Have ReSpecT for LogOp

Ronaldo Menezes Andrea Omicini Mirko Viroli

***Abstract*—**

In this paper, we exploit two different coordination models, namely LogOp [1] and ReSpecT [2], to discuss the design of a coordination infrastructure for distributed systems. LogOp model introduces coordinated primitives extending those of Linda towards usage on a multiplicity of tuple spaces, which can be dynamically associated by logical operators and then accessed altogether. These primitives are here given a mapping on top of the architecture of ReSpecT tuple centres, a coordination infrastructure suitable for distributed environments. A formal treatment of LogOp primitives is presented that allows us to reason about the applicability of the resulting model in the context of distributed environments.

I. Background & Motivation

Complex systems of today are usually built out of several heterogeneous components (objects, processes, agents, ...) dipped into dynamic and distributed environments. Managing the space of the interaction among components is a matter of coordination [3]: starting from the pioneering work on Linda [4], a number of different coordination models have been designed and proposed to handle the complexity of components interaction in system engineering [5]. Coordination models are typically conceived as providing a general solution to coordination issues: the Linda model, for instance, has been experimented to face a wide class of different application problems. However, no coordination model is really general enough to effectively cover the whole spectrum of the different coordination problems that may arise from the engineering of complex systems. Every coordination model promotes a particular level of abstraction over systems, by providing a powerful but limited number of coordination abstractions through which the space of interaction is modeled and ruled.

Consequently, several issues have to be addressed when choosing a coordination model that basically concern the expressive power of coordination models, and how do they compare to each other. In particular, given a coordination model, the first issue to be addressed is what class of coordination problems are actually addressed by the model, and how effectively. A measure of expressiveness for coordination models and languages is also required that allow designers to compare different models, and helps them choosing the right one for their particular application scenario (a first result in this direction is [6]). Furthermore, when complexity and applicability range of the coordination problem are large, it may be the case that more models have to be used together – how they could be combined, merged, blended together in a more or less clean and effective way is also a relevant matter to be solved.

According to the above considerations, in this paper we take two relevant examples of coordination models, namely LogOp [1] and ReSpecT [2], that are both Linda-based, and are seemingly meant to cover the same class of application problems. However, the difference in the approach and in the solutions they provide – namely, two opposite ways to extend the basic Linda model – makes them complimentary models that can be used altogether at different levels of abstraction in the coordination of distributed systems.

This paper is organized as follows: Section II introduces LogOp in short, defining the basic coordination language and providing a first informal description of its semantics. After a brief discussion about the main features of ReSpecT in Section III, Section IV summarizes the differences between the two models, and describes a possible mapping of LogOp upon ReSpecT. Finally, Section V introduces the issue of the semantics of coordination models, discussing the different models of the interaction space implicitly endorsed by different models of coordination, and also showing how a distributed and asynchronous environment could induce different interpretations of the intended effect of coordination primitives – which are not likely to be made unambiguous by any informal semantics.

II. LogOp

Linda provides primitives that enable processes to store and retrieve tuples from tuple spaces. Although the names of the primitives vary slightly in different implementations, the functionalities are normally very similar. Processes use the primitive `out` to store tuples and they retrieve tuples using the primitives `in` and `rd`. The primitives `in` and `rd` take a template (a definition of a tuple) and use associative matching to retrieve the desired tuple – while `in` removes a matching tuple, `rd` takes a copy of the tuple. 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 can be retrieved.

The blocking nature of these primitives and the fact that they can only deal with one tuple space at a time is somewhat limitating. The primitives in Linda force processes to see tuple spaces as disjoint. In other words, processes can only access one tuple space at a time; processes that require access to several tuple spaces have to do so by serializing the access. Although tuple spaces are indeed disjoint, the serialization of access on the client side limits the ability of Linda to express coordination where the contents of different tuple spaces need to be considered as a whole. This problem is the main motivation of Snyder and Menezes in the proposal of LogOp [1]. Linda is widely acceptable in the coordination community due to its

R. Menezes is with Department of Computer Science at Florida Institute of Technology, 150 West University Blvd., Melbourne, FL 32901, USA – email: rmenezes@cs.fit.edu

A. Omicini and M. Viroli are with DEIS, Università degli Studi di Bologna, via Rasi e Spinelli 176, 47023 Cesena (FC), Italy – email: {aomicini,mviroli}@deis.unibo.it

elegance and simplicity but it does not to maintain this elegance in problems where a single process need to concurrently access several tuple spaces. LOGOP is a coordination model that aims at improving the expressiveness of LINDA primitives, yet trying to achieve good performance at the implementation level as a consequence of expressiveness.

Consider the example where two independent processes offer a service, respectively expressed as a tuple occurring in tuple spaces `tsA` and `tsB`, and a process that wants to exploit the service, expressed as the withdrawal of one of the two tuples. In LINDA the problem would be inefficiently solved: on the one hand, primitive `in` can be applied only to one tuple space which may cause the process to block, thus never checking the second tuple space. On the other hand, exploiting primitive `inp` – the asynchronous version of primitive `in` – would require continuously polling both tuple spaces.

LOGOP was developed to tackle these kinds of problems. 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 noticed that the associations amongst the tuple spaces that are normally required relate to basic logical operators such as AND, OR, and XOR. These operators are used in LOGOP primitives to specify the target tuple spaces, specifying e.g. that a tuple has to be inserted in both `tsA` and `tsB` (`out` + AND), or has to be removed from either `tsA` or `tsB` (`in` + OR), or has to be read from one of `tsA` and `tsB` (`rd` + XOR).

Informally, the OR operator has the effect of combining tuple spaces so that processes can store and retrieve tuples from any of the tuple spaces from a list without having to impose an order on the way the tuple spaces are accessed. At times, it may be necessary for a process (say a consumer) to read tuples from *any* of several tuple spaces. The OR operator, when used to combine tuple spaces, allow processes to express this scenario. When storing tuples the OR operator adds another level of nondeterminism to the model – the tuple will be stored in some (at least one) tuple spaces from 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.

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: if n tuple spaces are involved in the operation, LINDA would have to execute the `out` primitive n times. In the case of blocking primitives `in` and `rd`, the semantics of AND is such that the process will block if one ore more tuple spaces in the list fail to contain a tuple matching the template given in the primitive – when the process unblocks, n tuples are returned.

III. ReSpecT

A. Tuple Centres

A *tuple centre* is basically a tuple space extended with the notion of behavior specification, that is, a set of rules defining how the tuple centre reacts to incoming/outgoing communication events, such as the insertion or the reading of a tuple [2]. This makes a tuple centre a *programmable coordination medium* [7], in that the behavior of the medium enabling the interaction – in this case a tuple space – is not fixed (as in the case of LINDA), but can be (dynamically) programmed by means of a specific language: in this way, the coordination laws governing the interactions and accomplishing the intended behavior of the whole system can be enforced by suitably programming the medium.

In the case of tuple centres, the behavior specification is expressed in term of a *reaction specification language* that associates a (possibly empty) set of computational activities, called *reactions*, to each communication event occurring in a tuple centre. Reactions are executed sequentially, and with a transactional semantics: only successful reactions affect the tuple center state. Also, the effect of all the reactions triggered by a single communication event is perceived *as a whole* by the coordinated entities, that is, they perceive one such effect as an atomic event occurring when all successful reactions have been executed.

B. ReSpecT as a specification language

ReSpecT is a logic language used as a reaction specification language for tuple centres. In ReSpecT, reactions are defined as sequences of *reaction goals*, built as logic atoms from the list of the ReSpecT reaction predicates, as reported for instance in [8]. In short, reaction predicates can be divided in three classes: *(i)* predicates to access and modify the space of the tuples, *(ii)* predicates to access information about the triggering communication event, and *(iii)* predicates to compute over logic terms. A complete and formal definition of ReSpecT is reported in [8], while in [9] (a subset of) ReSpecT is proven to be Turing-equivalent.

However, since Turing-equivalence has been shown to be not the only meaningful measure of expressiveness for coordination models [6], in [10] the ReSpecT specification language was extended with a new reaction predicate, belonging to none of the three classes reported above: `out_tc`. Informally, reaction goal `out_tc(n,t)` succeeds if and only if ground term `n` is an admissible identifier for a coordination medium, and results in inserting logic tuple `t` into tuple centre `n`. Roughly speaking, the result of `out_tc(n,t)` is perceived by tuple centre `n` as an `out(t)` operation issued by the same agent a responsible for the communication operation $o(t)$ that triggered the execution of the reaction goal within tuple centre n. ([10] reports on the formal semantics of `out_tc`.)

As a simple example, consider a tuple centre `tc` programmed with the follow naive reaction providing a simple logging capability to the medium:

```
reaction(out(Tuple),(
  current_agent(Who), current_tc(Where),
  out_tc(logsA,log(Who,Where,Tuple)),
  out_tc(logsB,log(Who,Where,Tuple))
)).
```

When an agent issues an `out` operation with any tuple on the `tc` tuple centre, such as `tc ? out(p(1))` the reaction is triggered, and its body executed: a `log` tuple with information about the agent identity, the tuple centre name `tc`, and the

```
% block 1 - initial setting on request
reaction(in(and(TCList,Tuple,_)),
  current_agent(A), current_time(T),
  out_r(serving(inand,A,T,TCList,Tuple)),
  out_r(count(A,T,0,up)),
  out_r(inand(A,T,TCList,Tuple))
).
%  block 2 - splitting in requests
reaction(out_r(inand(A,T,[TC|TCList],Tuple)),
  in_r(inand(A,T,[TC|TCList],Tuple)),
  in_r(count(A,T,N,up)), N1 is N + 1,
  out_r(count(A,T,N1,up)),
  out_tc(TC,inand(A,T,Tuple)),
  out_r(inand(A,T,TCList,Tuple))
).
%  block 3 - final setting on request
reaction(out_r(inand(A,T,[],Tuple)),
  in_r(inand(A,T,[],Tuple)),
  out_r(got(A,T,[]))
).
%  block 4 - handling responses
reaction(out(inand(TC,A,T,Tuple)),
  in_r(count(A,T,N,_)), N1 is N - 1,
  out_r(count(A,T,N1,down)),
  in_r(got(A,T,TupleList)),
  out_r(got(A,T,[(TC,Tuple)|TupleList]))
).
%  block 5 - building answer tuple
reaction(out_r(count(A,T,0,down)),
  in_r(count(A,T,0,down)),
  in_r(got(A,T,TupleList)),
  in_r(serving(inand,A,T,TCList,Tuple)),
  out_r(and(TCList,Tuple,TupleList))
).
```

Fig. 1. ReSpecT specification for tuple centre `logOp`: case `in` (AND (...)).

tuple `p(1)` is inserted by means of two `out_tc` invocations in two different tuple centres, `logsA` and `logsB`.

This extension enables ReSpecT to specify coordination laws that relate distinct tuple centres, and thus different coordination flows. In particular, it is possible to manifest/expose the occurrence of specific coordination events or states of a tuple centre to another tuple centre. As far as the expressiveness is concerned, this extension does not extend the power of ReSpecT in terms of Turing equivalence, but makes a step forward in the direction of expressiveness in the context of Sequential Interaction Machines (SIM) [11], i.e. in terms of SIM expressiveness [6].

IV. ReSpecTING LOGOP

While ReSpecT and LOGOP extend LINDA, they also adopt complementary approaches. On the one hand, LOGOP tries to address the limited expressiveness of the basic LINDA coordination language, by providing primitive aggregation in form of logical operators over tuple spaces, without changing the basic behavior of the single tuple space as a coordination medium. On the other hand, ReSpecT preserves the basic LINDA primitives and their default behavior, but extends the power of the coordination medium, adopting tuple centres as programmable tuple spaces [2]. As a result, LOGOP and ReSpecT conceptually act at two different levels of abstraction: while ReSpecT represents a sort of low-level language for coordination, empowering

```
% block 1 - serving requests immediately
reaction(out(inand(A,T,Tuple)),
  current_tc(TC),
  in_r(Tuple),
  in_r(inand(A,T,_)),
  out_tc(logOp,inand(TC,A,T,Tuple))
).
% block 2 - serving requests on updates
reaction(out(Tuple)),
  in_r(Tuple),
  in_r(inand(A,T,Tuple)),
  current_tc(TC),
  out_tc(logOp,inand(TC,A,T,Tuple))
).
reaction(out_r(Tuple)),
  in_r(Tuple),
  in_r(inand(A,T,Tuple)),
  current_tc(TC),
  out_tc(logOp,inand(TC,A,T,Tuple))
).
```

Fig. 2. ReSpecT behavior specification for other tuple centres: : case `in` (AND (...)).

the coordination space with programmability, LOGOP extend the high-level coordination language, providing agents with more expressive primitives. Since ReSpecT works as a sort of assembly language for coordination, it is natural to try to exploit it as a platform for implementing a high-level LOGOP-like language. In the following, we sketch a possible architecture of a ReSpecT-based LOGOP-like coordination language.

First, we suppose that ReSpecT tuple centre `logOp` represents our LOGOP kernel. This can be simply instrumented by suitably defining the `logOp` behavior as a ReSpecT program: Figure 1 presents the ReSpecT code for handling `in` primitive in the case of logical operator AND. Any agent willing to perform a LOGOP primitive can simply send the appropriate request tuple to the `logOp` tuple centre, that will process it according to the LOGOP semantics and produce the required response tuple. For instance, the query

```
in(and(TCList,Tuple,Answer))
```

represents the request from a logic agent for an $\texttt{in}_{and}$ operation asking one tuple matching `Tuple` to each tuple centre in `TCList`. The querying (logic) agent receives the required `Answer` as a list of pairs `(TC,Tuple)` from the `logOp` tuple centre. Correspondingly, whenever `logOp` receives a request of that form, it first reifies the request (block 1 in Figure 1), then splits it in the corresponding number of the requests for the involved tuple centres, (blocks 2 and 3), finally gets the tuples from the tuple centres (block 4) and builds up the response tuple (block 5).

The other tuple centres only need to embed the behavior specification that enables them to handle the requests from tuple centre `logOp`. For instance, Figure 2 shows the simple ReSpecT specification allowing tuple centres to properly implement LOGOP $\texttt{in}_{and}$.

Even though for the sake of brevity we do not show here the ReSpecT code for all the LOGOP primitives, the same basic architecture presented for $\texttt{in}_{and}$ can be used to make a ReSpecT tuple centre (`logOp` in the example) the LOGOP

core handling any kind of request, also sharing most of the behaviors (and of the ReSpecT code) between the different LOGOP primitives.

V. SEMANTIC ISSUES

In this section we provide a formal semantics to LOGOP by naturally extending the usual semantics framework of LINDA. We show that the semantics of LINDA is indeed different from the one induced by the ReSpecT mapping of LOGOP, the latter being intrinsically more non-deterministic due to asynchronous communication between tuple centres.

A. One-space Coordination: The Semantics of LINDA

Here we discuss the basic semantics aspects of the LINDA coordination model as typically described in the literature (see for instance [12]), which includes the safety properties of LINDA but abstracts away from the tuple matching mechanism. Coordinated entities can be modeled as finite, nondeterministic processes sequentially executing some LINDA coordination primitive, e.g. considering only the case of operations `out`, `rd`, and `in`. Formally, the syntax of these processes can be expressed using a CCS-like notation [13] as follows:

$$\begin{array}{lcl} \pi & ::= & out \mid in \mid rd \\ \alpha & ::= & \pi(x) \\ P & ::= & 0 \mid \alpha.P \end{array}$$

0 denotes the void (or terminated) process, $\alpha.P$ the process performing a primitive operation α and then executing the continuation P.

A (coordinated) system $S \in \mathcal{S}$ is then syntactically denoted by syntax:

$$S ::= 0 \mid x \mid P \mid S \parallel S$$

for which the following equivalence rules are supposed to hold:

$$\begin{array}{c} 0 \parallel S \equiv S \qquad S \parallel S' \equiv S' \parallel S \\ (S \parallel S') \parallel S'' \equiv S \parallel (S' \parallel S'') \end{array}$$

Each element S can be considered as a system configuration, which is a finite composition of processes and of items x, denoting tuples occurring in tuple space.

So, a semantics can be given to LINDA coordination model by describing the possible evolutions of a system configuration. This can be conveniently done by a transition system $\langle S, \longrightarrow \rangle$, where relation $\longrightarrow \subseteq S \times S$ is used to associate to a system configuration the possible configurations it may move to. This relation is defined as the smallest relation satisfying the three rules:

$$out(x).P \parallel S \longrightarrow P \parallel S \parallel x \qquad [L-OUT]$$

$$rd(x).P \parallel S \parallel x \longrightarrow P \parallel S \parallel x \qquad [L-RD]$$

$$in(x).P \parallel S \parallel x \longrightarrow P \parallel S \qquad [L-IN]$$

B. Multi-space Coordination: The Semantics of LOGOP

We provide an extension to this semantic framework so as to model LOGOP. The syntax of processes is as follows:

$$\begin{array}{lcl} \pi & ::= & out \mid in \mid rd \\ \lambda & ::= & ts \wedge \ldots \wedge ts \mid ts \otimes \ldots \otimes ts \mid ts \vee \ldots \vee ts \\ \alpha & ::= & \pi(\lambda, x) \\ P & ::= & 0 \mid \alpha.P \end{array}$$

Differently from LINDA model, here standard LINDA primitives `out`, `in`, and `rd` are applied specifying a logic-like formula involving the identifier ts of one or more tuple spaces, according to the intuition described in Section II. Given a set of tuple space identifiers $Ts = \{ts_0, ts_1, \ldots ts_n\}$, operation $\pi(ts_0 \wedge ts_1 \wedge \ldots \wedge ts_n, x)$ – which can also be conveniently denoted by notation $\pi(\bigwedge_{ts \in Ts} ts, x)$ – means that primitive π has to be executed on all tuple spaces in set Ts, underlying the concept of logic AND. Similarly, $\pi(ts_0 \otimes ts_1 \otimes \ldots \otimes ts_n, x)$ – or $\pi(\bigotimes_{ts \in Ts} ts, x)$ – means that primitive π has to executed on one of the tuple spaces in Ts (exclusive OR). Finally, $\pi(ts_0 \vee ts_1 \vee \ldots \vee ts_n, x)$ – or $\pi(\bigvee_{ts \in Ts} ts, x)$ – means that primitive π has to be executed on at least one of the tuple spaces in Ts (inclusive OR).

A (coordinated) system $S \in \mathcal{S}$ is now syntactically denoted by syntax:

$$S ::= 0 \mid \langle ts, x \rangle \mid P \mid S \parallel S$$

so that a system configuration is described as a finite composition of processes and of items $\langle ts, x \rangle$, denoting datum (tuple) x occurring in tuple space (with identifier) ts. Similar to the case of λ logic formulae, given a set of tuple spaces $Ts = \{ts_0, \ldots, ts_n\}$, we write $\prod_{ts \in Ts} \langle ts, x \rangle$ as a shorthand for $\langle ts_0, x \rangle \parallel \ldots \parallel \langle ts_n, x \rangle$, with $\prod_{ts \in \{\}} \langle ts, x \rangle$ naturally meaning void system 0.

We provide a semantics to LOGOP enforcing a property of atomic execution, that is, the effect of executing a LOGOP operation is perceived as atomic by all the coordinated entities living in the system. In order to define this semantics analogously to the simple case of LINDA, we first introduce a relation σ, so that $\sigma(\alpha, S)$ associates to an operation α the set of tuples that may be involved in its execution, i.e., the tuple inserted by an `out`, read by a `rd`, or removed by a `in`. This relation is defined as the smaller one satisfying the rules:

$$\sigma(\pi(\bigwedge_{ts \in Ts} ts, x), \prod_{ts \in Ts} \langle ts, x \rangle)$$

$$\sigma(\pi(\bigotimes_{ts \in Ts} ts, x), \langle ts_0, x \rangle) \qquad \text{with} \qquad ts_0 \in Ts$$

$$\sigma(\pi(\bigvee_{ts \in Ts} ts, x), \prod_{ts' \in Ts'} \langle ts', x \rangle) \quad \text{with} \qquad \{\} \subset Ts' \subseteq Ts$$

Then, the semantics of LOGOP is given by transition system $\langle \mathcal{S}, \longrightarrow \rangle$ over set $\mathcal{S}$, where transition relation $\longrightarrow \subseteq \mathcal{S} \times \mathcal{S}$ is

defined by rules:

$$\frac{\sigma(out(\lambda,x),S')}{out(\lambda,x).P \parallel S \longrightarrow P \parallel S \parallel S'} \quad [OUT]$$

$$\frac{\sigma(rd(\lambda,x),S')}{rd(\lambda,x).P' \parallel S \parallel S' \longrightarrow P \parallel S \parallel S'} \quad [RD]$$

$$\frac{\sigma(in(\lambda,x),S')}{in(\lambda,x).P \parallel S \parallel S' \longrightarrow P \parallel S} \quad [IN]$$

As an example, valid transitions include:

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

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

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

Since semantics is given to logic formulae λ only by means of relation σ, and since $\pi(\bigwedge_{ts\in\{ts_0\}} ts, x)$, $\pi(\bigvee_{ts\in\{ts_0\}} ts, x)$, and $\pi(\bigotimes_{ts\in\{ts_0\}} ts, x)$ are associated by σ to the same datum $\langle ts_0, x\rangle$, then we naturally denote these three kinds of λ by the shorthand ts_0. An operation involving a λ of this kind is called a *mono-space* operation, which can be easily shown to correspond to LINDA semantics as described in previous section. Other operations are called *multi-space*.

C. Distributed Coordination: Weak vs. Strong Semantics

The semantics provided to LOGOP in Subsection V-B can be referred to as *strong-synchronous*, in that by this semantics all the coordinated entities of the domain perceive the execution of a LOGOP primitive as atomic. On the other hand, the semantics given to LOGOP by the mapping in Section IV can be referred to as *weak-synchronous*: only the entity executing a primitive perceives it as atomic, while other entities may interact with tuple spaces affected in a partial way by the primitive. Consider the system configuration:

$$\begin{aligned} S \;=\; & in(ts_1 \wedge ts_2, x).out(ts_3, x) \parallel \\ & in(ts_1 \wedge ts_2, x).out(ts_3, x) \parallel \\ & \langle ts_1, x\rangle \parallel \langle ts_2, x\rangle \end{aligned}$$

describing two equivalent processes willing to consume tuple x from both spaces ts_1 and ts_2, and then inserting tuple x in space ts_3. By strong-synchrony, the execution of `in` is atomic, hence only one process may consume the two tuples and then put datum x in ts_3. In fact, the evolution represented by:

$$S \longrightarrow^* in(ts_1 \wedge ts_2, x).out(ts_3, x) \parallel \langle ts_3, x\rangle$$

is the only admissible one for the system S.

On the other hand, by weak-synchrony, the execution of a primitive can make another entity perceive the system state in a spurious state. For the above system configuration, the following evolution may occur. First of all, one process may remove x from ts_1 and still wait to remove it from ts_2 as well, which can be described by a transition of the kind:

$$\begin{aligned} S \longrightarrow_w \; & in(ts_2).out(ts_3, x) \parallel \\ & in(ts_1 \wedge ts_2).out(ts_3, x) \parallel \langle ts_2, x\rangle \end{aligned}$$

Then, because of non-atomicity, the other process may execute its primitive thus causing x to be removed from ts_2, with transition:

$$\begin{aligned} & in(ts_2).out(ts_3, x) \parallel in(ts_1 \wedge ts_2).out(ts_3, x) \parallel \langle ts_2, x\rangle \\ & \longrightarrow_w in(ts_2).out(ts_3, x) \parallel in(ts_1).out(ts_3, x) \end{aligned}$$

As a result, the whole system remains blocked in a state where both processes are waiting to complete their `in` primitive, and x has been removed from both ts_1 and ts_2.

VI. CONCLUSION

In the world of coordination models, the quest for expressiveness has driven researchers into proposing various models that claim to be expressive. It may not be surprising that some these models see expressiveness from a different angle. This paper focused on two of these models, namely LOGOP and ReSpecT, which are respective high level and low level coordination models.

We first described how can LOGOP be modeled in terms of reactions in ReSpecT tuple centres. The second part of the paper provided a formal semantics (referred to as strong-synchronous) for LOGOP induced by the straightforward extension to LINDA semantics, which naturally interprets LOGOP informal specification. This semantics is shown to be different than the one induced by the ReSpecT mapping (referred to as weak-synchronous), which is more appropriate for distributed systems and still conforms to the LOGOP informal description, hence retaining its expected advantages.

Providing a formal definition of weak-synchrony semantics, showing its conformance to the given ReSpecTmapping, and studying formal comparison with respect to strong-synchrony are left for future works.

REFERENCES

[1] Jim Snyder and Ronaldo Menezes, "Using logical operators as an extended coordination mechanism in Linda," in *Coordination Languages and Models*. 2002, vol. 2315 of *LNCS*, pp. 317–331, Springer-Verlag.

[2] Andrea Omicini and Enrico Denti, "From tuple spaces to tuple centres," *Science of Computer Programming*, vol. 41, no. 3, pp. 277–294, Nov. 2001.

[3] T. Malone and K. Crowston, "The interdisciplinary study of coordination," *Computing Surveys*, vol. 26, no. 1, pp. 87–119, 1994.

[4] David Gelernter, "Generative communication in Linda," *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 1, pp. 80–112, 1985.

[5] Andrea Omicini, Franco Zambonelli, Matthias Klusch, and Robert Tolksdorf, Eds., *Coordination of Internet Agents: Models, Technologies, and Applications*, Springer-Verlag, Mar. 2001.

[6] Mirko Viroli, Andrea Omicini, and Alessandro Ricci, "On the expressiveness of event-based coordination media," in *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'02)*, Hamid R. Arabnia, Ed., Las Vegas, NV, USA, 24–27 July 2002, vol. III, pp. 1414–1420, CSREA Press.

[7] Enrico Denti, Antonio Natali, and Andrea Omicini, "Programmable coordination media," in *Coordination Languages and Models*, David Garlan and Daniel Le Métayer, Eds. 1997, vol. 1282 of *LNCS*, pp. 274–288, Springer-Verlag, 2nd International Conference (COORDINATION'97), 1–3 Sept. 1997, Berlin (D), Proceedings.

[8] Andrea Omicini and Enrico Denti, "Formal ReSpecT," in *Declarative Programming – Selected Papers from AGP'00*, Agostino Dovier, Maria Chiara Meo, and Andrea Omicini, Eds., vol. 48 of *Electronic Notes in Theoretical Computer Science*, pp. 179–196. Elsevier Science B. V., 2001.

[9] Enrico Denti, Antonio Natali, and Andrea Omicini, "On the expressive power of a language for programming coordination media," in *1998 ACM Symposium on Applied Computing (SAC'98)*, Atlanta (GA), 27 Feb. – 1 Mar. 1998, pp. 169–177, ACM, Track on Coordination Models, Languages and Applications.

[10] Alessandro Ricci, Andrea Omicini, and Mirko Viroli, "Extending ReSpecT for multiple coordination flows," in *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'02)*, Hamid R. Arabnia, Ed., Las Vegas, NV, USA, 24–27 July 2002, vol. III, pp. 1407–1413, CSREA Press.

[11] Peter Wegner, "Why interaction is more powerful than computing," *Communications of the ACM*, vol. 40, no. 5, pp. 80–91, May 1997.

[12] Nadia Busi, Roberto Gorrieri, and Gianluigi Zavattaro, "A process algebraic view of Linda coordination primitives," *Theoretical Computer Science*, vol. 192, no. 2, pp. 167–199, 1998.

[13] Robin Milner, *Communication and Concurrency*, Prentice Hall, 1989.