the crucial issue of designing distributed systems where adaptiveness globally emerges out of local interactions in a predictable and controllable way.

Based on the above works, and on an existing trend of studies of concurrency "in space" [12,11], we introduce a new coordination model and language aiming at further bridging the gap between coordination and the continuum ideas of spatial computing. In our model, we make situated agents interact by injecting into the coordination substrate so-called *space-time activities*, namely, processes that manipulate the space-time configuration of tuples in the network. Such activities are expressed in terms of a process-algebra like language, composing atomic coordination primitives in the style of Linda with additional Proto-derived constructs that deal with spatial and temporal aspects: *(i)* spreading of an activity in a node's neighbourhood depending on its relative orientation; *(ii)* scheduling an activity at the next computation round of a node; and *(iii)* accessing space-time contextual information to link the configuration of tuples with the actual physical space.

Contribution: The main contribution of this work is hence the definition of a spatial computing coordination language, which we call $\sigma\tau$-Linda, extending Linda to flexibly enact coordinated activities tightly linked in space-time with their environment. It provides more advanced mechanisms for controlling space-time behaviour of coordination activities than those of existing coordination middlewares such as TOTA [17], and overcomes the restrictions of Proto [4] that make it unsuitable as an open coordination framework (as will be detailed in Section 5).

The proposed model can be used to design coordination mechanisms that emergently adapt to environment stimuli, such as agent interactions or changes due to mobility and faults. We describe two applications for this model: first we develop an adaptive crowd steering coordination service by which people in a structured environment can be guided (by signs appearing in their personal smartphone or public devices) towards a point of interest through the shortest

path that circumvents dynamically-formed crowded areas. Second, we provide a space-time extension of standard Linda coordination primitives (`out`, `in` and `rd`) working in a distributed mobile environment, and relying on the self-organisation pattern known as a computational gradient [2].

The remainder of this paper is organised as follows. Section 2 illustrates the proposed model and language, Section 3 describes a formalisation expressed as a core calculus, Section 4 presents application cases, Section 5 relates our model with prior work, and finally Section 6 concludes with final remarks.

2 Linda in Space-Time

2.1 Basic Model

Our coordination infrastructure runs on a (possibly very dense and mobile) set of situated computational devices, e.g., located in specific points (i.e. nodes) of the physical environment. Each node hosts software agents (the coordinated components) and a tuple space, and has ability of interaction with nodes in the neighbourhood—where proximity can be seen as a physical or virtual property. Differently from Linda, in which agents interact by atomic actions (inserting, removing and reading tuples in the local tuple space), in $\sigma\tau$-Linda, agents interact by injecting *space-time activities* (activities for short). These activities are processes composing atomic Linda-like actions with additional constructs allowing the activity to diffuse in space (i.e. to other nodes) and in time (i.e. to be delayed). The net effect of an activity is hence to evolve the population of tuples in the network, thereby affecting the distributed structures of tuples that agents use to for global coordination.

In our model, each node undergoes the following *computation round*: (i) it sleeps, remaining frozen; (ii) it wakes up, gathers all incoming activities (contained in messages received either from neighbour nodes or from local agents) and executes them; (iii) it executes the continuation of the activity executed in previous computation round; (iv) spreads asynchronous messages to neighbourhood; and (v) schedules an activity continuation for next round. The node then sleeps again, returning to the beginning of the cycle. The duration of the computation round is dictated by the underlying infrastructure, and could possibly change over time or from device to device (as in [1]). We only assume that it is long enough for executing steps (ii-iii-iv) above. This particular computational model for a tuple space has many similarities with the platform assumptions of the Proto language [4,27], which we adopt for its facility of situating computations in space-time—a thorough comparison is reported in Section 5.

2.2 The Coordination Language

A key role in the proposed coordination model is played by the concept of space-time activities. We here incrementally describe their features by presenting a surface language for their specification.

Primitive Actions. We being with three basic Linda actions for manipulating the tuple space: "`out` *tuple*", "`in` *tuple*", "`rd` *tuple*". These respectively insert, remove and read a tuple from the local tuple space. A tuple is a ground first-order term in our model—similarly to [22]. Read and removal specify a template (a term with variables, denoted as literals starting with an upper-case letter) that should be syntactically matched with the retrieved tuple. Read and removal are predicative: they are non-blocking and yield a negative result if no matching tuple is found. To these, we add a fourth primitive, "`eval` *pred*", which evaluates the predicate expression *pred*.

When such actions are defined for an activity injected by an agent, and are executed in the tuple space where the agent is situated in, then a notification result is shipped to the agent—although, following the spirit of [8], we shall not discuss internal aspects of agent interactions in this paper.

Protocols. Primitive actions can be sequentially composed in a protocol-like manner. Other standard operators of parallel composition and choice could be orthogonally added, but are not discussed in this paper for brevity. Additionally, `in`, `rd` and `eval` define branches leading to two different continuations, one for positive and one for negative outcome of the predicative action. Three examples of activities are:

```
out t(1,2,3); out t(a,1+2,b)
in r(X,2,3) ? out r(X,2,3) : out r(0,2,3)
(in r(X,2,3) ? (eval X=1 ? out r(X,2,3) : 0) : 0); out ok
```

The first expression inserts tuple `t(1,2,3)` and then `t(a,3,b)`. Note that tuples are evaluated before being used in actions: evaluation amounts to computing the result of (mathematical) expressions used in a tuple's arguments.

The second expression removes any tuple matching `r(X,2,3)` (variable `X` is bound to the value of first argument, and this substitution propagates through the remainder of the activity). If it succeeds (`?` branch) the tuple is inserted back, otherwise (`:` branch) a new tuple `r(0,2,3)` is inserted.

The third example attempts to remove any tuple matching `r(X,2,3)`. If it succeeds and `X` = `1` then it inserts it back, otherwise it does nothing (`0`). Independently of the outcome of such a removal, tuple `ok` is then inserted. We may also omit the denotation of a ":" branch when it leads to the execution of empty process `0`, writing e.g. "`(in r(X,2,3) ? eval X=1 ? out r(X,2,3)); out ok`" in place of the third example above.

Definitions. When desired, one can equip the specification of an activity with *definitions* (which can possibly be recursive), in the style of agent definition in π-calculus. These have the form "N`(`x_1`,...,`x_n`) is` *activity*", which define *activity* as having name N and arguments x_1,...,x_n. For instance, after declaration

```
in-out(T) is (in T; out T)
```

we have for instance that activity "`in-out(r(1,2,3))`" behaves just like "`in r(1,2,3); out r(1,2,3)`", namely, the tuple is added if it is not already there. Note that since no branches are used for the removal operation, this in turn is equivalent to "`(in r(1,2,3)?0:0); out r(1,2,3)`", or, similarly, to "`in r(1,2,3)?out r(1,2,3):out r(1,2,3)`".

Time. The language provided so far is still point-wise in space and time. We now expand it, beginning by adding construct `next` to situate activities in time. Executing action "`next` P" (where P is the protocol – also called process – defining an activity), amounts to scheduling P for execution at the next computation round of the current node. Special variable `$delay` can be used in P and evaluates to the amount of time passed in between the current computation round and the previous one. Similarly, variable `$this` can be used to denote the identifier of the node on which it is evaluated. Useful examples of definitions (along with a brief descriptions of them) are then the following ones:

```
% When a tuple matching T is found, it is removed and replaced with T2
chg(T,T2) is (in T ? out T2 : next chg(T,T2) )

% Inserts a tuple time(T,X), updated as time X passes
rep(T)  is (out time(T,0); next rep2(T))
rep2(T) is (in time(T,Y); out time(T,Y+$delay); next rep2(T))

% Inserts tuple T that is removed after X time units elapse
outt(T,X) is (in-out(T); eval X<=0 ? in T : next outt(T,X-$delay))
```

Note that by the use of `next` in conjunction with a recursive definition, `chg` actually declares an activity with a duration in time, which will be stopped only when a tuple matching `T` is eventually found. Concerning the use of `$delay`, we then observe that – in the spirit of a Proto-style space-time computing model – as the average duration of computation rounds tends to zero, activity `rep` tends to define a continuous update of tuple `time(T,X)` as time `X` passes, and similarly, `outt(T,X)` tends to remove tuple `T` precisely as time `X` passed.

Space. To situate activities in space we introduce construct `neigh`. Executing action "`neigh` P" amounts to sending a broadcast message containing P to all neighbours, which will then execute P at their next computation round. Special variable `$distance` is also introduced, which evaluates to the estimated distance between the node that sent the message and the one that received it. Similarly, variable `$orientation` can be used to denote the relative direction from the receiver to the sender (e.g., as a vector of coordinates [11]). Some examples are as follows:

```
% Broadcasts tuple T in the neighbourhood
bcast(T) is (neigh out T)

% Broadcasts tuple T in the neighbourood but only within range R
```

```
bcastr(T,R) is (neigh (eval $distance<R ? out T))

% Gossips tuple T in the whole network within range R
goss(T,R) is (eval R>=0 ? (in-out(T); neigh goss(T,R-$distance)))
```

Of particular interest is the last definition, which spreads one copy of `T` to all the nodes whose hop-by-hop distance from the source is smaller than `R`. As in the case of time, as devices become increasingly dense, and their distance tends to zero, the set of devices holding tuple `T` will actually form a continuous sphere with radius `R` around the origin of `goss`.

Note that it is an easy exercise to define processes dealing with both space and time—as will be developed in Section 4. For instance, one can define a process `gosst` that adds temporal aspects to the `goss` example, such as to make the sphere of tuples created by `goss` all disappear following a timeout. In the sense of spatial computing interpretation [4], the definition of `gosst(T,R,TO)` would be the definition of a geometric space-time activity called "sphere of tuple `T` with radius `R` and timeout `TO`"—useful to limit the spatial and temporal extent of some advertised information.

Finally. We conclude by introducing a construct named `finally`, used to simplify the task of structuring the activities executed at a given round. Executing action "`finally` P" makes activity P executed in the current round, but only when all the others actually completed. A typical use of this construct is to start an aggregation activity for incoming messages only when all of them have been processed, as in the following equivalent specification of gossiping:

```
gossf(T,R) is (eval R>=0 ? (out T; neigh gossf(T,R-$distance);
                            finally clean(T)))
% Cleans multiple copies of T, leaving just one of them
clean(T) is (in T ? (in-out(T); clean(T)))
```

Messages spread by gossiping cause the receiver to execute the `gossf` activity, which inserts tuple `T`, further spreads messages, and finally schedules the `clean(T)` process for a later time. Only when all such messages have been processed in a round (and there are typically more than one) will the set of all `clean` activities be executed . The result of their execution is that only one tuple `T` will remain in the tuple space. The `finally` construct can thus, e.g., serve a similar aggregation and simplification role to the `*-hood` constructs in Proto.

3 Core Calculus

In this section we introduce a formalisation of the proposed framework similar in spirit to those of [8,18,28], namely, by a core calculus taking the shape of a process algebra.

3.1 Syntax

Let meta-variable σ range over tuple space (or node) identifiers, x over logic variables, τ over real numbers used to model continuous time, and f over function names (each with a given arity, and used either in infix of prefix notation)—as usual we refer to functions with arity 0 as constants. Meta-variable t ranges over terms built applying functions to variables, numbers, identifiers, and constants, and will be written in typetext font. For simplicity, we shorten special variable `$orientation` to ω, and neglect `$distance` since it can be "compiled away" to term $length(\omega)$ where *length* is a function. We let ϵ range over evaluations (functions) for terms, write t^ϵ for application of ϵ to term t, and denote $\epsilon(\sigma, \tau)$ the evaluation that (other than computing mathematical functions) maps `$this` to σ and `$delay` to τ. For instance, we have $\texttt{a(\$this,1+\$delay)}^{\epsilon(\texttt{id23},5.1)} = \texttt{a(id23,6.1)}$. A substitution θ of variables $x_1, \ldots, x_n$ to terms $t_1, \ldots, t_n$ is expressed by notation $\{t_1/x_1, \ldots, t_n/x_n\}$, and is applied to a term t by syntax $t\theta$, e.g., $a(x, 1)\{x/2\}$ means $a(2, 1)$. We write $mgs(t, t')$ for the most general substitution θ such that $t'\theta = t$—such a notation makes no sense (as in partial functions) if $mgs(t, t') = \bot$, i.e., when t is not an instance of t'.

Given these premises, the core syntax of the model is expressed by the grammar in Figure 1 (a). P defines the syntax of a process (or activity): it includes empty process 0, action prefix, predicative actions with branches, and call of a definition. Note we skipped from this syntax the composition operator ";", which can be basically compiled away once we have action prefix "." and branching "? :"—by straightforward equivalences like $0; P \equiv P$, $(\pi?P : Q); R \equiv \pi?(P; R) : (Q; R)$ and $(\alpha.P); R \equiv \alpha.(P; R)$. A space S is a composition, by operator " | ", of processes and tuple sets. The topology of a network is modelled by a composition L of connections of kind $\sigma \overset{t}{\rightsquigarrow} \sigma'$, representing proximity of node σ' to σ with orientation vector t—e.g., expressed as term `coord(x,y,z)` or the like. Finally, a system configuration C is a composition, by operator $\otimes$, of nodes $[S]_\sigma^{\tau,\tau'}$ (with id σ, space S, current round at time τ and previous one at τ'), topology L, and messages $P \triangleright \sigma$ (with content P and recipient σ).

Figure 1 (b) introduces a congruence relation "$\equiv$", stating when two configurations are to be considered syntactically equal, and hence can be used one in place of the other. First line introduces standard multiset-like properties of operators " | " and "$\otimes$". Second line states that scheduling operators can be lifted out of action prefix placed in parallel with the continuation, and can also distribute in parallel processes. Last line states that when a `finally` and `next` actions are in parallel composition, the latter can enter the former: this will in fact leave scheduling policy for Q unchanged.

3.2 Operational Semantics

We define operational semantics by transitions $C \xrightarrow{\lambda} C'$, where labels λ can have the syntax described in Figure 1 (a). Label "$\cdot$" means a silent action internal

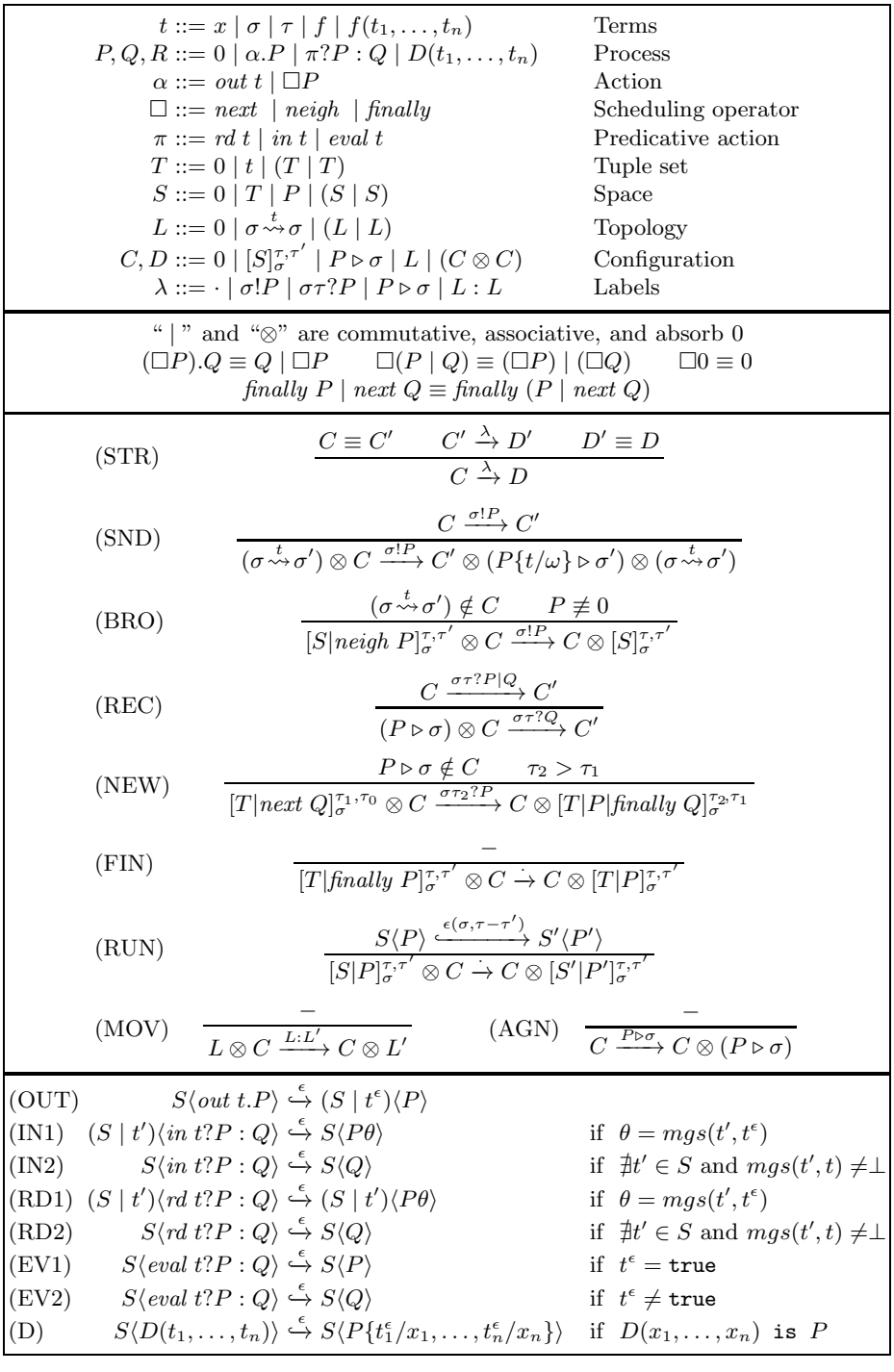

Fig. 1. (a) Grammar, (b) Congruence, (c) Global semantics and (d) Local semantics

to a node σ; "$\sigma!P$" means device σ is broadcasting a message with content P; "$\sigma\tau?P$" means device σ starts a new computation round at (its local) time τ and still needs to gather messages with content P (at the top level it will take the form $\sigma\tau?0$); "$P \triangleright \sigma$" means an agent is injecting process P in the tuple space σ; and "$L : L'$" means (sub)topology L changes to L' to reflect some mobility or failure in the system. Semantic rules are shown in Figure 1 (c).

Rule (STR) defines classical structural congruence. Rules (BRO) and (SND) recursively handle broadcasting (mostly in line with [26]), namely, create messages for all neighbours as soon as a process P is scheduled for broadcasting. Rule (SND) recursively selects a neighbour σ' at orientation t, and creates a message for it in which orientation variable ω is substituted with t. Rule (BRO) is the fixpoint: when all neighbours have been handled, scheduling action *neigh* P is removed. Note we do not send empty messages.

Similarly, rules (REC) and (NEW) recursively handle the reception of all messages when a new computation round starts. Rule (NEW) states that, given node σ in which Q is the process to execute at the next round, when a new round starts at time τ_2 and with overall incoming messages P, then the new process to start with is "$P|$*finally* Q", since we prescribe messages to be handled before Q as already described in previous section. Also note that this rule updates round times τ_1, τ_0 to τ_2, τ_1, and that it activates only when all incoming messages have been actually handled. Rule (REC) recursively gathers all incoming messages: it takes one with content P and proceeds recursively adding P to the set Q of messages considered so far.

Rule (FIN) handles semantics of *finally* P construct, by simply stating that when this is the only activity in a node, we can simply execute P—note all "finally-scheduled" processes can be gathered together (along with "next-scheduled" ones) because of congruence. Rule (RUN) handles one-step execution of a process, by simply deferring the task to transition relation $\overset{\epsilon}{\hookrightarrow}$, defined in Figure 1 (d)—its rules are quite straightforward, as they correspond to the standard semantics of Linda primitives in their predicative version [8]. Note that $\overset{\epsilon}{\hookrightarrow}$ takes the evaluation function to use, initialised in rule (RUN) with the proper value of `$this` and `$delay`. Finally, rule (MOV) addresses topological changes due to mobility or failures, and rule (AGN) models the injection of a process by an agent in the local node.

We conclude stating isolation and progress properties. First property allows one to reason about the execution of an activity into a node without considering its environment. Namely, we have that nodes get affected by the external environment only at the time a new computation round starts (because of reception of messages), otherwise they proceed in isolation possibly just spawning new messages.

Property 1. If $C \otimes [S]_\sigma^{\tau_0,\tau_0'} \xrightarrow{\lambda} C' \otimes [S']_\sigma^{\tau,\tau'}$ with $S \not\equiv S'$ then λ is either $\cdot$, $\sigma!P$, or $\sigma\tau?P$. In the former two cases (namely, unless we change computation round), $\tau_0 = \tau$, $\tau_0' = \tau'$, and $C' \equiv C \otimes C_m$ (where C_m is either 0 or a broadcast), and moreover, for each D we have also $D \otimes [S]_\sigma^{\tau_0,\tau_0'} \xrightarrow{\lambda} D \otimes [S']_\sigma^{\tau_0,\tau_0'} \otimes C_m$, i.e., computation is independent of the environment.

The progress property states instead that when a computation round is completed it is necessarily composed of a `next` scheduling: at that point (NEW) can surely fire for that node, starting a new computation round. This ensures that our computations never get stuck.

Property 2. $C \otimes [S]_\sigma^{\tau,\tau'} \not\dashrightarrow$ and $C \otimes [S]_\sigma^{\tau,\tau'} \overset{\sigma!P}{\dashrightarrow}$ iff $S \equiv (T \mid next\ P)$. In that case, we have $C \otimes [S]_\sigma^{\tau,\tau'} \xrightarrow{\sigma\tau_0?0} C' \otimes [S']_\sigma^{\tau_0,\tau}$ for any $\tau_0 > \tau$.

4 Case Studies

4.1 Adaptive Crowd Steering

As a first example we study a specification able to support the case study presented in [30,25], with the goal of showing how $\sigma\tau$-Linda can provide support to easily define complex, distributed and adaptive data structures, and how they can be used in practice in a pervasive computing scenario.

Crowd-aware gradient

```
% creating a gradient spreading tuple T
source(T) is (in-out(source(T)); grad(T,0,$this))
% gradient process for tuple T, at distance D, coming from node S
grad(T,D,S) is grad(T,D,S,$this)
grad(T,D,S,This) is (
   rd source(T)
     ? in-out(pre(T,0))
     : in pre(T,N) ? (eval N<D
                      ? out pre(T,N)
                      : (in target(T,M); out target(T,S); out pre(T,D)))
); finally (in pre(T,N)? (in field(T,M);
   (rd crowd(C)
      ? out field(T,N-1.2*C)
      : out field(T,N)); rd field(T,V); neigh grad(T,V+$distance,This)))
```

Fig. 2. Definitions for the crowd-aware computational gradient. At each site, if this is the source we consolidate `pre(T,0)`. Otherwise, we replace the `pre` tuple if a smaller distance `D` is found, and `target` tuple is inserted as well. Finally, we take the remaining `pre` tuple, and apply the `crowd` factor: the resulting distance `N` goes into the `field` tuple.

Our reference environment is a bidimensional continuous space made of various rooms connected by strict corridors. Inside rooms and corridors, a dense grid of computational devices (nodes) is set up. Each node hosts its own tuple space, receives coordination activities (programmed using our spatial language) by software agents running in it, interacts with nodes in its proximity, and has a sensor locally injecting a tuple `crowd(CrowdLevel)` where `CrowdLevel` is an estimation of the number of people sensed around. People want to reach a point

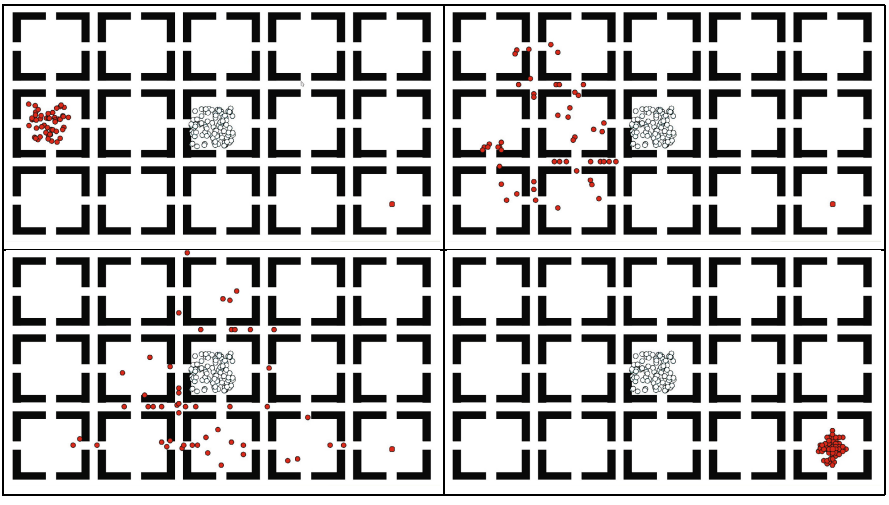

Fig. 3. Simulation snapshots: the coloured visitors reach its POI avoiding crowd

of interest (POI) by the fastest path, and receives directions suggested by their handheld device and/or by public displays on the walls. It is worth noting that the fastest path does not correspond to the shortest: if everybody followed the same way, in fact, corridors would become crowded. We want the system to be able, relying only on local interactions, to avoid crowded paths, dynamically adapting to any emerging and unforeseen situation. However, we will not implement algorithms to predict future situations, but rather make information about a crowded area spread around such that it becomes a less attractive transiting place to reach a POI.

Our strategy is to build a computational gradient injected by an agent located in the POI. A computational gradient holds in any node the estimated distance to the source by the shortest path [4], computed by further spreading and then aggregating at the destination the local estimation of distance. This distributed data structure must take into account also the crowding level, increasing estimated distance where a crowd is forming, and thus deflecting people towards longer but less crowded paths. This strategy can be encoded as in Figure 2, where the goal is achieved by maintaining a tuple `target(Poi,Id)` containing the `Id` of the neighbour node where to steer people to following a certain `Poi`. The crowding level influences the local field generation, and is weighted using a constant $K_{crowd} = 1.2$. Values between 1 and 1.5 have been established as good ones after running several simulations: more generally, the higher K_{crowd}, the more sensitive is path computation to the presence of crowd.

We implemented and ran simulations using Alchemist simulator [25], assuming that computation rounds are fired at the same rate for all nodes, and modelling such a rate following the Continuous-Time Markov Chain model. Four screenshots of a simulation run are provided in Figure 3, in which we built an environment of fifteen rooms with an underlying grid-like network of infrastructure

nodes, an initial configuration with two groups of people, and a POI of interest for the first group which is reachable by a path crossing a crowded area. Note that not only every visitor reached the POI, but they all bypassed the crowded room (even if it is part of the shortest path, the large amount of people inside makes the whole area rather disadvantageous to walk); additionally, the visitors group is subject to "self-crowding", in that when a group is following a path it forms crowded areas itself (e.g. near doors), hence people behind the group tend to follow different paths. Further simulations we do not describe here for the sake of space show that the above properties hold for a large set of situations, including presence of crowds in different locations and dynamic formation and movement of such crowds during simulation[1].

4.2 Linda in a Mobile Ad-Hoc Environment

As a second case study we show a possible extension for Linda standard primitives taking into account both time and spatiality. In particular, our aim is to show how would it be possible in a mobile ad-hoc environment to specify, along with an operation over a tuple, a spatial and temporal horizon of validity: only retrieval operations whose horizon embraces the respective target tuple will actually succeed. We will show an implementation for the spatio-temporal `out` (`stout`) and the spatio-temporal `in` (`stin`) primitives—the easier case of `strd` being a simple variant of `stin`.

The key idea is to make primitive Linda actions actually generate waveform-like space-time data structures, with limited extent in space and dissolving as a timeout expires. Those structures will be responsible to determine the pertinence in space and time of each operation. An example of such a structure is realised by the code shown in Figure 4 (top). A `wave` works similarly to the gradient in Figure 2, maintaining a `target` tuple reifying the shortest past through a similar specification. A main difference – other than the obvious absence of any crowd management – is the evaluation of the age and distance, which makes the wave disappear whenever and wherever the horizon is reached.

When a `stin` operation requiring retrieval of a tuple template `T` is triggered, it will spawn a messenger activity called `hermes` (with `Op` set to `in`) which will propagate to a matching tuple `T'` following the corresponding `wave` it generated. As soon as the tuple is found, a new `hermes` (with `Op` set to `in_back`) is spawned which will follow the `stin` gradient back. This behaviour can be coded as shown in Figure 4 (middle).

Given these two basic bricks, the `stout` and `stin` primitives would be encoded as in Figure 4 (bottom). For each, a tuple template, a spatial range and a validity time must be specified. `stout` implementation is concise, because it just needs to manifest itself trough a `wave` and make the tuple available; `stin`, instead, needs also to spawn a `hermes`, whose goal is to retrieve a tuple and move it to the tuple space where the operation was spawned.

[1] The interested reader can download an example clip at:
`http://apice.unibo.it/xwiki/bin/download/Publications/Coord2012/museum-small.avi`

Wave-form: a space-time gradient

```
wave(T,Range,Ttl) is wawe(T,Range,0,$this,$this,Ttl, 0)
wave(T,Range,D,Source,Ttl,Age) is wave(T,Range,D,Source,$this,Ttl,Age)
wave(T, Range, D, Dest, This, Ttl, Age) is (
  eval (Age>Ttl or D>Range)
    ? (in pre(T,D); in field(T,N))     % disappearing
    : rd source(T)
       ? (in-out(pre(T,0))             % default behaviour in a source
       : in pre(T,N) ? (eval N<D        % choosing minimum distance
                        ? out pre(T,N)
                        : ( in target(T,_); out target(T,Dest);
                            out pre(T,D))))
); finally (in pre(T,N) ? (            % consolidating target
      in field(T,M); out field(T,N);
      rd target(T, Dest); next wave(T,Range,D,Dest,This,Ttl,$delay);
      eval Age = 0 ? neigh wave(T,Range,N+$distance,This,Ttl,0)))
```

Tuple retrieval

```
hermes(Op, T, This) is
  eval This = $this
    ? (eval Op = in ? ( in T
        ? (rd target(op_in(T), Dest); neigh hermes(in_back, T, Dest))
        : (rd target(op_out(T), Dest); neigh hermes(in, T, Dest))))
    : (eval Op = in_back ? (in in_request(T)
         ? out(T)
         : (rd target(op_in(T), Dest); neigh hermes(in_back, T, Dest))))
```

Space-time Linda operations

```
stout(T,Range,Ttl) is out(T); wave(op_out(T), Range, Ttl)
stin(T, Range, Ttl) is out in_request(T);
                       wave(op_in(T), Range, Ttl);
                       hermes(in, T, $this)
```

Fig. 4. Definitions of Linda space-time operations

These new primitives allow agents to publish/retrieve information flexibly tuning the space-time horizons, relying on lower-level gradients (and routing paths) which adapt to the mobility of the network [2].

5 Related Work

Spatial Computing. The coordination model presented in this paper is very much in line with the motivations and basic mechanisms proposed in spatial computing research [6,5], and in particular by Proto [4]. Proto is a functional language used to specify the aggregate behaviour of all nodes in a space-filling network. It introduces specific space-time operators to situate computation in the physical world, and these operators form the inspiration for the space-time operators introduced in $\sigma\tau$-Linda. For example, there is a neighbourhood primitive `nbr` by which one can atomically compute an expression locally, spread

the result to neighbours, gather neighbours' messages previously sent, and return their collection. In Proto, the function computing a gradient data structure could be specified as:

```
(def distance-to (source)     % defining a unary function distance-to
  (rep d inf                  % d starts with value infinity
    (mux source 0             % d becomes 0 in the source, otherwise..
      (fold-hood* min inf     % d is the minimum value taken from
        (+ (nbr d) (nbr-range)) % neighbour's d plus neighbour's range
) ) ) )
```

As previously noted, the underlying execution on a node follows a cycle roughly similar to the one we use in Section 2 [27]. To achieve a similar expressiveness to Proto, we introduced the `next` and `neigh` constructs (playing a role similar to Proto's constructs `rep` and `nbr`), along with the space-time variables `#distance` and `#delay` (similar to Proto's constructs `nbr-range` and `dt`), and `finally`, which plays a role similar to Proto's *`-hood` constructs.

The main differences with respect to Proto are as follows: *(i)* in our model a node stores a tuple space, whereas in Proto only a fixed tuple of values is maintained, hence specific constructs to perform generative communication are lacking in Proto; *(ii)* being purely functional, Proto cannot easily deal with state transitions as typically required when programming coordination activities; and *(iii)* in Proto all nodes run the same program, which is assumed to be installed everywhere before computation starts (this is because the information to be exchanged and the structure of programs has to be known at design-time for construct `nbr` to properly work), whereas we assume nodes are initially empty, and computation starts from the run-time injection of activities by agents.

On the other hand, Proto provides functionalities that we neglected at this stage, though they are interesting for future works: Proto nodes can be programmed to move, a feature that could be interesting as a coordination metaphor for pervasive scenarios featuring physically-mobile devices; and Proto functions can be seen as operators applying to whole spatial structures and their behavior can be modified by changing the region of space on which they execute, a very important property for modularly building complex spatial computations.

It is also interesting to mention a trend in formal calculi for distribution converging to spatial computing. 3π was developed as an extension of π-calculus with the idea of modelling the space where processes execute as a 3-dimensional geometric space [11]. In 3π, each process has a position and an orientation in space (a *basis*), encoded in a so-called geometric data. Other than accessing it (symbolically), a process can also send or receive geometric data through channels and can evolve to new processes located elsewhere (i.e., movement). From 3π we inherited the idea of letting orientation vector of a node being accessible from a neighbour. An even more abstract approach is taken in the Ambient calculus [12] and its derivatives – like Brane Calculi [10] and P-systems [23] – in which processes execute in a spatial system of hierarchically nested compartments, which could be of interest as soon as one wants to considered the hierarchical structure of complex environments.

Traditional Coordination Models. Our approach relates to the idea of engineering the coordination space of a distributed system by some policy "inside" the tuple spaces, following e.g. the pioneer work of programmable tuple spaces like TuCSoN [22] or Mars [9]—and subsequent coordination frameworks such as those of *coordination artifacts* [20,19]. Though our coordination activities can be mapped to a certain extent on top of those fully-expressive programming models, we believe they are different in spirit in at least two ways: first, we foster the idea that agents inject the desired behaviour (which is not to be seen as a program for the space), and second, we push forward the idea of space-time computations which the above works typically neglect.

The KLAIM language and core calculus [18] extend the tuple-space concept with several notions that are related to our approach. KLAIM has a networked tuple-space model very similar to ours, since nodes host a tuple space, processes, and has interaction ability with a (virtual) neighbourhood; it also supports the idea of executing processes in a remote location, with a mechanism by which a process explicitly mentions the location of the action to be executed. Our approach differs in the use of broadcasts for node-to-node communication, in its ability of controlling temporal evolution and spatial location of a process continuation, and in the use of computation rounds for tuple spaces. It is an interesting future work to see to which extent KLAIM can be seen as a lower level model to describe our space-time activities, or vice versa.

The application example shown in Section 4.2 is also related to Geo-Linda [24], another example of spatial coordination approach combining the tuple manipulation of LINDA with the geometric addressing concepts of SPREAD [13]. In Geo-Linda, tuples are read and published over an assortment of geometric primitives, such as boxes, spheres, cylinders, and cones, all defined relative to a device. The language also introduces primitives to detect coarse movement of devices through the appearance or disappearance of tuples.

Self-organisation in Tuple Spaces. As described in [21], applications of coordination models and languages – and especially space-based ones – are inevitably entering the realm of self-organisation, where complexity of interactions becomes the key to make desired properties appear by emergence. Given the intrinsic difficulty of *designing emergence*, most approaches mimic nature-inspired techniques to organise and evolve tuples according to specified rules.

Among the many existing approaches, one that is very related to ours is TOTA (Tuples On The Air) [17], a tuple-based middleware supporting field-based coordination for pervasive-computing applications. In TOTA each tuple, when inserted into a node of the network, is equipped with a content (the tuple data), a diffusion rule (the policy by which the tuple has to be cloned and diffused around) and a maintenance rule (the policy whereby the tuple should evolve due to events or time elapsing). Compared with the language proposed here, and although TOTA was an evident inspiration to the idea of building dynamic and distributed structures of tuples, we observe a number of differences: *(i)* TOTA is a middleware and defines no true language or primitives to program spatial structures (content and maintenance rule are programmed directly in Java and

can access and manipulate the whole tuple space); *(ii)* TOTA has no specific mechanisms to keep track of physical space and time, for it only has a concept of "spreading to the neighbourhood", which allows to estimate distance in terms of number of hops to the source. TOTA could be possibly used as an underlying framework for implementing our language, provided additional ability to perceive the physical world are added.

A chemical-inspired self-organisation model is instead studied in [28,29]. There, tuples are associated with an *activity level*, which resembles chemical concentration and measures the extent to which the tuple can influence the state of system coordination—e.g., a tuple with low activity level would be rather inert, hence taking part in coordination with very low frequency. Chemical-like reactions following the CTMC model, properly installed into the tuple space, evolve activity level of tuples over time in the same way chemical concentration is evolved in chemical systems, and provide a diffusion mechanism that is shown to provide spatial notions like gradients as well. The SAPERE approach in [30] adds to this model the notion of semantic matching and tailors it to the pervasive computing context. We believe that, as density and speed of nodes grows, our language can be used to approximate the behaviour of those chemical rules.

6 Conclusions and Future Work

The current trend in ICT will shortly bring us distributed systems of huge size, density, mobility and opennes. Following the direction of a good deal of recent works – including [17,6,11] and many others – we claim that this will require to elect the notion of "spatial coordination" as first-class abstraction in coordination models and languages, and distributed systems in general. The present paper is a first exploration in the direction of filling the gap between Linda-based and spatial computing models, obtained by a coordination model incorporating – though in an innovative guise – mechanisms for the space-time situation of processes [4], used to realise adaptive coordination mechanisms. We argue that the proposed language can be rather easily implemented on top of those existing coordination middleware providing basic features of space-to-space interaction and space programmability, such as TuCSoN [22], Klava [7] and TOTA [17]. We also plan to implementation further case studies of self-organisation, according e.g. to the pattern-based approaches in [15,14].

Another interesting thread of future research activities will be devoted to clarify what would be a good notion of expressiveness, and what would be a minimal set of primitives for fully-expressive space-time computation—a problem already stated in [3] for the spatial computing settings. Accordingly, we plan to use the presented language to define basic calculi in the style of the one presented here, which would be able to model higher-level languages like, e.g., the eco-law language for pervasive service ecosystems [30], and paving the way towards formal methods for the predictability and control of emergent adaptation in collective systems.

Acknowledgments. This work has been supported by the EU FP7 project "SAPERE - Self-aware Pervasive Service Ecosystems" under contract No. 256873.

References

1. Bachrach, J., Beal, J., Fujiwara, T.: Continuous space-time semantics allow adaptive program execution. In: IEEE SASO 2007, New York, pp. 315–319. IEEE (July 2007)
2. Beal, J.: Flexible self-healing gradients. In: Proceedings of the 2009 ACM Symposium on Applied Computing, SAC, pp. 1197–1201. ACM (2009)
3. Beal, J.: A basis set of operators for space-time computations. In: Self-Adaptive and Self-Organizing Systems Workshop (SASOW 2010), pp. 91–97 (September 2010)
4. Beal, J., Bachrach, J.: Infrastructure for engineered emergence on sensor/actuator networks. IEEE Intelligent Systems 21(2), 10–19 (2006)
5. Beal, J., Dulman, S., Usbeck, K., Viroli, M., Correll, N.: Organizing the aggregate: Languages for spatial computing. CoRR, abs/1202.5509 (2012)
6. Beal, J., Michel, O., Schultz, U.P.: Spatial computing: Distributed systems that take advantage of our geometric world. ACM Transactions on Autonomous and Adaptive Systems 6, 11:1–11:3 (2011)
7. Bettini, L., Nicola, R.D., Pugliese, R.: Klava: a java package for distributed and mobile applications. Softw., Pract. Exper. 32(14), 1365–1394 (2002)
8. Busi, N., Gorrieri, R., Zavattaro, G.: On the expressiveness of Linda coordination primitives. Inf. Comput. 156(1-2), 90–121 (2000)
9. Cabri, G., Leonardi, L., Zambonelli, F.: MARS: A programmable coordination architecture for mobile agents. IEEE Internet Computing 4(4), 26–35 (2000)
10. Cardelli, L.: Brane Calculi. Interactions of Biological Membranes. In: Danos, V., Schachter, V. (eds.) CMSB 2004. LNCS (LNBI), vol. 3082, pp. 257–278. Springer, Heidelberg (2005)
11. Cardelli, L., Gardner, P.: Processes in Space. In: Ferreira, F., Löwe, B., Mayordomo, E., Mendes Gomes, L. (eds.) CiE 2010. LNCS, vol. 6158, pp. 78–87. Springer, Heidelberg (2010)
12. Cardelli, L., Gordon, A.D.: Mobile ambients. Theoretical Computer Science 240(1), 177–213 (2000)
13. Couderc, P., Banatre, M.: Ambient computing applications: an experience with the spread approach. Hawaii International Conference on System Sciences, HICSS 2003 (January 2003)
14. Fernandez-Marquez, J.L., Di Marzo Serugendo, G., Montagna, S., Viroli, M., Arcos, J.L.: Self-organising design patterns. Natural Computing (to appear, 2012)
15. Gardelli, L., Viroli, M., Omicini, A.: Design Patterns for Self-organising Systems. In: Burkhard, H.-D., Lindemann, G., Verbrugge, R., Varga, L.Z. (eds.) CEEMAS 2007. LNCS (LNAI), vol. 4696, pp. 123–132. Springer, Heidelberg (2007)
16. Gelernter, D.: Generative communication in Linda. ACM Trans. Program. Lang. Syst. 7(1), 80–112 (1985)
17. Mamei, M., Zambonelli, F.: Programming pervasive and mobile computing applications: The tota approach. ACM Trans. Softw. Eng. Methodol. 18(4), 1–56 (2009)
18. Nicola, R.D., Ferrari, G.L., Pugliese, R.: Klaim: A kernel language for agents interaction and mobility. IEEE Trans. Software Eng. 24(5), 315–330 (1998)

19. Omicini, A., Ricci, A., Viroli, M.: An algebraic approach for modelling organisation, roles and contexts in MAS. Applicable Algebra in Engineering, Communication and Computing 16(2-3), 151–178 (2005)
20. Omicini, A., Ricci, A., Viroli, M.: Coordination Artifacts as First-Class Abstractions for MAS Engineering: State of the Research. In: Garcia, A., Choren, R., Lucena, C., Giorgini, P., Holvoet, T., Romanovsky, A. (eds.) SELMAS 2005. LNCS(LNAI), vol. 3914, pp. 71–90. Springer, Heidelberg (2006)
21. Omicini, A., Viroli, M.: Coordination models and languages: From parallel computing to self-organisation. The Knowledge Engineering Review 26(1), 53–59 (2011); Special Issue 01 (25th Anniversary Issue).
22. Omicini, A., Zambonelli, F.: Coordination for Internet application development. Autonomous Agents and Multi-Agent Systems 2(3), 251–269 (1999)
23. Paun, G.: Membrane Computing: An Introduction. Springer-Verlag New York, Inc., New York (2002)
24. Pauty, J., Couderc, P., Banatre, M., Berbers, Y.: Geo-linda: a geometry aware distributed tuple space. In: IEEE 21st International Conference on Advanced Networking and Applications (AINA 2007), pp. 370–377 (May 2007)
25. Pianini, D., Montagna, S., Viroli, M.: A chemical inspired simulation framework for pervasive services ecosystems. In: Proceedings of the Federated Conference on Computer Science and Information Systems, pp. 667–674. IEEE Computer Society Press (2011)
26. Singh, A., Ramakrishnan, C.R., Smolka, S.A.: A process calculus for mobile ad hoc networks. Sci. Comput. Program. 75(6), 440–469 (2010)
27. Viroli, M., Beal, J., Casadei, M.: Core operational semantics of Proto. In: 26th Annual ACM Symposium on Applied Computing, SAC 2011, Tunghai University, TaiChung, Taiwan, March 21-25. ACM (2011)
28. Viroli, M., Casadei, M.: Biochemical Tuple Spaces for Self-organising Coordination. In: Field, J., Vasconcelos, V.T. (eds.) COORDINATION 2009. LNCS, vol. 5521, pp. 143–162. Springer, Heidelberg (2009)
29. Viroli, M., Casadei, M., Montagna, S., Zambonelli, F.: Spatial coordination of pervasive services through chemical-inspired tuple spaces. ACM Transactions on Autonomous and Adaptive Systems 6(2), 14:1–14:24 (2011)
30. Viroli, M., Pianini, D., Montagna, S., Stevenson, G.: Pervasive ecosystems: a coordination model based on semantic chemistry. In: Ossowski, S., Lecca, P., Hung, C.-C., Hong, J. (eds.) 27th Annual ACM Symposium on Applied Computing, SAC 2012, Riva del Garda, TN, Italy, March 26-30. ACM (2012)