Available online at www.sciencedirect.com
SCIENCE DIRECT

Electronic Notes in Theoretical Computer Science 97 (2004) 97–124

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

On the Semantics of Coordination Models for Distributed Systems: The LogOp Case Study

Ronaldo Menezes[a,1] Andrea Omicini[b,2] Mirko Viroli[b,3]

[a] *Florida Institute of Technology,
150 West University Blvd., Melbourne, FL 32901, USA*

[b] *DEIS, Università degli Studi di Bologna
via Venezia 52, 47023 Cesena, Italy*

Abstract

LogOp is a coordination model extending Linda by allowing a single coordination operation to dynamically address a multiplicity of possibly distributed tuple spaces. The design of LogOp raises relevant issues that are of general interest in the fields of coordination and distributed systems engineering. In particular, the design of an infrastructure that supports the enactment of coordination laws involving a multiplicity of physically distributed tuple spaces demands a careful treatment of the aspects related to synchrony, atomicity, locality of interactions, and global interpretation of coordination rules.
In this paper we elaborate on these general issues, starting from the study of the semantics of LogOp. First of all, the LogOp coordination model is introduced as an extension of Linda. Then, two different semantics, both conforming to the LogOp informal specification, are formally described and compared. Finally, the limitations of the traditional approach to the formal characterisation of coordination (coordination as a language) are pointed out, and addressed through a different approach (coordination as a service) whose benefits are illustrated by suitably re-formulating LogOp semantics. On the one hand, this provides crucial hints on how the LogOp coordination model could be deployed as an interactive service provided by a coordination infrastructure for distributed systems. On the other hand, the above results allow some general aspects of coordination in open and distributed systems to be clearly pointed out and adequately discussed.

Keywords: Coordination Models, Linda, distributed systems, formal models, software infrastructure

[1] Email: rmenezes@cs.fit.edu
[2] Email: aomicini@deis.unibo.it
[3] Email: mviroli@deis.unibo.it

1 Introduction

A key role in the design of infrastructures for today's and tomorrow's distributed systems is played by coordination models and languages, whose goal is to provide means for governing and ruling interactions in complex systems. Linda [15] is probably one of the best known examples of such models: tuple spaces playing the role of blackboards are used as a means to mediate the interactions between different entities, promoting temporal and spatial decoupling. Whereas it originated in the field of closed, parallel systems, the main features and characteristics of Linda are nowadays exploited in infrastructures — such as JavaSpaces [14], TuCSoN [23,20], Mars [8], and Lime [21] — aimed at tackling the complexity of interactions in open, distributed, and highly dynamic systems. It is clear that the quest for studying new solutions in the field of coordination models is still a main challenge, whose impact on the development of successful technologies will be increasingly relevant.

In this article we study LogOp [22], a coordination model extending basic Linda by adding the ability to invoke coordination primitives involving more than one tuple space at a time. By this feature, coordination rules are no longer restricted to the context of a single tuple space, but can be defined on-the-fly by transiently forming extended scopes of visibility, obtained by joining the scope of different and possibly distributed tuple spaces altogether.

However, the importance of this new model does not only reside in this enhanced expressiveness and flexibility. Relevant issues related to governing interactions in distributed systems are explicitly raised by the design of LogOp, which are of general interest in the fields of coordination and of distributed systems engineering. In particular, the very idea of expressing a coordination law involving physically distributed tuple spaces demands a careful treatment of the aspects related to synchrony, atomicity, locality of interactions, and global interpretation of coordination rules. Indeed, these issues are crucial in the more general context where the infrastructure is made of a distributed coordination space, populated by mutually interacting coordination abstractions.

In this paper, we elaborate on these general issues, by taking LogOp as a case study, and discussing its possible semantics. Inspired by the traditional application for building closed and parallel systems, in Section 2 we start by defining a formal model to Linda featuring a multiplicity of tuple spaces, each separately accessible by the invocation of a coordination primitive. Section 3 points out some of the limitations of the Linda model, and shows how LogOp — defined in term of its syntax and informal semantics (as they appear in [22]) — is meant to address them.

Section 4 formally discusses the semantics of LogOp by extending the formalisation of Linda reported in Section 2, and provides an example of a typical problem concerning the semantics of coordination models for distributed systems. In fact, two different interpretations are shown to be compatible with the informal semantics introduced in Section 3. In particular, the most straightforward extension of Linda towards LogOp, referred to as *strong-synchrony semantics*, features the atomic execution of coordination primitives. We observe that in the context of distributed systems, however, preserving atomicity of the operations that involves physically distributed tuple spaces may require a quite challenging infrastructural support, including e.g. locking and recovery mechanisms typical of transactions management. By relaxing this strong assumption we develop a semantics more plausible for an implementation over distributed systems, called *weak-synchrony semantics*. In this case, an operation is perceived as atomic only by the entity that executes it, while it is globally seen as a sequence of standard Linda operations. The two semantics are compared, and some general considerations are derived.

The semantic approach we exploit up to this point is the one traditionally used in the coordination field [3]: it is promoted by the traditional application of coordination models as languages for building the interactive part of applications, and is consequently based on the same framework used for specifying semantics of concurrent languages. On the other hand, the most interesting applications of coordination models — in today's computer systems — is to define coordination infrastructures for distributed scenarios. There, coordination is not easily seen as a concurrent language — e.g. supported by a compiler [26] — but rather, as a service provided by a number of *coordination media* [10], such as e.g. Linda-like tuple spaces. This viewpoint calls for a different semantic model, focusing on the *coordination medium* abstraction and its interactive behaviour. In [25,24], the differences between the two viewpoints are motivated, compared with the many related works, and technically analysed in detail, showing that they lead to two different semantic frameworks for coordination. In particular, the former is defined as the framework of *coordination as a language*, while the latter is referred to as the framework of *coordination as a service*. The most notable difference here is that viewing coordination as a service promotes the explicit representation of a number of run-time aspects, which quite crucially impact on the deployment of a coordination model as an interactive service provided by an infrastructure for distributed systems.

Accordingly, starting from the weak-synchrony semantic model of LogOp, Section 5 derives a formulation of LogOp as a coordination service provided by an infrastructure. Such a new formulation demonstrates the implications of using the two different semantic frameworks, but also provides a significant development of the LogOp specification. In fact, from the semantics of the LogOp primitives — providing an abstract specification of LogOp — a new model is derived that focuses on the LogOp interactive behaviour. In turn, this model can easily be interpreted as a design specification for a LogOp infrastructure, as it clearly identifies the different coordination abstractions involved, their roles in the whole coordination process, and the management of their mutual interaction. Some issues that would be crucial to the implementation of a LogOp service are hence pointed out and emphasised, which the traditional framework would have easily left underspecified.

Even more, the contributions of this paper are not limited to the field of coordination. Instead, this paper is also meant to provide some significant clues about the sound development of today's computer systems in general. Here, in fact, we go beyond the mere recognition that formal models are needed to non-ambiguously specify and correctly understand the behaviour of complex systems – and of coordination infrastructures in particular. What we suggest in this work is that formal models should be also used for the design of coordination infrastructures, and that a suitable formalism could actually make abstract specifications drive infrastructure development toward sound and effective implementations.

2 The Semantics of Linda

Linda is a coordination model providing primitives that enable processes to store and retrieve tuples from tuple spaces. In the very basic Linda model [15], processes use the primitive `out` to store tuples and the primitives `in` and `rd` to retrieve tuples. In particular, the primitives `in` and `rd` take a template and use associative matching to select a tuple to return — `in` removes the matching tuple, while `rd` takes a copy of it. Both `in` and `rd` are blocking primitives. That is, if a matching tuple is not found in the tuple space, the process executing the primitive blocks until a matching tuple is actually placed in the tuple space and can be retrieved. In order to keep the article short yet self-contained, we consider the subset of Linda which is generally used in the context of formal semantics: in particular, we consider only primitives `out`, `rd`, and `in`, and we abstract away from the tuple matching mechanism, assuming that primitives `in` and `rd` accept the exact tuple to be looked for. However, to pave the way towards the complex problems of coordination in distributed systems, we model Linda with multiple tuple spaces, each one characterised by a unique identifier $i \in I$, and individually accessible by every Linda primitive through its identifier.

In the traditional approach to this kind of formalisation, such as [5], Linda is interpreted as a language for building closed, parallel applications (which used to be the first application of the LINDA model). Therefore, the behaviour of LINDA is given by providing an operational semantics to this language, which describes the possible evolution of a coordinated system where interactions are expressed in terms of the LINDA coordination language. Coordinated entities can be modelled as finite, non-deterministic processes sequentially executing some LINDA coordination primitives. Formally, the syntax of these processes P can be expressed using a CCS-like notation [18] as follows:

$$\pi ::= \mathit{out} \mid \mathit{in} \mid \mathit{rd}, \qquad \beta ::= \pi(i,x), \qquad P ::= 0 \mid \beta \mid P;P \mid P+P$$

0 denotes the void (or terminated) process, β denotes the process performing a primitive operation $\pi(i,x)$ — executing primitive π on tuple space (with identifier) i, involving tuple x — $P;P'$ is the sequential composition of P and P', $P+P'$ is the non-deterministic choice between P and P'. In the remainder of this paper, operator ; is given higher priority than +, so that by $P;P'+P''$ we actually mean $(P;P')+P''$ — other ambiguities will be resolved by parenthesis as usual. For instance, the process $\mathit{out}(i_1,x_1);\ \mathit{rd}(i_2,x_2);\ 0 + \mathit{in}(i_3,x_3);\ 0$ is the process that non-deterministically chooses between either *(i)* sequentially performing operation $\mathit{out}(i_1,x_1)$ and $\mathit{rd}(i_2,x_2)$, or *(ii)* performing operation $\mathit{in}(i_3,x_3)$ only.

We write $\sum_{i\in I} P_i$ with $I = \{a,b,c,\ldots\}$ as a shorthand for $P_a + P_b + P_c + \ldots$, and assume $\sum_{i\in\{\}} P_i$ denotes the process 0. Then, we assume that the following congruence rules hold:

$$0 + P \equiv P \qquad P + P' \equiv P' + P \qquad (P + P') + P'' \equiv P + (P' + P'')$$
$$0;P \equiv P \qquad P;0 \equiv P \qquad (P + P');P'' \equiv (P;P'') + (P';P'')$$

That is, we consider two different processes P and P' as being the same if from the above rules one can infer $P \equiv P'$. Because of such rules, a non-void process can always be written either as a summation $\sum_{j\in J}(\beta_j;P_j)$ or in the form $(\beta;P) + P'$. A LINDA configuration $L \in \mathcal{L}$, representing the state of a LINDA system at a given time, is syntactically denoted by syntax

$$L ::= 0 \mid \langle i,x\rangle \mid P \mid (L \parallel L)$$

where symbol $\parallel$ is used for parallel composition as usual. The following congruence rules are also assumed to hold:

$$0 \parallel L \equiv L \qquad L \parallel L' \equiv L' \parallel L \qquad (L \parallel L') \parallel L'' \equiv L \parallel (L' \parallel L'')$$

Thus, each element L is a finite composition of processes and items of the kind $\langle i, x\rangle$, denoting a tuple x occurring in tuple space i. An operational semantics can be assigned to this language by a transition system $\langle L, \longrightarrow\rangle$, where transition relation $\longrightarrow$ is of the kind $\longrightarrow \subseteq L \times L$. As usual, syntax $L_0 \longrightarrow L_1$ is used as a shorthand for $\langle L_0, L_1\rangle \in \longrightarrow$, and means that from configuration L_0 the system may move to configuration L_1. Relation $\longrightarrow$ is defined as the smallest relation satisfying the rules:

$$\begin{array}{lll} out(i,x); P + P' \parallel L & \longrightarrow P \parallel L \parallel \langle i,x\rangle & \text{[L-OUT]} \\ rd(i,x); P + P' \parallel L \parallel \langle i,x\rangle & \longrightarrow P \parallel L \parallel \langle i,x\rangle & \text{[L-RD]} \\ in(i,x); P + P' \parallel L \parallel \langle i,x\rangle & \longrightarrow P \parallel L & \text{[L-IN]} \end{array}$$

These can be considered the safety conditions of LINDA: `out` emits the tuple on the tuple space, `rd` waits for the tuple occurring in the tuple space, and `in` waits for the tuple and then removes it. Additionally, these rules also give semantics to choice and sequential composition: they state that whenever a primitive operation is executed any other choice (P') is excluded, while the process sequentially composed to the executed primitive (P) is allowed to carry on.

3 LogOp

LINDA is without a doubt not only the first, but also the most successful coordination model: its main asset is the balance that it achieves between simplicity and expressiveness. Simplicity is guaranteed thanks to a limited set of primitives, while expressiveness is obtained as these few primitives can be used to model a vast collection of communication and synchronisation patterns. Despite its expressiveness, LINDA has been traditionally used as a core model for a number of extensions tackling issues raised by modern applications, featuring for instance programmable behaviour (ReSpecT [19]), transactions and expiring tuples (JavaSpaces [14]), Prolog-like features (Shared Prolog [1]), scalability (SwarmLinda [17]) and mobility (LIME [21]) to mention but a few.

A particularly relevant class of extensions has been developed to deal with the new application domain of coordination models, which moved from closed, centralised environments — the main early application of LINDA — to open, distributed environments — the most common framework for today's computer systems. There, it is sensible to consider coordination as provided by a number of different tuple spaces. Extensions of the LINDA model with multiple tuple spaces include, besides JavaSpaces, SwarmLinda, and LIME mentioned above, also others such as TuCSoN [20], PEERSPACES [6], and Bauhaus Linda [9].

In particular, in this new setting, the blocking characteristic of `in` and `rd` and the fact that these primitives can only deal with one tuple space at a time may hinder the ability of LINDA to express complex coordination patterns. More precisely, the primitives in LINDA force processes to see tuple spaces as disjoint: processes that require simultaneous access to several tuple spaces have to do so by serialising the operations — i.e, by accessing one tuple space after the other. So, tuple spaces are indeed disjoint and do exist to form different scopes for tuples, identified at design-time. However, it may happen that a different scope is required, possibly involving more than one space, and that such a scope needs to be formed dynamically (on-the-fly) and then discarded.

The ability to combine tuple spaces at any time, forming a new scope for a single operation, is the main motivation behind Snyder and Menezes's proposal of the LOGOP coordination model [22]. More specifically, LOGOP aims at improving the expressiveness of LINDA primitives, yet trying to achieve good performance at the implementation level as a consequence of expressiveness. While observing that the original LINDA model lacks the ability to express coordination that involves *simultaneous* access to two or more tuple spaces, it was also noticed that the associations amongst the tuple spaces that are normally required, relate to basic logical operators such as AND, OR, and XOR. The syntax of the LOGOP primitives we consider in this paper is as follows:

```
<LogOp_PRIMITIVE> ::= <PRIM_NAME>(<OP>(ts_id,..,ts_id), tuple)
<PRIM_NAME> ::= in | rd | out
<OP> ::= AND | OR | XOR
```

where `ts_id` is a tuple space identifier (or in general, a handler for the tuple space). Logical operators are used in LOGOP primitives to specify the target tuple spaces: for instance, primitive `out(`AND(i_A,i_B)`,x)` specifies that a tuple `x` has to be inserted in both i_A and i_B, `rd(`OR(i_A,i_B)`,x)` that a tuple matching `x` has to be read from either i_A or i_B (or both), and `in(`XOR(i_A,i_B)`,x)` that one tuple matching `x` has to be removed from either i_A or i_B, but not from both.

Informally, the OR operator has the effect of combining tuple spaces so that processes can store and retrieve tuples from any of the specified tuple spaces without having to impose an order on the way the tuple spaces are accessed. The OR operator adds another level of non-determinism to the model: the tuple will be stored in *some* of the tuple spaces defined in the list, that is, non-deterministically in one, more, or all tuple spaces in the list.

The XOR operator is similar to OR, but in this case the tuple can be inserted, read, or withdrawn from one and only one tuple space in the specified list. This operator is useful when the client must be guaranteed that no more than one tuple is actually retrieved or removed from the tuple spaces specified in the list. Another level of non-determinism is also added here since the tuple space from which the tuple will be retrieved is not previously known.

Finally, the AND operator allows processes to consider *all* tuple spaces in a list. The use of AND with an `out` allows processes to store a tuple in a list of tuple spaces in just one step. Contrast this with the LINDA case where if n tuple spaces are involved in the operation, the `out` primitive would have to be executed n times. In the case of the blocking primitives `in` and `rd`, the semantics of AND is such that the process will block if one or more tuple spaces in the list fail to contain a tuple matching the template given in the primitive.

4 The semantics of LogOp

In the literature, an informal description of the semantics of a coordination model like the one above has often been considered as acceptable, even sufficient to provide a specification suitable for an implementation. However, a traditional issue in the field of concurrent systems in general — and coordination models in particular — is the intrinsic inability of informal descriptions to give a precise account of the properties of a system of interest. For instance, in [4] three different semantics are described for the `out` primitive of LINDA — namely, instantaneous, ordered, and unordered — that all accomplish to existing informal specifications of the model. Even more, in [7] Busi et al. go beyond and show how informal specifications can even lead to incorrect systems, as in the case of serialisability of transactions in the current design of the JavaSpaces coordination model [14].

LOGOP represents a perfect case study for this issue — which is indeed of general interest for most non-trivial coordination models. Accordingly, in the following we provide a formal semantics for LOGOP that precisely accounts for the effect of executing a primitive operation on the tuple spaces and the possible dynamics due to concurrent accesses. Indeed, we show that different semantics can be actually provided that match the informal description of LOGOP, as presented in [22] and summarised above, and that different semantics would lead to implementations with different key properties. In particular, we show that the model obtained by naturally extending LINDA (Subsection 4.1) is not well-suited to the end of designing a LOGOP coordination infrastructure for distributed systems. So (Subsection 4.2), we proceed by providing a more general semantics model in order to cope with this issue, compare the two different approaches (Subsection 4.3) and finally devise out some general considerations.

4.1 Strong-Synchrony Semantics

We extend the semantics of LINDA provided in Section 2 to the case of LOGOP, where a multiplicity of tuple spaces can be associated together by logical operators. In this model, similarly to the case of LINDA, the execution of a primitive operation is still considered as just one single event in the system, whose effect is to atomically change the (distributed) state of the coordination space. As a result, the execution of a coordination primitive is perceived atomically by all coordinated entities in the domain. We therefore refer to this semantics of LOGOP as strong-synchrony semantics. The abstract syntax that we use in this formalisation to represent a primitive operation α (which is a generalisation of a LINDA operation β) is now of the kind:

$$\alpha ::= \pi(\lambda, x), \qquad \lambda ::= i \wedge \ldots \wedge i \ \mid\ i \otimes \ldots \otimes i \ \mid\ i \vee \ldots \vee i$$

Here, λ is a logical expression, where n-ary logical operators AND ($\wedge$), XOR ($\otimes$), and OR ($\vee$) are applied to tuple space identifiers ranging in set $\mathcal{T}$.

Given a finite set of tuple space identifiers $I = \{i_0, i_1, \ldots i_n\} \subseteq \mathcal{T}$, the operation $\pi(i_0 \wedge i_1 \wedge \ldots \wedge i_n, x)$ can also be written as $\pi(\bigwedge_{i \in I} i, x)$, and means that primitive π has to be executed on all tuple spaces in set I (logic conjunction AND). Similarly, $\pi(i_0 \otimes i_1 \otimes \ldots \otimes i_n, x)$ — equivalently written as $\pi(\bigotimes_{i \in I} i, x)$ — means that primitive π has to executed on exactly one of the tuple spaces in I (exclusive logic disjunction XOR). Finally, $\pi(i_0 \vee i_1 \vee \ldots \vee i_n, x)$ — or equivalently $\pi(\bigvee_{i \in I} i, x)$ — means that primitive π has to be executed on at least one of the tuple spaces in I (inclusive logic disjunction OR). A configuration for a LOGOP system $S \in \mathcal{S}$ is equivalent to the case of LINDA model:

$$S ::= 0 \ \mid\ \langle i, x \rangle \ \mid\ P \ \mid\ (S \,\|\, S)$$

Given a finite set of tuple space identifiers $I = \{i_0, \ldots, i_n\}$, we write $\prod_{i \in I} \langle i, x \rangle$ as a shorthand for $\langle i_0, x \rangle \ \| \ \ldots \ \| \ \langle i_n, x \rangle$, with $\prod_{i \in \{\}} \langle i, x \rangle$ naturally meaning void configuration 0.

In order to define the semantics of LOGOP as a simple extension of the case of LINDA, we first introduce a relation σ between operations and tuples, so that $\sigma(\alpha, S)$ associates to an operation α the set of tuples (along with the identifier of their tuple space) that may be involved in its execution, i.e., the tuples inserted by an `out`, read by a `rd`, or removed by a `in`. This relation is defined as the smaller one satisfying the rules:

$$\sigma(\pi(\bigwedge_{i\in I} i, x), \prod_{i\in I}\langle i, x\rangle) \qquad \text{[S-AND]}$$

$$\sigma(\pi(\bigotimes_{i\in I} i, x), \langle i_0, x\rangle) \qquad \text{for any } i_0 \in I \qquad \text{[S-XOR]}$$

$$\sigma(\pi(\bigvee_{i\in I} i, x), \prod_{i'\in I'}\langle i', x\rangle) \qquad \text{for any } I' \text{ so that } \{\} \subset I' \subseteq I \qquad \text{[S-OR]}$$

As reported by the informal specification, operator AND involves tuples in all the tuple spaces in the specified list, operator XOR in just one tuple space, while operator OR in any non-void subset of the set of tuple spaces. Then, the semantics of LogOp is given by transition system $\langle \mathcal{S}, \longrightarrow\rangle$, where the transition relation $\longrightarrow \subseteq \mathcal{S}\times\mathcal{S}$ is defined by rules:

$$out(\lambda, x); P + P' \parallel S \longrightarrow P \parallel S \parallel S' \qquad \text{if } \sigma(out(\lambda, x), S') \qquad \text{[LG-OUT]}$$

$$rd(\lambda, x); P + P' \parallel S \parallel S' \longrightarrow P \parallel S \parallel S' \qquad \text{if } \sigma(rd(\lambda, x), S') \qquad \text{[LG-RD]}$$

$$in(\lambda, x); P + P' \parallel S \parallel S' \longrightarrow P \parallel S \qquad \text{if } \sigma(in(\lambda, x), S') \qquad \text{[LG-IN]}$$

These rules directly extends rules [L-OUT], [L-RD] and [L-IN], reported in Section 2, to the case of LogOp, respectively inserting, reading and withdrawing the set of tuples obtained through relation σ. As an example, valid transitions of LogOp configurations include the following:

$$out(id_1 \vee id_2 \vee id_3, x) \longrightarrow \langle id_1, x\rangle \parallel \langle id_3, x\rangle$$

$$rd(id_1 \otimes id_2 \otimes id_3, x) \parallel \langle id_1, x\rangle \longrightarrow \langle id_1, x\rangle$$

$$in(id_1 \wedge id_2, x) \parallel \langle id_1, x\rangle \parallel \langle id_2, x\rangle \longrightarrow 0$$

Notice, that semantics is given to logic formulae λ only by means of relation σ. Therefore, since $\pi(\bigwedge_{i\in\{i_0\}} i, x)$, $\pi(\bigvee_{i\in\{i_0\}} i, x)$, and $\pi(\bigotimes_{i\in\{i_0\}} i, x)$ are associated by σ to the same item $\langle i_0, x\rangle$, then we naturally denote their three λ by the shorthand i_0. An operation involving a λ of this kind is called a *mono-space* operation, which can be easily shown to have the same semantics of Linda coordination model as described in Section 2. Other operations are here called *multi-space*.

4.2 Weak-Synchrony Semantics

Clearly, strong-synchrony is an unrealistic assumption in distributed systems, that would require an implementation supported by transactional mechanisms — this would hardly fit the openness, dynamism, and decoupling requirements mandated by today's distributed systems. An alternative semantics model for LogOp can be defined that satisfies the informal specification, but where this constraint is released. In particular, the execution of a primitive can be modelled as a sequence of events in the system, each corresponding to a mono-space Linda operation. So, while the entity invoking a multi-space operation still sees its execution as atomic, other entities of the system may actually perceive non-consistent, partial configurations of the whole multi-space, distributed system. This amounts to what we call the *weak-synchrony* semantics of LogOp. Since this new model still satisfies the informal specification of LogOp — which in fact only considers the viewpoint of the single process executing a primitive — the advantages expected by LogOp can be maintained by an implementation conforming to this semantics, which is in fact more easily realisable.

In order to define this new formal model so that it be easily comparable to strong-synchrony, we define it in the form of a translation from LogOp configurations to LogOp configurations, that is, as a function from $\mathcal{S}$ to $\mathcal{S}$. Informally, this function translates processes executing multi-space operations into processes that non-deterministically choose between different sequences of mono-space operations. The final effect of each of these sequences is the same expected by the corresponding multi-space operation, however any execution of one such sequence is not atomic and can be perturbed by some other coordinated entity interacting with the same tuples. At the top-level, this encoding is defined as a function $\Big|.\Big|_{\mathcal{S}}:\mathcal{S}\mapsto\mathcal{S}$ defined as:

$$\Big|0\Big|_{\mathcal{S}} \triangleq 0, \quad \Big|\langle i,x\rangle\Big|_{\mathcal{S}} \triangleq \langle i,x\rangle$$
$$\Big|S \parallel S'\Big|_{\mathcal{S}} \triangleq \Big|S\Big|_{\mathcal{S}} \parallel \Big|S'\Big|_{\mathcal{S}}, \quad \Big|\alpha;P+P'\Big|_{\mathcal{S}} \triangleq \Big|\alpha\Big|_{\mathcal{O}};\Big|P\Big|_{\mathcal{S}}+\Big|P'\Big|_{\mathcal{S}}$$

This function leaves the structure of a configuration unchanged, but translates operations α into processes P by the encoding $\big|.\big|_{\mathcal{O}}$ inductively defined as:

$$\begin{aligned}
&\big|\pi(\epsilon, x)\big|_{\mathcal{O}} \triangleq 0 && \text{[W-EMPTY]}\\
&\big|\pi(\bigwedge_{i\in I} i, x)\big|_{\mathcal{O}} \triangleq \sum_{i\in I}\Big(\pi(i,x); \big|\pi(\bigwedge_{i'\in I\setminus\{i\}} i', x)\big|_{\mathcal{O}}\Big) && \text{[W-AND]}\\
&\big|\pi(\bigotimes_{i\in I} i, x)\big|_{\mathcal{O}} \triangleq \sum_{i\in I}\pi(i,x) && \text{[W-XOR]}\\
&\big|\pi(\bigvee_{i\in I} i, x)\big|_{\mathcal{O}} \triangleq \big|\pi(\bigotimes_{i\in I} i, x)\big|_{\mathcal{O}} + \sum_{i\in I}\Big(\pi(i,x); \big|\pi(\bigvee_{i'\in I'\setminus\{i\}} i', x)\big|_{\mathcal{O}}\Big) && \text{[W-OR]}
\end{aligned}$$

The first rule deals with the case where the logical expression involves zero tuple spaces — represented by notation ϵ —, in which case we obtain the void process. Operator $\wedge$ is defined so as to non-deterministically choose between the execution on any tuple space in the set I, and then carry on with the corresponding remaining tuple spaces, recursively. Operator $\otimes$ is defined so as to non-deterministically choose just one of the tuple spaces in the set I for primitive execution. Finally, operator $\vee$ is defined so as to non-deterministically choose between the execution on any tuple space in the set I; After that, the process can either terminate (left choice) or keep recursively executing the operation on the remaining tuple spaces (right choice). For instance we have the following mappings:

$$\begin{aligned}
\big|out(i_1 \wedge i_2, x)\big|_{\mathcal{O}} &= out(i_1,x); out(i_2,x) + out(i_2,x); out(i_1,x)\\
\big|in(i_1 \otimes i_2, x)\big|_{\mathcal{O}} &= in(i_1,x) + in(i_2,x)\\
\big|rd(i_1 \vee i_2, x)\big|_{\mathcal{O}} &= rd(i_1,x) + rd(i_1,x); rd(i_2,x) +\\
&\quad\; rd(i_2,x) + rd(i_2,x); rd(i_1,x)
\end{aligned}$$

Now, weak-synchrony semantics can be described by means of a transition relation $\longrightarrow_w \subseteq \mathcal{S} \times \mathcal{S}$, defined so that $S \longrightarrow_w S'$ holds iff $\big|S\big|_{\mathcal{S}} \longrightarrow \big|S'\big|_{\mathcal{S}}$. That is, a configuration is first translated into the version with mono-space operations only, and then strong-synchrony semantics induced by relation $\longrightarrow$ is simply therefore applied. It is natural to consider the equivalence relation $\approx$ over $\mathcal{S}$ induced by encoding $\big|.\big|_{\mathcal{S}}$, that is, the one defined so that $S \approx S'$ iff $\big|S\big|_{\mathcal{S}} = \big|S'\big|_{\mathcal{S}}$. We obtain the following examples of association induced by $\approx$, which provide an alternative viewpoint on weak-synchrony semantics.

$$\begin{aligned}
out(i_1 \wedge i_2 \wedge i_3, x) &\approx out(i_1, x); out(i_2 \wedge i_3, x) + out(i_2, x); out(i_1 \wedge i_3, x) + \\
&\quad out(i_3, x); out(i_1 \wedge i_2, x) \\
in(i_1 \otimes i_2 \otimes i_3, x) &\approx in(i_1, x) + in(i_2, x) + in(i_3, x) \\
rd(i_1 \vee i_2 \vee i_3, x) &\approx rd(i_1, x) + rd(i_1, x); rd(i_2 \vee i_3, x) + \\
&\quad rd(i_2, x) + rd(i_2, x); rd(i_1 \vee i_3, x) + \\
&\quad rd(i_3, x) + rd(i_3, x); rd(i_1 \vee i_2, x)
\end{aligned}$$

Roughly speaking, the left sides of the examples correspond to some multi-space primitive invocations — seen as atomic by the performing entity — whereas the right sides represent the corresponding non-atomic system behaviour as possibly perceived by another system entity.

4.3 Comparison & Remarks

The basic semantic difference between the two formal models of LogOp is that weak-synchrony semantics allows for more evolutions of a system configuration than strong-synchrony semantics. Evolutions allowed by weak-synchrony and prevented by strong-synchrony include partial evolutions of the tuple spaces affected by the execution of a coordination primitive.

Consider the case of a system S_E in which two equivalent coordinated entities want to first consume tuple x from tuple spaces i_1 and i_2, by primitive `in`(AND(i_1,i_2),x), and then produce tuple x in tuple space i_3. This is expressed by the following configuration:

$$S_E = in(i_1 \wedge i_2, x); out(i_3, x) \parallel in(i_1 \wedge i_2, x); out(i_3, x) \parallel \langle i_1, x\rangle \parallel \langle i_2, x\rangle$$

In the strong-synchrony model, one of the two processes may consume x from both i_1 and i_2, produce x in i_3, and then terminate, with the other process remaining blocked. The only allowed evolution is:

$$S_E \longrightarrow out(i_3, x) \parallel in(i_1 \wedge i_2, x); out(i_3, x) \longrightarrow in(i_1 \wedge i_2, x); out(i_3, x) \parallel \langle i_3, x\rangle$$

In the case of weak-synchrony, instead, the system S_E may also include a different evolution: the two processes may concurrently execute their primitive, with tuple x in space i_1 being consumed by one process and tuple x in space

i_2 being consumed by the other:

$$\begin{aligned} S_E \longrightarrow_w & \; in(i_2,x);out(i_3,x) \parallel \\ & \quad \big(in(i_1,x);in(i_2,x)+in(i_2,x);in(i_1,x)\big);out(i_3,x) \parallel \langle i_2,x\rangle \\ \longrightarrow_w & \; in(i_2,x);out(i_3,x) \parallel in(i_1,x);out(i_3,x) \end{aligned}$$

As a result, both processes remain blocked without inserting x in i_3 — one waiting for x to occur again in space i_2, the other in space i_1 — until another process comes in and inserts tuple x in either space i_1 or i_2. Notice that this example is conceptually similar to the paradigmatic case of the *dining philosophers* [13], where clients requires a mutually-exclusive access to different, distributed resources.

Of course, it would be interesting to further deepen the technical difference between the two semantics from an algebraic point of view, but this is not studied here since it is out of the scope of the article. However, a step in this direction is developed in [24], where the compliance issue for coordination models and coordination media is studied. According to that notion, a coordination model $\mathcal{Y} = \langle Y_0, Y, \rightarrow \mathcal{Y}\rangle$ is said to comply to a model $\mathcal{X} = \langle X_0, X, \rightarrow \mathcal{X}\rangle$ if $\mathcal{X}$ is able to simulate any transition of $\mathcal{Y}$. More specifically, there should exist an encoding $\big|\big| \in Y \mapsto X$ such that: *(i)* from $y_1 \rightarrow y_2$ we have $\big|y_1\big| \rightarrow^* \big|y_2\big|$, and *(ii)* $y \in Y_0$ implies $\big|y\big| \in X_0$ — where X_0 and Y_0 are used to represent the sets of successful states (typically the empty configuration). It worth noting that according to this definition, the strong-synchrony model can be shown to comply with the weak-synchrony one, the latter being able to simulate any transition of the former, exploiting the mapping $\big|.\big|_{\mathcal{S}}$ introduced in previous section.

Another related work is developed in [2], where the expressiveness of transactional mechanisms applied to concurrent languages is evaluated. In their setting, transactions are modelled as sequences of primitives to be atomically executed, modelling features of languages such as PoliS [11] and Shared Prolog [1]. Their main result, obtained by exploiting the framework of modular embeddings [12], is that languages with transactions are more expressive than languages without. This basically conforms to our observation about the differences between strong- and weak-synchrony: strong-synchrony semantics is allowed to force a specific sequences of interactions to occur, whereas weak-synchrony cannot prevent interference by other coordinated entities. However, to further deepen the comparison with the approach in [2], it would be crucial to study modular embeddings in the presence of the choice operator as well, which in fact significantly enhances the expressiveness of the coordination language.

So, generally speaking, strong-synchrony ensures atomicity of multi-space operations at the system level. This approach guarantees more properties to the coordinated system, but barely fits the requirements of open and distributed systems, where it would require costly transactional mechanisms associated to fine-grained communication operations. On the other hand, weak-synchrony poses lighter requirements over the coordination infrastructure, but may lose some interesting properties. In the example above, for instance, the system ends in a deadlock situation, where both processes are blocked waiting for the insertion of a tuple. However, we notice that these issues appear to be less restricting in the context of open, distributed systems, where e.g. optimistic approaches are generally exploited to recover from deadlock situations. Still, the weak-synchrony semantics of LogOp addresses the expressive limitations of the Linda coordination language, by allowing multi-space operations to be perceived as atomic at the coordinated entity's level.

5 The Semantics of a LogOp Service

The formal semantics described thus far endorses a viewpoint of the LogOp coordination model in terms of a language, defined by syntax and operational semantics of coordination primitives, used to express the interactive part of a concurrent system. As claimed in [25], this viewpoint — referred to as *coordination as a language* — is particularly suitable to analyse abstract properties of a coordination model in a conceptually clean and convenient way, as we showed in this LogOp case study as well. However, as far as deploying the coordination model into an actual scenario is concerned — most notably, when designing an infrastructure supporting the model — this viewpoint is possibly not the most proper one, for it abstracts away from a number of relevant run-time issues. These includes the actual shape of the abstraction involved in the coordination process, most notably the coordination media, as well as the dynamics of the single interaction acts that occur between such abstractions.

An alternative viewpoint called *coordination as a service* is introduced in [25] to account for this issue. In this framework, a coordinated system is explicitly modelled as divided into a *coordinated space*, where entities subject to coordination (i.e. processes) live, an *interaction space*, where communication events occur reflecting the dynamics of coordination, and a *coordination space*, where coordination media (e.g. tuple spaces) live and interact to provide the coordination service. In particular, instead of considering a coordination model as a language with coordination primitives, this new framework views coordination as an interactive service provided by some coordination media. By focusing on the interactive behaviour of coordination media, this

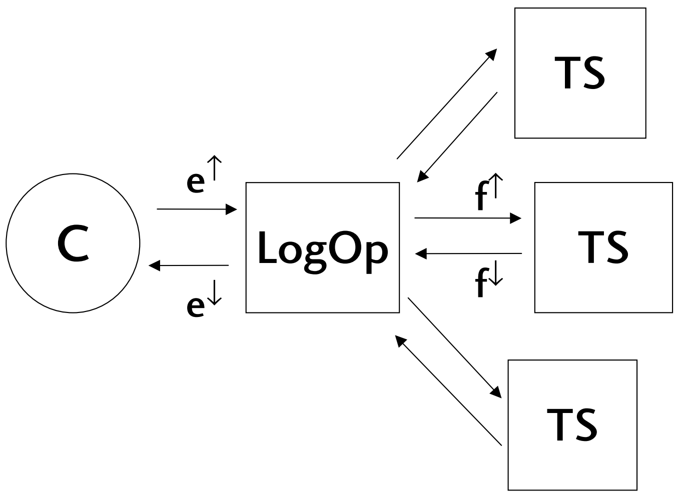

Fig. 1. The LogOp infrastructure model

framework considers crucial run-time issues of the model, thus evaluating its realisability as an infrastructure and sketching a design of its actual implementation. As shown in the methodological approach presented in [24], this kind of specification is meant to be a reference for implementors, providing the necessary information to check for the compliance of an actual implementation with respect to the original model.

In this section, we show how the weak-synchrony semantics of the LogOp coordination model can be turned into a specification of the LogOp coordination service, provided by a medium acting as a front-end to coordinated entities and implementing the core part of the LogOp service.

5.1 The LogOp Service

Our general aim here is to provide a design for the LogOp service on top of an infrastructure where standard Linda tuple spaces (TS) exist that conform to the semantics described in Section 2. So, we introduce a coordination medium implementing the further management required by the LogOp semantics. In particular, this abstraction is in charge of *(i)* accepting requests for executing a LogOp operation, *(ii)* handling and governing interactions with the tuple spaces in the system, and finally *(iii)* providing the proper reply to the requesting entity. A pictorial representation of the resulting infrastructure is reported in Figure 1. In general, no hypotheses are made on how many LogOp coordination media are actually implemented by the infrastructure, and used to provide the coordination service: possible and plausible scenarios include one LogOp coordination medium for each node, or one for each coordinated entity, or even one created on the fly for each LogOp coordination primitive invocation. For the sake of simplicity, in the following our formalism refers to the case of a single LogOp coordination medium: the extension to a multiplicity of LogOp coordination media is straightforward, and would simply requires a slightly more complex notation.

In the following, we let:

- meta-variable $f^{\uparrow}$ to range over the set $F^{\uparrow}$ of requests sent to a tuple space by the LogOp medium, which are of the kind $n \upharpoonleft \beta$ — where n is the name of the process local to the LogOp medium, called *handler*, in charge of handling replies, and β is the mono-space primitive operation to be executed;
- meta-variable $f^{\downarrow}$ to range over the set $F^{\downarrow}$ of replies sent from a tuple space to the LogOp medium, which are of the kind $n \downharpoonleft ok$ — where n is the handler's name and ok is the positive reply[4];
- meta-variable $e^{\uparrow}$ to range over the set $E^{\uparrow}$ of LogOp requests, which are of the kind $id \uparrow \alpha$ where id is the identifier of the requesting coordinated entity (C) and α is the multi-space primitive operation;
- meta-variable $e^{\downarrow}$ to range over the set $E^{\downarrow}$ of LogOp replies, which are of the kind $id \downarrow ok$.

We suppose that the set of identifiers for the coordinated entities is disjoint from the set of names of handlers, so that the four sets above are all disjoint.

5.2 Representing Tuple Spaces

We start by defining the behaviour of a Linda tuple space coordination medium conforming to the specification shown in Section 2. Following the work in [24], a Linda tuple space is easily represented by a transition system $\langle \mathcal{TS}, \rightarrow_{TS}, Act_{TS} \rangle$ with elements $TS \in \mathcal{TS}$ defined by:

$$TS ::= 0 \;\mid\; f^{\uparrow} \;\mid\; f^{\downarrow} \;\mid\; x \;\mid\; (TS \parallel TS)$$

Thus, the configuration of a Linda tuple space is seen as a composition of tuples, requests to be processed ($f^{\uparrow}$) and replies to be sent ($f^{\downarrow}$). The set of actions is defined as $Act_{TS} = \{\tau\} \cup F^{\uparrow} \cup F^{\downarrow}$, representing internal silent action, reception of requests, and production of replies. The transition relation is defined by rules:

[4] Only positive replies are considered here since we do not study primitives `inp` and `rdp`, which may fail.

$$TS \xrightarrow{f^\uparrow}_{\mathcal{TS}} TS \parallel f^\uparrow \qquad \text{[TS-REQ]}$$

$$TS \parallel f^\downarrow \xrightarrow{f^\downarrow}_{\mathcal{TS}} TS \qquad \text{[TS-REP]}$$

$$n \upharpoonleft out(x) \parallel TS \xrightarrow{\tau}_{\mathcal{TS}} TS \parallel x \qquad \text{[TS-OUT]}$$

$$n \upharpoonleft rd(x) \parallel TS \parallel x \xrightarrow{\tau}_{\mathcal{TS}} TS \parallel x \parallel n \downharpoonright ok \qquad \text{[TS-RD]}$$

$$n \upharpoonleft in(x) \parallel TS \parallel x \xrightarrow{\tau}_{\mathcal{TS}} TS \parallel n \downharpoonright ok \qquad \text{[TS-IN]}$$

These rules intuitively correspond to the semantics shown in Section 2. In fact, while the former two rules simply deal with interactions with the environment, respectively consuming requests and producing replies, the other three rules process the requests according to the LINDA semantics expressed by rules [L-OUT], [L-RD] and [L-IN] of Section 2, respectively. Most notably, when a *rd* or *in* primitive are served, a reply is reified in the tuple space that will be subsequently sent to the LOGOP medium, by rule [L-REP].

5.3 The LOGOP *Coordination Medium*

The LOGOP coordination medium receives *request events* of the kind $e^\uparrow$ from coordinated entities, process them by interacting with tuple spaces by means of events $f^\uparrow$ and $f^\downarrow$, and then possibly provides a *reply event* $e^\downarrow$ to the coordinated entity.

Similarly to the case of LINDA, the LOGOP medium can be defined by a transition system $\langle \mathcal{LO}, \rightarrow_{\mathcal{LO}}, Act_{LO}\rangle$ where the set of actions is defined as $Act_{LO} = \{\tau\} \cup F^\uparrow \cup E^\downarrow \cup F^\downarrow \cup E^\uparrow$. The configurations $LO \in \mathcal{LO}$ for the LOGOP medium are defined as:

$$LO ::= 0 \mid e^\uparrow \mid H \mid (LO \parallel LO)$$

At any time, a configuration is made of some pending request $e^\uparrow$ and some handler H. In particular, a handler is a local, non-deterministic process spawned when a new operation has to be processed, which is in charge of directing requests $f^\uparrow$ to tuple spaces, receiving the corresponding replies $f^\downarrow$, providing a reply $e^\downarrow$ to the coordinated entity, and eventually terminating. Handlers are defined by the syntax:

$$H ::= 0 \mid H + H \mid H ; H \mid H \parallel_h H \mid\ !f^\uparrow \mid\ !e^\downarrow \mid\ ?f^\downarrow$$

where operators ';', '+', and '$\parallel_h$' are sequential composition, choice, and parallel composition as usual — listed in decreasing priority order and equipped by the same congruence rules as defined in the previous sections. Handler $!f^\uparrow$

sends request $f^{\uparrow}$ to a tuple space, $!e^{\downarrow}$ sends reply $e^{\downarrow}$ to a coordinated entity, and $?f^{\downarrow}$ receives reply $f^{\downarrow}$ from a tuple space. Transition relation $\rightarrow_{\mathcal{LO}}$ for the LogOp medium is defined by the rules:

$$LO \xrightarrow{e^{\uparrow}}_{\mathcal{LO}} LO \parallel e^{\uparrow} \qquad \text{[LO-REQ]}$$

$$LO \parallel (!e^{\downarrow}; H + H') \xrightarrow{e^{\downarrow}}_{\mathcal{LO}} LO \parallel H \quad \text{[LO-REP]}$$

$$LO \parallel (!f^{\uparrow}; H + H') \xrightarrow{f^{\uparrow}}_{\mathcal{LO}} LO \parallel H \quad \text{[LO-LSND]}$$

$$LO \parallel (?f^{\downarrow}; H + H') \xrightarrow{f^{\downarrow}}_{\mathcal{LO}} LO \parallel H \quad \text{[LO-LRCV]}$$

$$\frac{h(e^{\uparrow}, H)}{LO \parallel e^{\uparrow} \xrightarrow{\tau}_{\mathcal{LO}} LO \parallel H} \qquad \text{[LO-SPAWN]}$$

Rule [LO-REQ] stores the received request into the configuration, rules [LO-REP], [LO-LSND], and [LO-LRCV] execute an action within some handler, causing an interaction with the environment. Finally, rule [LO-SPAWN] is used to consume a request and spawn the handler which is going to manage its processing from then onwards. Relation $h(e^{\uparrow}, H)$, which requires the creation of the handler H from the request $e^{\uparrow}$, is specified in the remainder of this section, providing the key semantics of the LogOp service.

5.4 *Handling the* `out` *Primitive*

We first deal with the case of primitive `out`. When executing an `out`, no reply is provided to the sending entity, so, as far the interactive behaviour of the LogOp medium is concerned, its specification is quite different from that of primitives `in` and `rd`.

In the case of primitive `out`, relation h is defined by the following three rules, which associate to a request for primitive `out` the set of single `out` requests to be sent to the tuple spaces:

$$h(id \uparrow \mathtt{out}(\bigwedge_{i \in I} i, x), \prod_{i \in I} n \uparrow \mathtt{out}(i, x))$$

$$h(id \uparrow \mathtt{out}(\bigotimes_{i \in I} i, x), n \uparrow \mathtt{out}(i_0, x)) \qquad \text{for any } i_0 \in I$$

$$h(id \uparrow \mathtt{out}(\bigvee_{i \in I} i, x), \prod_{i' \in I'} n \uparrow \mathtt{out}(i', x)) \qquad \text{for any } I' \text{ so that } \{\} \subset I' \subseteq I$$

Here, since we have no replies, the name n of the local handler can be any. Notice that the above rules simply correspond to the specification of relation

σ by rules [S-AND], [S-XOR] and [S-OR] as provided in Subsection 4.1, which describes the tuples involved in the execution of a LogOp primitive.

5.5 Handling the `in` *and* `rd` *Primitives*

Operations involving primitives `in` and `rd` are denoted by the syntax $\gamma(\lambda, x)$ where γ is defined as $\gamma ::= rd \mid in$. Dealing with these primitives clearly adds a further complication in the definition of the handler: replies of the kind $f^{\downarrow}$ have to be received, and based on them, a reply event $e^{\downarrow}$ should be properly sent to the requesting coordinated entity. To deal with the request and reply phases, we split the handler in two parts, H and H', one devoted to sending single requests to tuple spaces, and one devoted to receiving replies:

$$\frac{\mathit{send}(id\uparrow\gamma(\lambda, x), H, \nu) \qquad \mathit{recv}(id\uparrow\gamma(\lambda, x), \nu) = H'}{h(id\uparrow\gamma(\lambda, x), H \parallel H')}$$

These two parts are not completely independent, but interact by means of an injective function $\nu \in I \mapsto N$ associating to each tuple space identifier the name or reference of the handler that will manage its replies. Relation $\mathit{send}(e^{\uparrow}, H, \nu)$ associates to a request $e^{\uparrow}$ both *(i)* the part of the handler that sends replies H and *(ii)* the function ν associating a name to each tuple space $i \in I$ involved. On the other hand, function *recv* takes a request $e^{\uparrow}$ and a function ν and yields the (part of the) handler H' receiving replies.

The relation *send*, defining the handler sending requests, is defined analogously to the case of `out` primitive. The only difference is that in the case of operator XOR, more requests can be possibly sent, hence as far as only sending requests is concerned, it is treated as operator OR.

$$\mathit{send}(id\uparrow\gamma(\bigwedge_{i\in I} i, x), \prod_{i\in I}\nu(i)\uparrow\gamma(i, x), \nu)$$

$$\mathit{send}(id\uparrow\gamma(\bigotimes_{i\in I} i, x), \prod_{i'\in I'}\nu(i)\uparrow\gamma(i, x), \nu) \quad \text{with } \{\} \subset I' \subseteq I$$

$$\mathit{send}(id\uparrow\gamma(\bigvee_{i\in I} i, x), \prod_{i'\in I'}\nu(i)\uparrow\gamma(i, x), \nu) \quad \text{with } \{\} \subset I' \subseteq I$$

We suppose that names yielded by function ν are always unique within the current configuration, so as to avoid confusion with the handling of different replies.

Concerning the function *recv*, which defines the handler receiving replies, we first define the behaviour of `in` and `rd` in the case of operator AND, which is defined by the two rules:

$$recv\big(id{\uparrow}\gamma(\bigwedge_{i\in I} i, x), \nu\big) = \sum_{i\in I} ?n_i{\downarrow}ok; recv\big(id{\uparrow}\gamma(\bigwedge_{i'\in I\setminus\{i\}} i', x), \nu\big)$$

$$recv\big(id{\uparrow}\gamma(\epsilon, x), \nu\big) = !id{\downarrow}ok$$

The former rule specifies that non-deterministically any reply can be received, and then all the others, recursively. Then, the second rule — which is used later also by operator OR — deals with the final case where all replies are handled: in this case, a positive reply is sent to the requesting entity. Rules for operator XOR and OR are as follows:

$$recv\big(id{\uparrow}\gamma(\bigotimes_{i\in I} i, x), \nu\big) = \sum_{i\in I} ?\nu(i){\downarrow}ok;\ !id{\downarrow}ok;\ \mathit{finish}\big(id{\uparrow}\gamma(\bigotimes_{i'\in I\setminus\{i\}} i', x), \nu\big)$$

$$\begin{aligned} recv\big(id{\uparrow}\gamma(\bigvee_{i\in I} i, x), \nu\big) &= recv\big(id{\uparrow}\gamma(\bigotimes_{i\in I} i, x), \nu\big) \\ &+ \sum_{i\in I} ?\nu(i){\downarrow}ok; recv\big(id{\uparrow}\gamma(\bigotimes_{i'\in I\setminus\{i\}} i', x), \nu\big) \end{aligned}$$

In the case of operator XOR, *(i)* a reply is received from a tuple space ($?\nu(i){\downarrow}ok$), *(ii)* the reply event is sent to the requesting entity ($!id{\downarrow}ok$), and *(iii)* a function *finish* is applied creating a daemon in charge of consuming all other replies. In the case of operator OR, the handler yielded by function *recv* non-deterministically chooses between sending immediately the reply then invoking *finish* as for XOR (left choice) and keeping consuming other replies (right choice), recursively. Notice that these two rules are conceptually similar to the encoding for operator XOR and OR of weak-synchrony semantics, as reported in Subsection 4.2.

More precisely, while for the `rd` primitive the function *finish* creates a daemon consuming the remaining replies, in the case of the `in` primitive this daemon is also in charge of restoring the tuples unnecessarily removed, by sending `out` operations. In fact, because of our management, some tuples may be removed from a LINDA tuple space which are never actually processed by operator XOR, so they are sent back to be inserted again. Notice that this behaviour is indeed correct with respect to the rules of LINDA presented in Section 2, in that temporarily removing a tuple does not affect safety. Function *finish* is defined by the rules:

$$\begin{aligned}
\mathit{finish}\big(id\!\uparrow\!\texttt{rd}(\lambda,x),\nu\big) &= \sum_{i\in I} ?n_i\!\downarrow\!ok;\mathit{finish}\big(id\!\uparrow\!\texttt{rd}(\lambda_{I\setminus\{i\}},x),\nu\big)\\
\mathit{finish}\big(id\!\uparrow\!\texttt{in}(\lambda,x),\nu\big) &= \sum_{i\in I} ?n_i\!\downarrow\!ok;!\nu(i)\!\uparrow\!\texttt{out}(i,x);\mathit{finish}\big(id\!\uparrow\!\texttt{rd}(\lambda_{I\setminus\{i\}},x),\nu\big)\\
\mathit{finish}(id\!\uparrow\!\texttt{in}(\epsilon,x),\nu) &= 0
\end{aligned}$$

Here, term $\lambda_{I'}$ represents, for any set of identifiers I', the restriction of logical expression λ to only the tuple spaces in I', e.g. $(i_1 \wedge i_2 \wedge i_3)_{\{i_1,i_3\}} = i_1 \wedge i_3$. In particular, while for `rd` suffices it to consume replies, for the `in` primitive `out` requests are redirected to the tuple spaces to restore their proper state.

5.6 Representing the Infrastructure

The whole dynamics of the coordination space — including all the coordination media — can be characterised by a transition system $\langle \mathcal{CS}, \rightarrow_{\mathcal{CS}}, Act_{CS}\rangle$, where $\mathcal{CS}$ is the set of configurations (or distributed states), Act_{CS} is the set of actions, and $\rightarrow_{\mathcal{CS}} \subseteq \mathcal{CS} \times Act_{CS} \times \mathcal{CS}$ is the transition relation. Actions are either: *(i)* the silent action τ, representing internal changes of the configuration, *(ii)* request events $id\!\uparrow\!\alpha$ from coordinated entities, and *(iii)* reply events $id\!\downarrow\!ok$ to coordinated entities.

Configurations $CS \in \mathcal{CS}$ are formally defined by the syntax:

$$CS ::= LO \cdot CS^{TS} \qquad\qquad CS^{TS} ::= 0 \;\mid\; \langle i, TS\rangle \;\mid\; (CS^{TS} \parallel CS^{TS})$$

Each configuration is made of a LogOp coordination medium LO and by the tuple spaces in the system, each characterised by the configuration TS and the identifier i. Then, transition relation $\rightarrow_{\mathcal{CS}}$ is defined by rules:

$$\frac{TS \xrightarrow{\tau}_{\mathcal{TS}} TS'}{LO \cdot \left(CS^{TS} \parallel \langle i, TS\rangle\right) \xrightarrow{\tau}_{\mathcal{CS}} LO \cdot \left(CS^{TS} \parallel \langle i, TS'\rangle\right)} \quad [I - LND]$$

$$\frac{LO \xrightarrow{\tau}_{\mathcal{LO}} LO'}{LO \cdot CS^{TS} \xrightarrow{\tau}_{\mathcal{CS}} LO' \cdot CS^{TS}} \quad [I - LOGOP]$$

$$\frac{LO \xrightarrow{e}_{\mathcal{LO}} LO'}{LO \cdot CS^{TS} \xrightarrow{e}_{\mathcal{CS}} LO' \cdot CS^{TS}} \quad [I - EXT]$$

$$\frac{LO \xrightarrow{n\uparrow\pi(i,x)}_{\mathcal{LO}} LO' \qquad TS \xrightarrow{n\uparrow\pi(i,x)}_{\mathcal{TS}} TS'}{LO \cdot \left(CS^{TS} \parallel \langle i, TS\rangle\right) \xrightarrow{\tau}_{\mathcal{CS}} LO' \cdot \left(CS^{TS} \parallel \langle i, TS'\rangle\right)} \quad [I - SND]$$

$$\frac{LO \xrightarrow{n\downarrow ok}_{\mathcal{LO}} LO' \qquad TS \xrightarrow{n\uparrow ok}_{\mathcal{TS}} TS'}{LO \cdot \left(CS^{TS} \parallel \langle i, TS\rangle\right) \xrightarrow{\tau}_{\mathcal{CS}} LO' \cdot \left(CS^{TS} \parallel \langle i, TS'\rangle\right)} \quad [I - RCV]$$

Rules [I-LND] and [I-LOGOP] model internal computations within either a Linda tuple space or the LogOp medium. Rule [I-EXT] represents an interaction of the coordination space with a coordinated entity — where meta-variable e ranges over $E^{\uparrow} \cup E^{\downarrow}$ —, and finally rules [I-SND] and [I-RCV] model internal changes in the configuration due to requests to tuple spaces and corresponding replies.

5.7 On the Semantics of Operator or

The specification provided here to operator *or* is just one of the possible specifications complying to the formal model of LogOp described in Subsection 4.1, which requires some (at least one) of the tuples to be inserted/removed/read from the list of tuple spaces. In fact, the LogOp medium defined in this section deals with *or* operations by sending requests to any non-void subset of such tuple spaces, and furthermore, after the reception of the first reply it may possibly stop receiving further replies at any time. In particular, these choices are taken non-deterministically, by means of the summation operator $+$.

However, our choice for this specification is not arbitrary at all. It is rather a remarkable one, that is, the more general one – the one that an actual implementation has to comply with in order to follow the LogOp semantics. As a reference for this argument we consider [24], where the notion of compliance for coordination media is studied. There, a methodology for devising coordination media implementations out from coordination models is introduced which includes: *(i)* definition of a general coordination medium realising the coordination model, name it M, and *(ii)* definition of a compliance relation between coordination media using refinement of process algebras – i.e. preorder semantics [16]. In this framework, a coordination medium implementation I is considered compliant to the model if its interactive behaviour – understood as the multisets of all its possible interaction histories – is smaller than M's. This notion recalls the idea that I provides less behaviour than M, that I is in some sense more deterministic, more directly executable – namely, an implementation of M.

According to this very interpretation, the medium specification described in this section is the most general version possible. An actual implementation might deal with the *or* primitive in different ways: sending requests to all tuple spaces, to just one of them, to the 70% of them, waiting for just one reply, for 50% of the requests performed, or until a timeout expires, or, more generally, depending on run-time aspects such as load-balancing. All these behaviours can be understood as refinements of our general coordination medium. Here, non-determinism plays a crucial role, as it allows us to abstract away from a number of issues that may raise at implementation-time, depending on a number of constraints which are mostly unpredictable at design-time.

5.8 Remarks

In [25,24], the process of turning the specification of a coordination language into the corresponding service is shown in the case of the mono-space operations of LINDA. There, the main issue was to properly manage the replies to be sent to the requesting entities — the other aspects of the mapping being almost straightforward. The multi-space case of LOGOP analysed in this paper, being much more concerned with distribution, is instead more interesting and meaningful, and allows us to deepen the relationship between the two frameworks, as well as to consider a number of issues in the implementation of LOGOP.

The first problem to be faced when representing a coordination service is to characterise the shape and boundary of all the coordination media involved in the coordination process, in terms of the set of their interactions with coordinated entities and with each others as well. Notice that this conceptual step should generally take into account aspects related to effectiveness and efficiency, e.g. should enforce a principle of locality of interactions. In the LOGOP case, we decided for the sake of simplicity to define a single coordination medium receiving all the LOGOP requests, managing the necessary interactions with legacy LINDA tuple spaces. However, such an infrastructure specification could be easily adapted to the case where the LogOp medium is replicated, as mentioned above.

Not surprisingly, another issue raised by the LogOp case study is that the support of synchronous and asynchronous primitives as services generally requires quite a different management, whereas their operational semantics is similar. In the LogOp case, while e.g. rules [LG-OUT] and [LG-IN] (Subsection 4.2) describing the operational semantics of `out` and `in` primitives are quite similar (basically they are symmetrical), devising the corresponding interactive services is completely different. As primitive `out` only involves sending requests to all (and), one (xor), or some (or) tuple spaces — according to rules [S-AND], [S-XOR], and [S-OR] (Subsection 4.1) — primitive `in` (and `rd` as well) requires a handler for replies to be prepared, figuring out when a reply is to be sent to the coordinated entity and how to receive all the remaining replies.

In particular, our formulation of the LogOp service assumes the unordered semantics of primitive `out` [4], contrasting with ordered semantics of the model reported in Section 4. [5] This is motivated by simplicity: preserving the ordered semantics in the context of a distributed system would lead to a more complex interaction schema — basically dealing with replies for primitive *out* as well.

A final consideration concerns the difference between destructive (`in`) and non-destructive (`rd`) operations, which is easily underspecified from the viewpoint of LogOp as a language. While in the case of primitive `rd` it is safe to send more requests than actually needed, primitive `in` is much more crucial, as it requires to provide a procedure for restoring a safe state within the tuple spaces — which is however obtained from an interaction pattern emerged in the weak-synchrony semantic model. In particular, this is the case of operator xor, which requires only one tuple to be removed from a tuple space, hence introduces a notion of distributed mutual exclusion.

As the reader may notice, all these issues are crucial when designing a coordination model for distributed systems as LogOp, and also when designing an infrastructure that could support such a model effectively and efficiently. Whereas some of the details of such issues are easily underspecified by formalisations endorsing the notion of coordination as a language, the viewpoint of coordination as a service seems to provide a better framework for their description. Let us simply consider the services associated to weak-synchrony and strong-synchrony semantics. While in this paper we showed that the former allows for a reasonable specification — only the operator xor in conjunction with primitive `in` requiring a substantial overhead — defining the service of

[5] Notice however that the original description of LogOp in [22] neglects any specification about this issue.

strong-synchrony semantics would have required a rather overwhelming intricacy of messages, locks, and recovering protocols. From this viewpoint, the framework of coordination as a language is surprisingly counterintuitive, as it apparently shows that strong-synchrony is an even simpler model than weak-synchrony.

6 Conclusions

In this paper we analysed some issues in the semantics of coordination models for distributed systems. By using the LogOp coordination model as a case study, we showed that many different, non-equivalent interpretations are admissible for an informal specification of a coordination model, and that only a formal specification can make a coordination model specification non-ambiguous. In particular, we introduced two different interpretations for the LogOp informal semantics (weak-synchrony and strong-synchrony) that demonstrate how notions like atomicity in distributed systems can be addressed at different levels, with different effects on the systems and their components.

Also, we showed that two different semantic frameworks (coordination as a language and coordination as a service) can be used to provide a formal characterisation for coordination models, and we used LogOp to discuss their different impact on the specification of coordination infrastructures. As a vast literature is available on semantics for coordination models, we forward the reader interested in deepening the relationships between the two frameworks and the other known approaches to the discussion and bibliography in [25].

References

[1] Brogi, A. and P. Ciancarini, *The concurrent language Shared Prolog*, Transactions on Programming Languages and Systems **13(1)** (1991), pp. 99–123.

[2] Brogi, A. and J. Jacquet, *On the expressiveness of coordination models*, in: P. Ciancarini and A. L. Wolf, editors, *Coordination Languages and Models*, LNCS **1594** (1999), pp. 134–149, 3rd International Conference (COORDINATION'99), Amsterdam, The Netherlands, 26–28 April 1999. Proceedings.

[3] Busi, N., P. Ciancarini, R. Gorrieri and G. Zavattaro, *Coordination models: A guided tour*, in: A. Omicini, F. Zambonelli, M. Klusch and R. Tolksdorf, editors, *Coordination of Internet Agents: Models, Technologies, and Applications*, Springer-Verlag, 2001 pp. 6–24.

[4] Busi, N., R. Gorrieri and G. Zavattaro, *Comparing three semantics for Linda-like languages*, Theoretical Computer Science **240(1)** (1990), pp. 49–90.

[5] Busi, N., R. Gorrieri and G. Zavattaro, *A process algebraic view of Linda coordination primitives*, Theoretical Computer Science **192(2)** (1998), pp. 167–199.

[6] Busi, N., C. Manfredini, A. Montresor and G. Zavattaro, *Peerspaces: Data-driven coordination in peer-to-peer networks*, in: *18th ACM Symposium on Applied Computing (SAC 2003)*, ACM, Melbourne, FL, USA, 2003, pp. 380–386.

[7] Busi, N. and G. Zavattaro, *On the serializability of transactions in JavaSpaces*, in: U. Montanari and V. Sassone, editors, *ConCoord: International Workshop on Concurrency and Coordination*, Electronic Notes in Theoretical Computer Science **54** (2001).

[8] Cabri, G., L. Leonardi and F. Zambonelli, *MARS: a programmable coordination architecture for mobile agents*, IEEE Internet Computing **4(4)** (2000), pp. 26–35.

[9] Carriero, N., D. Gelernter and L. Zuck, *Bauhaus-Linda*, in: P. Ciancarini, O. Nierstrasz and A. Yonezawa, editors, *Object-Based Models and Languages for Concurrent Systems*, LNCS **924** (1995), pp. 66–76.

[10] Ciancarini, P., *Coordination models and languages as software integrators*, ACM Computing Surveys **28(2)** (1996), pp. 300–302.

[11] Ciancarini, P., F. Franzé and C. Mascolo, *Using a coordination language to specify and analyze systems containing mobile components*, ACM Transaction on Software Engineering **9(2)** (2000), pp. 167–198.

[12] de Boer, F. S. and C. Palamidessi, *Embedding as a tool for language comparison*, Information and Computation **108(1)** (1994), pp. 128–157.

[13] Dijkstra, E., "Co-operating Sequential Processes," Academic Press, London, 1965.

[14] Freeman, E., S. Hupfer and K. Arnold, "JavaSpaces: Principles, Patterns, and Practice," Addison-Wesley, 1999.

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

[16] Glabbeek, R. v., *The linear time – branching time spectrum I. The semantics of concrete, sequential processes*, in: J. A. Bergstra, A. Ponse and S. A. Smolka, editors, *Handbook of Process Algebra*, North-Holland, 2001 pp. 3–100.

[17] Menezes, R. and R. Tolksdorf, *A new approach to scalable Linda-systems based on Swarms*, in: *18th ACM Symposium on Applied Computing (SAC 2003)*, ACM, Melbourne, FL, USA, 2003, pp. 375–379.

[18] Milner, R., "Communication and Concurrency," Prentice Hall, 1989.

[19] Omicini, A. and E. Denti, *From tuple spaces to tuple centres*, Science of Computer Programming **41(3)** (2001), pp. 277–294.

[20] Omicini, A. and F. Zambonelli, *Coordination for Internet application development*, Journal of Autonomous Agents and Multi-Agent Systems **2(3)** (1999), pp. 251–269.

[21] Picco, G. P., A. L. Murphy and G.-C. Roman, *Lime: Linda meets mobility*, in: *1999 International Conference on Software Engineering (ICSE'99)* (1999), pp. 368–377, los Angeles, CA, USA.

[22] Snyder, J. and R. Menezes, *Using logical operators as an extended coordination mechanism in Linda*, in: F. Arbab and C. Talcott, editors, *Coordination Languages and Models*, LNCS **2315** (2002), pp. 317–331, 5th International Conference (COORDINATION 2002), York, UK, 8–11 April 2002. Proceedings.

[23] TuCSoN *at SourceForge*.
URL http://tucson.sourceforge.net

[24] Viroli, M., *Comparing semantic frameworks for coordination: on the conformance issue for coordination media*, in: *18th ACM Symposium on Applied Computing (SAC 2003)* (2003), pp. 394–401, Special Track on Coordination Models, Languages and Applications.

[25] Viroli, M. and A. Omicini, *Coordination as a service: Ontological and formal foundation*, in: A. Brogi and J.-M. Jacquet, editors, *Foundations of Coordination Languages and Software Architecture*, Electronic Notes in Theoretical Computer Science **68(3)**, Elsevier Science, 2003 1st International Workshop (FOCLASA 2002), Brno, Czech Republic, 24 August 2002. Proceedings.
URL http://www.elsevier.nl/gej-ng/31/29/23/121/52/40/68.3.013.pdf

[26] Zuck, L. D. and D. Gelernter, *On what Linda is: Formal description of Linda as a reactive system*, in: D. Garlan and D. Le Métayer, editors, *Coordination Languages and Models*, LNCS **1282** (1997), pp. 187–204, 2nd International Conference (COORDINATION'97), Berlin, Germany, 1–3 September 1997. Proceedings.