Concurrent Access to Multiple Tuple Spaces with \textit{LogOp}

Ronaldo Menezes \( ^{*} \) Andrea Omicini \( ^{†} \) James Snyder \( ^{‡} \) Mirko Viroli \( ^{§} \)

Abstract

In the last two decades, researchers working in coordination were able to demonstrate that distributed-systems models are made of two distinct parts: a computation and a coordination model. Among a number of coordination models available today, LINDA is perhaps the most successful. LINDA enables processes to interact via associative shared memories called tuple spaces—with no need for direct communication between processes, communication can be made decoupled in space and time. Out of the many changes and extensions to the original model, the development from a single-tuple-space model into a multiple-tuple-space variation is seminal. However, the coordination mechanism was never fully extended to explore the potential offered by the availability of multiple tuple spaces spread over a distributed environment.

Along this line, this paper describes an extension to the LINDA model, called \textit{LogOp} (Linda with Logical \textit{O}perators), where processes can use logical operators to combine tuple spaces on-the-fly. This paper argues that \textit{LogOp} provides a simple and elegant coordination mechanism to deal with the intricacies of complex distributed systems, by exploiting the full potential of the multiple-tuple-space abstraction. A formal semantics for the logical operators is given and the implementation described and tested. Results indicate that \textit{LogOp} can yield an implementation that efficiently deals with multiple tuple spaces in a distributed environment.

\textbf{Keywords:} Coordination Systems, Expressiveness of Coordination Models, Formal Semantics.

1 Introduction

Complex systems of today are usually built out of several heterogeneous components (objects, processes, agents, . . .) dipped into dynamic, open and distributed environments. Managing the space of the interaction among components is a matter of \textit{coordination} \cite{23}—many emerging fields such as mobile computing, grid computing \cite{8}, amorphous computing \cite{1}, and large scale Internet applications call for non-trivial models, patterns and technologies for handling component’s coordination.

Starting from the pioneering work by Gelernter \cite{10}, a number of different coordination models have been designed and proposed to harness the complexity of interaction in systems engineering. LINDA \cite{10} is likely to be the best known example of such models: tuple spaces working as shared blackboards are used as a means to mediate and govern the interactions between different entities,

\begin{itemize}
  \item \( ^{*} \)CCI, Department of Computer Sciences, Florida Tech, \texttt{mailto:rmenezes@cs.fit.edu}
  \item \( ^{†} \)DEIS, Alma Mater Studiorum—Universit\`a di Bologna a Cesena, \texttt{mailto:andrea.omicini@unibo.it}
  \item \( ^{‡} \)CCI, Department of Computer Sciences, Florida Tech, \texttt{mailto:Jim-Snyder@cfl.rr.com}
  \item \( ^{§} \)DEIS, Alma Mater Studiorum—Universit\`a di Bologna a Cesena, \texttt{mailto:mirko.viroli@unibo.it}
\end{itemize}
promoting temporal and spatial decoupling. The LINDA model has been experimented to face a wide class of different application problems, and is arguably the most important and most successful coordination model ever proposed. Such a statement can be easily verified by looking at both the academic (e.g., TuCSoN [22, 21], LIME [24], Ligia [15]) and commercial (e.g., TSpaces [12], JavaSpaces [9], and GigaSpaces [14]) implementations that spawned from LINDA’s simple idea of generative communication [10].

Since its initial proposal [10], LINDA has undergone a series of changes and modifications: among the many others, the extension from a single-tuple-space model to a multiple-tuple-space model of communication [11] can be surely said to be of prime importance, for it improved the model’s ability to deal with the intricacies and complexity of open distributed systems.

In spite of the vast amount of research in LINDA-based models and infrastructures, a considerable effort is still apparently needed in order to improve the suitability of the model to areas of computer science that require advanced coordination patterns. Despite the fact that some good specialized extensions of LINDA exist for a variety of fields or problems, it seems that most of them have neglected to fully explore the potential of the features that LINDA already provides. Along this line, this paper focuses on the multiple-tuple-space abstraction, by observing that the current semantics of the LINDA primitives prevents a process to access many tuple spaces simultaneously.

In order to address such a limitation, we introduce LogOp, a LINDA-based model meant to fully exploit the potential of the multiple-tuple-space abstraction, by providing an extended coordination mechanism that adopts logical operators at the primitive level to access more than one tuple space with a single coordination primitive. Since LogOp makes it possible to invoke coordination primitives involving more than one tuple space at a time, 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.

On the one hand, this extension enhances the expressiveness and flexibility of the basic LINDA model. On the other hand, and more generally, 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, and mandates for a non-ambiguous formal description of the model. In this paper, the \textit{LogOp} model is formally described alongside an implementation of the proposal model.

2 Coordination in LINDA

2.1 LINDA and Multiple Tuple Spaces

LINDA provides processes with primitives to store and retrieve tuples from tuple spaces. Although the names of the primitives vary slightly from implementation to implementation, the functionality they provide is usually very similar. On the one hand, processes are able to store tuples in a tuple space using the primitive \texttt{out}. On the other hand, processes can retrieve tuples from a tuple space using the \textit{getter} primitives \texttt{in} and \texttt{rd}; they take a tuple template (a definition of a tuple, or of a set of tuples) and use associative matching to non-deterministically retrieve one tuple of the desired sort: while \texttt{in} removes a matching tuple, \texttt{rd} takes a copy of the tuple. If a matching tuple is not found in the tuple space, the process invoking the primitive blocks until a matching tuple can be retrieved. In addition to these primitives, some LINDA implementations provide non-blocking versions of \texttt{in} and \texttt{rd}, called \texttt{inp} and \texttt{rdp} respectively. These primitives have the same semantics of their blocking counterparts when a matching tuple is found in the tuple space: however, they behave significantly different if a matching tuple cannot be found, as in this case they immediately fail instead of blocking.

All primitives in LINDA have the following characteristics:

1. They require the process to define the primitive’s scope of visibility, which always consists of only one tuple space—the place where the operation takes place.

2. Ordering is the norm. A sequence of \texttt{out} primitives is reflected in the order of the tuples inserted in the space—this is due to the \texttt{out}-ordering problem \cite{6}.\footnote{The out-ordering problem is connected to the synchronicity of \texttt{out}: if \texttt{out} is synchronous, a sequence of \texttt{outs} emitted by a process will reach the tuple space in the same order in which it is sent, viceversa ordering is not} A sequence of \texttt{in} and
rd primitives is instead ordered owing to their blocking characteristic—an answer to a primitive must be provided before the next is processed.

3. They deal with one tuple at a time. Either one tuple is retrieved, removed or stored.²

While standard LINDA assumes a single tuple space, its many multiple-tuple-spaces variants [11, 5] allow for multiple disjoint tuple spaces. By disjoint it is meant that processes can only access one tuple space at a time: each primitive invocation then gets/puts a single tuple from/to a single tuple space. Once the multiple-tuple-space abstraction is adopted, different coordination flows are typically built upon different tuple spaces, where process coordination is expressed in terms of the occurrence/absence of some tuples in distinct and possibly distributed tuple spaces [26].

In particular, shared access to distributed resources is naturally mapped upon distributed tuple spaces, each one representing a resource. Consider for instance the case of a vacuum cleaner vc, and of a number of cleaning agents, each one in charge of a different room in the same building, and sharing the vacuum cleaner for their activity. The typical LINDA-based solution would involve a tuple space associated to the vacuum cleaner—say, tvc—containing all the useful information on the tool (like, its state and location), as well as a tuple used for agent synchronisation—say, ["vcFree"]—whose occurrence in tvc indicates that the vacuum cleaner is available for use. Correspondingly, an agent willing to start cleaning would ask for the vacuum cleaner by performing an in(tvc, ["vcFree"]) invocation on the tvc tuple space. In case the tuple ["vcFree"] is already there, vc is available, the tuple is removed from the tvc tuple space, and the agent can start cleaning. When done cleaning, the agent makes the vacuum cleaner available again to other agents by restoring the synchronization tuple to the tuple space tvc with an out(tvc, ["vcFree"]) operation. In case the tuple ["vcFree"] is not there—some other agent is using the vacuum cleaner—the operation is just suspended, and resumed only when the tuple ["vcFree"] is restored to the tuple space, and the vacuum cleaner vc is ready for another use.

 ensured to be maintained. According to [6], this dramatically affects the expressive power of the coordination model.

²Some Linda-based systems include non-standard primitives that are able to deal with bulk movement of tuples, often called bulk-primitives [4].
2.2 Limits of LINDA with Multiple Tuple Spaces

In the same way they cannot handle more than one tuple, LINDA primitives cannot handle more than one tuple space at a time. So, any coordinated activity involving more than one tuple space at a time cannot be handled atomically, through a single coordination operation, but requires instead the composition of many accesses to the tuple spaces involved. Although this composition (typically, serialization) is reasonable in some cases, it seriously limits the ability of LINDA to express coordination whenever the contents of different tuple spaces need to be considered as a whole. More generally, any coordination policy based on the simultaneous occurrence of some tuples in more than one tuple space cannot be properly expressed in LINDA: basically, non-trivial coordination policies are beyond the LINDA capabilities.

While this could seem just a matter of expressive power, which could be addressed through some suitably-built software layer on top of LINDA, this is no longer true when considering the blocking nature of getter primitives in and rd—which was the main motivation behind LogOp along with the fact that they can only deal with one tuple space at a time. In fact, when coordination depends on the absence/occurrence of some tuples in more than one tuple space, it is often the case that it cannot be expressed by any composition/serialization of LINDA primitives.

Consider a simple extension to the case discussed in previous subsection: instead of a single vacuum cleaner vc, a number N of vacuum cleaners vc1, vc2, ..., vcN are available to cleaning agents. Since they are physically distributed, each vacuum cleaner vc_i is associated to a tuple space tvc_i, and shared through the use of a synchronizing tuple ["vcFree"] in the associated tuple space, in the same way discussed above for a single vacuum cleaner. Since vacuum cleaners are supposed to be interchangeable, a cleaning agent just need (any) one of them to complete its task: so, it can proceed if at least one tuple ["vcFree"] occurs in any of the tuple spaces tvc_i.

In LINDA, this cannot be expressed through any composition of in operations: so, no LINDA-based library or API can be built to solve this problem. In fact, since in is suspensive, it can only be used to wait for a single vacuum cleaner to get free—how can this be chosen a priori, independently of its current state. This potentially leads to inefficient use of the resources, such as when all the agents are waiting for a given vacuum cleaner vc_j, while other vacuum
cleaners $vc_i, i \neq j$ are ready for use. Moreover, it may also have disruptive effects, such as when a vacuum cleaner $vc_j$ is unusable for any reason, thus any cleaning agent waiting for it is suspended indefinitely. Figure 1 shows the naïve pseudo-code of a LINDA-based cleaning agent trying to get any of the vacuum cleaners $vc_i$ by iteratively consuming a tuple "$vcFree" from any of the tuple spaces $tvc_i$.

Before we can go any further it should be clear that this solution is not only incorrect but it also does not express what the agent actually wants to do. The use of $\text{in}$ would force the agent to block on the tuple space $tvc_1$ first, and unblock only when a tuple "$vcFree" can be retrieved from there—only when $vc_1$ is available, independently of state of the other vacuum cleaners. Once the first $\text{in}$ succeeds, the $\text{break}$ statement completes the $\text{for}$ cycle: the rest of the cycle is completely irrelevant as it is never executed—so, each agent is able to use only the first resource in the list, as defined at design time. So, for instance, if $vc_2$ becomes free when the cleaning agent is waiting for $vc_1$, the agent stays blocked until $vc_1$ becomes free, and can not use $vc_2$ before. In general, once the access to the tuple spaces is serialized (once an order is imposed), the agent always looks for the tuple in the first tuple space in the sequence and then remain blocked on that tuple space until the operation is completed successfully.

Some LINDA variants assume that blocking operations can time-out—then have apparently more chances to solve the problem of our example. A solution like the one in Figure 2 may apparently work, even though in a very unsatisfactory way: once an $\text{inp}$ times-out, the subsequent $\text{if}$ statement condition becomes false$^3$, so that all the resources are searched iteratively until one of them is found available for use. Instead, introduction of timeouts to the semantics of the blocking primitives does not solve the problem.

First of all, this solution would be extremely inefficient. The polling caused by the agent

\footnote{Assuming that timing out generates an empty tuple.}
while (isEmpty(t)) do
    for (l := {tvc_1, tvc_2, ..., tvc_N}; l := next(l)) do
        tvc := first(l); t := inp(vc, ["vcFree"]);
        if (isEmpty(t)) then
            useVacuumCleaner(tvc);
            out(tvc, ["vcFree"]);
            break;
        fi
    od
od

Figure 2: Pseudo-code of a LINDA-based cleaning agent using inp.

execution make its use prohibitive, in particular in distributed scenarios: at each loop iteration, the agent makes up to as many requests as it is the number \( N \) of the resources, and the loop is repeated indefinitely until one available resource is found. In general, is not difficult to see that if \( P \) processes constantly rely on this solution to get data from \( N \) tuple spaces, and \( C \) is the average number of cycles through all tuple spaces required to each process before the required tuple is found, then the LINDA kernel may quickly become a bottleneck: for the total number of requests will grow proportionally to \( P \times N \times C \).

Nevertheless, the problem of the inp solution is not merely its inefficiency. Instead, the problem lays exactly in the absence of synchronization of the in primitive: inp does not wait for a tuple to become available in the tuple space, as the in does—so agents cannot wait for any vacuum cleaner to become available. So, it may in principle happen that every vacuum cleaner is not available when an agent is looking for it, then get free when the agent is looking through the others vacuum cleaners, and then become busy again for the use by other cleaning agents—so, in general, starvation is possible when using this approach.

Even though quite simple, this example is quite realistic indeed: any problem of shared distributed resources can present issues of such sort—but much more complicated in real-world scenarios. Despite its popularity, and the number of applications where it is currently used, LINDA is essentially unable to properly express a large number of relevant coordination problems, ranging from the simplest to the most complex ones. Accordingly, this paper proposes to extend the LINDA primitives addressing the issue of concurrent access to multiple tuple spaces. While this is not meant to solve all the possible coordination problems, it is anyway a necessary step to be taken toward the full exploitation of the potential of the LINDA coordination model in

---

4Here we abstract from the obvious dependence of \( C \) on both \( P \) and \( N \).
complex, real-world scenarios.

3 The LogOp Model

While expressiveness was the most relevant argument for LogOp model (as reported in Section 2), efficiency was never a minor concern in its development. In fact, it could be argued that models have no direct connection with the efficiency issues, and that any feasible model can be then implemented more or less efficiently. However, a good, well-thought model is likely to enable and promote better implementations, where the model features—in terms of expressive, powerful and orthogonal abstractions—are straightforwardly embedded into an effective and well-performing implementation.

In the context of LINDA-based models, very few have looked into improving the efficiency of their implementations by working on the model features. As a main example, Rowstron’s work on efficiency of open LINDA systems yielded two interesting models whose implementation were shown to be more efficient than implementations of the standard LINDA model. It is assumed that tuple spaces are organized in a hierarchy [27], and the concept of asynchronous primitives is added [28]. While Rowstron’s approaches adopt performance as the main target—so that implementations are meant to be efficient even if expressiveness is lost—LogOp’s primary concern is expressiveness, without forgetting performance concerns. In fact, LogOp tries to achieve good performance at the implementation level as a consequence of suitable choices at the model level.

LogOp was developed to tackle problems such as the one described in the previous section, where tuples have to be searched through more than one tuple space at a time. The choice made by LogOp was to experiment with logical operators, allowing tuple spaces to be combined in a single LINDA operation via basic logical operators such as AND, OR and XOR. So, LogOp extends LINDA with the feature of combining tuple spaces with logical operators.

LogOp logical operators may be used in conjunction with most primitives. However, for the sake of simplicity and clarity, this paper focuses on the original standard LINDA primitives (in, rd and out), whose semantics is formally defined in Section 4. Instead, non-blocking primitives (inp and rdp) are discussed only informally in the rest of this section.

The BNF (Backus-Naur Form) grammar in Figure 3 describes the general syntax of the LogOp
where

\[
\begin{align*}
<\text{PRIMITIVE}> & ::= \text{handle} \mid <\text{LOP}> (<\text{TSS}>)
\text{\textit{< SCOPE >}} & ::= \text{handle} \mid \text{< TUPLE >}
\text{\textit{< TSS >}} & ::= \text{handle} \mid \text{< TUPLE >}
\text{\textit{< LOP >}} & ::= \text{and} \mid \text{or} \mid \text{xor}
\text{\textit{< PRIMITIVE >}} & ::= \text{in} \mid \text{rd} \mid \text{out}
\text{\textit{< TUPLE >}} & ::= \text{tuple} \mid \text{template}
\end{align*}
\]

Figure 3: The BNF for the LogOp primitives.

primitives. Symbol TUPLE ranges over tuples or templates; PRIMITIVE over LINDA primitives in, rd and out, LOP over LogOp logical operators, TSS over non-empty lists of handles—identifiers of tuple spaces. Finally, SCOPE ranges over expressions denoting scopes, which can be either a single handle or a logical operator applied to a list of handles. The grammar is meant to show what type of expression can be used in the LogOp model. For instance, it can generate expressions such as \(\text{in}(\text{OR}(\text{ts1,ts2,ts3}),[?\text{string}])\) or \(\text{out}(\text{AND}(\text{tvc}_1,\text{tvc}_4),["\text{vcFree}"]))\).

Another issue to be addressed is the subset of logical operators being considered. Since at the best of our knowledge LogOp is the first and only model extending LINDA with logical operators for combining multiple tuple spaces, we intentionally choose here to limit full description to the operators AND, OR and XOR (as shown in Figure 3): any further addition of logical operators (such as NOT, NAND and NOR) would have added in complexity with no obvious gain, and would have possibly hindered some of the main contributions of this paper.

To help understanding, we first introduce the main extensions of LogOp to LINDA informally in the rest of the section, in a very simple and straightforward way—the semantics of each logical operator is described extensively in the context of the different coordination primitives. Then, in Section 4 we present the LogOp model formally in order to avoid the risk of ambiguities and misunderstandings—which is typical of any distributed setting—, and to support for reasoning about abstract properties of LogOp.

### 3.1 The OR Operator

The OR operator has the effect of combining tuple spaces so that processes can store and retrieve tuples from \textit{any} of the tuple spaces from a list without having to impose an unnecessary order on the way the tuple spaces are accessed. The idea is relatively simple but it has very interesting
consequences. Roughly speaking, the word *any*, as used here, refers to the ability of a process to
deal with tuple spaces as if all the tuples where part of a bigger tuple space containing all the
tuples.

The usefulness of such an operator can be easily understood by recalling the example of
previous section—which cannot be faced satisfactorily by single-space Linda primitives. When
a cleaning agent needs a vacuum cleaner—*any* of the vacuum cleaners—it can simply combine
the $tvc_1, tvc_2, \ldots, tvc_N$ tuple centres with the OR operator, and ask for a ["vcFree"]
tuple with an \texttt{in} operation over such a combined space of tuples. As a result, if and when a
tuple ["vcFree"] appears in any of the OR-combined tuple spaces—that is, whenever any of the
vacuum cleaners gets free—then the tuple is removed and returned to the requesting agent, that
is, the cleaner is assigned to the agent for use.

The semantics of OR can be first expressed independently for each \texttt{LogOp} primitive. Let $LT$
be the list of tuple spaces specified in a primitive, $|LT|$ its cardinality, and $t$ be the number of
tuples involved in the operation (being retrieved or stored):

- The informal semantics of OR is such that all tuple spaces are considered as one when the
  request is received, but the result is given as if the operation had been applied to some of
  the tuple space individually. In order words, the number $t$ of tuples involved is such that
  $0 < t \leq |LT|$.

- The above introduces another level of non-determinism to the model. The number $t$ of
tuples (and tuple spaces) involved in the operation is non-deterministic.

The non-determinism introduced by the OR operator in \texttt{LogOp}, is better understood if explained
separately for getter (\texttt{in/rd}) and setter (\texttt{out}) primitives. In the context of \texttt{out}, non-determinism
is quite simple, and just comes from the fact that $t$ tuple spaces (where $0 < t \leq |LT|$) are non-
deterministically chosen and then subject to insertion.

Non-determinism is less trivial when involving \texttt{in} and \texttt{rd} alongside with the OR operator,
when a tuple matching the given template is looked for in the tuple spaces defined in $LT$. Each
tuple space $ts \in LT$ may return (at most) one tuple. Non-determinism here comes from the fact
that not all tuple spaces are in principle guaranteed to be actually searched if the execution of
vclist := in(OR(tvc_1,tvc_2,...,tvc_N),["vcFree"])
tvc := chooseVacuumCleaner(vclist);
out(AND(vclist\{tvc}, ["vcFree"]);
useVacuumCleaner(tvc);
out(tvc, ["vcFree"]);

Figure 4: Pseudo-code of a LogOp-based cleaning agent using in along with the OR operator.

the primitive has already been satisfied—at least one tuple has already been found in other tuple spaces.

Since both in and rd are blocking primitives, LogOp must define a blocking/failure semantics for the primitives, that is, the primitive (provisionally) fails and is then suspended if none of the tuple spaces in LT contain a matching tuple at the time of the invocation—so the process blocks. Then, the process unblocks when in the end the primitive succeeds, that is, when (at least) one tuple matching the process’ request is placed in any of the tuple spaces involved, and can be finally retrieved.

Of course, this indicates that the OR operator in combination with an in/rd may return a list of tuples—not merely one. For the primitive to succeed, one matching tuple must be found in at least one of the tuple spaces in the list—the use of OR may return a list of tuples if more than one tuple space searched has a matching tuple. For instance, when a process executes in(OR(ts1,ts2,ts3),[?string]), it blocks if no tuples matching the template [?string] can be found in any of the tuple spaces listed. Instead, if a matching tuple can be found in at least one of these tuple spaces, the process will not block—the number of tuples returned can be up to 3 in this example.

With a better understanding of the primitive, we can now show the solution for the agent problem introduced in Section 2 expressed in LogOp using OR. Such a solution is reported in Figure 4: from there, one may easily observe that LogOp is able to express the solution in a very elegant way. This simple example already shows a case where LogOp expressiveness has an impact on the solution: the cleaning agent obtains a list of free vacuum cleaners with an in primitives over an OR-scope of tuple spaces, chooses the one to use among them, makes the other vacuum cleaners available to other agents, uses the cleaner of choice, finally sets it free when done.

Predicative versions of in and rd—that is, inp and rdp—never blocks, but do not guarantee
that a tuple will be removed. In both LINDA and LogOp, they either succeed or fail: when a matching tuple can be found (and is retrieved) in at least one tuple space, they succeeds, behaving exactly like their blocking counterparts, returning a list of matching tuples. The difference is when a matching tuple cannot be found in any of the tuple spaces: while the primitives in and rd block, inp and rdp would fail, returning the empty list.

The idea that primitives may return a list of tuples instead of a single tuple is also a consequence of the extension of the primitives to the multiple-tuple-space model. The decision to make the getter primitives (in, inp, rd, rdp) return a list of tuple spaces is essentially based on the logic behind the operator OR. If one assumes that there is a true or false value associated with a tuple space (depending on whether the tuple space contains a matching tuple or not), one could use boolean logic to find out the action that results from the execution of a getter primitive. Since any number of false values combined with or result in false, the primitive either blocks or fails—this causes the process to block in the case of in and rd, to fail in the case of inp and rdp. On the other hand, if at least one tuple space has the value true the result of the or operator will also be true meaning that the primitive succeeded in getting one or more tuples.

Here we highlight the fact that some variations of LINDA are able to deal with bulk tuples [28, 12]. We do no claim to be introducing such a concept but rather the fact that the standard primitives in LogOp also have a bulk semantics. This has minimum influence on the need for bulk primitives which are still required. In fact, LogOp and bulk primitives are meant to address different issues: bulk primitives were originally introduced to deal with multiple tuples within a single tuple space, LogOp is essentially designed to deal with individual tuples within multiple tuple spaces. For instance, the LogOp semantics of the standard primitives does not allow a programmer to get all tuples matching a template that are stored in a tuple space. For this we still either use bulk primitives, or simulate them using e.g. programmable tuple spaces à la ReSpecT [20].

In some cases we might want to get only one tuple no matter how many tuple spaces are defined in the list LT. For instance, in the example of the cleaning agent discussed just above, one may decide that giving a choice to the cleaning agent makes things unnecessarily complicated.
Instead, we might prefer the \textit{LogOp} primitive to return just a single free vacuum cleaner among the possibly many available. This is done with the operator \texttt{xor}, explained in the next subsection.

\section{The XOR Operator}

The \texttt{xor} operator is similar to the \texttt{or} operator to the extent it considers \textit{any} of the tuple spaces from a list without imposing an order to the way tuples are accessed. The difference here is that only \textit{one} tuple is returned. In essence, the \texttt{xor} also combines all the tuple spaces in the list $LT$ as one but the number $t$ of tuples returned must be equal to 1.

So, let $t$ the number of tuples involved in the operation (being retrieved or stored):

- The \texttt{xor} operator conceptually considers \textit{all} tuples spaces as one ($t = 1$ independently of $|LT|$).
- One can still observe the non-determinism present in \texttt{xor}. The tuple involved in the operation can be stored or retrieved from any of the tuple spaces defined in $LT$.

Again, the non-determinism mentioned above is better understood if explained separately for getter (\texttt{in/rd}) and setter (\texttt{out}) primitives. The use of an \texttt{in} and \texttt{rd} alongside with the \texttt{xor} operator makes \textit{LogOp} look for a matching tuple in all the tuple spaces defined in $LT$. The primitives only block if a tuple cannot be found. If more than one tuple is available the primitive \textit{always} returns a single tuple, non-deterministically chosen among all the matching tuples in all the tuple spaces included in the list. For instance, if a process executes \texttt{in(xor(ts1,ts2),[?int])}, it blocks if neither \texttt{ts1} nor \texttt{ts2} have a matching tuple. If they (combined) have at least one matching tuple, the primitive succeeds and a tuple is returned. It is crucial to point out that, like \texttt{or}, there is no ordering being imposed in the access of the tuple spaces defined in the list $LT$.

With respect to the example discussed in the previous subsection, a further, simpler solution to the cleaning agent problem can be easily expressed in \textit{LogOp} using operator \texttt{xor} instead of \texttt{or}, as shown in Figure 5. There, the agent is just returned with a single handle to an available vacuum cleaner: so, no choice is required, the vacuum cleaner is just obtained, used, then set free again.

The semantics of \texttt{xor} for any of the getter primitives when the a tuple is not present is exactly the same as the semantics for the operator \texttt{or}—the process either blocks (\texttt{in, rd}) or
\[ tvc := \text{in}(\text{XOR}(tvc\_1, tvc\_2, \ldots, tvc\_N), ["vcFree"]) \]

\[ \text{useVacuumCleaner(tvc);} \]

\[ \text{out}(tvc, ["vcFree"]); \]

Figure 5: Pseudo-code of a LogOp-based cleaning agent using \text{in} along with the XOR operator. receives a failure signal (inp, rdp) depending on the getter primitive being used.

In the case of the use of an \text{out}, the operator XOR is used to non-deterministically choose a tuple space in the list to store the tuple. Contrasting with OR, XOR always chooses one tuple space from the list LT defined in the primitive. For instance, \text{out}(\text{XOR}(ts1, ts2), ["Hello"]) will store the tuple ["Hello"] in either ts1 or ts2—not in both.

### 3.3 The AND operator

The AND operator allows processes to consider all tuple spaces in a list. The motivation for the AND operator as means of combining tuple spaces ranges from event notification to broadcast of information.

The use of AND with an \text{out} allows processes to store a tuple in all tuple spaces defined in LT in just one step. For instance, the OR solution to the cleaning agents problem shown in Figure 4 handles the phase after the choice of the vacuum cleaner in a very simple way: a simple \text{out} of the ["vcFree"] over an AND-scope of tuple spaces (\text{out}(\text{AND}(vclist\_tvc), ["vcFree"])) is required to set all the unneeded vacuum cleaners free with just a single agent operation.

The use of AND with an \text{out} is an example of how the LogOp model also affects efficiency. In fact, although broadcast is easily expressed in LINDA, the LogOp version of this primitive is not merely more elegant, but is also likely to generate a more efficient implementation. If \( n \) tuple spaces are involved in the operation, LINDA would have to execute the \text{out} primitive \( n \) times. This operation could be very expensive due to the startup overhead associated with sending \( n \) messages separately to the kernel. In LogOp only one message is sent to the kernel. Once the message is in the kernel, it is stored in the individual tuple spaces. For a better understanding of the costs associated with LINDA primitives refer to [28].

The AND operator can also be used in getter primitives. The semantics of these primitives is such that the process will block if one or more tuple spaces in the list LT fail to contain a tuple matching the template given in the primitive. Similar to the other cases, the decision on whether
the primitive fails—the process blocks—is made based on the concept that tuple spaces have the \texttt{true} or \texttt{false} values associated with them. For an \texttt{AND} to produce \texttt{true} all tuple spaces that are considered in the primitive must have value \texttt{true}—they all must have a matching tuple. When the getter primitives succeed, the number of tuples returned is exactly $|LT|$—one from each tuple space.

The predicative version of the getter primitives do not block. Therefore, their semantics consists of failing and returning an empty list when at least one tuple space in $LT$ does not contain a matching tuple. That is, when they are used with an \texttt{AND} operation to combine tuple spaces, these primitives either succeed, returning a list containing $|LT|$ tuples, or fail, returning the empty list.

4 The \textit{LogOp} calculus

In this section we introduce a formal foundation to the \textit{LogOp} coordination model, with the goal of providing a precise description of some of its more distinctive features. Technically, we introduce \textit{LogOp} as a concurrent language, or calculus, borrowing a number of standard techniques from the process algebraic settings [2]. This approach is standard for coordination models—it has been used for LINDA in [3] and is generally applicable when endorsing the viewpoint of coordination as a language [29]—for it provides a good support to represent aspects related to concurrency and interaction.

Syntactically, the language expresses the allowed configurations of a \textit{LogOp} distributed system, including the state of all tuple spaces (or dataspace for short in the following) as well as the interactive state of all client processes. An operational semantics is introduced by means of a transition system, operationally characterizing the allowed dynamics for a given system configuration, that is, the admissible evolutions of the processes and the tuple spaces. In the end, this formalization allows us to precisely characterize the possible outcomes of executing a \textit{LogOp} primitive on the tuple spaces in the system, and can e.g. be used to investigate on possible deadlock situations.

In this model we focus on synchronization aspects while neglecting the true communication part: this is obtained by avoiding the tuple matching mechanism, namely, exact tuples instead
of templates are looked for reading or removing. This simplification is a quite typical approach in the formalization of coordination models (see e.g. [4]); it is worth noting that LogOp does not introduce peculiar novelties on communication aspects with respect to the standard Linda model.

4.1 Syntax

Developing on the concrete syntax expressed in Section 3, the structure of LogOp system configurations is defined by the abstract syntax:

\[
\begin{align*}
\sigma & ::= 0 \mid [\sigma] \mid h \mid \sigma \land \sigma \mid \sigma \lor \sigma \mid \sigma \oplus \sigma & \text{Scopes} \\
p & ::= \text{in} \mid \text{rd} \mid \text{out} & \text{Primitives} \\
a & ::= p(\sigma, x) & \text{Actions} \\
P & ::= 0 \mid a.P \mid P + P & \text{Processes} \\
S & ::= 0 \mid P \mid \langle h, x \rangle \mid S \parallel S & \text{Systems configurations}
\end{align*}
\]

Meta-variable \( h \) ranges over handles, which are tuple space identifiers; non terminal symbols of the grammar are used for the other meta-variables: \( \sigma \) ranges over scopes of LogOp primitives; \( p \) over Linda primitives; \( a \) over the set \( A \) of actions (also called operations); \( P \) over processes, and \( S \) over system configurations. Scopes can be void (0) or obtained by composing handles through the binary operators \( \land \) (AND), \( \lor \) (OR), and \( \oplus \) (XOR). A scope can also be of the kind \([\sigma]\), meaning the scope \( \sigma \) can be optionally considered: this concept is used in the operational semantics to generate those temporary cases due to operator OR, where as much tuples as possible are to be considered, non-deterministically. Processes are either the void process 0, a process \( a.P \) that executes action \( a \) and then proceeds with \( P \), or the choice \( P + P' \) between a process \( P \) or \( P' \).

A system \( S \) is the parallel composition through operator “\( \parallel \)” of processes \( P \) and elements \( \langle h, x \rangle \), respectively representing a client of LogOp tuple spaces and a tuple \( x \) occurring in space \( h \).

We assume operators “\( \parallel \)” for system configurations, “\( + \)” for processes, and “\( \land \)”, “\( \lor \)”, and “\( \oplus \)” for scopes be associative, commutative, and absorb the term 0.

An example configuration is the following

\[
\text{in}(h_1 \land h_2, x) . \text{out}(h_3, x) . 0 \parallel \langle h_1, x \rangle \parallel \langle h_2, x \rangle
\]

modelling one process executing action \( \text{in}(h_1 \land h_2, x) \) and then \( \text{out}(h_3, x) \) immersed in a dataspace with tuple \( x \) in space \( h_1 \) and \( h_2 \).
4.2 Semantics for Scopes

As for the common practice with concurrent languages, operational semantics is defined here in terms of a transition system (TS). This is divided in two parts, one which gives semantics to the language for scopes, and the other which defines the impact of the former on the overall system configurations. The first TS is a triple \( \langle \Sigma, \rightarrow, K \rangle \), where the set \( \Sigma \) of all possible scopes is used as carrier set of the transition system, \( K = H \cup \{ \tau \} \) (ranged over by meta-variable \( k \)) is the set of transition labels including handles \( h \in H \) or the silent action \( \tau \), and \( \rightarrow \subseteq \Sigma \times K \times \Sigma \) is the transition relation. Write \( \sigma \xrightarrow{k} \sigma' \) for \( \langle \sigma, k, \sigma' \rangle \in \rightarrow \): in particular \( \sigma \xrightarrow{h} \sigma' \) means scope \( \sigma \) allows for accessing tuple space \( h \) and correspondingly moves to scope \( \sigma' \), and \( \sigma \xrightarrow{\tau} \sigma' \) means scope \( \sigma \) might spontaneously move to scope \( \sigma' \). Rules for this transition relation are as follows:

\[
\begin{align*}
 h & \xrightarrow{h} 0 & \text{[ACT]} \\
 \sigma \land \sigma' & \xrightarrow{k} \sigma'' \land \sigma' & \text{if } \sigma \xrightarrow{k} \sigma'' & \text{[AND]} \\
 \sigma \oplus \sigma' & \xrightarrow{k} \sigma'' & \text{if } \sigma \xrightarrow{k} \sigma'' & \text{[XOR]} \\
 \sigma \lor \sigma' & \xrightarrow{k} \sigma'' \land [\sigma'] & \text{if } \sigma \xrightarrow{k} \sigma'' & \text{[OR]} \\
 [\sigma] & \xrightarrow{h} \sigma' & \text{if } \sigma \xrightarrow{h} \sigma' & \text{[PICK]} \\
 [\sigma] & \xrightarrow{\tau} 0 & \text{[SKIP]}
\end{align*}
\]

This operational semantics is at the core of the LogOp model, as it expresses the allowed sequences of tuple spaces that might be accessed due to a given scope. For instance, an evolution

\[
\sigma_0 \xrightarrow{h_1} \sigma_1 \xrightarrow{\tau} \sigma_2 \xrightarrow{h_2} \sigma_1 \xrightarrow{h_3} \sigma_3 \xrightarrow{\tau} 0
\]

would mean that an operation over scope \( \sigma_0 \) may result in accessing space \( h_1, h_2, \) and then \( h_3 \)—with some internal, silent action \( \tau \) in between.

Rule [ACT] trivially manages the case where a scope is simply a single handle. Rule [AND] deals with logic conjunction: the left alternative is picked \( (\sigma \xrightarrow{k} \sigma'') \) and correspondingly moves to its new state, with the right alternative being unchanged \( (\sigma \land \sigma' \text{ moves to } \sigma'' \land \sigma') \). Recall the \( \land \) operator is commutative and associative: this means that because of rule [AND] any alternative out of \( n \) \( (n > 0) \) can be actually picked. Similarly, rule [XOR] picks one of the alternatives, but then exclude the others \( (\sigma \oplus \sigma' \text{ moves to } \sigma'') \). Then rule [OR] picks one of the alternatives and allows it to proceed, but then the others become optional extensions for the scope \( (\sigma \lor \sigma' \text{ moves to } \sigma'' \land [\sigma']) \). Finally, An optional scope can either be picked and proceed by rule [PICK], or can spontaneously move to the void scope by rule [SKIP].
As a first example, we consider the scope $h_1 \oplus h_2 \oplus h_3$, which should allow either $h_1$ or $h_2$ or $h_3$ to be accessed. Because of commutativity and associativity of $\oplus$, and because of rule [XOR] we have:

$$h_1 \oplus h_2 \oplus h_3 \equiv h_1 \oplus (h_2 \oplus h_3) \xrightarrow{h_1} \Sigma 0$$
$$h_1 \oplus h_2 \oplus h_3 \equiv h_2 \oplus (h_1 \oplus h_3) \xrightarrow{h_2} \Sigma 0$$
$$h_1 \oplus h_2 \oplus h_3 \equiv h_3 \oplus (h_1 \oplus h_2) \xrightarrow{h_3} \Sigma 0$$

With operator $\land$, instead, all handles have to be accessed using any order, and this is correctly dealt with by rule [AND]:

$$h_1 \land h_2 \land h_3 \equiv h_3 \land (h_1 \land h_2) \xrightarrow{h_3} \Sigma 0 \land (h_1 \land h_2) \equiv h_1 \land h_2 \xrightarrow{h_1} \Sigma 0 \land h_2 \equiv h_2 \xrightarrow{h_2} \Sigma 0$$

Finally, rule [OR] along with [PICK] and [SKIP] handle the non-determinism required by operator OR, in that at least one of the tuple spaces should be accessed, but possibly more:

$$h_1 \lor h_2 \lor h_3 \equiv h_3 \lor (h_1 \lor h_2) \xrightarrow{h_3} \Sigma 0 \lor (h_1 \lor h_2) \equiv [h_1 \lor h_2] \xrightarrow{\tau} \Sigma 0$$
$$h_1 \lor h_2 \lor h_3 \equiv h_3 \lor (h_1 \lor h_2) \xrightarrow{h_3} \Sigma 0 \lor (h_1 \lor h_2) \equiv [h_1 \lor h_2] \xrightarrow{h_1} \Sigma 0 \lor [h_2] \equiv [h_2] \xrightarrow{h_2} \Sigma 0$$

The combination of all rules is then able to deal with the interplay between the operators, as in the following case (where equivalences between scopes are not reported for brevity):

$$(h_1 \land h_2) \lor (h_3 \oplus h_4) \xrightarrow{h_3} \Sigma [h_1 \land h_2] \xrightarrow{h_2} \Sigma h_1 \xrightarrow{h_1} \Sigma 0$$
$$(h_1 \land h_2) \lor (h_3 \oplus h_4) \xrightarrow{h_1} \Sigma h_2 \land [h_3 \oplus h_4] \xrightarrow{h_3} \Sigma h_2 \xrightarrow{h_2} \Sigma 0$$

...
\[
\begin{align*}
p(0,x).P & \rightarrow P & \text{[SEQ]} \\
p(\sigma,x).P & \rightarrow p(\sigma',x).P & \text{if } \sigma \tau \rightarrow \Sigma \sigma' & \text{[TAU]} \\
\text{out}(\sigma,x).P & \rightarrow \text{out}(\sigma',x).P \parallel \langle h,x \rangle & \text{if } \sigma \rightarrow \Sigma \sigma' & \text{[OUT]} \\
\langle h,x \rangle \parallel \text{rd}(\sigma,x).P & \rightarrow \text{rd}(\sigma',x).P \parallel \langle h,x \rangle & \text{if } \sigma \rightarrow \Sigma \sigma' & \text{[RD]} \\
\langle h,x \rangle \parallel \text{in}(\sigma,x).P & \rightarrow \text{in}(\sigma',x).P & \text{if } \sigma \rightarrow \Sigma \sigma' & \text{[IN]} \\
P + P' & \rightarrow P'' & \text{if } P \rightarrow P'' & \text{[SUM]} \\
S \parallel S' & \rightarrow S'' \parallel S' & \text{if } S \rightarrow S'' & \text{[PAR]}
\end{align*}
\]

Rule [SEQ] allows a process continuation \( P \) to carry on when the prefix action has void scope, i.e. it has just finished. Rule [TAU] propagates a scope spontaneous move at the process level, modelling the case where an internal \( \lor \) operator is stopping considering optional handles. Rules [OUT], [RD] and [IN] define the semantics of standard LINDA primitives out, rd and in, as defined e.g. in [3]: out has the effect of inserting tuple \( \langle h,x \rangle \) in the space, rd is executed if the tuple occurs in the space, and in removes the tuple from the space; in all the three cases as the scope moves from \( \sigma \) to \( \sigma' \) (with label \( h \)) the process \( p(\sigma',x).P \) is allowed to carry on after the operation is executed. Finally, rules [PAR] and [SUM] handle the semantics of parallel composition and choice as in standard process algebras [19].

As an example, we consider the process \( \text{out}(h_1 \land h_2,x).P \), inserting tuple \( x \) in both tuple spaces \( h_1 \) and \( h_2 \). According to rules [AND] and [OUT], two evolutions are admissible from this state, as the tuple \( x \) can be inserted first either in \( h_1 \) or \( h_2 \):

\[
\begin{align*}
\text{out}(h_1 \land h_2,x).P & \rightarrow \text{out}(h_2,x).P \parallel \langle h_1,x \rangle \rightarrow P \parallel \langle h_1,x \rangle \parallel \langle h_2,x \rangle \\
\text{out}(h_1 \land h_2,x).P & \rightarrow \text{out}(h_1,x).P \parallel \langle h_2,x \rangle \rightarrow P \parallel \langle h_2,x \rangle \parallel \langle h_1,x \rangle
\end{align*}
\]

As another example, we finally consider rules [OR] and [IN] by system configuration \( \text{in}(h_1 \lor h_2 \lor h_3,x).P \parallel \langle h_1,x \rangle \parallel \langle h_2,x \rangle \). Operator \( \lor \) makes tuple \( x \) being removed either from \( h_1 \), \( h_2 \) or both. Some allowed evolutions are:

\[
\begin{align*}
\text{in}(h_1 \lor h_2 \lor h_3,x).P \parallel \langle h_1,x \rangle \parallel \langle h_2,x \rangle & \rightarrow \text{in}(h_2 \lor h_3,x).P \parallel \langle h_2,x \rangle \\
& \rightarrow \text{in}(h_3,x).P \rightarrow \text{in}(0,x).P \rightarrow P \\
\text{in}(h_1 \lor h_2 \lor h_3,x).P \parallel \langle h_1,x \rangle \parallel \langle h_2,x \rangle & \rightarrow \text{in}(h_2 \lor h_3,x).P \parallel \langle h_2,x \rangle \\
& \rightarrow \text{in}(0,x).P \parallel \langle h_2,x \rangle \rightarrow P \parallel \langle h_2,x \rangle
\end{align*}
\]

The syntax and semantics of this calculus as expressed in this section have a one-to-one mapping to a MAUDE model checker specification [7]. This allows for the automatic verification

\[\text{19}\]
of properties: starting from a configuration of given tuples and processes, properties concerning safety, liveness and deadlock-freedom can be expressed using LTL logic and proved by model checking.

5 Implementation

A LogOp prototype has been implemented using Java, which is used here to show the model’s feasibility and to report some measurement of performance. LogOp relies on an open client-server architecture where the number of servers and clients is not pre-determined. Communication is solely via sockets. In order to avoid bottlenecks, multi-threading is extensively used. Each LogOp server has four threads: two dealing with server requests and two dealing with client requests. A pair of threads is used to connect servers and clients in order to make the task of receiving objects and sending objects concurrent.

On the client side, in order to deal with large quantities of primitives to be processed, the two LogOp primitive types—getters and setters—have their own thread. The execution of the setters (out) is taken independently so as to guarantee out-ordering [6].

Servers can be added to the system at any time. When a server starts it broadcasts to the all the current servers that it is now available in the system. A list of current servers is maintained in a centralized repository and used as the basis for the broadcast.

Currently, the client tries to connect to a server in the same machine (same IP address) and if one is not available a server is chosen randomly from the list of servers available in the system. After the connection is established, the server chosen becomes the default server for this client. We intend to avoid the random choice for a default server by using heuristics to choose the best server based on metrics such as: physical proximity (by analyzing the IP addresses), logical proximity (by keeping track of delays in contacting servers), or load balancing (by knowing the current load of the servers and then using this as part of the decision process when choosing a server).

LogOp uses the concept of distributed central servers, where the contents of each individual tuple space are not distributed across several servers—each tuple space is entirely stored in one server. When a server receives a request to create a tuple space, it creates it locally. One of
the properties of a tuple space is its location (IP number of the server where the tuple space is stored). This property is used to route requests that processes make to different tuple spaces.

When a client executes a getter primitive, the request is sent to the client process’ default server where the command is parsed and processed as needed. In the LogOp server, the operation is broken into smaller operations by a server component named the decomposer—executing a task conceptually similar to the expansions described in previous section. The server then analyze the list of tuple spaces used in the primitive and group them based on their IP addresses—the assumption is that tuple spaces with the same IP address are stored in the same server and can be handled together. The second stage of this process consists of sending requests to all servers involved in the primitive to be executed. These servers receive their list of tuple spaces, the primitive being executed, and the logical operator used to combine the tuple spaces. The client’s default server waits for the answers from all the other servers involved. As responses come back from other servers, their tuples (if any) are placed into a list object that will be returned to the client process. When all responses are received, the object is sent back to the client.

It should be noted that even though LogOp assumes that responses from the servers must be received, the responses may not consist of an actual tuple (or tuples). If no tuple can be found the answer may be a null value to indicate that no tuple is current available.

When a client executes a setter primitive such as an out, the request is sent directly to the client’s default server. Again the decomposer analyzes the list of tuple spaces and groups them based on their IP addresses. The tuple is then sent to each server involved in the operation so that they, in turn, can store the tuple in the tuple spaces. This process is slightly different when OR or XOR are the operator being used since the default server chooses one (or many) of the tuple spaces in the list to send the tuple. After the default server acknowledges that the message was received, the client is allowed to proceed (the primitive returns). It should be noted that the above does not mean to say that the an out is a blocking primitive. The need for an acknowledgment is mainly due to the out-ordering requirement [6].

In general terms, the decomposer is a parsing engine augmented with some control to handle the different forms of logical operators. The decomposer may need to cancel requests made to other servers if the primitive is satisfied with other responses. The decomposer also controls
the consistency of the system. For instance it ensures that tuples are not removed more than once from a tuple space. When requested, tuples are placed in a “under request” status which locks the tuple from other in operations. The “under request” status need to be canceled if the tuple has not been used or is no longer needed. Note that LogOp is not a fault-tolerant system. Cancelations of “under request” status requires safeguards (e.g. be transactional) if failures are possible—there are LINDA systems that are fault-tolerant and implement the necessary safeguards to avoid inconsistencies.

6 Experimental Results

This section covers results on the performance of the LogOp primitive rd. All tests were conducted on Sun Ultra 5 UNIX machines running Solaris 8 containing a single CPU with 128 MB of RAM. An additional Solaris 8 Unix box was used that contained four CPUs and GB of RAM. The Java version used is JRE 1.3.

Several external factors can influence the execution time of these experiments. The first is the Java Virtual Machine’s (JVM) garbage collection (GC). The point in time when and how long the GC would occur could influence the results of these tests. To lessen the effect of garbage collection each tuple space contained only one tuple.

Another factor deals with the ObjectOutputStream class provided by Java. Each object in the JVM contains an object identification (OID) number. As an object is written (serialized) to the output stream, its OID number is stored. If the same object is written more than once, the reference OID is sent on the socket instead of the whole object again. For Java, this saves the time it takes for an object to be serialized. To turn off the feature of storing OIDs, a reset() method provided by the ObjectOutputStream can be called after each object write is performed. To conserve memory during testing, a call to the reset() method every 5000 object writes flushes the OID table. This allows the tests to be executed without running out of memory, and avoiding an extensive amount of page swapping occurring on the 128 MB UNIX boxes. The tests here represent an average of at least 20 runs of the experiment. The results in Figure 7 describe the performance of LogOp when the OID table is flushed and when it is not.
6.1 Distributed Implementation

This suite of tests focuses on the distributed implementation of LogOp. Note that LogOp can be used as a standard LINDA implementation. If the cardinality of the list of tuple spaces is exactly one, LogOp would behave as a standard LINDA. In these experiments, the reference to LINDA pertains to using LogOp as a LINDA implementation.

This experiment demonstrates the concurrency aspect of the primitives by performing the same operation in LINDA-like style and LogOp-like style. The tests involve from two to eight servers containing a range of one to four hundred tuple spaces on a particular server—each tuple space contains exactly one tuple.

In order to deal with the serialization issue described above, these experiments are performed by calling the reset() method on the stream object, which in this first experiment we do after every five thousand operations.

Figure 6 shows the comparison between LogOp and LINDA. The experiments were run on respectively (a) two, (b) four, and (c) eight servers. The x-axis represents the number of tuples retrieved per primitive call and thus each numerical value along this axis represents the total number of tuples retrieved per rd primitive execution. The y-axis represents the average time in milliseconds to retrieve those tuples. For example, in Figure 6(a) , the last x-axis value is 800 and thus executing the LogOp rd primitive retrieving 800 tuples distributed across two servers took less than half a second; likewise, the LINDA rd primitive took over two and half seconds. LogOp performs this operation in one primitive call (using an AND of the tuple spaces involved) whereas the LINDA alternative is performed using 800 separate primitives calls (one for each tuple space).

6.2 The Effect of Serialization

The experiments showed in the previous section may be affected by Java serialization. As explained before, Java attempts to optimize the transmission of objects by maintaining a history of objects that have already been sent. In order to test the effect of serialization, we repeated the experiments above making sure that the entire object is serialized and sent between client and the LogOp server. This is achieved by calling the reset() method every time a tuple is being
transmitted. This clears the out any stored object IDs. Figure 7 shows that LogOp is still more efficient than LINDA even though the reset operation is still being used in the LINDA case.

7 Related Work

Merrick and Wood [18] have proposed to replace the idea of tuple spaces with the concept of Scopes. In their model tuples can belong to multiple scopes. Scopes are created either empty (no tuples) or by combining other existing scopes. In terms of expressiveness, Scopes is better than LINDA and it can indeed model more coordination patterns than LINDA. However, in order to solve problems as the one depicted in Section 2, a new scope will have to be created combining the two existing ones. While this assumption is not a problem in small systems, the creation of these scopes in large dynamic systems may force many extra structures to be created. Furthermore, a scope is a “permanent” structure that needs to be explicitly created. In LogOp, the model (LINDA) is not changed nor any extra structure needs to be created—tuple spaces are combined on-the-fly and only from the point of view of the process executing the primitive.

Bulk primitives, as well as many other LINDA extensions, can be also built as programmable behaviors of tuple spaces using ReSpecT tuple centres [20]. In particular, in [16] we showed how
LogOp could be built on top of ReSpecT, exploiting the linking primitive out_{tc}.

Another solution that aims at modeling the examples described in this paper is realized in event-driven coordination models such as Lime [25] and TSpaces [12]. In these models, processes can register for events in the kernel. Once processes are registered, the kernel notifies them every time events of interest take place. For instance, a process could register with two tuple spaces to receive any tuples that match the template [?int]. From the moment processes register onwards, if a tuple arrives in any of the two tuple spaces and this tuple matches the given template, the tuple is immediately sent to the process.

Event-driven coordination models are very powerful and are able to express diverse coordination patterns. Yet, these models are aimed at avoiding blocking of processes. Processes normally register for an event but carry on with their execution. Event-driven models are analogous to the idea of subscribing to a mailing list; once you say you want to receive messages, the messages will keep arriving in your mailbox until you cancel your subscription (cancel the event).

Jacquet and De Bosschere describe an extension of the μLog model where relations between blackboards can be defined [13]. However, the relations are normally defined as rules of pertinence of tuples in the blackboards. For instance, they are able to express with these rules that if a
tuple t is placed in blackboard A it should also be placed in blackboard B. Surely this is somewhat similar to an AND; the difference in LogOp is that rules are not created per se and no relation between tuple spaces is created in the context of the entire system. LogOp offers more flexibility as it allows users to define the scope of the primitive on-the-fly.

LogOp primary concern is with blocking primitives. It extends the model to allow a process to wait on several tuple spaces at the same time, without imposing any ordering. Another difference between LogOp and event-driven models is that events (normally tuples) are sent one at a time from the kernel to the process, while LogOp allows for several tuples to be retrieved at the same time. For instance the use of AND forces the kernel to return either N tuples or none. In an event-driven model this would be difficult to achieve since tuples would be sent to the process as they appear in the tuple space.

8 Future Work

Future work in LogOp involves first of all implementing a fully functional system where logical operators can be combined, e.g. as in in(or(ts1,AND(ts1,ts2)),[?int]). It would be desirable to have a module in LogOp that can apply simplification rules to the expressions involving tuple spaces. This on itself is a difficult task but should allow the implementation to work with more elaborate expressions. Yet, one may still need to understand the practical use of combine logical operators. It is unclear what should be the behavior of the processes when faced with more elaborate expressions such as the above one. Again, the implementation of a logical expression evaluator should shed some light on the semantics of this extension of LogOp.

It would be interesting to see how logical operators could be applied to templates. This would lead to a LogOp implementation with two levels of logical operators. In this system, one would be able to express: retrieve a tuple matching either template A or template B which is stored in either tuple space X or tuple space Z. This extension is being planned to be added to LogOp after the implementation of a more elaborate logical expression evaluator.

Moreover, deadlock issues may need be considered. LogOp implements a semantics with a weak notion of synchrony—as described in [16]—that may yield deadlocks due to the lack of global atomicity. Yet, the implementation of a strong-synchrony semantics may be too expensive
in practice, for it would require a full transactional management [16]. For instance, deadlocks may occur with the operation AND if two (or more) processes are accessing the same tuple spaces but the order they actually access them is different. A naive locking mechanism may lead to the deadlock. Suppose a process executes an in combining two tuple spaces, ts1 and ts2, using an AND. If the implementation is such that a matching tuple is retrieved from tuple space ts1 before knowing whether tuple space ts2 also has matching tuple, a deadlock may happen if these operations are performed while another process grabs a matching tuple from ts2 and blocks on ts1. This situation is not uncommon but it was not introduced by LogOp, standard LINDA primitives can have the same problem. The occurrence or not of the problem above related to the synchrony semantics for the primitives (as described in [16]).

Finally, the control of the blocked process by LogOp has to be done with care. The kernel may maintain a list of blocked processes and keep track of tuples being stored or retrieved using a tuple-monitoring sub-system [17] so as to be able to unblock processes whenever possible.

9 Conclusion

This paper describes an extension to the LINDA model meant to fully exploit the expressive power of multiple distributed tuple spaces. In fact, only simple coordination problems can be faced straightforwardly through the standard primitives of LINDA, even when multiple tuple spaces are available.

By introducing logical operators to be used in conjunction with basic LINDA primitives, LogOp allows a single coordination primitive to combine tuple spaces on-the-fly by need, and to access them altogether. Such an extension was showed to be quite a natural one, and its expressiveness apparently covers a wide class of non-trivial coordination problems.

A LogOp prototype has been implemented in order to test the model feasibility and measure its performance—the results of the experiments are promising. This seemingly confirms the expectation that suitably-designed changes at the model level would positively affect the performance of implementations.
References


