Florida Institute of Technology

Scholarship Repository @ Florida Tech

Theses and Dissertations

2-2002

Coordination using logical operators in LINDA

James Kendall Snyder
Florida Institute of Technology

Follow this and additional works at: https://repository.fit.edu/etd

Part of the Computer Sciences Commons

Recommended Citation

Snyder, James Kendall, "Coordination using logical operators in LINDA" (2002). *Theses and Dissertations*. 696.
https://repository.fit.edu/etd/696

This Thesis is brought to you for free and open access by Scholarship Repository @ Florida Tech. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of Scholarship Repository @ Florida Tech. For more information, please contact kheifner@fit.edu.

COORDINATION USING LOGICAL OPERATORS IN LINDA

James Kendall Snyder

A THESIS
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AT FLORIDA INSTITUTE OF TECHNOLOGY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE
IN
COMPUTER SCIENCE

Melbourne, Florida
February 2002

© Copyright by James Kendall Snyder 2002
All Rights Reserved

The author grants permission to make single copies

Approved for the University Committee on Graduate Studies:

"Coordination using Logical Operators in Linda"
a thesis by James Kendall Snyder

Ronaldo Menezes, Ph.D.
Assistant Professor
Computer Sciences
Major Adviser

Ryan Stansifer, Ph.D.
Associate Professor
Computer Sciences

Pei-feng Hsu, Ph.D.
Associate Professor
Mechanical and Aerospace Engineering

William Shoaff, Ph.D.
Associate Professor and Department Head
Computer Sciences

Abstract

TITLE: Coordination Using Logical Operators In Linda

AUTHOR: James Kendall Snyder

MAJOR ADVISOR: Ronaldo Menezes, Ph.D.

During the past 20 years, research in coordination has had success in demonstrating that distributed systems are made of two distinct parts: computation and coordination. One of the models that has contributed to the success of coordination is LINDA. While it is an extremely powerful coordination model, LINDA has limitations expressing parallel access to tuple spaces. This thesis proposes an extension to LINDA called LOGOP LINDA which combines tuple spaces using logical operators and thus enabling a single primitive to access multiple tuple spaces in parallel. Essentially this concept replaces serial access to multiple tuple spaces using the same primitive, tuple or template. Based on a single implementation consisting of LINDA and LOGOP LINDA, it is shown in the empirical results that LOGOP LINDA is more expressive, scalable, and efficient at both the model and implementation level.

Table of Contents

List of Figures

List of Tables

Declaration

This thesis is the result of my own work and the concept has been published at the following conference on Coordination:

- Fifth International Conference on Coordination Models and Languages (COORDINATION '02) in York, England, April 2002. Published by Springer, in the LNCS (Lecture Notes in Computer Science) [SM02].

Acknowledgments

I would like to first thank Dr. Ronaldo Menezes for his knowledge of LINDA and for his help in conceptualizing LOGOP LINDA. My appreciation also goes to my wife Laura, Alane, Jenn, Barb, Andy and Sarah for watching Austin on those special days so I could work on my thesis. I would also like to thank my proof readers Alane, Dan, Mom and Dad.

Lastly, a special and close to my heart thanks goes to my wife Laura for her endearing patience.

Dedication

To my loving wife, Laura and our beautiful son, Austin.

Chapter 1

Introduction

Computers exist to perform complex tasks in a short amount of time. If the software is written correctly, tasks should be completed with correct results every time they are executed. In addition, the more complex tasks become, the longer they will take to finish. Decomposing these tasks and distributing them among idle computer systems normally decreases the amount of time needed to complete the tasks.

Access to distributed shared resources such as hardware, software and data allows users not to be restricted to a single source of failure as with a centralized resource. If the centralized system is inaccessible, all computing power is lost. One of the first architectures to support distributed systems is Client/Server. The client software has access to one or more of the servers' resources. This type of access decreases the chance of a single point of failure. When a failure does occur, clients could access the same software or data on the backup server. As software accessed distributed resources, it became apparent that in order to have distributed computing, there had to be coordination among the systems.

Coordination is an important characteristic in distributed computing. It allows concurrent processes to synchronize and communicate with one another. Without synchronization and communication these processes could never interact. Unknowingly to each of them, they would be running concurrently and never be able to work together to achieve the intended result. As a consequence, three common models exist for coordination in a distributed system: message passing, monitors, and remote operations [And81]. In 1985, Gelernter [Gel85] proposed a fourth model that is different from the first three: the generative model. The generative model has two distinguishing properties: *communication orthogonality* and *free naming.* Communication orthogonality implies that a sending process does not need to know who the receiving process is; likewise, the receiving process does not need to know who was the sending process. This type of communication has two important consequences: time and space uncoupling. Time uncoupling is defined as two processes that do not have to exist at the same time in order to communicate. Space uncoupling means any process may receive data from any other process; similarly, any process may send data to any other process. A third effect based on space and time uncoupling is distributed sharing. Distributed sharing allows disjoint processes to share data that exists outside of the processes themselves; hence, the data exists in a global location accessible by all processes. The second property is free naming: this is the *type (or name)* of data being passed. This second property leads to *support for continuation passing* and *structured naming.* Support for continuation passing implies that a process can wait for data in the globally assessable location. It will stop waiting when a second process inserts the appropriate data into the global location. Structured naming implies that in order for a process to retrieve data from a global location the process must know the data's signature. More about these concepts will be covered in Chapter 2. Gelernter states that the message passing model most closely resembles the generative model: Message passing allows two or more processes in a system to directly communicate amongst themselves by common shared procedures. Hence, the generative model has some enhanced characteristics as described above that distinguish it enough from a message passing model.

Gelernter and Carriero [GC92] argue that a complete programming model consists of two separate pieces: a coordination model and a computation model. A computation model allows processes to create a solution while a coordination model allows process creation, synchronization and communication between processes. The authors state:

> "A coordination model is the glue that binds separate activities into
>
> an ensemble."[GC92]

Ensembles are an important concept and will be even more critical since software has migrated from centralized computing to distributed computing. Ensembles exemplify requirements of processes communicating and synchronizing to get a single task completed. Without these requirements, tasks could never be completed by more than one process attempting to work together. Furthermore, tasks would never be known to complete without communication.

Ciancarini [Cia97] formalized the definition of coordination. He stated that in order for two or more processes to synchronize and communicate, coordination could be described as a triple consisting of (E, M, L):

***E*:** Coordination Entities: An ensemble of processes that communicate with themselves.

***M*:** Media: The mechanism in which the ensembles communicate.

***L*:** Laws: The rules that allow ensembles to communicate using a finite number of primitives.

The generative communication paved way to the LINDA coordination created by Gelernter [Gel85]. This thesis extends Gelernter's work by focusing strictly on the media and laws of LINDA. Chapter 2 details some LINDA models that have been previously proposed. Based on Chapter 2, Chapter 3 illustrates the motivation for this thesis followed by the proposal in Chapter 4 of LINDA adopting logical operators creating LOGOP LINDA. An implementation written in Java is covered in Chapter 5 followed by a chapter on empirical results. The final chapter covers future research and a conclusion.

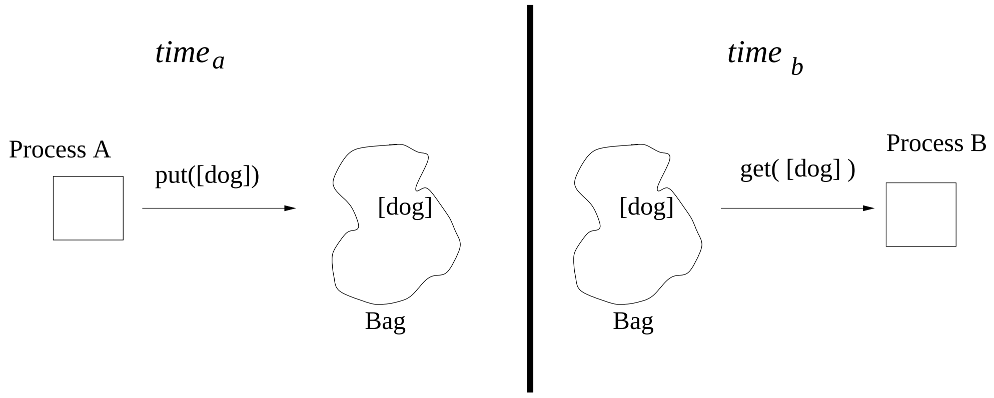
Chapter 2

Review

This chapter describes several models that are relevant to this thesis. The models are not described in complete detail but to a level that supports the thesis. The differences of these models compared to the proposal of LogOp Linda will be explained in detail in the next chapter. Complete description of each model can be found in the appropriate references.

In the Linda model, processes do not communicate directly with each other (peer to peer) but through the use of a "bag". A "bag" represents a medium of storage, something that is globally assessable or well known by all processes. The concept of the "bag" implies that the communicating processes are both time and space uncoupled. Time uncoupling allows one process (A) to insert an "item" into a well known "bag". At a later time, another process (B) can retrieve that "item" from the well known "bag". Thus, A and B are time uncoupled. An "item" represents a value that is placed into and removed from a "bag". Space uncoupling allows for processes to never know from which process the "item" came from; processes communicate through the use of the "bag". Recall that time and space uncoupling is a characteristic of communication orthogonality in the generative model described earlier. Figure 2.1 is a high level picture of this concept: At $time_a$ process A inserts an item "[dog]" into the bag. At a later time, $time_b$, process B retrieves the item "[dog]".

Figure 2.1 High level architecture of two processes communicating via a bag



Before moving forward, it is important to clarify the generic terms "bag" and "item" used earlier. LINDA has several new concepts that support the foundation of its model. A tuple, represented before as an "item", contains an ordered list of actuals (values) and are stored in tuple spaces (analogous to the term "bag"). Processes can retrieve a tuple from a tuple space if and only if a template associatively matches that tuple. A template consists of an ordered list of actuals, formals (definition of a value) or both. The concept of a template and tuple are similar with the exception that a tuple can never physically contain a formal: they only contain actuals. The term LINDA kernel, which can be interchanged with LINDA system, contains tuple spaces. In other words, LINDA has the characteristics of the generative model. The tuple space allows time and space uncoupling, and distributed sharing among disjoint processes. The associative matching, described more in depth below characterizes the structured naming property of the generative model.

Associative matching means that if $field_x$ is a formal or an actual in a template then it is equivalent to the actual in $field_x$ of the tuple. Successful associative matching implies all $fields_i$ in the template and the tuple are equivalent. Note, too, that the tuple and template must have the same number of fields. Table 2.1 exemplifies this concept. For example, in row #1 the tuple has "dog" and 3 as its field's values respectively and the template has String and Integer as its field's formals respectively: the template and tuple match. In row #2, the tuple has "dog" and 3 as its field's values respectively and the template has “dog” and Integer as its fields; likewise, $field_1$ of the template and tuple match and $field_2$ is an Integer 3; therefore a successful match. On the other hand, in row #4, the first formal is a Float in the template and it does not match the first field “dog” (String); thus, no successful associative matching.

Table 2.1 Associative Matching of template and tuple

Row	template	tuple	Match?
1.	[String,Integer]	[“dog”,3]	yes
2.	[“dog”,Integer]	[“dog”,3]	yes
3.	[“dog”,3]	[“dog”,3]	yes
4.	[Float,Integer]	[“dog”,3]	no
5.	[Integer,String]	[“dog”,3]	no

When there are two or more tuples in a tuple space that match a template, LINDA uses nondeterminism to choose which tuple to retrieve. A template containing the two formals String and Integer in $field_1$ and $field_2$, respectively, will match zero or more tuples in a given tuple space. If there is one tuple matching that template, then LINDA will return that tuple. If there are two or more tuples matching that template, then LINDA will select any tuple out of the set of matching tuples to return to the calling process. Menezes [Men98] defines nondeterminism as follows: Given relation R, R is non-deterministic if there exists x, y_1 and y_2 such that $x\ R\ y_1$, $x\ R\ y_2$ and $y_1 \neq y_2$.

2.1 Linda Primitives

In LINDA, processes do not communicate directly with each other; instead through the use of a tuple space called Universal Tuple Space. LINDA consists of two types of primitives used for communication: setters and getters. The getter (or “get”) primitives are `in`, `rd`, `inp` and `rdp` while the setter (or “set”) primitives are `out` and `eval`. It is these primitives that support the last characteristic (from Chapter 1), continuation passing, in the generative model.

The getter primitives can be decomposed into two groups: blocking and non-blocking. If the tuple does not immediately exist, blocking primitives will wait for a tuple that matches a `template` in the tuple space. Non-blocking primitives, on the other hand, return immediately if the tuple does not exist. The blocking getter primitives are `in` and `rd` while the non-blocking primitives are `inp` and `rdp`. Rows #1 through #4 in Table 2.2 summarizes the syntax of these primitives. All four primitives return a single tuple that matches the `template` parameter. The only difference is the `in` and `inp` primitives physically *remove* the tuple from the tuple space; unlike the `rd` and `rdp` primitives, they return a *copy* of the tuple leaving the original tuple in the tuple space.

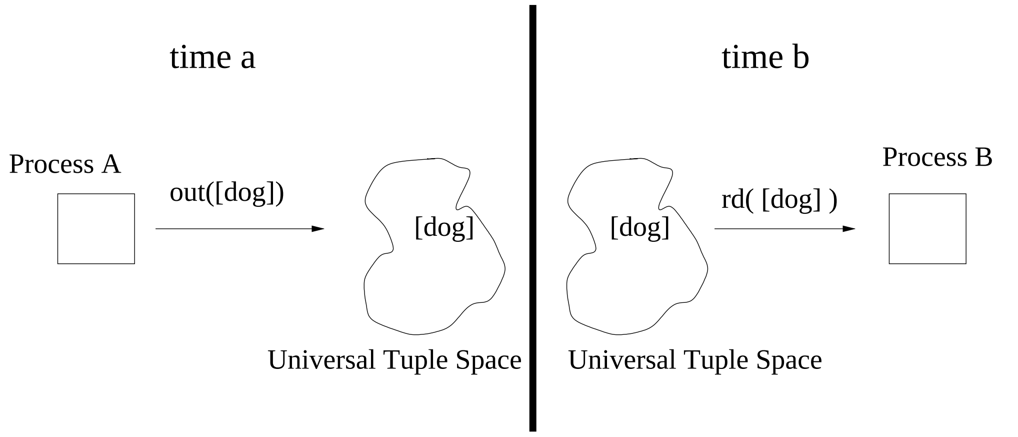
The setter primitives are `out` and `eval` and their syntax is illustrated in rows #5 and #6 of Table 2.2. The `out` primitive inserts the tuple into the Universal Tuple Space. The `eval` primitive is more complex. A tuple is inserted into the Universal Tuple Space and is marked as active. Each field in this tuple is a function that gets evaluated in parallel in the LINDA kernel. Once each function is complete the active tuple becomes a normal tuple.

Table 2.2 Getter and Setter primitive syntax

	Primitive	Syntax	Blocking?
1.	`in`	`in (template)`	Yes
2.	`rd`	`rd (template)`	Yes
3.	`inp`	`inp (template)`	No
4.	`rdp`	`rdp (template)`	No
5.	`out`	`out (tuple)`	Not Applicable
6.	`eval`	`eval (tuple)`	Not Applicable

With the LINDA primitives defined, Figure 2.1 can be drawn using the standard LINDA primitives of `rd` and `out` as illustrated in Figure 2.2. The definitions and concepts just described form what is known as the original LINDA.

Figure 2.2 Revised diagram of Figure 2.1 using LINDA Primitives `rd` and `out`



2.2 Multiple Tuple Spaces Linda Model

While the original LINDA is a powerful model, Gelernter realized it had a weakness: multiple processes share a common single tuple space known as the Universal Tuple Space. This sharing of a single tuple space creates side-effects with multiple processes. For instance, process A `out`'s a tuple intended for process B. Process C is blocked on the `in` primitive using a template that, unknowingly, associately matches process A's tuple. Process C can and does retrieve the tuple. These side-effects can be circumvented by the use of identifiers (IDs). Thus, each tuple sent to the Universal Tuple Space had an extra field in the tuple used as an identifier. The identifier is used to decrease the likelihood of side-effects. However, this was not a solution but a masking of the original problem. Even though processes may not know the explicit identifier of a particular tuple, the formal of that identifier could be used to side effect other processes. What was needed was a way for two or more processes to communicate via a different tuple space other than the Universal Tuple Space.

Gelernter later revisited the LINDA model and added another primitive to allow the creation of new tuple spaces. The revised model became known as MTS-LINDA (Multiple Tuple Spaces Linda). This primitive, called `tsc` (), returns to the calling process a handle (reference) of the new tuple space created inside the LINDA system. The advantage of creating a tuple space is that it allows for better encapsulated communication between processes. If only two processes know the handle of a tuple space, then a third process cannot guess that handle and side effect the other two processes.

The question remains, how would process B know about the tuple space that process A created? Process A would have to explicitly put the handle into a tuple and, using the LINDA `out` primitive, send it to the Universal Tuple Space. This implies that the LINDA primitives need to be slightly changed. Table 2.3 illustrates that all MTS-LINDA primitives are prefixed with the destination tuple space. In the table, ts_1 is referenced; however, ts_1 can simply be replaced with any tuple space reference including one to the Universal Tuple Space.

Table 2.3 MTS-LINDA

original LINDA	MTS-LINDA
`in (template)`	ts_1`.in (template)`
`rd (template)`	ts_1`.rd (template)`
`inp (template)`	ts_1`.inp (template)`
`rdp (template)`	ts_1`.rdp (template)`
`out (tuple)`	ts_1`.out (tuple)`
`eval (tuple)`	ts_1`.eval (tuple)`

Therefore, process A that created the new tuple space could explicitly send this tuple space value via a tuple to the Universal Tuple Space. Process B could be blocked waiting for a tuple that associatively matches a template containing tuple space formal. If process B is the only process blocked waiting for a tuple containing tuple space formal then no side-effects occur. Once process B removes that tuple containing the tuple space value, process A and B can communicate privately without having side-effects by another process.

Although processes can communicate via new tuple spaces, there can still be an unwanted side effect. Consider three processes A, B, and C. Assume process C and B are blocking in the Universal Tuple Space using a template that will match a tuple space formal and process C is using a `rd` primitive and process B is using the `in` primitive. If process A creates a new tuple space and sends a tuple containing a tuple space value to Universal Tuple Space, then there is a chance process C can receive that tuple without process A and B knowing of the retrieval. Pinakis [Pin91] proposed a solution alleviating this side effect that provides directed communication in LINDA using capabilities. Further discussion of directed communication is beyond the scope of this thesis.

Figure 2.3 illustrates how two processes (A and B) can communicate via a newly created tuple space. In 2.3(a), process A creates a new tuple space. In 2.3(b), a tuple is created and the handle of the created tuple space is `out` to the Universal Tuple Space. In 2.3(c), process B uses the `in` primitive consisting of a template containing the formal of a tuple space handle. In 2.3(d), process B uses the `out` primitive to put a tuple into tuple-space ts_1. In 2.3(e), the arrows to and from both processes illustrate the means of communication using the LINDA primitives and tuple-space ts_1. In 2.3(e), in the ts1.* statement, the * represents the LINDA primitives from Table 2.2. At this point, a third process, process C, can start and not side effect process A and B.

For the rest of this thesis, original LINDA and MTS-LINDA will be known simply as LINDA.

2.3 York Linda II Model

The YORK LINDA II [RDW96] is another coordination model that extends LINDA by solving the multiple read problem. The multiple read problem occurs when two or more processes are simultaneously accessing the same tuple space using the same template and `rd` primitive. Since the `rd` primitive does not remove the tuple, it is possible for every `rd` primitive call to return the same tuple; hence, this problem also occurs when a single process is executing this primitive. The issue is: How does a single process or multiple processes access the same tuple space using the same template efficiently and without obstructing other processes? The question is related to which LINDA primitive to use. A possible solution is each process could use the `rd` primitive. However, LINDA cannot guarantee that each tuple retrieved will be different from any previous retrieved tuple because of the nondeterminism nature of Linda. Furthermore, the *same* tuple could be continuously read. Therefore, this is not a plausible solution.

Figure 2.3 Description of two processes setting up a new tuple space for communication

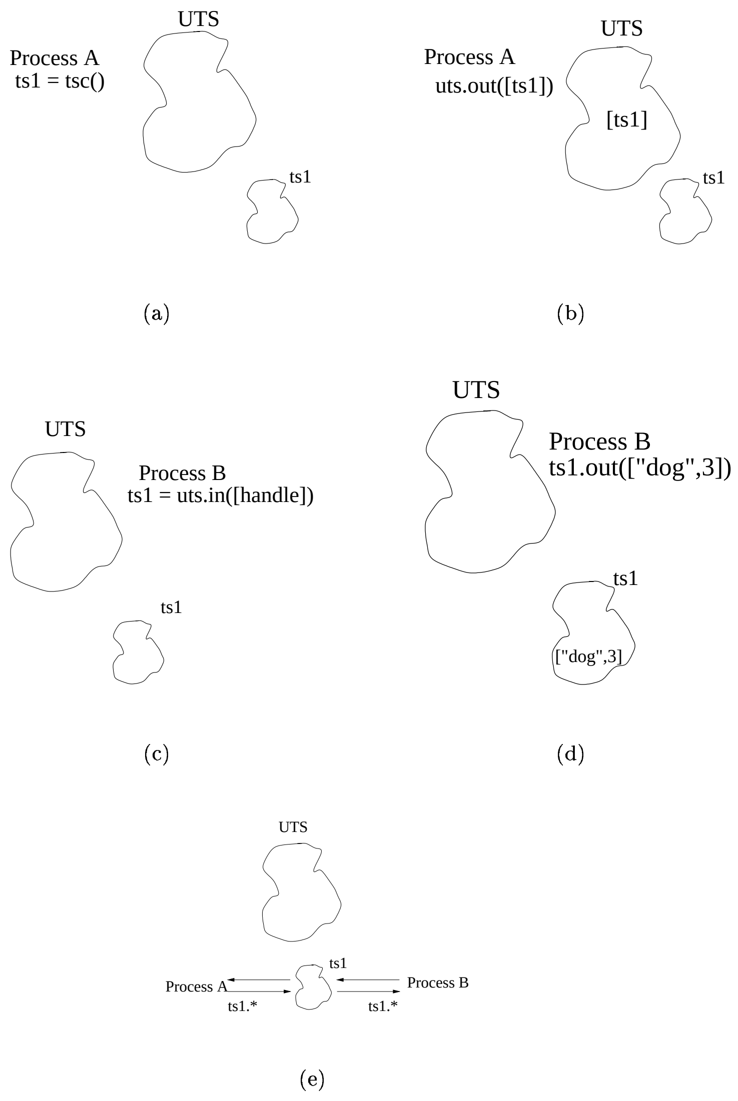
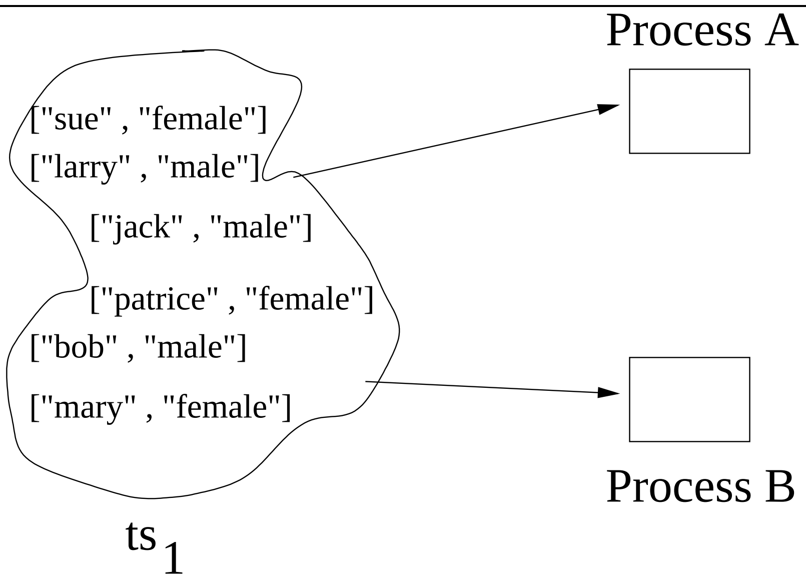


Figure 2.4 Linda multiple read problem



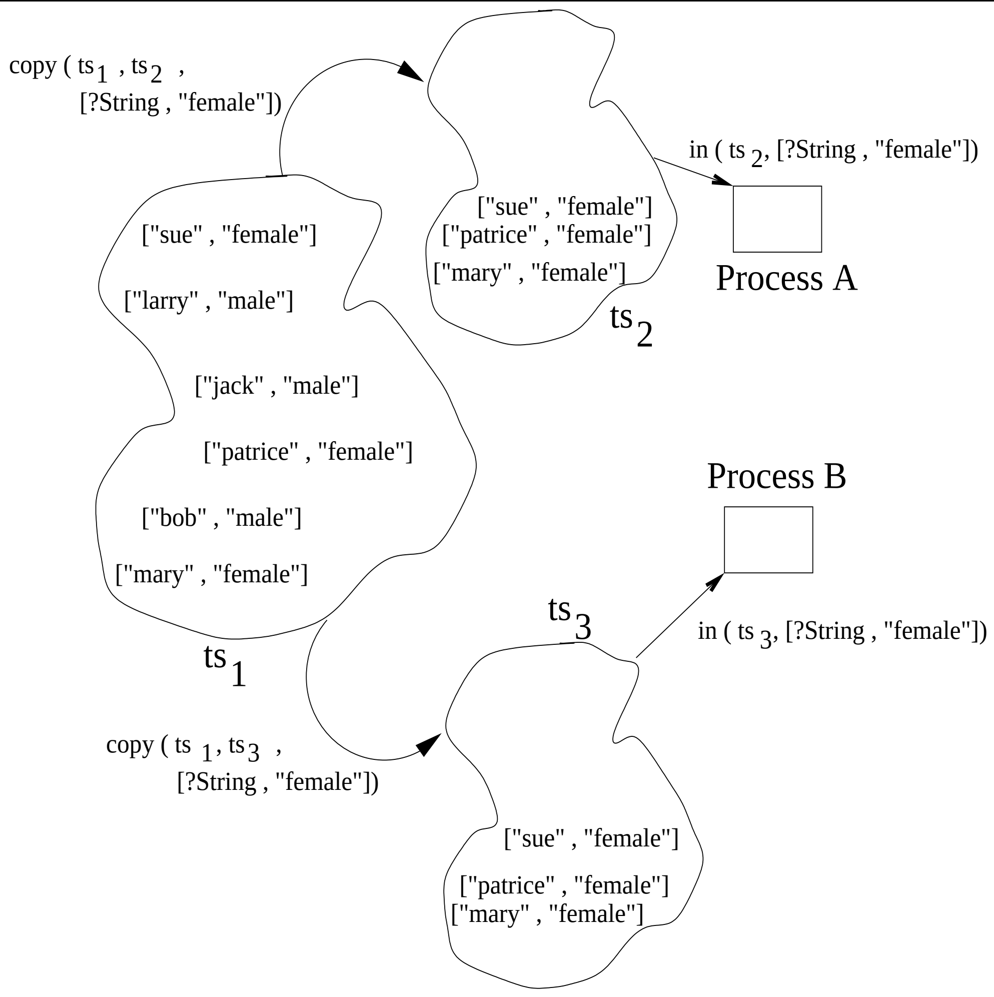
Another possible solution is for each process to use the `in` primitive. However, this solution is also problematic. The first obstacle is the `in` primitive removes the tuple from the tuple space. Thus, when process A is removing tuples, process B cannot access them. Likewise, if process B is using the `in` primitive, process A cannot access the ones process B has retrieved. The second obstacle is if process A would `out` the tuple back into the tuple space, process A cannot be sure that the next `in` would not retrieve the same tuple because of nondeterminism.

Rowstron, et al. [RDW96] proposed a solution to the multiple read problem. They introduced a new primitive called `copy` that *copies* all the tuples matching `template` in the source tuple space to the destination tuple space. The primitive returns the number of tuples that were copied. The syntax for this primitive is in Table 2.4. This primitive solves the multiple read problem because it copies all tuples in bulk using one primitive call. Thus for process A and B above, each process can create temporary tuple-spaces called ts_2 and ts_3, respectively. Both processes can issue the following `copy` primitive below without affecting one another. Figure 2.5 illustrates the new tuple spaces and their contents after the `copy` primitive has finished. Processes A and B can now work independently and efficiently without obstructing the other process.

Process A: `copy` (ts_1 , ts_2 , [?String , “female”]);

Process B: `copy` (ts_1 , ts_3 , [?String , “female”]);

Figure 2.5 `copy` primitive in use



As an option to the `inp` primitive, Butcher [BWA94] proposed another new primitive to *move* (destructive primitive) all the tuples matching `template` from a source tuple space to a destination tuple space. The primitive, called `collect`, returns the number of tuples moved. The primitive's syntax is in Figure 2.4

Table 2.4 `Copy` and `collect` syntax

Primitive	Syntax
`collect`	count := `collect` (ts_1 , ts_2 , `template`);
`copy`	count := `copy` (ts_1 , ts_2 , `template`);

2.4 Scopes

Merrick and Wood [MW00] proposed a new concept called Scopes. The term SCOPE is analogous to a set of tuple spaces with one exception, that a SCOPE can contain other SCOPES: unlike LINDA, a tuple space cannot contain other tuple spaces. Thus a set relationship is achievable using SCOPES. The contents of a SCOPE is one or more sets of tuple spaces: A tuple space in SCOPES is known as a *name*. A set can contain a single *name* or a list of *names*. For example, SCOPE A containing [{a} , {b}] illustrates two sets, each containing one *name*: a and b. Another example of a SCOPE is [{a,b} , {c}]. This, too, contains two sets of names. The first is {a,b} and the second set is {c}. For clarity purposes, the above SCOPES are written without the curly braces as [a,b] and [ab,c]. For this section, a SCOPE is represented by a capital letter and a name is represented by a lower case letter.

SCOPES also introduce four additional operations ($\oplus$, $\ominus$, $\oslash$, and $\odot$) and Table 2.5 list examples of these operators. The $\oplus$ operator does a union of two SCOPES. For example, row #1 in Table 2.5 creates a new SCOPE containing two sets “a” and “b” represented as [a,b]. The $\ominus$ operator, shown in row #4, creates a new SCOPE containing sets that are in the left side of the operator that are not in the right side of the operator. This operator is also known as a set difference. In row #7, the $\odot$ operator creates a new SCOPE containing sets based on applying a cross product algorithm and then the union operator. A cross product algorithm takes each of the sets on the right side of the operator and concatenates it to each set on the left of the operator. The $\oslash$ operator in row #9 does the cross product and then the set difference algorithms. Note, too, in Table 2.5 newly created SCOPES, on the right side of the equivalent sign, contain no duplicate sets of names. In rows #2 and #8 the equivalent statement on the right side contains only one b name and not two b names.

1. A $\oplus$ B $\equiv$ A U B

2. A $\ominus$ B $\equiv$ A / B

3. A $\odot$ B $\equiv$ { x U y | (x,y) $\leftarrow$ A x B }

4. A $\oslash$ B $\equiv$ { x / y | (x,y) $\leftarrow$ A x B }

Table 2.5 SCOPES examples of its operators: $\oplus$, $\ominus$, $\oslash$, and $\odot$. Adapted from [MW00]

1:	[a] $\oplus$ [b]	$\equiv$ [a,b]
2:	[a,b] $\oplus$ [b,c]	$\equiv$ [a,b,c]
3:	[ab] $\oplus$ [bc]	$\equiv$ [ab,bc]
4:	[a,b] $\ominus$ [b,c]	$\equiv$ [a]
5:	[ab] $\ominus$ [bc]	$\equiv$ [ab]
6:	[a] $\odot$ [b]	$\equiv$ [ab]
7:	[a,b] $\odot$ [b,c]	$\equiv$ [ab,ac,b,bc]
8:	[ab] $\odot$ [bc]	$\equiv$ [abc]
9:	[ab,ac] $\oslash$ [a]	$\equiv$ [b,c]
10:	[ab,ac] $\oslash$ [b,c]	$\equiv$ [a,ab,ac]

To summarize, the changes to the LINDA model consists of the following (Adapted from [MW00]):

1. A primitive NEWSCOPE for creating new scopes.

2. Four operations: $\oplus$, $\ominus$, $\oslash$, and $\odot$ for combining existing SCOPES.

3. The usual LINDA primitives, `out`, `in`, and `rd`, are modified to incorporate scoping. The `in` and `rd` primitives return just a single tuple and the SCOPE it was contained in. The `out` primitive puts the tuple into the SCOPE.

Finally, SCOPES uses an additional matching rule. This matching rule determines if two or more SCOPES have at least one element in common, then those SCOPES match. The matching rule and examples are exemplified in the following:

$$\mathrm{A} \sim \mathrm{B} \leftrightarrow \mathrm{A} \cap \mathrm{B} \neq \emptyset$$

Table 2.6 SCOPES matching rule. Adapted from [MW00]

1.	[a,b]	$\sim$	[b,c]	(both contain b)
2.	[ab]	!$\sim$	[bc]	(since ab *neq* bc)
3.	[a,b,ab]	$\sim$	[a,b,c]	(both contain a and b)
4.	[a,b,ab]	$\sim$	[ab,c]	(both contain ab)
5.	[a,b,ab]	!$\sim$	[abc]	(no element in common)

Row #1 containing SCOPE [a,b] and SCOPE [b,c] match because their intersection is not empty (both have b in them). Likewise, row #3 match because both SCOPES contain a and b in their intersection. The SCOPES in row #2 do not match because SCOPE ab does not match SCOPE bc: the intersection is empty. This matching rule determines the visibility of a tuple: Assume process A uses the `out` primitive to insert a tuple into SCOPE A. If a second process, B, uses an `in` primitive to retrieve a tuple from SCOPE B, SCOPE B could actually remove and return the tuple that process A inserted if A $\sim$ B.

2.5 Bonita Model

With respect to efficiency, one limitation of LINDA is its blocking primitives. As they block, client processes cannot continue executing. As a result, Rowstron and Wood [RW97] proposed a new model, known as BONITA, to be more efficient than LINDA. The authors claim that the time for a LINDA primitive to complete can be decomposed into seven components. The sum of these times is shown in Figure 2.6.

T_{Pack}: The time it takes to prepare the primitive in the client process.

$T_{SendRequest}$: The time it takes to send the primitive over the communication medium to the LINDA kernel.

T_{Queue}: The time it takes for a message to be serviced.

$T_{Process}$: The time it takes to service the primitive by finding a tuple and packaging the result.

T_{Block}: The time it takes because no tuple is found.

$T_{SendReply}$: The time it takes to send the resulting tuple back over the communication medium.

T_{Unpack}: The time it takes to interpret the result.

Figure 2.6 Total time of a blocking LINDA getter primitive

$$T_{Pack} + T_{SendRequest} + T_{Queue} + T_{Process} + T_{Block} + T_{SendReply} + T_{Unpack}$$

They claim the LINDA primitives could be made more efficient by enabling the computational language to work in parallel with the LINDA primitives. In doing so, the formula in Figure 2.6 could be reduced for the getter LINDA primitives and setter LINDA primitives shown in Figure 2.7 and 2.8, respectively.

Figure 2.7 BONITA's time for getter primitives

$$T_{Pack} + T_{Unpack}$$

Figure 2.8 BONITA's time for setter primitives

$$T_{Pack}$$

To achieve an increase in efficiency, four new primitives were created to replace the LINDA primitives: dispatch and dispatch_bulk primitives (which are asynchronous), arrive, and obtain. They are described below:

rqid = dispatch (ts1 , tuple | [template , destructive | nondestructive]) :

an overloaded primitive that either returns a tuple to a process or inserts a tuple into tuple-space ts1. If template is specified then this will remove a tuple and additional information is required: destructive or nondestructive. Destructive would actually remove the tuple from ts1 while nondestructive simply makes a copy of the tuple.

rqid = dispatchBulk (ts1 , ts2 , template, destructive | nondestructive) :

requests the tuples matching template in tuple-space ts1 be moved into tuple-space ts2.

arrived(rqid) :

detects if the tuple arrives that is associated with rqid.

tuple = obtain(rqid) :

a blocking primitive that waits for the tuple associated with rqid to arrive.

For example, consider the two code fragments, 1 and 2 respectively, in Program 2.1. Assume that retrieving a matching tuple from tuple-space ts_1 and the method call doExpensiveCalculation() take x and y seconds to process, respectively. Since the dispatch method on line 1a is asynchronous, it immediately returns an id. At this time, the dispatch method can, and is, working in parallel with the execution of line 1b in the client process. Since both are working in parallel, by the time line 1d executes, the max(x,y) seconds would have elapsed. Consider now the code fragment on lines 2a-2c. On line 2a, the process executes a blocking `in` primitive which of course takes x seconds to process. Once complete, the client process executes the method doExpensiveCalculation(), which takes y seconds. By the time line 2c is executed, x + y seconds have elapsed.

Program 2.1 BONITA's primitive and LINDA's primitive

```
(1a)  id1 := dispatch( ts1 , template , destructive)
(1b)  doExpensiveCalculation()
(1c)  tuple := obtain(id1)
(1d)  useIt(tuple)                   // max(x,y) seconds have elapsed

(2a)  tuple := in( ts1 , template )
(2b)  doExpensiveCalculation()
(2c)  useIt(tuple)                   // x + y seconds have elapsed
```

2.6 Event-driven Models

Event-driven models are based on a client process registering and unregistering for occurrences of a tuple that matches a specific template being inserted into a specific tuple space. As these tuples enter the tuple space, the registered client process receives notifications of the tuple and the tuple space to which it will enter. This type of notification is not used as a blocking (synchronizing) mechanism but for a client to register and continue processing. Thus, as a client process executes, a notification can arrive by calling a predefined method. As the event is sent to the client process, the execution flow in the client process is interrupted and goes to the notified method.

Several LINDA derived models exist that support event-driven facilities including LIME (Linda in Mobile Environment) [PMR99], JavaSpaces [Sun97] and TSpaces [IBM98]. Among LIME's primitives is one that is called `reactsTo`. The syntax of this primitive is `T.reactsTo (cf,p)` where T is the tuple space that is registering this reaction, cf is a code fragment containing non-reactive statements that are to be executed when a tuple matches the pattern p. Hence, when a tuple is being inserted into tuple-space T that matches pattern p, the code fragment is executed. One example of a code fragment is it can `out` the tuple matching the template to another tuple space.

JavaSpaces, written using the Java programming language, is Sun Microsystem's implementation of LINDA. In addition to the implementation of the LINDA primitives, this model has an event-driven application programming interface (API). The JavaSpace interface has a method allowing events to be registered. This method, called on a javaspace (analogous to a tuple space), is called *notify()*. One of the parameters is the template that will be matched against the tuples entering into the javaspace. Another parameter is an instance of a class that a developer writes that implements the RemoteEventListener interface for a client process. The RemoteEventListener interface has one method called *notify()* that is passed a RemoteEvent object:

void notify(RemoteEvent theEvent)

Before a tuple that matches a registered template is inserted into a javaspace that has an event listener, the client process is immediately notified of that tuple. The event-driven capability in JavaSpaces sends the event notification to the registered client process. Program 2.2 illustrates an example of an event-driven implementation. Note on line 15 of the example a method called *take()* is executed: For JavaSpaces, this method is analogous to the `in` primitive in Linda. The data type Entry on line 15

Program 2.2 JavaSpaces event-driven model

```
(1)   public class Example implements RemoteEventListener
(2)   {
(3)     JavaSpace aJavaSpace;
(4)
(5)     public Example()
(6)     {
(7)        ...
(8)        aJavaSpace.notify( template, .. , this , .. , null);
(9)        ...
(10)     }
(11)
(12)
(13)    public void notify(RemoteEvent theEvent)
(14)    {
(15)      Entry tuple = aJavaSpace.take( template , .. , ..);
(16)      ...
(17)    }
(18)
(19)  }
```

TSpaces, created by IBM and written in the Java programming language, is also based on Linda. It, too, has developed an event-driven API that is associated with a single tuple space object. Registering for an event involves invoking the *eventRegister()* method for that tuple space. Two parameters of particular interest are the template and the callback object. The template is used for matching every tuple coming into a particular tuple space. The callback object is an instance of a class that implements the Callback interface. This interface contains one method: *call()*. One of the parameters of this method is the tuple matching the template. Another parameter is the tuple space name where the tuple came from. Like JavaSpaces, the event-driven capability in TSpaces sends events externally to the registering client process. Program 2.3 is an example of this event model.

Program 2.3 TSpaces event-driven model

```
public class Example implements Callback
{

  public Example()
  {
    ...
    tupleSpace.eventRegister( .. , template , this , .. );
    ...
  }

   public boolean call( .. , String tupleSpaceName , .. ,
                        SuperTuple tuple , .. )
   {
     ...
   }

 }
```

2.7 Conclusion

This chapter discussed some semantics and descriptions of the Linda model. The preliminary foundation of Linda and its primitives were conceptualized. Finally, Scopes, Bonita, York Linda II and Event-driven models were described. Note that these models focused the semantics of their primitives returning a single tuple. Additionally, semantics for sending a tuple to a tuple space involved one tuple space at a time.

Chapter 3

Motivation

This chapter focuses on the differences of the semantics described in Chapter 2. While all the described models are descendants, none completely addresses the issue of expressing multiple-tuple-space primitives. For clarification, accessing multiple tuple spaces at once means several *different* tuple spaces or the *same* tuple space several times being accessed using the same primitive. Furthermore, defining semantics that express accessing multiple tuple spaces at once could produce efficient and scalable primitives. Efficiency is defined as the amount of time it takes to execute a particular primitive. For a *consecutive* list of the same getter primitives using the same template accessing several tuple spaces, a scalable primitive implies that it is not necessary to execute the same primitive more than once: it is sufficient to somehow combine those tuple spaces and execute a single primitive call.

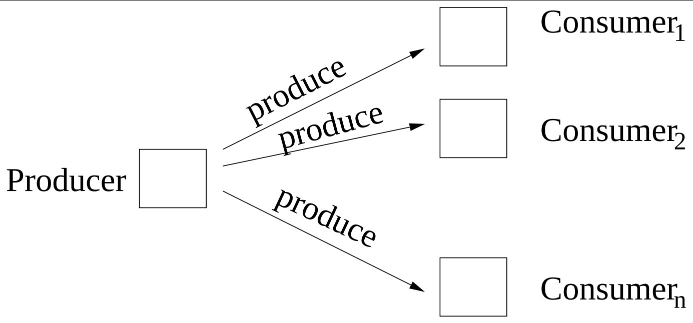
3.1 Linda Model

This section covers the getter and setter primitives provided by LINDA. Some differences and expressiveness of this model are exposed that pertain to the proposal of this thesis. Another motivation for this thesis is the difference of expressiveness with SCOPES.

3.1.1 Linda Model Setter Primitives

A common architecture in software is the producer/consumer: A single process "produces" topics that subscriber-processes have requested. Figure 3.1 illustrates process P (a producer) sending n topics to consumer processes C_1 ... C_n. Consumers generally ***subscribe*** to and ***unsubscribe*** from a producer process's topics. Naive implementations of a common producer process can end up sending n copies of the same topic to n different consumers. If these topics are all the same, could one topic simply be broadcasted to n consumers? Yes, LINDA feasibly expresses sending a topic to a *single* tuple space and multiple consumers retrieving a *copy* of that topic (via the `rd` primitive). Taken one step further, a problem with this approach is how would the producer easily know that the last consumer has retrieved the tuple and the producer can remove the tuple. Menezes states that tuples are not garbaged collected: tuple spaces are [Men98]. What is also important is a producer in LINDA sending the same topic to n *different* tuple spaces: It would have to make n `out` primitive calls. If n is large, this could create a scalability issue.

Figure 3.1 Producer/Consumer Architecture



LINDA supports the producer/consumer architecture through the use of the setter and getter primitives discussed earlier. Figure 3.2 illustrates this architecture where the producer sends two identical items to each of the two consumers' tuple space. "Subscription" means that a consumer does a "get" using the `in` primitive and a template[1] matching the produced item. Code is shown in Figures 3.3 and 3.4 for the producer and two consumers,

[1]This example purposely overlooks how the producer knows there are two consumers and for what they have subscribed.

respectively.

Figure 3.2 LINDA producer/consumer architecture

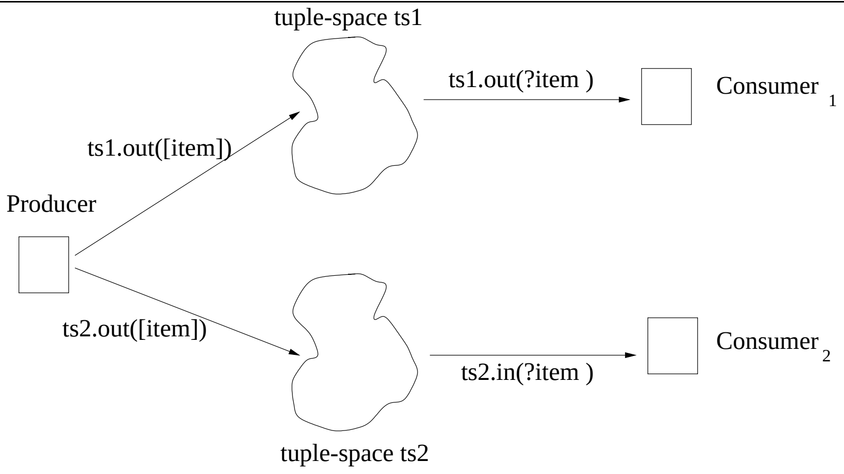


Figure 3.3 LINDA producer code from Figure 3.2

```
while (true) do
    $item = computeIt();$
    $ts_1$.out($newTuple(item)$);
    $ts_2$.out($newTuple(item)$);
end
```

Figure 3.4 LINDA consumer code for $Consumer_1$ and $Consumer_2$, respectively, from Figure 3.2

```
while (true) do
    $item = ts_1$.in([?$item$]);
    $doSomething(item);$
end
```

```
while (true) do
    $item = ts_2$.in([?$item$]);
    $doSomething(item);$
end
```

The code in Figure 3.3 illustrating LINDA's ability to send the "item" serially to different tuple spaces has limitation on scalability. If the number of consumers is large, then LINDA's `out` semantics state that for each consumer, there must be an `out` primitive executed by the producer. How well can this primitive scale? What is needed is semantics that allow more than one tuple space, possibly the same tuple space, to be accessed with one primitive call using the same tuple or template.

For the code in Figure 3.3, Property 3.1.1 reflects the scalability of consecutive serial calls using the LINDA's `out` primitive with the same tuple.

Property 3.1.1 (Scalability of LINDA's `out` primitive.) *Let S be a set of tuple-spaces ts_1, ts_2, ... , ts_n. Then $\forall$ ts_i, where i = 1..n, $\exists$ n* `out` *primitive calls being executed.*

Consider now LINDA's `eval` primitive and rewriting the code in Figure 3.3 to be a producer using an `eval` primitive. Figure 3.5 illustrates this code. LINDA is trying to achieve sending the same results of $f_1(x_1)$,$f_2(x_2)$, ... ,$f_n(x_n)$ to both ts_1 and ts_2. Consider that each `eval`'s $f_1(x_1)$ in Figure 3.5 returns an object consisting of the date and time. Furthermore, consider that $f_n(x_n)$ is an expensive function taking no less than x seconds to process. Then, the values for $f_1(x_1)$ for each `eval` will be different by no less than x seconds. Again, two different tuples will be placed into ts_1 and ts_2. Therefore, LINDA cannot provide a good solution.

In addition to the above argument, if the client is multi-threaded, further data anomalies could occur between the execution of the first `eval` and the second `eval` in Figure 3.5. Consider that thread A is executing the while loop in Figure 3.5. Thread A could have just finished the execution of the first `eval` primitive when thread B starts and changes a data value affecting the result of $f_2(x_2)$. When thread A resumes, the result of $f_2(x_2)$ in the second `eval` primitive will be different from the first `eval` primitive. One solution could be that $f_2(x_2)$ needs to be considered a critical method and thus allowing only one thread to be executed. However once the first `eval` primitive is complete ($f_2(x_2)$ has completely been evaluated), thread B can enter into that method and thus causing the same data anomaly for the second `eval` primitive. Another solution could be that the two `eval` primitive calls are the critical section and thus thread B needs to be aware of this. However as more solutions are proposed, it must be asked: What is the real reason that the second `eval` has this data anomaly? $f_2(x_2)$ already needs to be a critical method because of its execution in the first `eval` primitive. The real problem is that the `eval` primitive does not allow the tuple resulting from the evaluation of each $f_i(x_i)$ to be sent to two tuple spaces using one primitive call. The solution and this type of expressive power will be illustrated in the next chapter.

As with the `out` primitive, semantics is an issue with the `eval`. It can be argued that if $f_1(x_1)$ needs to be executed twice and $f_n(x_n)$ is an expensive function to execute once let alone twice, then why use the `eval` and not just an `out` primitive. Recall one of the benefits of the `eval` primitive is that each $f_i(x_i)$ is executed in parallel in the LINDA kernel. Unarguably, the computational language can implement its own generic parallel processing algorithm so that each $f_i(x_i)$ can execute in parallel (outside the LINDA kernel). However, why should the computational language implement another algorithm to duplicate parallel and synchronization that LINDA inherently provides just to avoid using the `eval` primitive? In addition, this solution does not solve the semantics argument described above with the `out` primitive using Property 3.1.1. It will be shown later how LOGOP LINDA feasibly solves this problem.

Figure 3.5 LINDA producer code using the `eval` primitive

```
while ($true$) do
    $ts_1$.eval($[f_1(x_1), f_2(x_2), ..., f_n(x_n)]$);
    $ts_2$.eval($[f_1(x_1), f_2(x_2), ..., f_n(x_n)]$);
end
```

3.1.2 Linda Model Getter Primitives

Figure 3.6 illustrates a process called P_1 blocking on both ts_1 and ts_2 *at the same time* using the `in` primitive. Because LINDA can also do synchronization, P_1 cannot proceed until receiving a tuple from both ts_1 and ts_2 matching `template`. How can LINDA express this situation?

Figure 3.6 P_2 and P_3 both executing an `out` that will match template

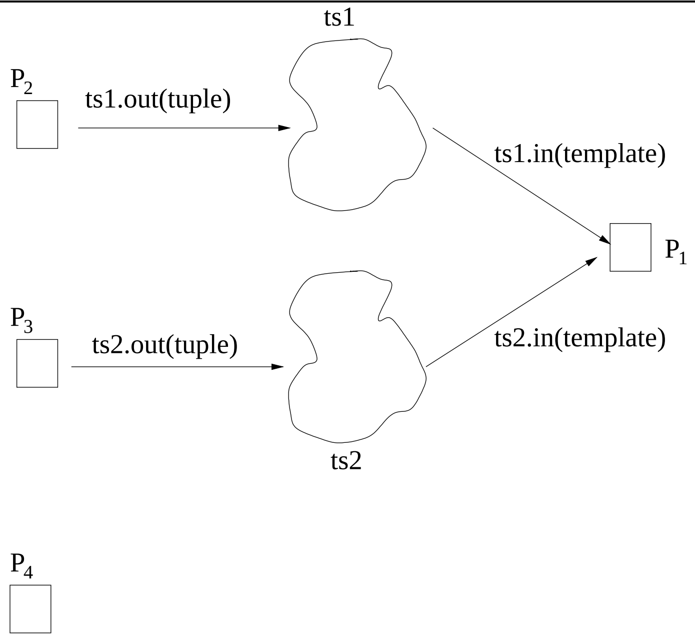


Figure 3.7 Linda consumer code for P_1 in Figure 3.6

$$tuple1 = ts_1.\texttt{in}(\texttt{template});$$
$$tuple2 = ts_2.\texttt{in}(\texttt{template});$$
$$proceed();$$

Consider the code in Figure 3.7 that synchronizes on both tuple-spaces ts_1 and ts_2 and once both primitives unblock, calls a method proceed() for further processing. The code first blocks on ts_1 and then on ts_2. If the order of blocking in the tuple spaces is important, then this is a justifiable solution. As the code is currently written in Figure 3.7, the total time for both `in` primitives to retrieve existing tuples would be:

$$time_1 + time_2$$

where $time_1$ is the time for the first `in` primitive and $time_2$ is the time for the second `in` primitive. However, if order of blocking is not important, then the total time for both primitives to retrieve existing tuples would still be the same.

If order is not important, why could not the two primitives be done in parallel? If the primitives could be done in parallel, the total time for retrieval would be

$$\max(time_1 \ , \ time_2)$$

Furthermore, speculating on scalability, if n tuple spaces were involved, the time for a non-parallel solution would be:

$$\textstyle\sum_{i=1}^{n} time_i$$

Whereas a parallel solution time would be:

$$\max \ (\ time_1 \ , \ time_2 \ , \ ... \ , \ time_n)$$

If order is not important, it can be argued there is a coordination pattern that LINDA cannot achieve due to serial access to tuple spaces. Figure 3.8 illustrates a time line of this access. Consider at $time_1$, process P_1 begins blocking on the first `in` primitive in Figure 3.7. Immediately after $time_1$, process P_3 does a

ts_2`.out ( tuple);`

at $time_2$. And at a much later time, $time_3$, another process (P_4 from Figure 3.6) starts up and begins to block on

ts_2`.in ( template);`

Since P_4 is the only process blocking on ts_2 and a tuple exists from process P_3 that presumably matches `template`, it will remove the matching tuple and proceed. At this point a side effect occurs because of serial access to tuple spaces and the blocking characteristic of the `in` primitive.

To avoid the blocking, Figure 3.7 is rewritten using the `inp` primitive and is now shown in Figure 3.9.

Figure 3.8 Side effect using `in`. The ellipses in the time line represent *"at a much later time"*

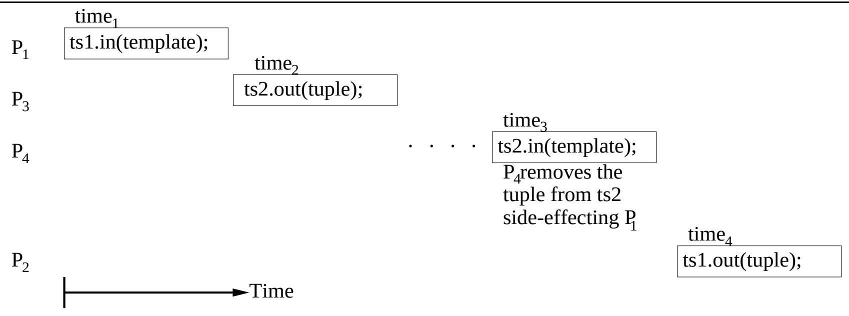


Figure 3.9 LINDA consumer code

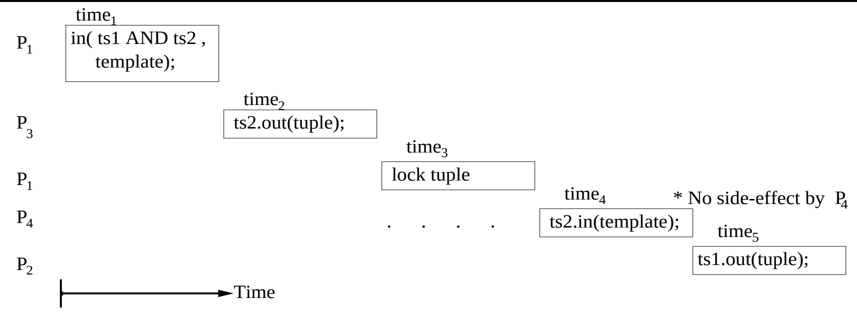
```
while ( Either tuple1 or tuple2 is null ) do
    if ( tuple1 is null ) then
                    tuple1 = ts_1.inp (template);
    fi
    if ( tuple2 is null ) then
                    tuple2 = ts_2.inp (template);
    fi
end
process();
```

Because the `inp` primitive is nonblocking, it immediately returns a null if a tuple is not found matching `template`. The code in Figure 3.9 suffers from another efficiency problem: a polling mechanism [Hoa85]. The while loop coupled with an `inp` primitive is doing a continuous poll for the data. Polling can be quite expensive and be architecturally prohibitive. Another issue of the polling is the number of tuple spaces: As they increase, the efficiency decreases; hence, it suffers from a lack of scalability.

The only feasible solution is for LINDA to execute each `in` primitive in parallel. Executing an `in` primitive in parallel solves both problems of scalability and polling. Figure

3.10 solves the timing problem noted in Figure 3.8.

Figure 3.10 Parallel execution of the LogOp Linda `in` primitive in Figure 3.6 for P_1



Note that under $time_1$ in Figure 3.10, the `in` primitive, accessing tuple-spaces ts1 and ts2, is executed and both are blocking on their respected tuple spaces *at the same time.* Thus at $time_3$, P_1 locks the tuple (inserted by P_3) in tuple-space ts_2. The concept of locking a tuple will be explored further in Chapter 4.

The important concept between Figures 3.8 and 3.10 is the concept of *"at a much later time"*. For both diagrams, the calculation of time starts after P_3 executes the `out` (ts_2, `tuple`). This *"later time"* is the same quantity in both 3.8 and 3.10. Conceptually, the second `in` primitive from Figure 3.7 suffers from not being able to execute because the first `in` primitive is blocking. In addition, note the ellipses from Figures 3.8 and 3.10. They both start immediately after P_3 executes

ts_2`.out (tuple)`

Again the advantage allowing P_1 to lock a matching tuple in tuple-space ts_2 (and eventually remove it) is depicted in Figure 3.10.

3.1.3 Scopes Model

The authors of the Scope concept covered in the previous chapter never mention in their semantics efficiency and scalability of their model. However, their proposal is primarily about the semantics of overlapping tuple spaces: in other words the concept of orthogonal tuple spaces in the context of Venn diagrams [MW00]. Additionally, the authors offer another interpretation of overlapping tuple spaces: the orthogonal intersection of tuple spaces is a hidden area. These interpretations are what makes SCOPES a powerful extension to LINDA.

Consider the SCOPES code in Figure 3.11. When the $\oplus$ operator is used, an additional data structure for the new SCOPE needs to be updated on the server. Furthermore, the operators do not remove actual SCOPES from existing SCOPES: Based on the semantics, the operators create new SCOPES not containing certain names or other SCOPES. In a large scale environment, this could prove prohibitive.

Figure 3.11 SCOPES producer code

```
$combinedScope = (a \oplus b);$
**while** $(true)$ **do**
    $item = computeIt();$
    out$(combinedScope, [item]);$
**end**
```

Recall from Chapter 2 in Table 2.5 that the $\odot$ and $\oplus$ eliminate duplicate names when a new SCOPE is created from existing SCOPES. This type of semantics prohibits ever being able to express inserting n copies of a single tuple into a SCOPE using a single primitive call.

While SCOPES offers this expressiveness to block on a single SCOPE, it does not express well what gets returned based on the SCOPES' contents. When using an `in` primitive, SCOPES will remove and return a single tuple that matches `template` and the SCOPE where it is found [MW00]. However, this is not scalable. Consider that a SCOPE contains two sets, $name_1$ and $name_2$, and because of associative matching in LINDA, $name_1$ and $name_2$ could both contain different tuples that match `template`. However, SCOPES would only remove from either $name_1$ or $name_2$ but not from both. It will be shown later that LOGOP LINDA returns the proper number of tuples and tuple spaces listed in the primitive.

Since SCOPES consists of multiple sets, this implies that a tuple can go into x number of *different* sets' name using one `out` primitive call. This is scalable and also efficient; however, this one tuple is shared by all sets. However, how can SCOPES put the same tuple into the same SCOPE x times using one `out` primitive call? Recalling the examples of SCOPE operations from Table 2.5, the answer is SCOPES cannot express this effectively due to the results in lines 2 and 8 shown below from that table:

$$2\text{: [a,b]} \oplus \text{[b,c]} = \text{[a,b,c]}$$

$$8\text{: [ab]} \odot \text{[bc]} = \text{[abc]}$$

In the first example (row labeled #2), note the equivalent SCOPE only contains one "b"; likewise the same is shown in row labeled #8. So retrieving x tuples or inserting x tuples requires x number of primitive calls. It will be shown later that LOGOP LINDA solves this limitation of scalability and efficiency.

3.2 Bonita Model

Recall that the BONITA Model proposed better efficiency by introducing asynchronous primitives. Consider Figure 3.12 (which is a rewrite of code from Figure 3.3).

Figure 3.12 BONITA producer code

```
while ($true$) do
    $item = computeIt()$;
    dispatch($ts_1$, [$item$]);
    dispatch($ts_2$, [$item$]);
end
```

Rowstron and Wood [RW97] state that the time factor for the asynchronous dispatch primitive is:

$$T_{Pack}$$

Figure 3.12 has two dispatch calls. This implies that the total time inside the while loop for each iteration is:

$$2 * T_{Pack}$$

Although the `dispatch` primitive is asynchronous, it does not express well when the same tuple needs to be sent to different tuple spaces or several copies of the same tuple need to be placed into a single tuple space. Property 3.2.1 states the time it takes for n `dispatch` calls to be made.

Property 3.2.1 (Number and time of BONITA's `dispatch` primitive calls in setter mode.) *Let S be a set of tuple spaces ts_1, ts_2, ... , ts_n. Then $\forall$ ts_i, where i = 1..n, $\exists$ a* `dispatch` *primitive being called. Therefore, the time taken for all n* `dispatch` *calls is* $\sum_{i=1}^{n} T_{Pack}$.

The BONITA model also claimed in Section 2.5 that its asynchronous `dispatch` method can retrieve a tuple from a tuple space in

$$T_{Pack} + T_{Unpack}$$

time. If the process expresses that n tuples need to be retrieved using the same template from a single tuple space or n different tuple spaces then the total time taken is going to be

$$\sum_{i=1}^{n}(T_{Pack} + T_{Unpack})$$

Furthermore, it will take n primitive calls. Such time efficiency and number of primitive calls can be better. In fact, by defining new semantics that will access multiple tuple spaces with a single (either getter or setter) primitive call could prove more expressive and may even, during runtime, provide better time efficiency.

While BONITA expresses powerful efficiency through the use of asynchronous primitives, this thesis takes a different approach and will be explained during the discussion of semantics covered in Chapter 4.

3.3 York Linda II Model

Recall the YORK LINDA II [RW96] has two additional primitives called `copy` and `collect`. While these are considered bulk primitives, there are scalability concerns when tuples from more than one source tuple spaces need to be copied or moved into more than one destination tuple spaces. For example, if there are five tuple spaces and at time t each need to copy their own tuples matching template to the other four tuple spaces, then there will be 5 * 4 primitive calls. Program 3.1 lists the pattern of primitive calls.

Program 3.1 York Linda II `copy` primitive

```
(1)  count = copy( ts1, ts2, template );
(2)  count = copy( ts1, ts3, template );
(3)  count = copy( ts1, ts4, template );
(4)  count = copy( ts1, ts5, template );
(5)  count = copy( ts2, ts1, template );
(6)  count = copy( ts2, ts3, template );
...
(17)  count = copy( ts5, ts1, template );
(18)  count = copy( ts5, ts2, template );
(19)  count = copy( ts5, ts3, template );
(20)  count = copy( ts5, ts4, template );
```

If n tuple spaces do this, then there would be n * (n-1) primitive calls: This would not scale well if n is a large number. Furthermore, on row #5, when ts_2 copies its contents back into ts_1, ts_1 will receive copies of the same tuples that it sent to ts_2 in row #1. This contradicts the initial coordination pattern in which the tuples in the five tuple spaces were to copy their contents at time t over to the other four tuple spaces. Time t for row #1 and row #5 are different thus creating an incorrect result for the pattern. Semantics that achieve the desired results of accessing multiple source and destination tuple spaces at once would alleviate the coordination pattern just described.

3.4 Event-driven Models

The Event-driven JavaSpaces and TSpaces models discussed in Section 2.6 both allow one notification at a time to be called back to the client process that has registered the event.

Furthermore, if several registrations exist across different tuple spaces, the semantics do not support receiving a group of events in a single notification.

The LIME semantics of `reactsTo` executes a code fragment containing non-reactive statements based on a tuple matching a template entering into a tuple-space T. Assume the non-reactive code calls back on the client process and several tuple spaces each have a `reactsTo` listener for the same template. If a tuple is placed in each tuple space matching the template at the same time, then the client process will be notified in serial of all events. If a client process is blocked on receiving event notifications, then at a single point in time the client process will unblock when only one event has occurred. However, since several events occurred at the same time, the client process will proceed when it receives the first event. It may be incorrect for the client process to proceed with only one event when several occurred at the same time.

Although semantics for an event model discussion in this thesis does not exist, further discussion of this topic will be continued in Chapter 7.

3.5 Conclusion

All models discussed is this chapter have limitations of expressiveness and in most cases execute too many primitive calls. The number of executed primitive calls can be improved. The next chapter addresses how LogOp Linda solves these differences with better expressiveness and decreased number of primitive calls; in addition, without increasing or decreasing the number of Linda primitives, a new concept will be introduced for Linda called parallel primitives: using a single getter primitive to access and retrieve from more than one tuple space simultaneously and for a single setter primitive to access more than one tuple space simultaneously when the same template and tuple, respectively, is specified.

Chapter 4

Logical Operators

Chapter 3 illustrated the differences of semantics with certain LINDA models. These differences are the main motivation behind the proposal of LOGOP LINDA. This chapter defines the semantics of LOGOP LINDA operations. Additionally, some of the coordination patterns that are achievable by LOGOP LINDA will be covered.

It is important to understand that LOGOP LINDA does not combine primitives but combines tuple spaces. For example, the following could be combined into one primitive call

`in` (ts_1,template)

`in` (ts_2,template)

as such

`in` (combine(ts_1, ts_2) , template)

whereas the following can be combined into three separate calls

`in` (ts_1,template)

`in` (ts_2,template)

`rd` (ts_3,template)

`rd` (ts_2,template)

`out` (ts_2, tuple)

`out` (ts_1, tuple)

`out` (ts_4, tuple)

as such

`in` (combine (ts_1, ts_2) , template)

`rd` (combine (ts_3, ts_2) , template)

`out` (combine (ts_2, ts_1, ts_4) , tuple)

How tuple spaces are combined is the basis of this thesis and will be explored in detail throughout this chapter. In addition, when tuple spaces are combined, the primitive does not force order of how tuple spaces are processed: the tuple spaces are processed in parallel.

4.1 Semantics

LogOp Linda is an extension of Linda[1] and does not add any new primitives. The semantics described below will be an extension of the semantics described in Table 2.2. Because LogOp Linda deals with multiple tuple spaces per primitive, Table 2.3 will be modified and merged into the discussion to show how LogOp Linda semantically accesses multiple tuple space. The logical operators covered are `And`, `Or`, and `Not`.

In this section, the remove or copy characteristics of `in`/`rd` and `inp`/`rdp` respectively have no effect on the semantics of each primitive because the logical operator affects what tuple spaces are accessed and not whether a tuple is removed or copied from them. In addition, this chapter will compare and contrast the properties listed in the previous chapters to new properties addressed in this chapter. Because LogOp Linda combines two or more tuple spaces for the `in`, `inp`, `rd`, `rdp` primitives, a list of pair values are returned and will consist of the tuple itself along with the tuple space where the tuple is found. Similarly the `in` and `inp` primitives are still blocking primitives while `rd` and `rdp` are still nonblocking primitives. The value returned from the `collect` and `copy` primitives will be

[1]For clarification, Linda will contain primitives from the original Linda, MTS-Linda and York Linda II.

a list of countPair. A countPair contains the number tuples copied/moved and the source tuple space.

Processes create new tuple spaces by using the `tsc` primitive and such processes can put that new tuple space into an existing tuple space by creating a tuple containing the new tuple space actual and using the `out` primitive to put the tuple into the Universal Tuple Space or other existing tuple spaces that have been created. The process can retrieve tuples containing a tuple space actual by using the getter primitives and specifying a tuple space formal. The term “well known tuple spaces” is in context of the client process executing a LogOp Linda primitive. A client process can obtain tuple space handles through two mechanisms. First, the process creates a new tuple space using the `tsc` method, or second, the process retrieved a tuple containing a tuple space handle. Based on these mechanism, as it executes, the client process knows several existing tuple spaces in the LogOp Linda kernel: These tuple spaces are termed well known. Furthermore, if the client process is executing a non-`tsc` primitive, its well known tuple spaces do not and cannot change: An external client process or the kernel cannot remove a well known tuple space from another client process. Furthermore, even garbage collection cannot reclaim a tuple space inside the kernel because a handle is stored inside the client process [Men98].

4.2 `And` Logical Operator

Table 4.1 contains code that will aid in discussion of the `And` operator. The `in` and `rd` primitives in Table 4.1 return a list of pairs to the calling process from each of the tuple spaces (ts_1, ... ,ts_n) that matches `template`. Since these primitives are blocking, they will not return until one tuple has been obtained from each tuple space listed. The `inp` and `rdp` primitives are non-blocking and they return a list of pairs of minimum length 0 (when no tuples match the template) up to n, where n is the number of tuple spaces that are checked. The `out` primitive places `tuple` into each of the tuple spaces (ts_1, ... ,ts_n) listed. The `eval` is similar to the Linda `eval` primitive. However, once the tuple is created, it is placed inside each of the tuple spaces listed. For each tuple space that is in the source list of `collect`, its tuples that match `template` are moved to each listed destination tuple space.

The `collect` returns a list of countPair. Each countPair consists of the number of tuples moved from the source tuple space and the tuple space itself. The `copy` primitive works similarly to the `collect` but just copies the tuples matching `template`. For example, when `out` (∧ (ts_1,ts_2),[“henry”,“male”]) executes, the tuple will be placed in both tuple-spaces ts_1 and ts_2. Assume that $f_1(x_1)$ returns “mike” and $f_2(x_2)$ returns “male”, then when `eval` (∧ (ts_1,ts_2),[$f_1(x_1)$,$f_2(x_2)$]) is executed, the tuple [“mike”,“male”] will be placed into both ts_1 and ts_2.

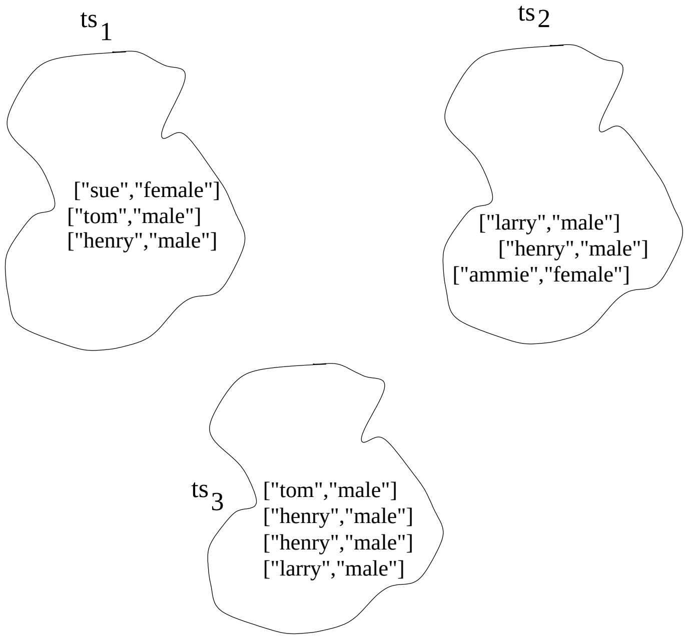
Table 4.1 Code to aid in the semantics of the LogOp Linda primitives using the `And` (∧) operator

Primitive	Code
`in`	results := `in` (∧ (ts_1, ts_2 {, ... , ts_n }) , `template`)
`inp`	results := `inp` (∧ (ts_1, ts_2 {, ... , ts_n }) , `template`)
`rd`	results := `rd` (∧ (ts_1, ts_2 {, ... , ts_n }) , `template`)
`rdp`	results := `rdp` (∧ (ts_1, ts_2 {, ... , ts_n }) , `template`)
`out`	`out` (∧ (ts_1, ts_2 {, ... , ts_n }) , `tuple`)
`eval`	`eval` (∧ (ts_1, ts_2 {, ... , ts_n }) , `tuple`)
`collect`	list := `collect` (∧ (ts_{a1}, ts_{a2} {, ... ,ts_{an} }) , ∧ (ts_{b1}, ts_{b2} {, ... , ts_{bn} }) , `template`)
`copy`	list := `copy` (∧ (ts_{a1}, ts_{a2} {, ... ,ts_{an} }) , ∧ (ts_{b1}, ts_{b2} {, ... , ts_{bn} }) , `template`)

Figure 4.1 contains three tuple spaces to assist with the following examples. When `in` (∧ (ts_1, ts_2), [String , “female”]) is executed it will remove and return two pairs: ([“sue” , “female”] , ts_1) and ([“ammie”,“female”] , ts_2). When `inp` (∧ (ts_1, ts_2) , [“tom” , “male”]) is executed, it will return one pair: ([“tom”,“male”] , ts_1). When `rd` (∧ (ts_1, ts_2) , [“larry” , “male”]), it will simultaneously get a copy of ([“larry”,“male”] , ts_2) and block in ts_1 for a matching tuple. When `rdp` (∧ (ts_1, ts_2), [“joe” , “male”]) is executed it will immediately return (will not block) an empty list because a tuple containing [“joe” , “male”] does not exist in tuple-spaces ts_1 and ts_2. When `copy` (∧ (ts_1, ts_2), ts_3, [

?Name , "male"]) is executed, it will copy a total of four (two from each tuple-spaces ts_1 and ts_2) tuples into tuple-space ts_3. When the primitive returns to the calling process, a list of length two of countPair: 2 , ts_1 , 2 , ts_3. Tuple-Space ts_3 consists of these tuples. The `collect` behaves similarly but moves the matching tuples.

Figure 4.1 Three simple tuple spaces used in illustrating the `and` operator



4.2.1 Semantics Description of the `And` Operator

Blocking for the `in` and `rd` primitives occurs for two reasons. The first happens because of the blocking nature of these primitives in a LINDA model. The second, deals with the new semantics in LOGOP LINDA. The result of a template matching a tuple in a tuple space can either be a 1 or a 0. The value of 1 represents that a tuple exists and can be sent back to the calling process; the value of 0 represents no tuple existing and thus the primitive blocks. These primitives will only unblock if all individually listed tuple spaces return a value of 1; otherwise, if a single tuple space returns 0, the whole primitive is still blocking. Consider the following scenario:

$$\text{results} := \texttt{in} \; (\; \wedge \; (ts_1, ts_2, ts_3) \; , \; \texttt{template})$$

Internal to the LogOp Linda kernel, the primitive would unblock if and only if after all tuple spaces return a 1 value shown below:

$$\text{results} := \texttt{in} \; (\; 1 \wedge 1 \wedge 1 \; , \; \texttt{template})$$

and would remain blocked if any of the tuple spaces (ts_1- ts_3) did not have a tuple matching `template`. Below illustrates ts_2 not having a tuple matching template. This explanation is the same for the `rd` primitive.

$$\text{results} := \texttt{in} \; (\; 1 \wedge 0 \wedge 1 \; , \; \texttt{template})$$

Property 4.2.1 (`in`/`rd` primitive internal blocking/unblocking in the LogOp Linda kernel) *Let S be a set of tuple spaces ts_1, ts_2, ... , ts_n that need to be checked using either the* `in` *or* `rd` *primitive. Then $\forall$ ts_i, where i = 1..n, a value of 1 must be returned for each tuple space before the primitive's result can be returned to the calling process.*

Because of the `And` operator and the nonblocking characteristic of the `inp` and `rdp` primitives, each tuple space listed will be checked only one time in parallel. The possibility of these tuple spaces being checked more than once will be discussed in Section 4.3.2. Once all tuple spaces are checked, the primitive returns to the calling program. The length of the list returned to the calling process depends on the number of tuples retrieved. If no tuple space contained a tuple matching the template, then the list will be of length 0. If, on the other hand, all tuple spaces contained a matching tuple, then the list would be of length n where n is the number of tuple spaces.

One important concept about each of the getter primitives is as soon as a tuple matching a template is found in ts_i, that tuple is locked by the primitive for the calling client process. For example, consider process A executing `in` (ts_1 `and` ts_2, template). If a tuple is only found in ts_1, then this primitive should mark that tuple a being locked by process A. This ensures if process B executes `in` (ts_1, template), B will block if no other tuple exists in ts_1 (other than the one locked by process A). How a tuple is "locked" depends on the implementation of this model.

This also describes an important concept in the LogOp Linda: deadlock. Consider processes A and B and both are executing the following primitive

$$\text{in} \ (\ \wedge \ (ts_1, \ ts_2) \ , \ \text{template})$$

where template is the same for both processes. Consider now that in tuple-spaces ts_1 and ts_2 there is only one tuple that matches template. If processes A and B issue the above command and A checks ts_1 while process B checks ts_2, both tuples will be "locked". Since this is an `And` operator, each process must also check the other tuple space: process A checks ts_2 and process B checks ts_1. At this point, a deadlock will occur for both process A and B because there will not be any tuple that matches the template in the most recently checked tuple spaces. Deadlock is an important issue and will be discussed in Chapter 5.

The `out` and `eval` semantics are very similar to the getter primitives: all tuple spaces are accessed. The obvious difference with these primitives is the tuple spaces receive the tuple specified in the respected primitive. In addition, recall in section 3.1.1 the argument of evaluating each f_i function listed in the serial `eval` primitive calls. This will be shown in chapter 5, but speculating on efficiency, since multiple tuple spaces can be specified in one `eval` primitive call, all f_i functions should be evaluated only once, no matter how many tuple spaces are listed.

The bulk primitives `copy` and `collect` have improved semantics. Similar to the getter and setter primitives, all involved tuple spaces are accessed with a single primitive call. The semantics enables a single source tuple space to be broadcasted to n destination tuple spaces. Likewise, n source tuple spaces can be sent to a single destination tuple space and similarly, the semantics allows n source tuple spaces to be sent to m destination tuple spaces with one primitive call. Furthermore, a requirement of sending the tuples matching a template at time t from n source tuple spaces to n destination tuple spaces is now achievable with this semantics where as in York Linda II it was not. Recall the code in Program 3.1. It can now be written as shown in Program 4.1. Recall the requirements about Program 3.1: at time t each source tuple space needs to copy their own tuples matching template to the destination tuple spaces.

Program 4.1 York Linda II `copy` primitive

```
(1)  countPair = collect( and (ts1, ts2, ts3, ts4, ts5),
                          and (ts1, ts2, ts3, ts4, ts5),
                          template );
```

4.3 `Or` Logical Operator

This section discusses the semantics of the `Or` operator. To assist the explanation, code in Table 4.2 is provided. The `in` and `rd` primitives return a list of pairs. The minimum number of pairs returned using the `or` operator is one and the maximum is based on the number of listed tuple spaces. Since the `inp` and `rdp` primitives do not block, the minimum number of pairs returned is zero and the maximum is the number of listed tuple spaces. The `out` and `eval` primitives nondeterministically chooses between one and n tuple spaces from the list and places in each chosen tuple space the `tuple`. The `collect` primitive nondeterministically chooses between one and n source tuple spaces and moves the `tuples` matching `template` into between one and n nondeterministically chosen destination tuple spaces. The `copy` primitive works similarly to the `collect` except it copies the matching `tuples`. Both of these primitives returns a list of countPair.

Table 4.2 Code to aid in the semantics for the LogOp Linda primitives using the `Or` ($\vee$) operator

Primitive	Code
`in`	results = `in` ($\vee$ (ts_1, ts_2 {, ... , ts_n }) , `template`)
`inp`	results = `inp` ($\vee$ (ts_1, ts_2 {, ... , ts_n }) , `template`)
`rd`	results = `rd` ($\vee$ (ts_1, ts_2 {, ... , ts_n }) , `template`)
`rdp`	results = `rdp` ($\vee$ (ts_1, ts_2 {, ... , ts_n }) , `template`)
`out`	`out` ($\vee$ (ts_1, ts_2 {, ... , ts_n }) , `tuple`)
`eval`	`eval` ($\vee$ (ts_1, ts_2 {, ... , ts_n }) , `tuple`)
`collect`	countPairs = `collect` ($\vee$ (ts_{a1}, ts_{a2} {, ... ,ts_{an} }) , $\vee$ (ts_{b1}, ts_{b2} {, ... , ts_{bn} }) , `template`)
`copy`	countPairs = `copy` ($\vee$ (ts_{a1}, ts_{a2} {, ... ,ts_{an} }) , $\vee$ (ts_{b1}, ts_{b2} {, ... , ts_{bn} }) , `template`)

4.3.1 Semantics Description of the `Or` Operator

The LogOp Linda kernel determines how and when the `in` and `rd` primitives unblock can be viewed as a series of 1s and 0s. The `in` and `rd` primitives will remain blocked until at least one of the tuple spaces listed return a 1 value stating it found a tuple which matched the template. The `Or` operator has a distinct characteristic from the `And` operator: It will not block and wait for *all* tuple spaces to return a value of 1. Hence, given the following `in` primitive below

results = `in` ($\vee$ (ts_1, ts_2, ts_3) , `template`)

will unblock if, and only if, one or more tuple spaces returns a one (1). Below illustrates that ts_1 and ts_2 have a tuple that matches `template` and thus will return them and which tuple spaces they came from.

results = `in` (1 $\vee$ 1 $\vee$ 0 , `template`)

The `Or` operator has different semantics than the `And` operator using non-blocking primitive `inp`. Because it does not block, a tuple that matches the given `template` does not need to exist in any tuple space in order for the primitive to return to the calling process. Hence the following statement shows no matching tuples and will return immediately to the calling client process with a list containing no entries.

$$\text{results} = \texttt{inp}\ (\ 0 \vee 0 \vee 0\ ,\ \texttt{template})$$

The above examples and the discussion using the `in` and `inp` primitives can be interchanged with `rd` and `rdp` primitives, respectively.

The `out` and `eval` semantics for `Or` operator imply the LINDA kernel can nondeterministically choose the destination tuple spaces. This type of nondeterminism of choosing tuple spaces can possibly increase efficiency during runtime and will be discussed shortly.

If a client process is blocked on two tuple-spaces ts_1 and ts_2 using the `Or` operator and another process inserts simultaneously a tuple into both ts_1 and ts_2 that would unblock the client process, the client process should receive both of the tuples.

4.3.2 Location-aware

The semantics for the `Or` operator leads itself toward a very interesting coordination pattern: choosing which tuple spaces out of the original given list can be deferred to the LOGOP LINDA kernel. Using such a deferral, the kernel could choose which tuple spaces based on physical locality of the tuple spaces. This could increase performance of the primitive by only (or by first) accessing tuple spaces that are close to the client process executing the primitive. It is very difficult to define "closer" with respect to tuple spaces and a client process. Tuple Spaces that are close could be on the same subnet as the client process; or tuple spaces on a different subnet that is in different room. Close could be defined as high bandwidth between the client process and where the tuple spaces are physically located. From the point of view of a client process, this type of closeness could mean that the same set or subset of tuple spaces are being considered for every execution of a primitive. Thus for the getter primitives, if these tuple spaces do not contain a tuple matching the template, the whole list of tuple spaces (contained in the primitive) would then be checked. Again the concept of location-aware could make the model more efficient with respect to speed.

Furthermore, if the client process is on a mobile device, what tuple spaces are closer at $time_a$ might not be closer at $time_b$. For example, consider a global positioning system (GPS) broadcasting signals out to tuple spaces at $time_a$. As the device moves, it may take longer to broadcast to the original tuple spaces because a new set of tuple spaces maybe closer to the mobile device. This is an important concept: because the device is mobile, it is not bias towards always choosing a particular set of tuple spaces.

In the case that a client process is not mobile and executing a getter primitive, if the bias tuple spaces do not contain a matching tuple, then the whole list of tuple spaces, including the closer tuple spaces, should be checked.

4.4 `Not` Logical Operator

This section covers the `Not` operator for the LogOp Linda primitives listed in Tables 4.1 and 4.2. Table 4.3 shows the code of the `And` and `Not` operators.

The semantics of the `Not` operator is to determine which tuple spaces are to be accessed during the execution of a primitive. For example, consider process A obtaining over the time of execution (via the getter primitives or the tsc() primitive) the following tuple spaces:

$$\{ \text{uts}, ts_1, ts_2, ts_3, ts_4 \}$$

Table 4.3 Code to aid in the semantics for the LogOp Linda primitives using `And` and `Not` operators

Primitive	Code
`in`	results := `in` ($\wedge$ (`not` (ts_1, { ts_2, ... ,ts_n }), `template`))
`inp`	results := `inp` ($\wedge$ (`not` (ts_1, {ts_2, ... ,ts_n }), `template`))
`rd`	results := `rd` ($\wedge$ (`not` (ts_1, {ts_2, ... ,ts_n }), `template`))
`rdp`	results := `rdp` ($\wedge$ (`not` (ts_1, {ts_2, ... ,ts_n }), `template`))
`out`	`out` ($\wedge$ (`not` (ts_1, {ts_2, ... ,ts_n })) , `tuple`)
`eval`	`eval` ($\wedge$ (`not` (ts_1, {ts_2, ... ,ts_n })) , [tuple])
`collect`	countPairs := `collect` ($\wedge$ (`not` (ts_{a1}, {ts_{a2}, ... ,ts_{an} })) , $\wedge$ (not(ts_{b1}, {ts_{b2}, ... ,ts_{bn} }),`template`))
`copy`	countPairs := `copy` ($\wedge$ (`not` (ts_{a1}, {ts_{a2}, ... ,ts_{an} })) , $\wedge$ (not (ts_{b1}, {ts_{b2}, ... ,ts_{bn} })) , `template`)

Thus, the set difference of ts_1, ts_2 from the well known tuple spaces would be uts, ts_3, ts_4. Snyder and Menezes note that this set difference is also considered the complement of the list of tuple spaces [SM02]. Once the set difference is applied, which operator, `And` or `Or`, should be used? Table 4.3 illustrates the `And` operator; however, the `Or` operator can be easily substituted. Combining these logical operator semantics, consider process A having well known tuple spaces:

$$\{ \text{ uts}, ts_1, ts_2, ts_3, ts_4 \}$$

Now process A issues the following `in` primitive to the Linda kernel:

results := `in` ($\wedge$ (`not` (uts , ts_1)) , `template`)

After applying the set difference semantics, the Linda kernel would process the `in` statement as:

results := `in` ($\wedge$ (ts_2, ts_3, ts_4) , `template`)

Note the resulting statement executes exactly the same as the `And`'s semantics.

There are a couple points of interest that are associated with the `not` operator. The first is a side effect to the `Not` operator. If process A only knows one tuple space, uts, then after applying the set difference algorithm to

$$\text{results} = \texttt{in}\ (\ \wedge\ (\ \texttt{not}\ (\text{uts})\ ),\ \texttt{template})$$

the result will be an empty set of tuple spaces. At the model level, two actions can occur when the blocking getter primitives are executed. The first action is they block indefinitely. While this is an undesirable action, it is a valid semantics. Furthermore, a deadlock detection algorithm, covered in Chapter 7, could alleviate blocking indefinitely. A second action would be for the blocking primitives to just return an empty list. As for the nonblocking primitives, this issue is irrelevant because they would return an empty list as a result. For the setter primitives, the tuple would be inserted into no tuple space. At the implementation, the LINDA kernel should throw an error/exception back to the client process stating there are no tuple spaces listed in the primitive. The second point is after the set difference is applied, there may be only one tuple space used in the primitive. For example, if the well known tuple spaces are uts, ts_1, and ts_2 and the following executes

$$\text{results} := \texttt{in}\ (\ \wedge\ (\ \texttt{not}\ (\text{uts},\ ts_1)\ )\ ,\ \texttt{template})$$

LOGOP LINDA will issue the `in` primitive with tuple-space ts_2 using the original LINDA semantics and ignore the `And` operator because there is no second tuple space.

4.5 Model Comparisons

This section covers an in depth comparison to all the models previously discussed in Chapter 2. It will illustrate the semantics differences of these models against the semantics of LOGOP LINDA and speculate on some possible improvement of efficiency at runtime.

4.5.1 LogOp Linda and Linda Comparison

LOGOP LINDA has several advantages over the original LINDA model. One advantage is its ability to access multiple tuple spaces using one primitive call. This type of access, one primitive call and many tuple spaces, adheres to a new concept of parallel primitives. Recall that the semantics of LINDA state when accessing n tuple spaces using the getter/setter primitives n primitive calls are made. The semantics of LOGOP LINDA are quite different: One primitive call can be made instead of consecutive primitive calls. This type of semantics could lead to a more efficient implementation because only one primitive call is made to access several tuple spaces. However, this efficiency concept is difficult to show at the model level but is shown at the implementation level described in the next chapter. Since only one primitive is used, the time it takes to retrieve n tuples from n tuple spaces is

$$\max(time_1 \ , time_2 \ , ... \ , time_n \ )$$

where each $time_i$ represents the time taken to retrieve a tuple from tuple-space ts_i. Similarly, the time it takes to send (using either an `out` or an `eval`[2] primitive) a tuple to n tuple spaces is the time it takes to execute only one `out` or one `eval` primitive. Contrast this efficiency with the original LINDA, as stated earlier,

$$\textstyle\sum_{i=1}^{n} time_i$$

These parallel primitives could lead to better efficiency and scalability which are illustrated in Properties 4.5.1 and 4.5.2.

Property 4.5.1 (Parallel access for LOGOP LINDA getter primitives.) *Given a single template that matches a tuple from n tuple spaces, the time it takes will be max($time_1$, $time_2$, ... , $time_n$) where each $time_i$ represents the time it takes to retrieve a tuple from tuple-space ts_i.*

Property 4.5.2 (Number of parallel primitive calls with respect to tuple spaces.) *Let S be a set of tuple spaces ts_1, ts_2, ... , ts_n. Then $\forall$ ts_i, where i = 1..n, $\exists$ only a single getter/setter primitive is being executed using a logical operators.*

4.5.2 LogOp Linda and Bonita Comparison

BONITA achieves efficiency through semantics defining asynchronous primitives. What BONITA does not offer with its efficiency is the ability to access multiple tuple spaces in

[2]This time does not include the time it takes to process $f_1(x_1)$ ... $f_n(x_n)$.

parallel using a *single* primitive call. Consider the following LogOp Linda `in` primitive:

$$\text{pairs} = \texttt{in}\ (\wedge\ (ts_1{,}ts_2{,}ts_3{,}ts_4),\ \texttt{template})$$

Although the LogOp Linda `in` primitive is not asynchronous, there is still only one primitive call being executed. Consider now Bonita's `dispatch` primitive and its attempt to retrieve four tuples from four tuple spaces:

Program 4.2 Bonita's dispatch and obtain primitives

```
(1)  id1 := dispatch( ts1 , template , destructive)
(2)  id2 := dispatch( ts2 , template , destructive)
(3)  id3 := dispatch( ts3 , template , destructive)
(4)  id4 := dispatch( ts4 , template , destructive)
(5)  tuple1 := obtain(id1)
(6)  tuple2 := obtain(id2)
(7)  tuple3 := obtain(id3)
(8)  tuple4 := obtain(id4)
```

The asynchronous feature of Bonita makes the model powerful. However, it still has to make eight primitive calls to retrieve four tuples matching the template. LogOp Linda's parallel primitive feature stated in Properties 4.5.1 and 4.5.2 could be combined with Bonita's asynchronous primitives using logical operators to create an even more efficient and expressive coordination model. Chapter 7 expands on this concept.

4.5.3 LogOp Linda and Scopes Comparison

Scopes has limitations with expressiveness if n copies of a tuple need to be placed in Scope S, then n serial calls need to occur. Contrasting this with Property 4.5.2, LogOp Linda is feasibly capable of this by combining the same tuple space n times using the `And` operator. The execution below illustrates a tuple being sent to tuple-space ts_1 in parallel using the `And` operator to allow n copies of tuple to exist in ts_1.

$$\texttt{out}\ (\ \wedge\ (ts_1,\ ts_1,\ ....\ ,\ ts_1),\ \texttt{tuple})$$

Furthermore, in Scopes, there is a containment relationship: A single Scope can encompass one or more Scopes (sets). In addition, each set of an existing Scope can itself be a Scope to other sets. Hence, a single Scope could consist of n-levels of descendants. Consider Figure 4.2 as an illustration: Scope [a,b,c] consists of sets [a], [b] and [c] while Scope [b,c] consists of the same sets [b] and [c]. This containment is based on the arrows' directions. If process A inserts a tuple into Scope [a,b,c], then the tuple can be retrieved by process B: assuming B has a handle to Scope [b,c] and uses a template that matches. This concept is exactly how Scopes operate: a strength of the semantics itself. However, consider now how process B actually retrieves the Scope [b,c]. If process A puts that Scopes' handle into the Universal Tuple Space and process B retrieves that Scope, how does process B know exactly what sets are involved with this particular Scope. Another question is, should process B have the capability (via a new primitive) to know what sets are contained inside this Scope? Consider the code in Programs 4.3 and 4.4

Program 4.3 Process A code

```
aScope = NewScope();
bScope = NewScope();
cScope = NewScope();

abcScope = aScope U bScope U cScope;
bcScope = bScope U cScope;

out( uts , [ bcScope ] );
out(abcScope , [‘‘dog’’] );
```

Program 4.4 Process B code

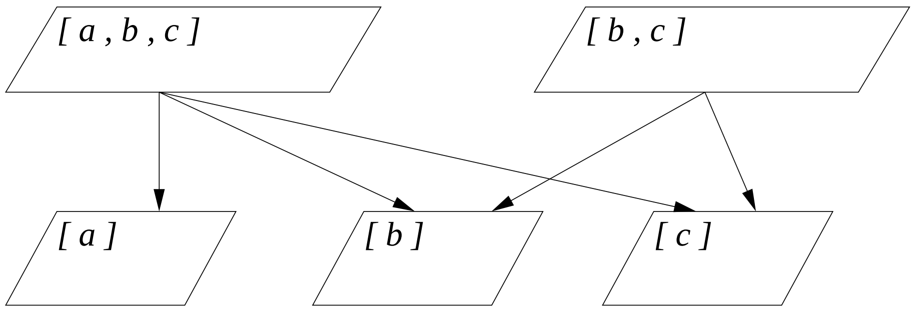
```
theScope = in( uts , [ScopeFormal] ).getScope();
values = in (theScope , [StringFormal] );
value = values.getValue();
scope = values.getScope();
```

LogOp Linda does not (nor can it) hide the intended tuple spaces. Hence, when several tuple spaces are considered with the setter primitives using the `And` operator, it is known where the tuple will reside. When several tuple spaces are considered with the setter primitives using the `Or` operator, there is no ability of another process retrieving that tuple from a different tuple space not listed in the original primitive call. Hence, the following

$$\text{out}(\vee\ (ts_1, ts_2, ts_3, ts_4)\ ,\ \texttt{tuple})$$

states that any one tuple space or a combination of the listed tuple spaces will contain `tuple`. In LogOp Linda, another tuple space, for example ts_5, could not contain the tuple that is placed in any of the tuple spaces listed above.

Figure 4.2 Scope [a,b,c] sharing the same sets as Scope [b,c]



As stated by Merrick and Wood, Scopes requires an extra data structure to be maintained on the server side in Linda. Based on the number of Scopes in a system, this required data structure could be expensive to maintain all the Scopes and their references. This could get expensive if no garbage collection is implemented [Men98].

LogOp Linda does not require an extra data structure to be maintained across several primitive calls during the execution of a process. During the execution lifetime of the LogOp Linda kernel the *logical* combination of tuple spaces solely exists during execution of that single primitive call, thus tuple spaces remain ***physically*** disjoint in the kernel. Further explanation of the implementation of LogOp Linda for this thesis will be discussed in Chapter 5.

4.6 Coordination Patterns

There are several major contributions LogOp Linda makes to Coordination. This section describes some coordination patterns that achieve better expressiveness with less primitive calls in LogOp Linda that are not achievable in other Linda models.

4.6.1 1-Producer, n-Consumer Architecture

The classic example of distributed systems is the producer/consumer architecture which LINDA can obviously express through serial access of the tuple spaces. However, as noted previously in Chapter 3, the number of primitive calls it takes for the producer to send is proportional to the number of tuple spaces accessed. The code is depicted in Figure 4.3. In another scenario, even if there is only one tuple space receiving the producer's tuple, it would still need to be sent the same number of times as there are consumers. Figure 4.4 illustrates this. Contrast this type of expressiveness with LOGOP LINDA's. LOGOP LINDA only needs one primitive call in both scenarios and is illustrated in the following two calls:

`out` ($\wedge$ (ts_1,ts_2,ts_3, ... , ts_n), new Tuple(item))

`out` ($\wedge$ (ts_1,ts_1,ts_1, ... , ts_1), new Tuple(item))

The above code is scalable because no matter how many tuple spaces are going to receive the same tuple, it only takes one primitive call. This is an especially important concept: whether the same destination tuple spaces or different tuple spaces, there is only one primitive execution.

A consumer can take advantage of LOGOP LINDA's getter primitives by retrieving several tuples from several tuple spaces (different or the same tuple spaces) with one primitive call using the same template. This type of expressiveness is similar to the producer's `out` or `eval` primitive discussed above. Using the code from Table 4.1, the following illustrates a consumer retrieving three tuples from three different tuple spaces and also from three of the same tuple space.

Additionally, accessing all tuple spaces at once ensures that blocking on a single tuple space does not occur and hinder other tuple spaces listed in the same primitive from locking a matching tuple.

results := `in` ($\wedge$ (ts_1,ts_2,ts_3), template)

results := `in` ($\wedge$ (ts_1,ts_1,ts_1), template)

Figure 4.3 The producer executes n `out` primitive calls to n different tuple spaces

```
$item := computeIt();$
out($ts_1, newTuple(item)$)
out($ts_2, newTuple(item)$)
out($ts_3, newTuple(item)$)
....
out($ts_n, newTuple(item)$)
```

Figure 4.4 The producer executes n `out` primitive calls to the same tuple space

```
$item := computeIt();$
out($ts_1, newTuple(item)$);
out($ts_1, newTuple(item)$);
out($ts_1, newTuple(item)$);
....
out($ts_1, newTuple(item)$); // the nth call.
```

4.6.2 Master/Worker Architecture

The previous section covered the 1-Producer/n-Consumer problem in order to highlight LogOp Linda's expressiveness, scalability and efficiency using the `out` primitive. This section examines LogOp Linda's `in` primitive call using the `Or` operator in a master/worker architecture.

Consider a network of equally powerful computer systems dedicated to solving many N-Queens problems simultaneously. The N-Queens problem is stated as follows: On an 8x8 chess board, place 8 queens on the board in such a way that neither queen can attack another queen. As N increases, the computation time exponentially increases. Each system in the network has its own tuple space, and each system is required to solve an N-Queen problem. In this example, there consists 100 systems which known as the worker systems.

The network consists of 100 worker computers required to solve 100 N-Queen problems.

Another computer, the master system, distributes each N-Queen task to each tuple space. The pseudo code for this master system to insert these 100 tuples into 100 tuple spaces is shown in Figure 4.5. Note that, since there are 100 different tuples, the code uses the LINDA `out` primitive. In addition, the chessboard sizes will range from 15 to 30 queens[3]. Each worker's code is listed in Figure 4.6 which retrieves the tuple from its known tuple-space ts_i, computes the solution, and sends the result back to their known tuple-space ts_i.

Figure 4.5 Master's LOGOP LINDA pseudo code to produce 100 tuples into 100 tuple spaces

```
$results$ := in($\wedge(uts, uts,$ ... $, uts), ?TupleSpace)$ //uts is repeated 100 times.
$i := 0$
while ($i < 100$) do
    $n := randomNumberBetween(15, 30)$
    out(ts_i, [$n, chessboard$])
    $i := i + 1$
end
```

Figure 4.6 Each worker's LINDA code to compute it's own N-Queens problem

```
$result$ := in($ts_i$ , [$?Integer$ , $?ChessBoard$])
$startTime := getTime()$
$chessBoardSolution := computeSolution(result.getN()$ , $result.getChessBoard())$
$stopTime := getTime()$
out($ts_i$ , [$(stopTime - startTime)$ , $result.getN()$ , $chessBoardSolution$])
```

Because the number of queens range from 15 to 30, these solutions will vary in the time it takes to solve. However, one of the requirements of the architecture is to print the resulting computation times as they become available. In other words, as a worker finishes the solution and puts the solution into its tuple space, the master system must be ready to

[3]The values of 15 and 30 could have been any other range of values.

retrieve the resulting tuple from the tuple space as quickly as possible and print the result to the user.

The requirement of the architecture to print the result as it is available requires the master process to poll in the LINDA model and do a blocking-poll in the LOGOP LINDA model. A blocking-poll is different from doing an actual poll. In an actual poll, if the information is not available, the primitive returns immediately with nothing and continuously does this. In a blocking-poll, if the information is not there the process blocks. As information is obtained, the process unblocks and it is returned to the calling process. The calling process then reissues the primitive call again. To solve this architectural problem, LOGOP LINDA will use its `in` primitive and the `Or` operator. Recall the semantics of LOGOP LINDA's primitives using the `Or` operator, the `in` primitive will block until at least one tuple matches a `template`. It will be shown below how more efficient LOGOP LINDA is compared with LINDA.

Figure 4.7 illustrates the polling (via the `inp` primitive) that LINDA must do in order to meet the requirements as described above. The process iterates in two loops. The outer while loop will continue to execute as long as max never decreases down to zero (0). The inner loop goes from 1 to max where max is initialized at 100 (the number of worker processes). Inside the for loop is the `inp` primitive. This will not block if a tuple does not match the template [?Time, ?count, ?ChessBoardSolution] in the tuple space array[i].getTS(). If there is a valid resultTuple, then that time is printed and that tuple space does not need to be checked again. After each for loop, the array entries marked as done are removed and max is decremented by the amount of tuples found. Thus, each complete iteration of the for loop has a chance to reduce the value of max. However, if no tuples are found, the for loop continuously polls each tuple space in the array. This is the inefficiency of this primitive in LINDA and in the worst case continuous execution of the for loop occurs until all tuples are found. It is continuous because the `inp` does not block. Thus, the number of `inp` calls is the amount that can be done in the time it takes for all worker processes to be completed.

The code in Figure 4.8 illustrates how LOGOP LINDA would implement the architectural requirements. Recall the semantics of LOGOP LINDA's `in` primitive using the `Or` operator.

Figure 4.7 LINDA code to solve the N-Queens problem

```
$array := getAllTupleSpaces()$
$max := 100$
$tmpMax := max$
while ($max$ != 0) do
        for $i := 1$ to $max$ step 1 do
            $resultTuple := inp(array.\mathrm{ts\_i}, [?Time, ?count, ?ChessBoardSolution])$
            if ($resultTuple! = null$) then
                                          $print((Time)resultTuple.get(0))$
                                          $array[i].markedAsDone()$
                                          $tmpMax := tmpMax - 1$
            fi
        od
        $array = removeMarkedAsDone(array)$
        $max := tmpMax$
end
```

The while loop continues until there are no more tuple spaces to be checked. This occurs when all worker processes have solved their N-Queens problem and have each put the resulting tuple into their tuple space (illustrated in Figure 4.8). Each execution of the LOGOP LINDA's `in` primitive has a chance to return between 1 and N tuples where N is the number of tuple spaces being checked in parallel. The worst case is that a 100 calls are made using the blocking `in` primitive. This only happens if two or more workers do not finish at the same time. The requirement above is that the information needs to be polled. As shown in Figure 4.8, the LOGOP LINDA primitive combined with the `Or` operator is far more efficient than LINDA's `inp`. LINDA's worst case is far less efficient than LOGOP LINDA's worst case. In addition, the decrease of the number of `in` primitive calls in LOGOP LINDA implementation is strictly due to the fact LOGOP LINDA can check multiple tuple spaces in parallel using logical operators.

As stated in [MW00], if a tuple matches a template in a SCOPE, then that tuple and SCOPE are returned. Hence, the minimum and maximum number of `inp` primitive calls using SCOPES is 100. Figure 4.9 illustrates the SCOPES code. Although this is efficient, LOGOP LINDA could possibly do it with less calls because the `in` primitive can return more than one tuple at a time.

Figure 4.8 LogOp Linda code to retrieve the N-Queens solution

```
$set := getOrListOfTupleSpaces() // returns \ \vee (ts_1, ts_2, \ ... \ , ts_{100})$
**while** $(set.length \ != 0)$ **do**
    $results := in(set, [?Time, ?count, ?ChessBoardSolution])$
    **for** $i := 1$ **to** $results.length$ **step** $1$ **do**
        $print((Time)results.getTuple(i).get(0))$
        $set.remove(results.getTupleSpace(i))$
    **od**
**end**
```

Figure 4.9 Scopes code

```
$allScopes := getAllScopes() // returns(scope_1 \oplus scope_2 \oplus \ ... \ \oplus scope_{100})$
**while** $(count! = 100)$ **do**
    $result := in(allScopes, [?Time, ?count, ?ChessBoardSolution])$
    $print((Time)result.getTuple().get(0))$
    $count := count - 1$
**end**
```

4.7 LogOp Linda Summary

Tables 4.4 and 4.5 summarizes the semantics of LogOp Linda's primitives discussed in this chapter. As stated earlier, for the `Or` operator, if a location-aware concept is used, tuple spaces may be checked more than once.

Table 4.4 Semantics for LogOp Linda primitives using the `And` ($\wedge$) operator

Primitive	Semantics Overview
`in`	Removes a tuple from each of the tuple spaces ts_1 ... ts_n that associately matches `template`. It returns the list of matching tuples and tuple spaces to the calling process. This primitive will not return until one tuple has been removed from each tuple space listed. The length of the list is n, where n is the number of tuple spaces.
`inp`	Removal characteristic is the same as LogOp Linda `in`. Recall this is a non-blocking primitive and only checks each tuple space (ts_1... ts_n) one time and then returns to the calling process. The length of the list is between 0 and n, where n is the number of tuple spaces listed.
`rd`	Returns to the calling process a copy of a tuple that associately matches `template` from each of the tuple spaces ts_1 ... ts_n. This primitive will not return until one tuple has been copied from each tuple space listed. The length of the list is n, where n is the number of tuple spaces.
`rdp`	Removal characteristic is the same as LogOp Linda `rd`. Recall this is a non-blocking primitive and only checks each tuple space (ts_1... ts_n) one time and then returns to the calling process. The length of the list is between 0 and n, where n is the number of tuple spaces listed.
`out`	Places tuple into each of the tuple spaces ts_1 ... ts_n.
`eval`	Evaluates all functions $f_1(x_1)$ to $f_m(x_m)$ once. Each of the functions' return value is an actual in the newly created tuple that is inserted into the tuple-space ts_1... ts_n.
`collect`	*Removes* the tuples matching `template` in all the source tuple spaces (ts_{a1}, ts_{a2}, ... , ts_{an}) and inserts them into the destination tuple spaces (ts_{b1}, ts_{b2}, ... ,ts_{bn}). Returns a list of countPairs with each entry stating the source tuple space and the number of tuples removed.
`copy`	*Copies* the tuples matching `template` in all the source tuple spaces (ts_{a1}, ts_{a2}, ... , ts_{an}) and inserts them into the destination tuple spaces (ts_{b1}, ts_{b2}, ... ,ts_{bn}). Returns a list of countPairs with each entry stating the source tuple space and the number of tuples copied.

Table 4.5 Semantics for LogOp Linda primitives using the `Or` ($\vee$) operator

Primitive	Semantics Overview
`in`	Checks and tries to remove a tuple from each of the tuple spaces ts_1 ... ts_n that associately matches `template`. It returns the list of matching tuples and tuple spaces to the calling process. This primitive will not return until at least one tuple has been removed by any tuple space listed. If at a given time, (n - x) tuples match the `template`, then a list of length (n - x) is returned.
`inp`	Removal characteristic is the same as LogOp Linda `in`.
`rd`	Checks and tries to return a copy of a tuple from each of the tuple spaces ts_1 ... ts_n that associately matches `template`. It returns the list of matching tuples and tuple spaces to the calling process. This primitive will not return until at least one tuple has been copied from any tuple space listed.
`rdp`	Removal characteristic is the same as LogOp Linda `rd`.
`out`	Nondeterministically chooses one or more tuple spaces from the given list and places tuple into them.
`eval`	Evaluates all functions $f_1(x_1)$ to $f_m(x_m)$ once. Each of the functions return value is an entity in the newly created tuple to be inserted into one or more nondeterministically chosen tuple-spaces ts_1... ts_n.
`collect`	Nondeterministically chooses one or more tuple spaces from the source and destination lists and *removes* the tuples matching `template` in the chosen source tuple spaces and inserts them into the chosen destination tuple spaces.
`copy`	Nondeterministically chooses one or more tuple spaces from the source and destination lists and *copies* the tuples matching `template` in the chosen source tuple spaces and inserts them into the chosen destination tuple spaces.

Chapter 5

Implementation

This chapter describes the implementation of LogOp Linda. Java [GJSB00] was chosen as the language of implementation due to its write once, run every where and thus could execute in a heterogeneous environment without requiring recompiling.

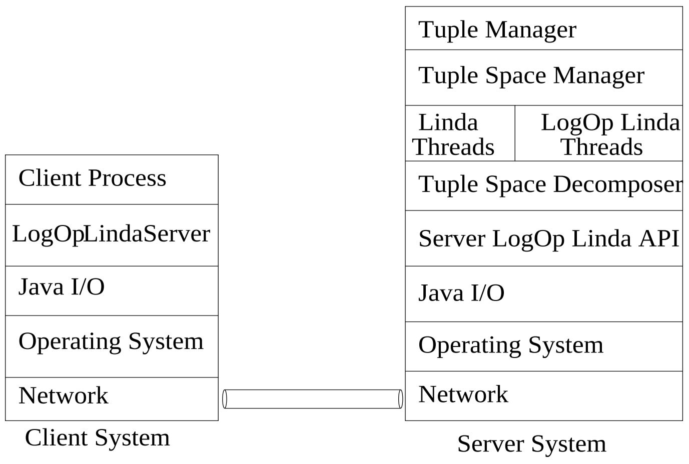
The implementation is written as a proof of concept to verify the theory of using logical operators as a more expressive, scalable and efficient Linda model. The system does not support fault tolerance [Som92] or anomalies in the network or on hardware that execute either the client processes or the LogOp Linda kernel processes. Furthermore, LogOp Linda does not support memory exhaustion in the hardware system; thus, it does not migrate parts of tuple spaces from one hardware system to another and does not store tuples out onto the hard drive when virtual memory is low in the virtual machine.

In order to do test comparisons between a Linda model and the LogOp Linda model, both models had to be implemented. The implementation of both will be discussed below. Unless stated otherwise, the explanation of implementation will represent both models; however, there are some differences with respect to how each model's primitives are processed through the system.

5.1 LogOp Linda Architecture

The LogOp Linda kernel is composed of two systems: a client system and a server system. Sockets are used for the inter-process communication between each server and their clients. LogOp Linda's layer hierarchy of subsystems is illustrated in Figure 5.1. The horizontal lines represent the current subsystem's interaction to the next subsystem that is either above or below. The subsystems implemented for LogOp Linda are the client processes, Linda API, LogOp Linda API, Server LogOp Linda API, TupleSpaceDecomposer, Linda Threads, LogOp Linda Threads, TupleSpaceManager, TupleManager.

Figure 5.1 LogOp Linda subsystem hierarchy



5.1.1 Client System Architecture

The client system provides developers with an API that supports the primitives described in Chapter 4 with the exception of `collect` and `copy`: For these a design is covered in Section 5.4. The API also provides exception handling for the side effect using the `Not` operator as covered in Chapter 4. Hidden from the developer is the API needed for socket communication between the client process and the LogOp Linda kernel. If a client process is started on a server system (one executing a LogOp Linda kernel), it will connect to that server; otherwise, a server is randomly chosen from a properties file.

5.1.2 Server System Architecture

The server system contains inter-process communication between the client processes and the LogOp Linda servers. Several LogOp Linda servers can participate at once; likewise, several client processes can be connected to these servers. The socket connection between a server and a client process is synchronous while the connection between each of the LogOp Linda servers is asynchronous. In order for the LogOp Linda kernel to operate correctly, all hardware systems must have already started their LogOp Linda server. Servers know of other servers because all participating servers are listed in the property file. Any server that suffers an anomaly may cause the other servers involved in the LogOp Linda kernel to fail.

Each LogOp Linda server involved has one socket connecting to all the other LogOp Linda servers. This socket allows for reading and writing information between the servers and thus each server has 2(n-1) (n is the number of participating servers) threads for each socket connection to other servers: a thread for reading from the socket and a thread for writing to the socket. The advantage of having 2(n-1) threads approach for server inter-process communication is each thread has a single responsibility: reading from or writing to another server. The disadvantage of this approach is if there are many servers involved in the LogOp Linda system, then having a single thread for each server's read/write sockets may exhaust the LogOp Linda kernel. A solution (not implemented) to this issue of several LogOp Linda kernels is to use a thread pool. A thread pool is a finite number of threads allocated to a particular LogOp Linda kernel. Thus no matter how many LogOp Linda kernels are connected together, in order to write to a socket, a information is queued and the next available thread would write the data to the socket. Each of the LogOp Linda setter and getter primitives use their own thread to process tuples and tuple spaces. The use of separate threads allows each primitive to execute in parallel on a multiple processor machine.

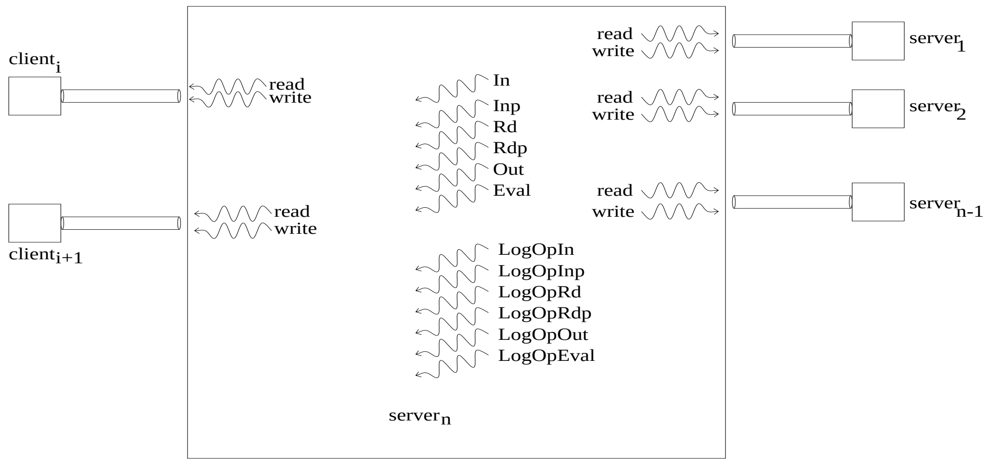
For example, consider a scenario where there are three servers: A, B and C. Assume server C contains ts_1, ts_2, ts_3 and ts_4. A client process D connects to server A and another client process E connects to server B. At the same time, client process D executes a LogOp Linda `in` primitive accessing both ts_1 and ts_2; likewise, client process E executes a LogOp

LINDA `rd` primitive accessing ts_3 and ts_4. If server C has a read (and write) thread for each server connection, then these two primitives can access the respected tuple spaces in parallel; as they should, since all primitives and tuple spaces are disjoint.

5.1.3 Architecture Summary

Figure 5.2 is a pictorial representation of the client and server system architecture. It is a generic view consisting of $client_i$ and $client_{i+1}$ processes connecting to LOGOP LINDA's $server_n$ where there are n servers involved in the system. Each inter-process communication is represented as a pipe and the threads involved are represented as squiggly lines.

Figure 5.2 Detailed LOGOP LINDA architecture



5.2 Design

The design of the client and server systems uses object oriented methodologies (encapsulation, inheritance, and polymorphism), design patterns [GHJV94], and a properties file for configuration is used. The methodologies will be covered throughout this chapter and the design patterns that are used follows:

Singleton: Insures that only one instance of a class is created at run-time and there is global access to it.

Chain of Responsibility: Decouple the sender of a request from its receiver. This allows more than one object to handle the request and leads to a more flexible and reusable implementation. Additionally it promotes better concurrency. This is also known as a delegation.

Command: Encapsulates a request as an object.

A properties file called LogOp.properties is used for configuring the participating servers, the port, and the server that contains the Universal Tuple Space. The Universal Tuple Space server must be one of the participating servers. Appendix I contains an example of the properties file.

The design is decomposed into two categories: the client and server. The client design consists of the code for the communication infrastructure to the server and the developer's API. The server design consists of APIs that communicate with the client's communication infrastructure, APIs that communicate between LogOp Linda servers, and general classes that implement tuple and tuple space concepts and LogOp Linda primitives.

The communication infrastructure for client and server processes is implemented using Java sockets. Java sockets contain an input stream and an output stream. These streams are wrapped inside a BufferedInputStream and BufferedOutputStream, respectively, for efficiency. Java objects are passed between the client process and the server process and thus the buffered streams are wrapped inside an ObjectInputStream and an ObjectOutputStream, respectively. The code for this is listed in Appendix H.

5.2.1 Client Design

Recalling Figure 5.1, a developer has access to the LogOp Linda API and Linda API subsystems. This subsystem contains several classes and interfaces. Discussed below are LogOpLindaServer, LogOpLinda, Linda, TupleSpace, Tuple, Tuples, Template, ActiveTuple and LogOp classes and interfaces.

Java does not support variable length parameters method signatures. As such many constructors take only up to a finite number of parameters (most of the time 8). However, where applicable in the discussion below, a public method called *add()* allows the developer append the 9th, 10th, etc. objects to the instance of that class.

LogOpLindaServer Class

LogOpLindaServer is the only class used by the client process to connect with a LogOp Linda server. It is a singleton object that has a public connectLogOpLinda class method. The return object for the connectLogOpLinda method is of type LogOpLindaServer. This class implements both Linda and LogOpLinda interfaces. These interfaces also extends an interface that contains methods to retrieve the tuple-space Universal Tuple Space and to create new tuple spaces. Program 5.1 illustrates actual Java source used to connect to the LogOp Linda server, to retrieve the Universal Tuple Space object (from a local private data member in the LogOpLindaServer object), and to create a new tuple-space object assigned to variable ts1. All tuple spaces that a client process creates using a createTupleSpace method implemented in the LogOpLindaServer resides on the server the client process is connected to. Appendix A contains documentation for these methods and interfaces described.

Program 5.1 Connection, Universal Tuple Space, and new tuple space from LogOpServer class

```
(1) LogOpLindaServer server = LogOpLindaServer.connectLogOpLinda();

(2) TupleSpace uts = server.getUTS();

(3) TupleSpace ts1 = server.createTupleSpace();
```

Note that most of the LogOp Linda primitives in Appendix A have as their first parameter a TupleSpaces object. The TupleSpaces object has two constructors: public TupleSpaces(AndLogicalOperator) and public TupleSpaces(OrLogicalOperator). These two constructors take an object that contains a list of tuple spaces. The logical operator used by this list of tuple spaces is represented by the class itself: either AndLogicalOperator or OrLogicalOperator. There is no constructor in the TupleSpaces class for NotLogicalOperator because an extra parameter specifying either `And` or `Or` operator would have been needed. Support for the `Not` operator is done through the LogOp class discussed next. For convenience, a helper class called LogOp consist of class methods for `And`, `Or` and `Not` operators.

These methods are called *and*, *or*, *andSetDiff*, and *orSetDiff* which are all overloaded each taking upto eight tuple spaces and each return a TupleSpaces object. LogOp's class method signatures are listed in Appendix B. An example of these classes is in use on lines 5-8 in Figure 5.2: A client process is sending an Integer object to tuple-spaces ts1, ts2, ts3 using the LogOp.and convenience method.

Tuple Class

In Figure 5.2, lines 5-7, a Tuple object is created and consists of an Integer object. Tuple objects are used to insert information into a particular tuple space or tuple spaces. The Tuple constructor is overloaded taking upto eight Java Objects. A convenience method called *add()* allows for more than eight objects in a tuple. When retrieving a Tuple from a getter primitive call, convenience methods exist to get the actual values of the Tuple object. The *get(int)* method is used to retrieve the i^{th} parameter of the Tuple object. Another convenience method in the Tuple object is *getTS()* which returns a TupleSpace object where the tuple came from. The Tuple class is listed in Appendix D. Lines 12-13 in Figure 5.2 illustrate the *get(int)* method of the Tuple class.

Template Class

The Template class is used as the second parameter of all the getter primitive methods. This class extends the Tuple class thus inheriting all public and protected data members and methods. One public method in the Template class called *addFormal(Object, IS_FORMAL)* is used to add formal objects to a template instance. The IS_FORMAL argument is used to distinguish between "java.lang.String" being a Formal and "java.lang.String" being an Actual. The code in Figure 5.2 on lines 9-10 illustrates usage of the Template class. The constructors of the Template class assume the value of each parameter is IS_FORMAL. The Template class is listed in Appendix E. In order for formal $field_i$ in the template to match the actual of $field_i$, the formal must consist of the package name and the class name.

Tuples Class

The Tuples class is a containment class consisting of lists of Tuple objects. The convenience methods are *getTuple(int)* and *getTS(int)*, which based on the length of the list of Tuple objects, retrieve the i^{th} Tuple or TupleSpace object, respectively. The Tuples class is listed in Appendix F and an example of its usage is in Figure 5.2 on lines 10-14. A Tuples object is returned from all LogOp Linda getter methods. Developers would not be creating instances of this class.

ActiveTuple Class

When a developer calls an `eval` method, the second parameter is an ActiveTuple object. The ActiveTuple has five constructors which take from one up to five ActiveObject objects. A convenience method called *add(ActiveObject)* is implemented if there are more than five ActiveObject objects. An ActiveObject is an interface that a developer implements on a class if the class is to be evaluated for use in an `eval` primitive. The ActiveObject interface has one method called *eval()* that returns an Object. Figure 5.3 is an example of the `eval` primitive using the ActiveTuple and ActiveObject classes. In this example there are two classes the developer writes, FutureDate and PastDate (Figures 5.4 and 5.5, respectively), that must implement the ActiveObject interface. When each *eval* method is complete, their return object creates the single Tuple object that will be inserted into the three tuple-spaces ts1, ts2 and ts3. The first parameter of this Tuple object is the result of the FutureDate class's return object; likewise, the second parameter of this Tuple object is the result of the PastDate class's return object. For clarification, the duration of each evaluated ActiveObject object does not determine the field order in the newly created Tuple object. The order of the fields in the new Tuple object are determined where each ActiveObject is added when the ActiveTuple object is created.

5.2.2 Server Design

When a client process executes a primitive, a Command object is sent over to the LogOp Linda server where the client process is connected. A Command object consist of the

LogOp Linda primitive, logical operator to be used, a list of tuple spaces, and either a Template or Tuple object. On the server process, a *ReadThread* object is waiting on the socket to receive a Command object coming from a client process. Once a Command is received off a socket, it is delegated to an object called TupleSpaceDecomposer.

Program 5.2 LogOp Linda out primitive using classes: LogOp, LogOpLindaServer, TupleSpace, Template, Tuples and Tuple

```
LogOpLindaServer server = LogOpLindaServer.connectLogOpLinda();

TupleSpace ts1 = server.createTupleSpace();
TupleSpace ts2 = server.createTupleSpace();
TupleSpace ts3 = server.createTupleSpace();

Integer anInteger = new Integer(1);
Tuple tuple = new Tuple(anInteger);
System.out.println((Integer)tuple.get(0));

server.out( LogOp.and ( ts1 , ts2 , ts3 ) , tuple );

Template template = new Template(``java.lang.String'',
                                 ''java.lang.String'');
Tuples tuples = server.in( LogOp.and ( ts1 , ts2 ) ,
                           template);

for( int index = 0; index < tuples.getCount(); index++ ) {
  tuple = tuples.getTuple(index);
  print(tuple.getTS() ,
        (String)tuple.get(0) ,
        (String)tuple.get(1) );
}
```

Program 5.3 LogOp Linda `eval` primitive

```
LogOpLinda server = LogOpLindaServer.connectLogOpLinda();

TupleSpace ts1 = server.createTupleSpace();
TupleSpace ts2 = server.createTupleSpace();
TupleSpace ts3 = server.createTupleSpace();

ActiveObject estStockPrices = new FutureDate(``10 days'');
ActiveObject pastStockPrices = new PastDate(``100 days'');

server.eval( LogOp.and ( ts1 , ts2 , ts3 ) ,
             new ActiveTuple( estStockPrices ,
                              pastStockPrices ) );
```

Program 5.4 ActiveTuple example #1

```
public class FutureDate
  implements ActiveObject
{
  private String timeStamp;

  public FutureDate(String timeStamp)
  {
    this.timeStamp = timeStamp;
  }

  public Object eval()
  {
    return calculateFutureStock(timeStamp);
  }
}
```

Program 5.5 ActiveTuple example #2

```
public class PastDate
  implements ActiveObject
{
  private String timeStamp;

  public PastDate(String timeStamp)
  {
    this.timeStamp = timeStamp;
  }

  public Object eval()
  {
     return calculatePastStock(timeStamp);
  }
}
```

The TupleSpaceDecomposer class takes the Command (known as the "original Command"), and by the list of tuple spaces, decomposes this original Command object into other Command objects. The new Command objects are created based on the physical location of the LogOp Linda server that created the tuple space. For example, consider a client process (connected to LogOp Linda server A) knows tuple-spaces ts_1, ts_2, ts_3, ts_4 and ts_5. Consider that tuple-spaces ts_1, ts_3, and ts_5 were created on LogOp Linda server B and tuple-spaces ts_2 and ts_4 were created on server C. The TupleSpaceDecomposer would create two new Commands: one for the group of tuple spaces created on LogOp Linda server B and another for the group of tuple spaces created on LogOp Linda server C. These new Command objects consist of the same information as the original Command object with the exception that they now consist of the grouped tuple spaces.

Once the TupleSpaceDecomposer determines the groups of new Commands, the original Command is placed into a hash table. The key in the hash table for this Command object is based on the IP address that the client process is executing on and the port number that the client is attached to the LogOp Linda server. The groups of the new Commands are then delegated to other LogOp Linda servers. This delegation is never any more than one LogOp Linda server away from the current LogOp Linda server because of how the

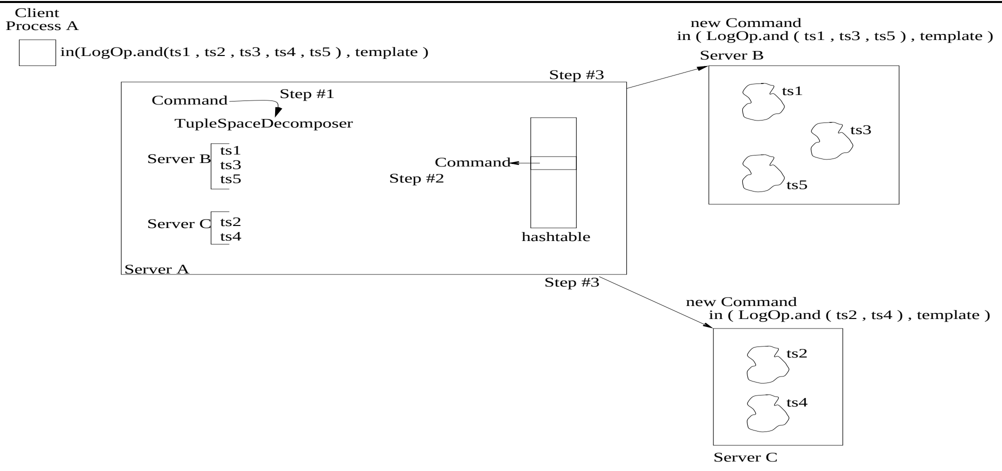
LogOp Linda servers are inter-connected via their socket communication.

Figure 5.3 illustrates receiving a Command from a client process, decomposing it (step #1), inserting it into the hash table (step #2), and delegating new Commands to other LogOp Linda servers (step #3). As in the previous example, this figure shows three LogOp Linda servers A, B and C. The client process is attached to LogOp Linda server A and is executing the `in` primitive

server.in(LogOp.and (ts_1, ts_2, ts_3, ts_4, ts_5) , `template`);

When a new Command object consisting of a group of tuple spaces is on the LogOp Linda server where all the tuple spaces in the group are created, this group of tuple spaces is then delegated to a thread that is responsible for a particular LogOp Linda primitive. If the group consists of one tuple space, then it is sent to a Linda thread supporting that particular primitive. Thus, if the original command is executing an `in` primitive, then the group of tuple spaces is put into a particular list so the *LogOpInThread* thread could process. Note the threads for each primitive are not continuously executing in a tight loop. A thread will execute when a Command object is inserted into its list.

Figure 5.3 LogOp Linda server receiving a Command object and processing it

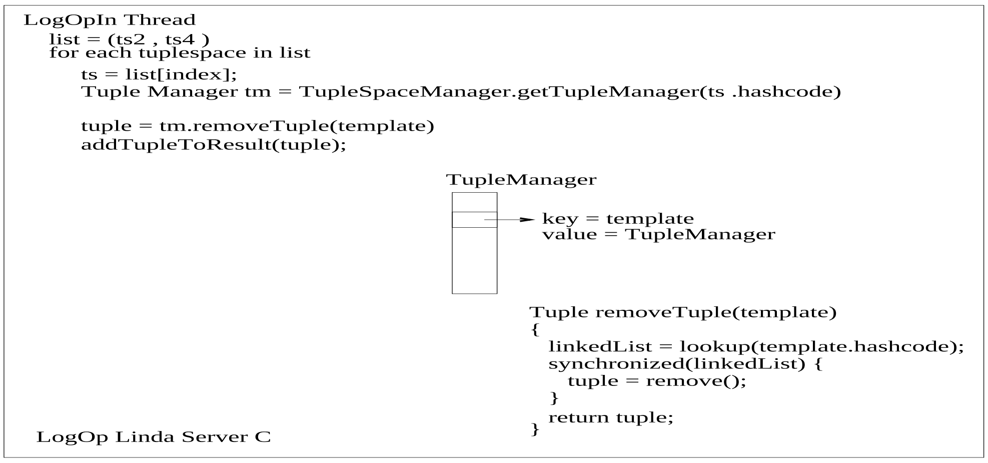


The `out` and `eval` primitives (*LogOpOutThread* and *LogOpEvalThread* setter threads) both insert tuples into tuple spaces. After this occurs, these setter threads notify the other getter threads (*LogOpInThread*, *LogOpInpThread*, *LogOpRdThread*, *LogOpRdpThread*, *InThread*, *InpThread*, *RdThread* and *RdpThread*) that a tuple has been placed into some tuple space. The terms "setter threads" and "getter threads" are known as "Primitive Threads" and are illustrated in Figure 5.1 as Linda Threads and LogOp Linda Threads.

When a primitive thread accesses a tuple space to retrieve or insert a tuple, two classes are involved: TupleSpaceManager and TupleManager. The TupleSpaceManager is a Singleton that manages what tuple spaces exist on this particular server. For efficient access, the TupleSpaceManager inserts new tuple spaces into a hash table. The key used for each tuple space is it's hash code. Since tuple spaces contain tuples, the value entry of the hash table is a TupleManager object.

The TupleManager is an object that manages Tuple objects. Inside the TupleManager class, a hash table is used to hold Tuple objects. Associated with each Tuple object is its template value. This template value is the key into the hash table. The value object of this key is a LinkedList object consisting of Tuple objects. The TupleManager has three public methods: *addTuple(Tuple)*, *removeTuple(Template)* and *readTuple(Template)*. The *addTuple()* is called by the *LogOpEvalThread*, *LogOpOutThread*, *EvalThread* and *OutThread* threads. The *removeTuple()* is called by the *LogOpInThread*, *LogOpInpThread*, *InThread* and *InpThread* threads. Finally, the *readTuple()* is called by the *LogOpRdThread*, *LogOpRdpThread*, *RdThread* and *RdpThread* threads. Because multiple client processes can be executing at once, the `out` and `eval` threads can both call the *addTuple()*; thus, this method is a critical section and is synchronized. Furthermore, if the Tuple object is going into a LinkedList object that exists, the LinkedList object is also synchronized because any of the other primitive threads can be accessing that same LinkedList object. Neither of the *removeTuple()* or *readTuple()* methods are synchronized because any number of threads are allowed into those methods at the same time. However, those methods access the LinkedList object based on the Template object passed into those methods. Hence access to the LinkedList object associated with a Template key is synchronized because only one thread can be accessing the LinkedList object. Synchronizing at this level allows multiple threads to call the *removeTuple()* method and access different LinkedList objects. Likewise, multiple threads can access the *readTuple()* method but not the same LinkedList object. This type of design and implementation facilitates greater concurrency in the LogOp Linda system executing on a multi-processor LogOp Linda system. Figure 5.4 illustrates the `in` primitive executing in LogOp Linda server C.

Figure 5.4 LogOp Linda server removing a tuple



```
LogOpIn Thread
   list = (ts2 , ts4 )
   for each tuplespace in list
       ts = list[index];
       Tuple Manager tm = TupleSpaceManager.getTupleManager(ts .hashcode)

       tuple = tm.removeTuple(template)
       addTupleToResult(tuple);
```

```
Tuple removeTuple(template)
{
  linkedList = lookup(template.hashcode);
  synchronized(linkedList) {
     tuple = remove();
  }
  return tuple;
}
```

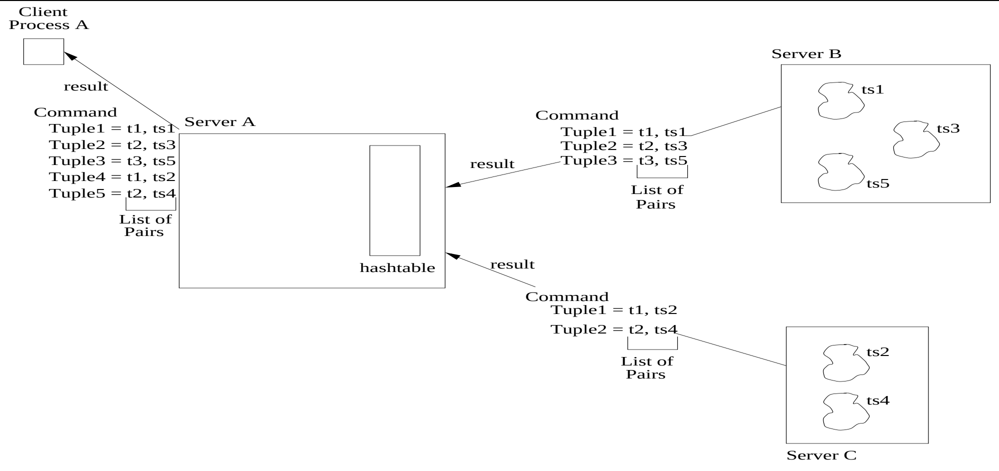
For the blocking primitives, as each Tuple object is found in the TupleManager, it is removed from (because of the `in` primitive) or copied (because of `rd` primitive) from the TupleManager and inserted into the original Command object as a pair object on that LogOp Linda server. Recall that a pair object consists of the Tuple object and the TupleSpace object where the Tuple object is found. When all Tuple objects are found on a particular LogOp Linda server, the original Command object is removed from the hash table and is sent back to the original calling system (either the client process or the LogOp Linda server). If the original calling system is the the client process, the client process retrieves a Tuples object (or Tuple object if it is a Linda primitive). It is up to the developer as what is to be done with the Tuples object. If the original calling system is a LogOp Linda server, then that Command object is inserted back into the original Command (the one from the client process). If all Tuple objects are found in the original Command, then the original Command with a list of pair objects is sent back to the client process. Figure 5.5 illustrates all Tuple objects being found on servers B and C sent back to server A. Once server A retrieves all resulting Tuple objects, the Command returns to the calling client process.

The above works the same for the nonblocking primitives `inp` and `rdp`. However, if there are no Tuple objects that match the Template object in all the specified tuple spaces, then the Command object containing an empty list is returned to the calling client process.

In the case of blocking primitives, if a Tuple is not found, the Command is placed into the back of its Thread's list for future processing. Thus the blocking is not occurring on the *removeTuple()* and *readTuple()* methods in the TupleManager class.

One question arises. When a Command object is sent from one LogOp Linda server back to the original LogOp Linda server, how does the original LogOp Linda server know this is a "response" Command? It cannot be determined by the number of tuples in the "response" Command object because no tuples may have been found (the client process originally sent a nonblocking primitive). Associated with each Command object is an identification number that is set back in the client process. When a Command object enters into a LogOp Linda server, its Command identification number is checked in the hash table on that LogOp Linda server. If the Command identification exists, the LogOp Linda server knows this is a response from another LogOp Linda server.

Figure 5.5 LogOp Linda server returning Tuple objects back to the calling client process



5.2.3 The `And` Operator

The `And` operator involves no processing on the client. The Command containing the tuple spaces and the `And` operator are sent to the LogOp Linda server. Once received on the server, the *ReadThread* sends it to the TupleSpaceDecomposer object. From there, processing of this Command object follows what is described in Section 5.2.2.

5.2.4 The `Or` Operator

The `Or` operator is implemented using a location-aware algorithm. A property called *OrLocale* in the properties file has a mask that determines which range of IP addresses are more closer than other IP addresses. An example of this property's value is 33.56.45.* which means all IP addresses in the range of 33.56.45.0 to 33.56.45.255 are considered more closer to an actual LogOp Linda server than other LogOp Linda servers. Since it is a property, each LogOp Linda server can have different ranges than other LogOp Linda servers in the LogOp Linda kernel. Recall associated with a tuple space is the IP address where the tuple space is stored.

Consider that process A knows of tuple spaces ts_1 through ts_4 and the closer tuple spaces are ts_2 and ts_4 and executes the following:

server.out(LogOp.or(ts_1, ts_2, ts_3, ts_4) , `tuple`);

When the LogOp Linda kernel receives this primitive, it will determine, through the location-aware algorithm, that the `tuple` is placed in tuple-spaces ts_2 and ts_4. This is how the other setter primitive, `eval`, works as well (after of course `eval`'s functions are evaluated).

The getter primitives, blocking and nonblocking, are more involved and both implementations will be described later. The following `in` primitive will be used and following the above description of which tuple spaces are closer

list = server.in(LogOp.or(ts_1, ts_2, ts_3, ts_4) , template);

In the LogOp Linda kernel (known as the "original server"), a list of closer tuple spaces will be determined using the location-aware algorithm. This new list of tuple spaces (ts_2

and ts_4) are then decomposed using the TupleSpaceDecomposer object and sent, via a Command object(s) marked as "TRY_ONCE", to their respected LogOp Linda servers. Once on those servers, the respected tuple spaces are checked only once and if a tuple matches the template, it is removed. The Command object is returned to the original server. Once all of these "TRY_ONCE" Command objects are returned, it will be determined if the collection of them contain at least one tuple. If so, the original server returns to the client process a list of pair objects containing tuples and tuple spaces where the tuples were retrieved (as described earlier). In the event that no "TRY_ONCE" Command consisted of a tuple, the original command consisting of the original list of tuple spaces (ts_1, ts_2, ts_3, and ts_4) are then decomposed and sent via a Command object(s) to their respected LogOp Linda server. Once at their respected servers, the Command blocks (if the `in` or `rd` primitive is used) waiting for a tuple that matches the template (for the non-blocking primitives, the tuple spaces are checked once and returned to the original server). Once a Command finds a tuple (termed the "initial tuple"), the Command object returns to the original server. Once on the original server, the original server sends out *KillOff* Command object(s) to all other LogOp Linda servers who received a Command object based on this `in` primitive executing to tell those Commands to immediately execute one more time and return to the original server (whether having a tuple or not). The term immediately implies the *LogOpInThread* is started and the Command objects in its list are executed. The only delay of this Command object not immediately executing is if the *LogOpInThread*'s list contains several Command objects. A tuple that is found and returned at this point is considered being inserted into its respected tuple space "at the same time" as the "original tuple" was being retrieved. Once all of the Command object(s) are returned to the original LogOp Linda server, the primitive returns a list of pair objects to the client process. This waiting and executing the "KillOff" Command allows one or more tuples to be returned to the client process.

The LogOp Linda implementation is flexible enough to allow a developer to implement a location-aware algorithm. In the properties file, a property called OrDeterminismClass specifies a class in which the developer implements to determine the tuple spaces to be used. The class specified must implement the OrDeterminism interface that has one method that gets passed the tuple spaces and the LogOp Linda primitive id that was executed. The primitive identification values are listed in Appendix C. An example of a class that implements the OrDeterminism interface is shown in Appendix G. Once the tuple spaces are determined by the developer's algorithm, the resulting tuple spaces are sent as a Command to the LogOp Linda server using the `And` operator. The Command follows the description of Section 5.2.2.

The advantage of allowing the developer to implement their own location-aware algorithm is for flexibility. Consider during a company's work hours that all LogOp Linda primitives should be sent to a certain set of tuple spaces and that after work hours the primitives should be sent to a different set of tuple spaces. The OrDeterminism interface allows for the LogOp Linda primitive ID values to be passed to the compute method. Hence, the time- and location-awareness, as just described, is also at the level of each primitive. Another advantage, from a object oriented approach, is the logic of choosing location is separated from the code of calling the LogOp Linda primitives.

5.2.5 The `Not` Operator

The `Not` operator is implemented using two class methods and two convenience class methods in the LogOp class. The class methods are *andSetDiff()* and *orSetDiff()* and both are overloaded passing from one upto eight tuple spaces. If eight is not enough, two convenience methods exist that pass an array of tuple spaces and are called createAndSetDiff(TupleSpace []) and createOrSetDiff(TupleSpace []). The result of all the methods discussed return a TupleSpaces object containing the actual tuple spaces that will be sent to the LogOp Linda server. The class methods throw one exception called EmptySetTupleSpaceException which occurs if the resulting TupleSpaces object contains no TupleSpace objects; hence an empty list of tuple spaces. Figure 5.6 illustrates a process using the *andSetDiff()* method.

Program 5.6 LogOp Linda *andSetDiff()* method

```
(1)  LogOpLinda server = LogOpLindaServer.connectLogOpLinda();

(2)  TupleSpace uts = server.getUTS();
(3)  TupleSpace ts1 =
           ((TupleSpace)(server.in(uts,
             new Template(``logoplinda.TupleSpace'')))).get(0);
(4)  TupleSpace ts2 = server.createTupleSpace();
(5)  TupleSpace ts3 =
           ((TupleSpace)(server.in(uts,
             new Template(``logoplinda.TupleSpace'')))).get(0);
(6)  TupleSpace ts4 = server.createTupleSpace();
(7)  TupleSpace ts5 = server.createTupleSpace();

(8)  server.out( LogOp.andSetDiff ( ts1, ts3 ) , aTuple );
     // the out actually goes to uts, ts2, ts4 and ts5
     // assuming ts1 and ts3 are not close.
```

5.2.6 Scopes Data Structure Comparison

Recall the necessary data structure required by the Scopes implementation and that this data structure is necessary between Linda primitive calls from the same client process due to the fact it contains the information that associates what Scopes belong to what Scopes. There is no data structure kept between LogOp Linda primitive calls from the same client process. There is, however, the Command object that is placed into a hash table. When the Command is complete, though, and a result is being returned to the client process, the Command object is removed from the hash table. Therefore, the hash table's contents are transient: As a primitive is executed, a value is inserted into the hash table; likewise, the value gets removed when the primitive is done. In addition, no set of tuple spaces remain logically combined when a LogOp Linda primitive is finished. In conclusion, Scopes does require a data structure to keep consistent what Scopes are combined where as LogOp Linda does not.

5.3 Deadlock

In the LINDA model, consider what happens when all processes, in a closed system, are accessing the same tuple space using a blocking primitive. Consider if all processes in LINDA block for a tuple that does not exist in the tuple space, then deadlock occurs. Until another process connects to the system and does an `out` of a tuple that matches a template by a blocked process, the system will remain deadlock.

Similar deadlocks occur in LOGOP LINDA. Whether all client processes are accessing several tuple spaces at once using a single blocking primitive, if there is no tuple that matches the template, then the system will remain in a deadlock state.

To solve this problem, a deadlock detection can be implemented to monitor such situations. The implementation is beyond the scope of this thesis due to significant research that must be done. See Chapter 7 for further information.

5.4 `Collect` and `Copy` Design

This section covers a design of the `collect` and `copy` primitives: The `collect` and `copy` are not implemented. Recall the difference between these primitives is the `copy` primitive makes copies of the matching tuples in the source tuple spaces whereas the `collect` primitive physically removes the matching tuples. Other than this difference, the design for both primitives is the same. Hence, the design will be with respect to both primitives and termed "bulk primitive" is this discussion.

A bulk primitive can be executed with a list of tuple spaces and a logical operator. The invocation of this primitive with a `Not` operator executes exactly as the implementation discussion in Section 5.2.5. The `Or` operator, too, executes exactly as the implementation is described in Section 5.2.4. Hence both of the operators are first resolved in the client process. After this is done, the primitive along with the lists of source and destination tuple spaces is sent over to the client process's LOGOP LINDA server. Once the primitive is received on the server, it is sent to the TupleSpaceDecomposer. Then, the groups of source tuple spaces are created. From here, the bulk primitive design is different than the other primitive implementations: the bulk primitives are executed in two phases. The first phase is copying or removing the tuples from all the source tuple spaces: a message from each LogOp Linda server is sent back to the original LogOp Linda server when this is done. Once all LogOp Linda servers have sent this message, another message is sent back to the participating servers to physically copy or move the tuples into the destination tuple spaces. Once all tuple spaces have received the tuples, a message containing the source tuple spaces and the number of tuples moved or copied is sent to the original LogOp Linda server. Once all LogOp Linda servers have done this, the bulk primitive returns to the client process the list of source tuple spaces and the number of tuples that were moved or copied.

It is important to note that retrieving the tuples from all the source tuple spaces must occur in parallel. For example, the following expresses *moving* two copies of the tuples from ts_1 into ts_2:

$$\text{resultList} = \texttt{collect}\ (\ \wedge\ (\ ts_1, ts_1)\ ,\ ts_2, \text{template});$$

The resulting list returned to the client process should contain two entries with each entry containing the tuple-space ts_1 reference. The number of tuples moved from ts_1 in each entry should be the same value x, where x could be 0 based on no tuples matching template. It is incorrect for entry #1 value of x to be five (five because there were five tuples matching template) and for entry #2 the value of x being 0 because the first execution of `collect` removed all the matching tuples.

5.5 Conclusion

Figure 5.6 illustrates how a setter command using the `And` operator flows through the LogOp Linda kernel starting from a client process going to the LogOp Linda server. The process box sends the tuple to the appropriate tuple spaces. Likewise, Figure 5.7 illustrates how a getter command using the `And` operator flows through the LogOp Linda kernel starting from a client process and going through the LogOp Linda server. Both figures have a diamond with "Send to Remote System". If the answer is yes, then the group of tuple spaces that are decomposed are on a particular remote server. If no, then the tuple spaces reside on this LogOp Linda server. The diamond question in Figure 5.7

that asks "Seen Command Already" is used to determine if this Command just read from the *ReadThread* is new or already seen before. If the answer is no, then this Command is from the client process. If yes, then this is a Command coming from another LogOp Linda server and must proceed to put these results back into the original Command.

Figures 5.8, 5.9 and 5.10 illustrates the setter and getter primitives flowing through the LogOp Linda kernel using the `Or` operator. The setter primitives in Figure 5.8 are simple to illustrate: When the server receives a Command with the `Or` operator, it applies the location-aware algorithm and then decomposes those location-aware tuple spaces to their appropriate LogOp Linda server. The getter primitives, as described in Section 5.2.4, are more involved and are illustrated in Figures 5.9 and 5.10.

Figure 5.6 Setter command, using the `And` operator, flowing through the LogOp Linda kernel

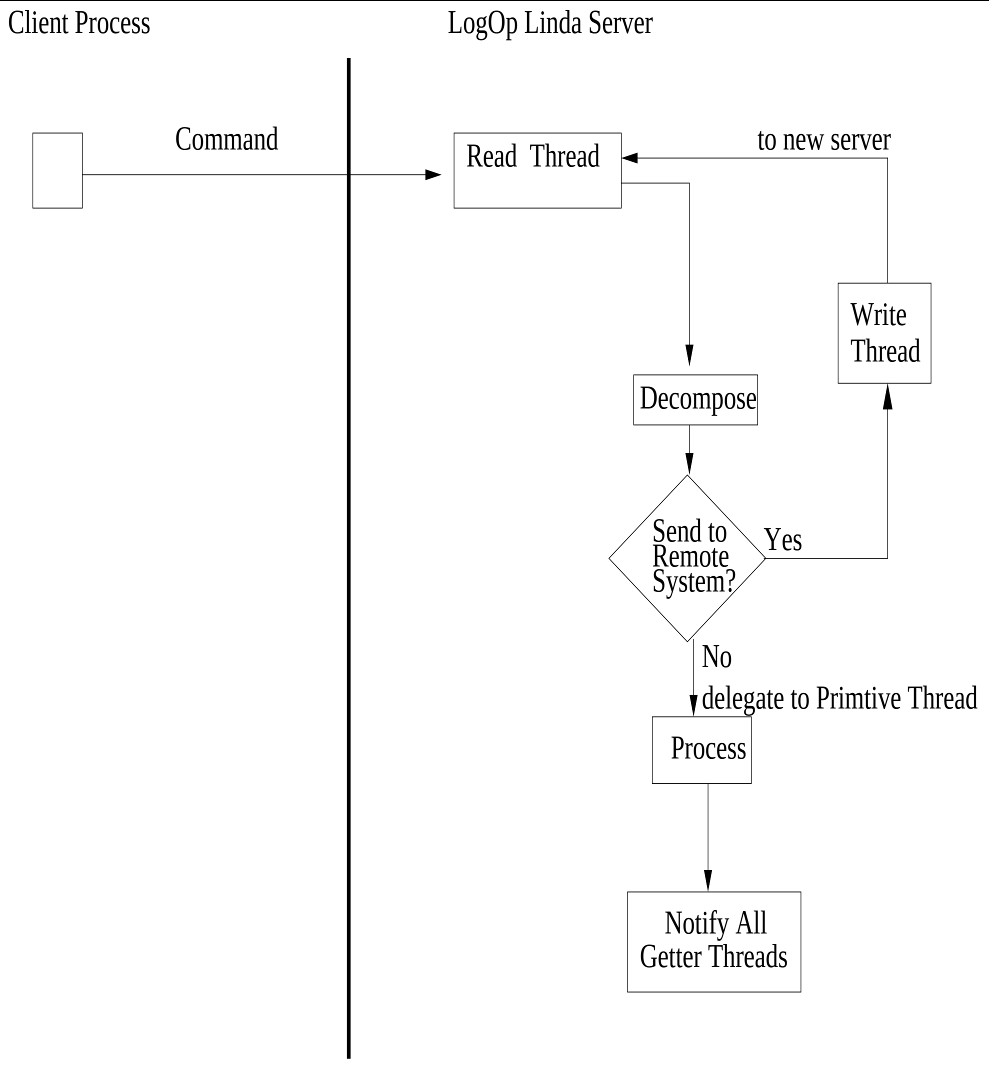


Figure 5.7 Getter command, using the `And` operator, flowing through the LogOp Linda kernel

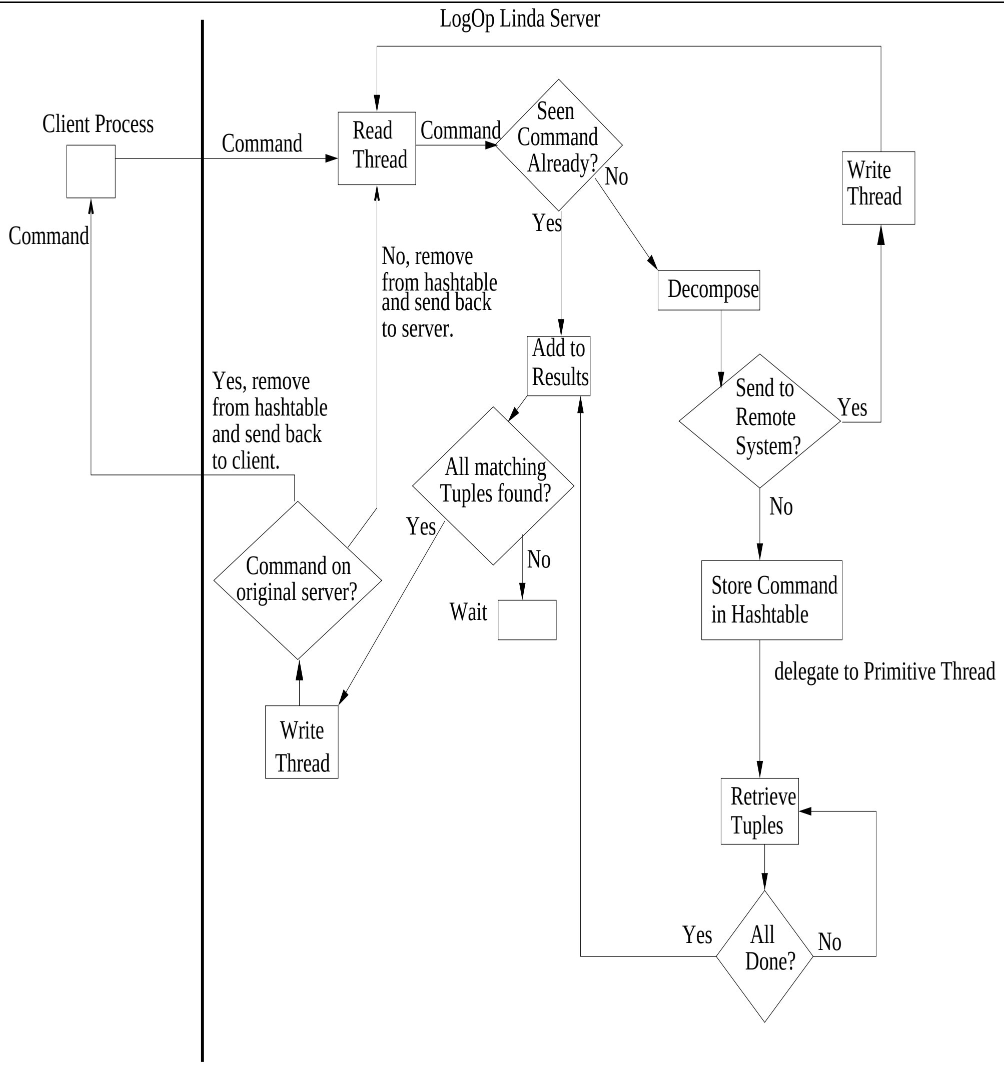

LogOp Linda Server
Client Process
Command
Command
Read Thread
Command
Seen Command Already?
No
Yes
Write Thread
Decompose
No, remove from hashtable and send back to server.
Add to Results
Yes, remove from hashtable and send back to client.
Send to Remote System?
Yes
All matching Tuples found?
Yes
No
No
Command on original server?
Wait
Store Command in Hashtable
delegate to Primitive Thread
Write Thread
Retrieve Tuples
Yes
All Done?
No

Figure 5.8 Setter command, using the `Or` operator, flowing through the LogOp Linda kernel

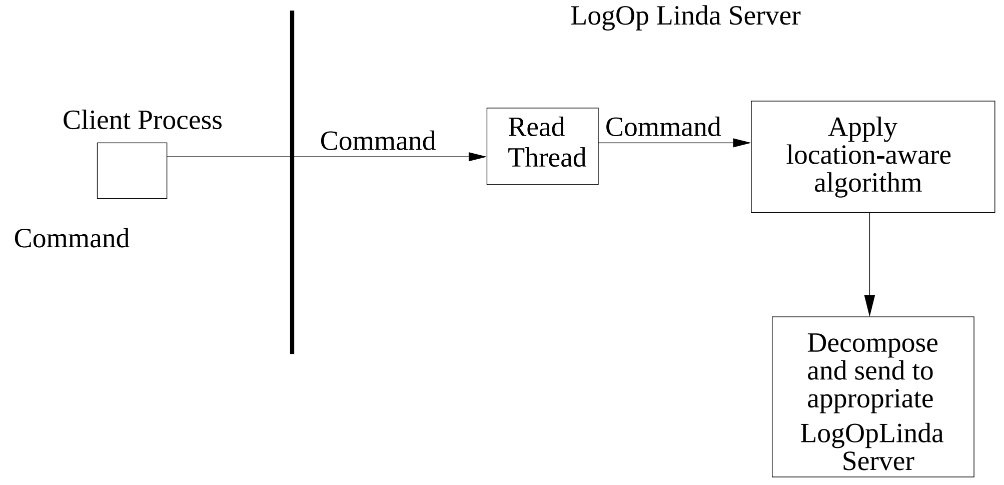

LogOp Linda Server
Client Process
Command
Read Thread
Command
Apply location-aware algorithm
Command
Decompose and send to appropriate LogOpLinda Server

Figure 5.9 Getter command, using the `Or` operator, flowing through the LogOp Linda kernel

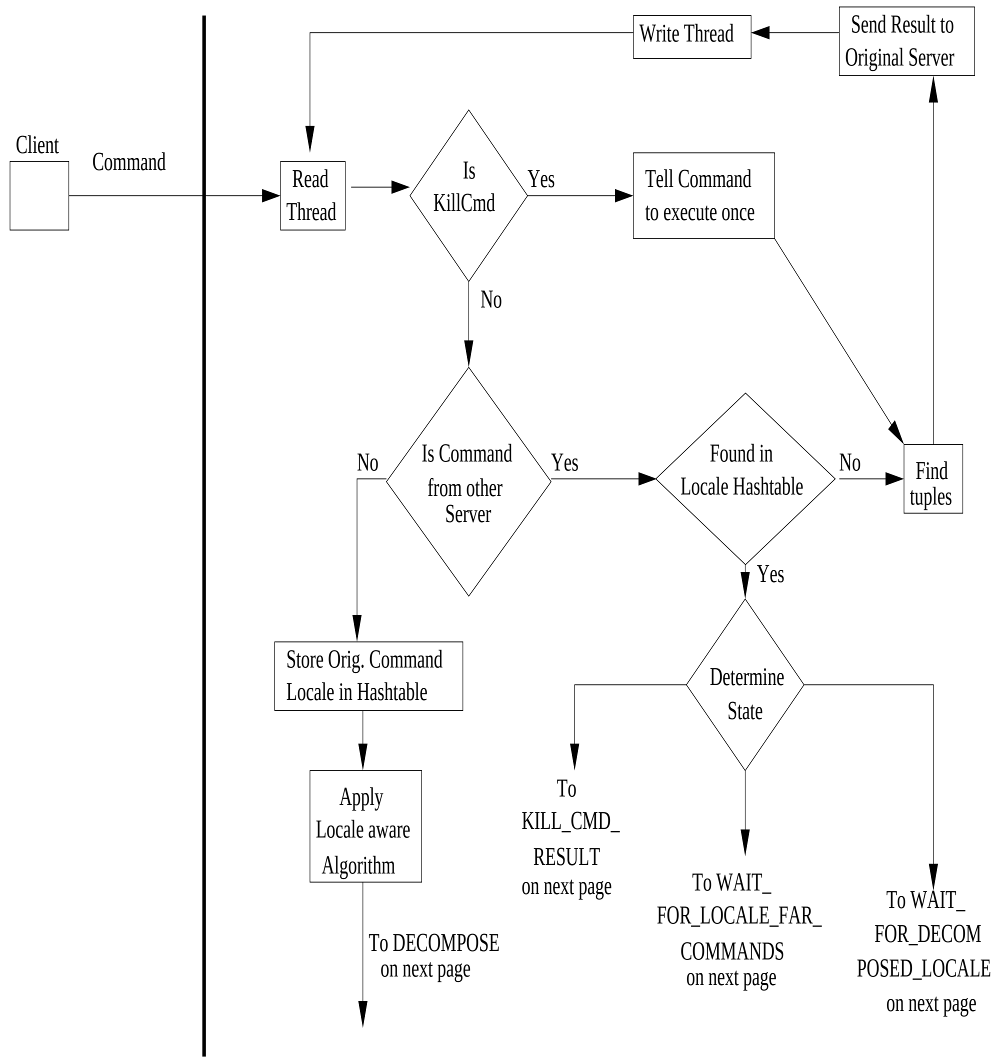
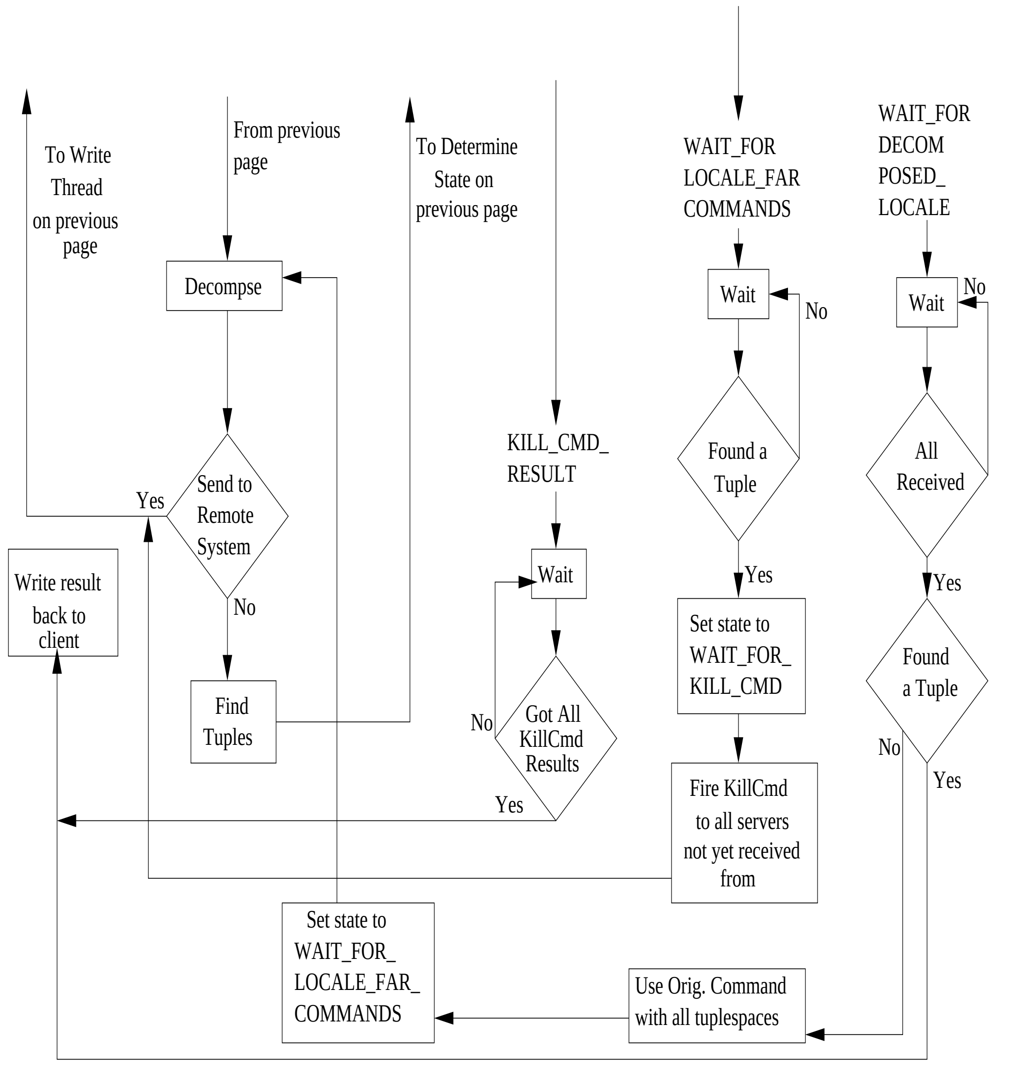

Write Thread
Send Result to Original Server
Client
Command
Read Thread
Is KillCmd
Yes
Tell Command to execute once
No
Is Command from other Server
No
Yes
Found in Locale Hashtable
No
Find tuples
Yes
Store Orig. Command Locale in Hashtable
Determine State
Apply Locale aware Algorithm
To KILL_CMD_ RESULT on next page
To WAIT_ FOR_LOCALE_FAR_ COMMANDS on next page
To WAIT_ FOR_DECOM POSED_LOCALE on next page
To DECOMPOSE on next page

Figure 5.10 Getter command, using the `Or` operator, flowing through the LogOp Linda kernel (continued)


To Write Thread on previous page
From previous page
To Determine State on previous page
WAIT_FOR LOCALE_FAR COMMANDS
WAIT_FOR DECOM POSED_ LOCALE
Decompse
Wait
No
Wait
No
KILL_CMD_ RESULT
Found a Tuple
All Received
Yes
Send to Remote System
Write result back to client
Wait
Yes
Yes
No
Set state to WAIT_FOR_ KILL_CMD
Found a Tuple
Find Tuples
No
Got All KillCmd Results
No
Yes
Yes
Fire KillCmd to all servers not yet received from
Set state to WAIT_FOR_ LOCALE_FAR_ COMMANDS
Use Orig. Command with all tuplespaces

Chapter 6

Empirical Results

This chapter covers several empirical results based on the single implementation (described in Chapter 5) of both LINDA and LOGOP LINDA. All tests were conducted on Sun Ultra 5 UNIX machines running Solaris 8 containing a single CPU with 128 megs of RAM. An additional Solaris 8 Unix box was used that contained four CPUs and 2 gigs of memory. The Java version used is JRE 1.3. The tests were conducted to see how efficient LOGOP LINDA and LINDA could access several tuple spaces.

Several external factors can influence the execution time of these experiments. The first is the Java Virtual Machine's (JVM) garbage collection (GC). During a testing phase, the point in time when and how long the GC would occur could influence the results of these tests. Specific hardware machines containing 128 megs of RAM were used to hold several tuple spaces. However, each tuple space contained only one tuple thus alleviating extraneous and time consuming garbage collecting. Another factor deals with the ObjectOutputStream class provided by Java [Mah01]. 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 versus 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 or an extensive amount of page swapping occurring on the UNIX boxes having 128 megs of RAM. The final influence is other executing processes external to the testing environment. For this, the tests were executed several times taking the average time.

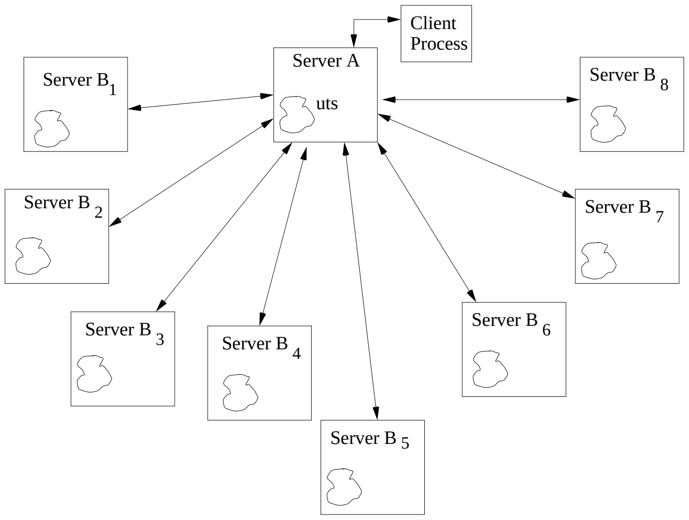
During these tests, between 2 and 8 UNIX boxes (servers B_1 through B_8 in Figure 6.1) with 128 megs of RAM were used as the LogOp Linda servers. Another server (A) is involved, for all tests, that the client process connects to. Server A has 2 gigs of memory and thus alleviated extensive garbage collection interruptions when retrieving several tuples. The UNIX machines B_1 - B_8 were located in the same lab while server A was in another lab. This setup decreased the network latency between systems.

Figure 6.1 shows the architecture setup for these tests. Note that it has 8 LogOp Linda servers B_1 - B_8 connecting to one server A. The client process is connected to server A and executes the `rd` primitive. The figure illustrates only one tuple space in each server B_1 - B_8, however these tests where executed using between 1 to 100 tuple spaces per server. The lines with arrows leaving server A represents the LogOp Linda primitive executed and the lines coming back from each B_i server represents the results of those primitives.

The tests were done with a single tuple in each tuple space to compare scalability and efficiency between LogOp Linda and Linda. Thus no significant amount of time is spent non-deterministically choosing a tuple. The tuple was inserted into each tuple space on each LogOp Linda servers (B_1 - B_8) by another client process whose duty was to create the tuple spaces on that LogOp Linda server B_1 - B_8, insert the single tuple and send those tuple space handles to the Universal Tuple Space residing on LogOp Linda server A. Afterwards the client process inserting the tuple exits.

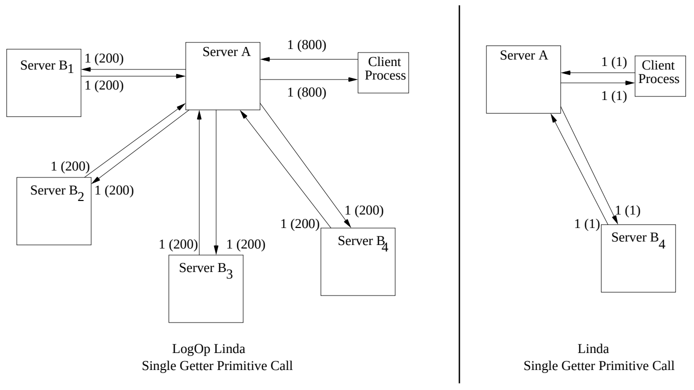
Tests involved each LogOp Linda server B_1 - B_8 containing between one tuple space and 100 tuple spaces. If one tuple space exists on each of the 8 servers, this meant that the client process executing Linda's `rd` primitive would have to make eight separate calls to retrieve eight tuples. In contrast, LogOp Linda's `rd` primitive would execute one call to the server to retrieve all eight tuples at once. In terms of scalability, when there are fifteen tuple spaces on each server, the client process executing LogOp Linda's `rd` primitive would again make one primitive call to the LogOp Linda server; where as, the client process executing Linda's `rd` primitive would execute 120 primitive calls to the LogOp Linda server. The number of inter-process communication between servers is actually less efficient in Linda than in LogOp Linda.

Figure 6.1 Testing environment



For example, once the single LogOp Linda primitive call reaches server A, the TupleSpaceDecomposer groups the tuple spaces by IP address (where each tuple space actually resides). Once the grouping is done, then a *single* call for each group is sent to the respected LogOp Linda server. This means, if there are four servers involved (B_1, B_2, B_3 and B_4) and each server contains two hundred tuple spaces, then at most there will be four individual calls: one to each server B_1, B_2, B_3 and B_4. This efficiency is based on the implementation; however it is the semantics of LogOp Linda that enables such an implementation. Thus the total number of calls between a client process executing a getter primitive and the five servers (A, B_1, B_2, B_3 and B_4) is: one from the client process to server A (1), one to send each call server A makes to B_1, B_2, B_3 and B_4 (4), one for each individual server B_1, B_2, B_3 and B_4 to reply back to server A (4), and one for the single call back to the client process (1). The total number of interprocess calls made is ten to retrieve eight hundred tuples. Since LINDA, BONITA, and SCOPES only support a single tuple space at a time, the total number of calls made is four times the number of tuples: 3200 calls. Figure 6.2 exemplifies this phenomenon. Shown in this figure on the left side is LOGOP LINDA making the ten calls and on the right side LINDA making a single call. The numbers in parenthesis illustrates the count of tuple spaces that will be accessed with each call (represented as arrows). Although the picture does not show a tuple space, it is assumed that two hundred tuple spaces exist on each server B_1, B_2, B_3 and B_4. LOGOP LINDA is not a simple wrapper around LINDA primitives. Most importantly, it takes a group of tuple spaces specified in a single primitive and sends that one primitive to the server. The server decomposes and sends those new grouped tuple spaces to their respected servers in parallel. This proof of scalability is important to the LOGOP LINDA theory and actual implementation. As will be shown in the charts, LOGOP LINDA scales much better and is more efficient than LINDA with greater number of tuple spaces.

Figure 6.2 A single primitive by both LOGOP LINDA and LINDA. The number in parenthesis represents the number of tuple spaces sent and the number of tuples received



6.1 Test Suite A

This suite of tests focuses on the distributed implementation of LOGOP LINDA and as described in Chapter 5 uses the implementation of both LINDA and LOGOP LINDA to allow a fair comparison. The tests involve from two to eight servers containing a range of one to four hundred tuple spaces on a particular server. To avoid memory exhaustion, a *reset()* method is called after every five thousand object writes. This section has three charts summarizing the results. Each chart consists of testing on two, four, and 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 6.3, the last x-axis value is 800 and thus executing the LogOp Linda `rd` primitive retrieving 800 tuples distributed across two servers took less than a half a second; likewise, the Linda `rd` primitive took over two and half seconds.

In the first chart, since hardware resources are limited on the two servers (B_1 and B_2) involved in Figure 6.3, a spike begins to occur on the last hash mark. Note that LogOp Linda's average time in milliseconds is much less than Linda's as the number of tuple spaces accessed per primitive increases. In fact, with the number of servers at four (B_1, B_2, B_3 and B_4) and eight (B_1 - B_8) in Figures 6.4 and 6.5, respectively, LogOp Linda performance is more efficient compared to Linda whose scalability is drastically getting worst.

There is evidence of Java's serialization at lesser number of tuple spaces that affects the performance of the LogOp Linda's primitives. Recall that LogOp Linda is sending out several tuple spaces per primitive call. Apparently if the number of tuple spaces is small, Linda is efficient in retrieving tuples from the same number of tuple spaces in more calls than it is for LogOp Linda primitives to retrieve the same number of tuples with less calls. Contrary to this, LogOp Linda's performance on a large number of tuple spaces surpasses that of Linda's.

Further evidence of Java's serialization side effecting results is when there is a total of 240 tuple spaces involved in the system. The single difference between all three figures is the number of servers involved. This thesis does not investigate the serialization algorithm that Java provides; however, there can be more tests executed. For this suite of tests the *reset()* is called every five thousand object writes, further tests can illustrate that continuous serialization does effect the execution. These test results are in Section 6.2

Appendix K contains tables of numbers representing these charts.

Figure 6.3 Average time in milliseconds using two servers. Number of tuples retrieved are across the x-axis (one tuple per tuple space). LogOp Linda and Linda calling *reset()* every 5000 object writes

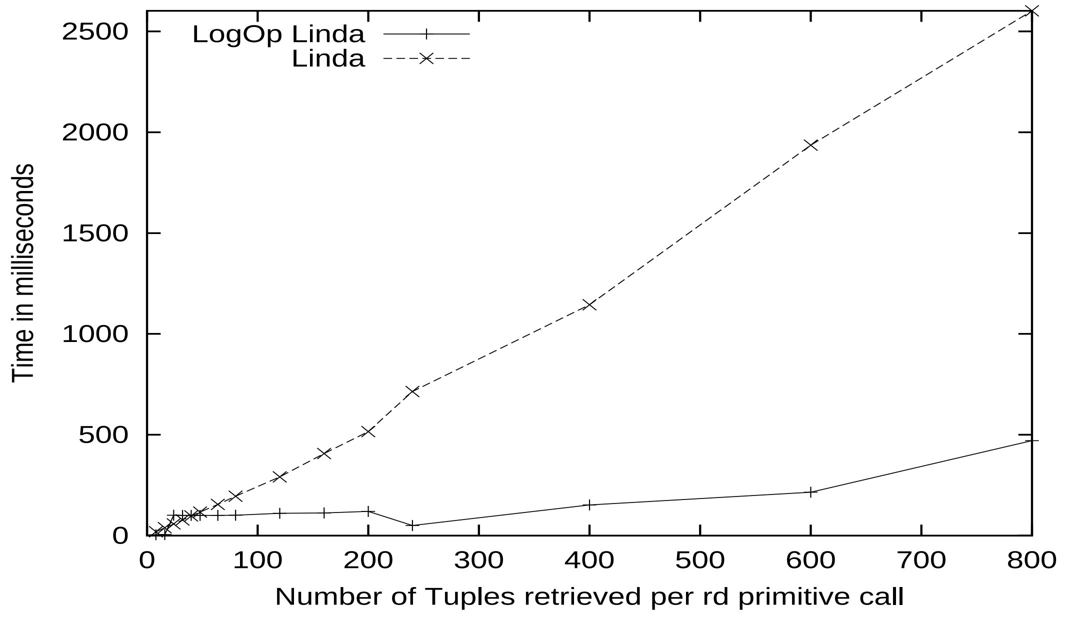


Figure 6.4 Average time in milliseconds using four servers. Number of tuples retrieved are across the x-axis (one tuple per tuple space). LogOp Linda and Linda calling *reset()* every 5000 object writes

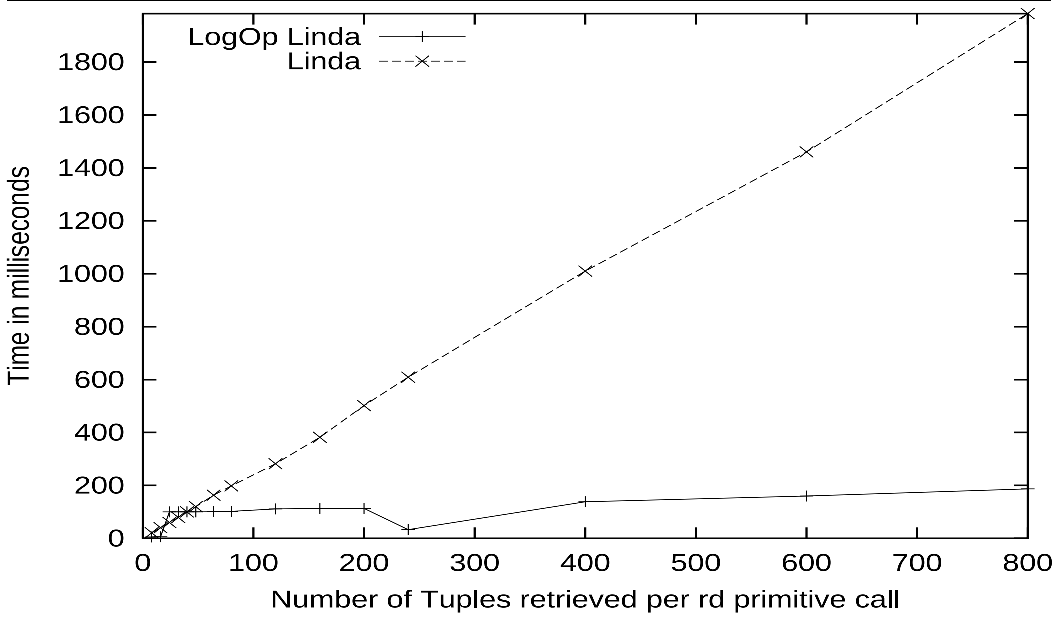
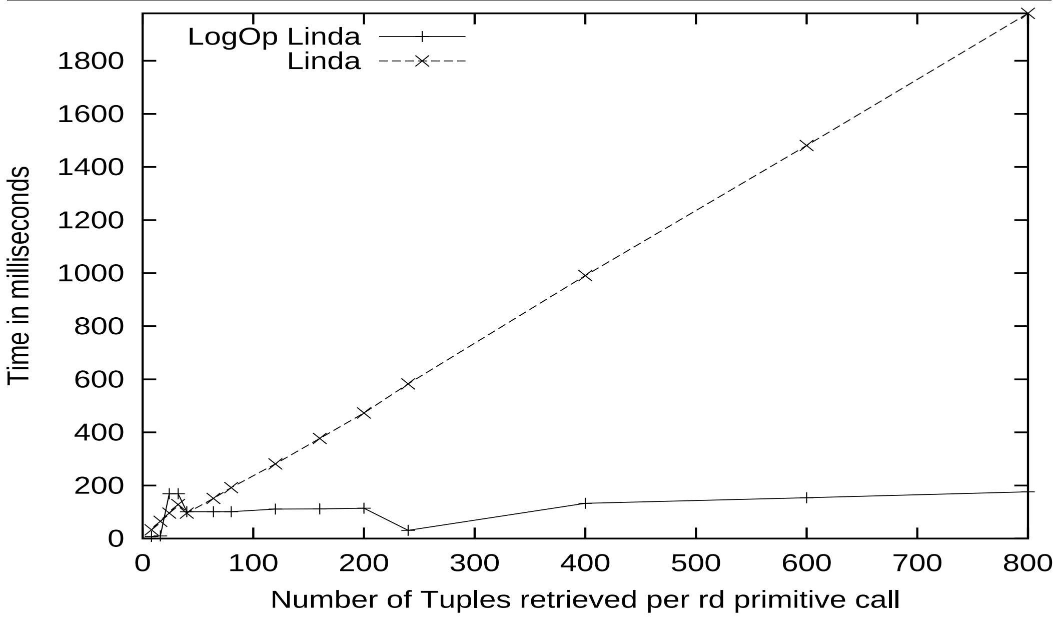


Figure 6.5 Average time in milliseconds using eight servers. Number of tuples retrieved are across the x-axis (one tuple per tuple space) LogOp Linda and Linda calling *reset()* every 5000 object writes



6.2 Test Suite B

Due to serialization and non-serialization occurring in the JVM, this suite tests the same scenarios from Section 6.1. However, the difference is after each Java object is written a *reset()* method is called to clear out any stored object IDs.

For example, consider in Figure 6.6 when eight hundred tuple spaces are accessed. The time taken to retrieve 800 tuples is only 563 milliseconds which is a little above the value shown in Figure 6.3 for LogOp Linda.

Figures 6.6 through 6.8 also include the Linda time values from Figures 6.3 through 6.5. Recall from Figures 6.6 through 6.8 that *reset()* was not called after each Java object was written to the socket (a much quicker method). Note in these figures that LogOp Linda is still more efficient than Linda.

Appendix K contains tables of numbers representing these charts.

Figure 6.6 LogOp Linda time in milliseconds using two servers. Number of tuples retrieved are across the x-axis (one tuple per tuple space). LogOp Linda calling *reset()* and Linda not calling *reset()*

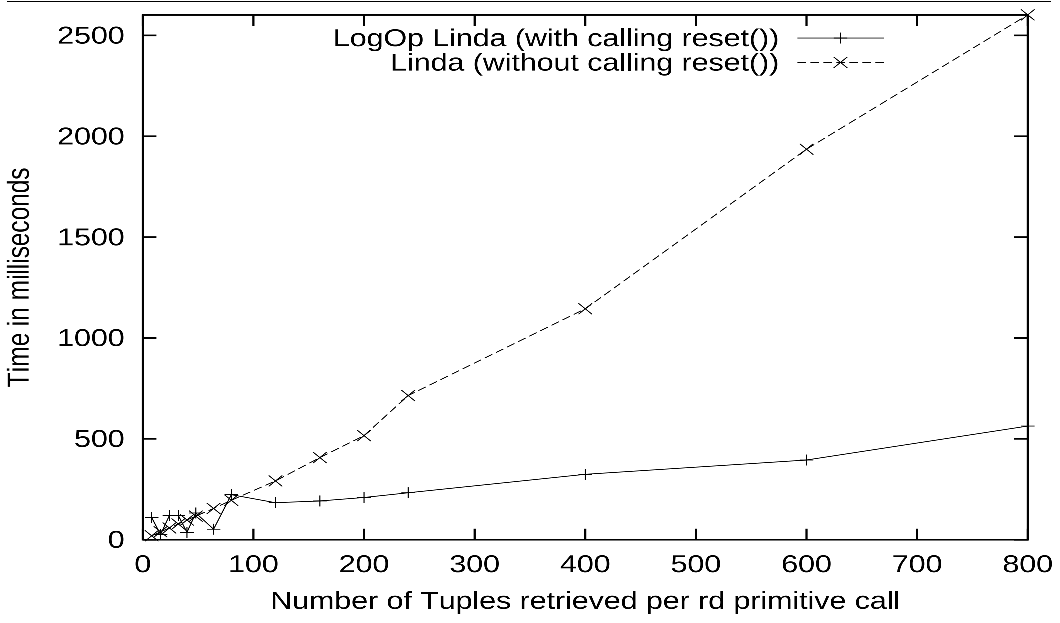


Figure 6.7 LogOp Linda time in milliseconds using four servers. Number of tuples retrieved are across the x-axis (one tuple per tuple space). LogOp Linda calling *reset()* and Linda not calling *reset()*

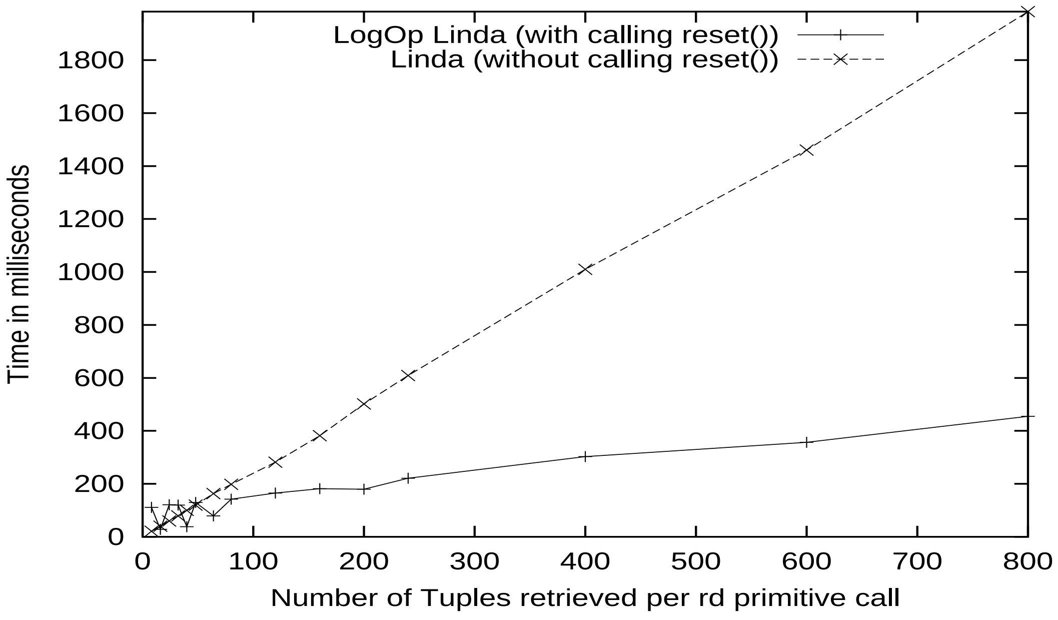
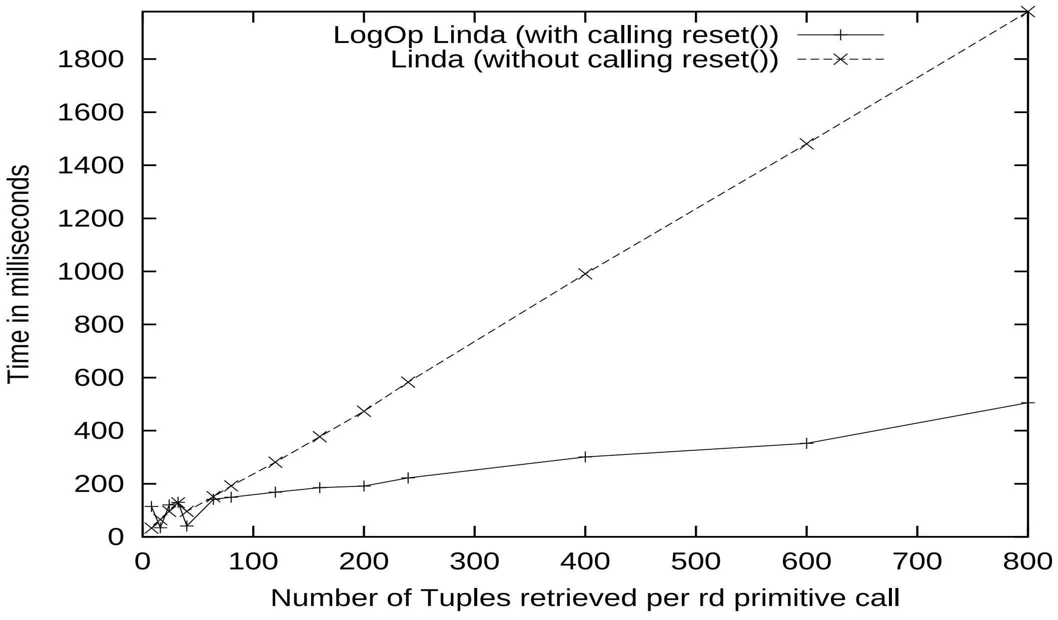


Figure 6.8 LogOp Linda time in milliseconds using eight servers. Number of tuples retrieved are across the x-axis (one tuple per tuple space). LogOp Linda calling *reset()* and Linda not calling *reset()*



Chapter 7

Conclusion and Future Research

7.1 Conclusion

This thesis is based on the semantics and efficiency of other coordination models. An extension of these models creates a scalable, efficient and expressive model called LogOp Linda. The ability to access several tuple spaces in parallel applies itself to several genres of software development: to name a few, network monitoring and security. Several areas of future research needs to still be done with LogOp Linda and are touched upon below.

7.2 Future Research

LogOp Linda is a highly expressive, efficient and scalable model for the coordination of distributed processes. This section discusses some future work for further enhancements to LogOp Linda's semantics.

7.2.1 Deadlock Implementation

As the problem was described in Sections 4.2.1 and 5.3 , deadlocking can occur in the LogOp Linda model. This section addresses a possible solution for the implementation described LogOp Linda in this thesis in Chapter 5. An additional Command object (possibly called DeadlockCmd) needs to be implemented for determining any deadlocks. A deadlock thread could be executed on each server monitoring the LogOp Linda and Linda Command objects stored in the hash table depicted in step #2 of Figure 5.3. Periodically, the state of each hash table on each server could be sent to a tuple space (internally known as the deadlock tuple space), only known by the LogOp Linda kernel, so analysis could be done to determine where a potential deadlock is occurring[1]. Once a deadlock is found, a (new) RestartCmd command object could be sent to each of the servers containing the original LogOp Linda Commands causing the deadlock to restart themselves and allow only one of the LogOp Linda Commands to proceed and complete, thus resulting in only one less LogOp Linda to be blocked. This scenario only describes the case where two LogOp Linda Commands are deadlocked. This could be a solution where n LogOp Linda Commands are deadlocked; however, only further research in the deadlock detection/recovery could determine the best algorithmic solution.

Note that each LogOp Linda server delivering the state of its hash table would execute the following `out` primitive:

`out` (deadlockTupleSpace, hashtableStateObject)

while the LogOp Linda server that is going to process each of the hash table states would be blocked using the `in` primitive (where n represents the number of LogOp Linda servers read from the LogOp.properties file):

$$\text{tuples} = \texttt{in}\ (\ \text{LogOp.and}\ (deadlockTupleSpace_1\ ,\ \ldots\ ,\ deadlockTupleSpace_n\ )\ ,\ \text{?hashtableStateObject}\ );$$

In this description, note that the tuple-spaces

[1]The algorithm for determining where a potential deadlock is occurring is beyond the scope of this thesis.

$$deadlockTupleSpace_1 \text{ , ... , } deadlockTupleSpace_n$$

are the same tuple space because all LogOp Linda servers would send the hash table state to the same tuple space.

7.2.2 Other Logical Operators

Other logical operators exist that were not researched are: `XOR`, `XNOR`, `NOR`, `NAND`. Consider Figure 7.1 consisting of the above operators and the `And` operator using the `in` primitive. Recall that the `in` primitive will only unblock using an `And` logical operator if all of the listed tuple spaces contain a tuple matching the given template. Consider the following primitives and the respected logical operator used

1. result = `in` (`xor` (ts_1, ts_2) , template)

2. result = `in` (`xnor` (ts_1, ts_2) , template)

3. result = `in` (`nor` (ts_1, ts_2) , template)

4. result = `in` (`nand` (ts_1, ts_2) , template)

and their respected truth tables shown in Figure 7.1. These truth tables illustrate when the `in` would unblock and return to the client process. However, what tuples would be returned when the primitive unblocks? Consider the truth table for the `NOR` in Figure 7.1. According to the unblock column, the primitive would unblock only if *none* in row #1 (recall that a zero in the ts_1 and ts_2 states a matching tuple does not exist and that a one means that a matching tuple exist) of the tuple-spaces ts_1 and ts_2 contained a matching tuple. This might possibly imply that a tuple *not matching* the template would be removed from each tuple space and returned to the client process.

Consider the truth table for the `XOR` operator and that the primitive should unblock only in rows #2 and #3. Does this mean for row #2 (and similarly row #3) that the tuple that is *matched* in tuple-space ts_1 and a tuple that *does not* match a tuple in tuple-space ts_2 should be removed? Another question is, what happens in ts_2 for row #3 if there are only tuples that match the template? Furthermore, since the `in` primitive in Linda only unblocks when a tuple matching the template exists: Does this contradict what the `in` primitive would do in the case of `XOR`? What type of coordination patterns do these types of logical operators allow? These questions and others for the other LogOp Linda's primitives imply further research beyond the scope of this thesis should be done.

The `NOR` and `NAND` are indeed different than the LogOp Linda's `Not` combined with either `Or`, or `And` operator. In LogOp Linda, the semantics for the `Not` first apply a set-difference algorithm and then either the `And` or `Or` operator is applied. The semantics for the `NOR` and `NAND` the reverse as follows:

`NOR:` is an `Or` followed by a `Not`

`NAND:` is an `And` followed by a `Not`

and would require further research beyond the scope of this thesis to determine what type of expressiveness and coordination patterns could be applied.

Table 7.1 `And`, `XOR`, `XNOR`, `NOR` and `NAND` operators using an `in` primitive

`And`	ts_1	ts_2	unblock
#1	0	0	0
#2	0	1	0
#3	1	0	0
#4	1	1	1

`XOR`	ts_1	ts_2	unblock
#1	0	0	0
#2	0	1	1
#3	1	0	1
#4	1	1	0

`XNOR`	ts_1	ts_2	unblock
#1	0	0	1
#2	0	1	0
#3	1	0	0
#4	1	1	1

`NOR`	ts_1	ts_2	unblock
#1	0	0	1
#2	0	1	0
#3	1	0	0
#4	1	1	0

`NAND`	ts_1	ts_2	unblock
#1	0	0	1
#2	0	1	1
#3	1	0	1
#4	1	1	0

7.2.3 Bonita Version of LogOp Linda

Currently, the LogOp Linda primitives are synchronous. Their setter methods are efficient due to the ability of using logical operators to combine the necessary tuple spaces. Bonita's `dispatch` methods are efficient due to their asynchronous semantics. However, recalling Properties 3.2.1 and 4.5.1, Bonita is still inefficient when dealing with multiple tuple spaces with the same tuple or template. It could be proposed that by combining Bonita's asynchronous primitives with logical operators, the total time taken would be $T_{StartUp}$ for the setter primitives and for the getter primitives take $T_{Check} + T_{Block} + T_{Extract}$ and are shown in Properties 7.2.1 and 7.2.2.

Property 7.2.1 (Combination of BONITA and LOGOP LINDA models setter primitives.) *Let S be a set of tuple spaces ts_1, ts_2, ... , ts_n. Then $\forall$ ts_i, where i = 1..n, $\exists$ a single* `dispatch` *primitive call. Therefore, the time taken for n* `dispatch` *calls is ($T_{StartUp}$).*

Property 7.2.2 (Combination of BONITA and LOGOP LINDA models getter primitives.) *Let S be a set of tuple spaces ts_1, ts_2, ... , ts_n. Then $\forall$ ts_i, where i = 1..n, $\exists$ a single* `dispatch` *primitive call. Therefore, the time taken for n* `dispatch` *calls is (T_{Check} + T_{Block} + $T_{Extract}$).*

7.2.4 LogOp Linda for Tuples and Templates

Currently, LOGOP LINDA uses logical operators to combine tuple spaces during primitive calls. While it is efficient for inserting the same tuple into a set of tuple spaces, it is inefficient when taking a set of tuples and using the `out` or `eval` primitive to insert into a set of tuple spaces. This same argument of inefficiency can be stated when using a getter primitive using a particular set of templates. How expressive would each extension of LOGOP LINDA's primitives be if logical operator were applied to `tuples` and templates? Figures 7.1 and 7.2 outline the proposal of efficiency. The `and` operator used in Figures 7.1 and 7.2 could be exchanged for the `Or` operator. The `Not` operator inherently contributes to the expressive power.

Figure 7.1 LogOp Linda efficiency

LogOp Linda on tuple spaces	Expressiveness and Efficiency for the number of Primitive calls

`out` ($\wedge$ (ts_1, ... , ts_n) , $tuple_1$);
`out` ($\wedge$ (ts_1, ... , ts_n) , $tuple_2$);
...
`out` ($\wedge$ (ts_1, ... , ts_n) , $tuple_n$);
	n

`in` ($\wedge$ (ts_1, ... , ts_n) , $template_1$);
`in` ($\wedge$ (ts_1, ... , ts_n) , $template_2$);
...
`in` ($\wedge$ (ts_1, ... , ts_n) , $template_n$);
	n

`rd` ($\wedge$ (ts_1, ... , ts_n) , $template_1$);
`rd` ($\wedge$ (ts_1, ... , ts_n) , $template_2$);
...
`rd` ($\wedge$ (ts_1, ... , ts_n) , $template_n$);
	n

Figure 7.2 logical operators for tuples and templates

Logical operator on tuple spaces, tuples and templates	Expressiveness and Efficiency for the number of Primitive calls
`out` ($\wedge$ (ts_1... ts_n) , $\wedge$ ($tuple_1$, ... , $tuple_n$));	1
`in` ($\wedge$ (ts_1, ... , ts_n) , $\wedge$ ($template_1$, ... , $template_n$));	1
`rd` ($\wedge$ (ts_1, ... , ts_n) , $\wedge$ ($template_1$, ... , $template_n$));	1

One concept with using the `Not` operator on a template is to choose a tuple from a tuple space that does not match the given template. For example, if the two templates (*template*$_1$ and *template*$_2$) are combined using the `And` and `Not` operators, the resulting templates used could allow the LogOp Linda kernel to choose a tuple that does not match *template*$_1$ and *template*$_2$. Further research in this particular area needs to be done to define the semantics of the `and`, `or` and `not` operators for tuples and templates.

In addition to the efficiency of the number of primitive calls, the Tuple Space Decomposer discussed in Section 5 would be more efficient. Currently in LogOp Linda, if *tuple*$_1$... *tuple*$_n$ are all sent to tuple-spaces ts_1... ts_m, the set of those tuple spaces are decomposed n times (based on n tuples). Again, if LogOp Linda could incorporate logical operators for tuples and templates, the Tuple Space Decomposer would execute once and achieve better parallelism.

7.2.5 LogOp Linda and Event-driven Models

Recall that models supporting Event Driven semantics are based on a notification occurring when a tuple that matches a template is being inserted into a particular tuple space. Thus, if events occur on several tuple spaces having the same event listener, it could be more correct for a single event to be called back to the client process containing a list of tuples and their tuple spaces. For example, registering an event for ts_1, ts_2 and ts_3 to occur could be expressed using an `And` operator:

registerEvent(`and` (ts_1, ts_2, ts_3) , template)

Several different implementations of the semantics can be achieved using event driven notification if LogOp Linda supported it. In the above statement, an event can be generated back to the registering client process if all three tuple spaces *already contain* a tuple matching a template. This event can continue to occur until all of the tuple spaces do not contain the template.

Another implementation could be after the event is registered, a client process receives an event notification when each tuple space receives a new tuple. Consider the following using the `Or` operator:

registerEvent(`or` (ts_1, ts_2, ts_3) , template)

From a location-aware perspective, events can be fired based on tuple spaces closer to the client process. For example, if ts_1 and ts_3 are closer, then the event is only registered for those two tuple spaces.

Once the semantics are defined for supporting an event driven capability in LogOp Linda, an advantage that may occur using logical operators is a single event notification is generated for a group of tuples. Contrast this with the semantics of JavaSpaces and TSpaces. Recall that their semantics only supported a client process receiving one tuple and tuple space at a time. Thus, if two tuples trigger an event, two serial calls will be made back to the calling client process.

7.2.6 The `Or` Operator and Nondeterminism

One question of concern is with the `Or` operator and nondeterminism: Does location awareness break nondeterminism? Further research needs to be done. As a start, consider mobile and stationary devices. If a device is not mobile, then the location-aware algorithm would keep choosing the same tuple space(s) continuously. However if the device is mobile, then this may improve performance. In other words, if it is designed to travel around and gather information based on closer (for speed purposes) tuple spaces, then there is a chance that at point B the device would access tuple-space b' which is now closer, but at point A tuple-space b' is too far away to be accessed.

7.2.7 Logical Operator Evaluator

Currently simple logical expressions have been defined in LogOp Linda. Future research needs to be investigated for more complex expressions that combine logical operators. For example, given the following:

$$\text{list} = \texttt{in} \; (\; \vee \; (\; \wedge \; (\; ts_1, \, ts_2) \; , \, ts_3) \; , \, \text{template})$$

Obviously the above expresses "remove a tuple from both ts_1 and ts_2 or remove a tuple from ts_3 that matches template" and return to the calling process. What tuple spaces should be accessed with this expression? Recall that the `in` primitive blocks until all tuple spaces contain a tuple matching template. Since the `Or` operator is used, a decision could be determining the "closer" (location-aware) tuple spaces. However if ts_1 is the only "close" tuple space, why should LINDA choose the expression $\wedge$ (ts_1, ts_2) over the more simpler ts_3? Aside from location-awareness, should the choice be made based on which tuple spaces physically have a tuple that matches template? If so, this implies that all LOGOP LINDA servers must continuously update all other LOGOP LINDA servers their contents. This type of updating could be expensive to maintain. Or, if this updating is not done, then the primitive would physically go to each of the specified tuple spaces a check for a tuple, lock the tuple and return that the tuple space has a marked tuple. When all results come back from checking the tuple spaces, there is still a decision that must be done in order to choose which tuple spaces will remove the tuple. This implies another trip to each of the tuple spaces to either remove the tuple or unlock the tuple if it is not to be removed.

Regarding LOGOP LINDA's blocking primitives, creating complex expressions can lead to the satisfiability problem. The satisfiability problem is concerned with determining whether there is an assignment of truth values associated with an expression that makes the expression true [Coo71]. Before determining if an expression is satisfiable, the expression must be transformed to conjunctive normal form. An expression in conjunctive normal form consists of the following format

$$u_1 \wedge u_2 \wedge \ \ldots \wedge u_n$$

where each u_i is a clause. A clause is a well-formed expression consisting of disjunctive variables or the negation of variables. The term variables here represents the tuple spaces in LOGOP LINDA. The negation of variables for LOGOP LINDA is simply applying a set difference algorithm as previously discussed. The above LOGOP LINDA execution of the `in` primitive can be viewed as a satisfiability problem. In essence, the following expression must return true in order for the primitive to unblock

$$(\ (\ ts_1 \wedge ts_2) \vee ts_3)$$

However this expression needs to be transformed into conjunctive normal form by distributing the $\vee$ operator over $\wedge$ operator as follows:

$$(\ ts_1 \vee ts_3) \wedge (\ ts_2 \vee ts_3)$$

Now that the expression is in conjunctive normal form, it can be determined what combination of tuple spaces makes it satisfiable or not. In order to determine this, the tuple spaces must be placed into a truth table shown in Figure 7.2. From this table, the results on rows #2 - #4 will unblock the primitive returning the list of tuple spaces and the tuples to the calling process. Note though, the solution of satisfiability is based on checking the results of each of the rows in both tables. In essence, the total number of rows checked is 2^n where n is the number of tuple spaces involved. The complexity of this problem is exponential implying that the satisfiability problem is in *NP* complete class of problems[2]. Due to this complexity, evaluating complex expressions can be very expensive in time.

Table 7.2 Truth table for tuple-spaces ts_1, ts_2 and ts_3

	ts_1	ts_3	$(ts_1 \vee ts_3)$
#1	0	0	0
#2	0	1	1
#3	1	0	1
#4	1	1	1

	ts_2	ts_3	$(ts_2 \vee ts_3)$
#1	0	0	0
#2	0	1	1
#3	1	0	1
#4	1	1	1

[2]The proof of this is beyond the scope of this thesis but can be found in [Sud88].

Appendix A

LogOpLindaServer class

```
public interface AbstractServer
{

 /**
  * disconnect
  *
  */
  public void disconnect();

}// AbstractServer

public interface MTS
  extends AbstractServer
{

  /**
  * createTupleSpace
  *
  * @param dbgName a value of type 'String'
  * @return a value of type 'TupleSpace'
  */
  public TupleSpace createTupleSpace(String dbgName);


  /**
  * getUTS
  *
  * @return a value of type 'TupleSpace'
  */
  public TupleSpace getUTS();

}// MTS

public interface LogOpLinda
```

```
  extends MTS
{
  /**
  * in
  *
  * @param tss a value of type 'TupleSpaces'
  * @param template a value of type 'Template'
  * @return a value of type 'Tuples'
  */
  public Tuples in(TupleSpaces tss, Template template);

  /**
  * inp
  *
  * @param tss a value of type 'TupleSpaces'
  * @param template a value of type 'Template'
  * @return a value of type 'Tuples'
  */
  public Tuples inp(TupleSpaces tss, Template template);

  /**
  * rd
  *
  * @param tss a value of type 'TupleSpaces'
  * @param template a value of type 'Template'
  * @return a value of type 'Tuples'
  */
  public Tuples rd(TupleSpaces tss, Template template);

  /**
  * rdp
  *
  * @param tss a value of type 'TupleSpaces'
  * @param template a value of type 'Template'
  * @return a value of type 'Tuples'
  */
  public Tuples rdp(TupleSpaces tss, Template template);

  /**
  * out
  *
  * @param tss a value of type 'TupleSpaces'
  * @param tuple a value of type 'Tuple'
  */
  public void out(TupleSpaces tss, Tuple tuple);

  /**
  * eval
  *
  * @param tss a value of type 'TupleSpaces'
  * @param activeTuple a value of type 'ActiveTuple'
  * @exception InvalidReturnObjectForEvalException if an error occurs
  */
  public void eval(TupleSpaces tss, ActiveTuple activeTuple)
    throws InvalidReturnObjectForEvalException;

}// LogOpLinda

public interface Linda
  extends MTS
{

  /**
  * in
  *
  * @param ts a value of type 'TupleSpace'
  * @param tuple a value of type 'Template'
  * @return a value of type 'Tuple'
  */
  public Tuple in(TupleSpace ts, Template tuple);

  /**
  * inp
  *
  * @param ts a value of type 'TupleSpace'
  * @param tuple a value of type 'Template'
  * @return a value of type 'Tuple'
  */
  public Tuple inp(TupleSpace ts, Template tuple);

  /**
  * rd
  *
  * @param ts a value of type 'TupleSpace'
  * @param tuple a value of type 'Template'
  * @return a value of type 'Tuple'
  */
  public Tuple rd(TupleSpace ts, Template tuple);

  /**
  * rdp
  *
  * @param ts a value of type 'TupleSpace'
  * @param tuple a value of type 'Template'
  * @return a value of type 'Tuple'
  */
  public Tuple rdp(TupleSpace ts, Template tuple);

  /**
  * out
  *
  * @param ts a value of type 'TupleSpace'
  * @param tuple a value of type 'Tuple'
  */
  public void out(TupleSpace ts, Tuple tuple);

  /**
  * eval
  *
  * @param ts a value of type 'TupleSpace'
  * @param activeTuple a value of type 'ActiveTuple'
  * @exception InvalidReturnObjectForEvalException if an error occurs
  */
  public void eval(TupleSpace ts, ActiveTuple activeTuple)
    throws InvalidReturnObjectForEvalException;

}// Linda
```

Appendix B

LogOp class

```
public class LogOp
{
  /**
  * and
  *
  * @param ts1 a value of type 'TupleSpace'
  * @param ts2 a value of type 'TupleSpace'
  * @return a value of type 'TupleSpaces'
  */
  public static TupleSpaces and(TupleSpace ts1, TupleSpace ts2)
  {
    return createAnd(ts1,ts2,null,null,null,null,null,null);
  }

  /**
  * and
  *
  * @param ts1 a value of type 'TupleSpace'
  * @param ts2 a value of type 'TupleSpace'
  * @param ts3 a value of type 'TupleSpace'
  * @return a value of type 'TupleSpaces'
  */
  public static TupleSpaces and(TupleSpace ts1, TupleSpace ts2,
                                TupleSpace ts3)
  {
    return createAnd(ts1,ts2,ts3,null,null,null,null,null);
  }

  /**
  * and
  *
  * @param ts1 a value of type 'TupleSpace'
  * @param ts2 a value of type 'TupleSpace'
* @param ts3 a value of type 'TupleSpace'
* @param ts4 a value of type 'TupleSpace'
* @return a value of type 'TupleSpaces'
*/
public static TupleSpaces and(TupleSpace ts1, TupleSpace ts2,
                              TupleSpace ts3, TupleSpace ts4)
{
  return createAnd(ts1,ts2,ts3,ts4,null,null,null,null);
}

/**
* and
*
* @param ts1 a value of type 'TupleSpace'
* @param ts2 a value of type 'TupleSpace'
* @param ts3 a value of type 'TupleSpace'
* @param ts4 a value of type 'TupleSpace'
* @param ts5 a value of type 'TupleSpace'
* @return a value of type 'TupleSpaces'
*/
public static TupleSpaces and(TupleSpace ts1, TupleSpace ts2,
                              TupleSpace ts3, TupleSpace ts4,
                              TupleSpace ts5)
{
  return createAnd(ts1,ts2,ts3,ts4,ts5,null,null,null);
}

/**
* and
*
* @param ts1 a value of type 'TupleSpace'
* @param ts2 a value of type 'TupleSpace'
* @param ts3 a value of type 'TupleSpace'
* @param ts4 a value of type 'TupleSpace'
* @param ts5 a value of type 'TupleSpace'
* @param ts6 a value of type 'TupleSpace'
* @return a value of type 'TupleSpaces'
*/
public static TupleSpaces and(TupleSpace ts1, TupleSpace ts2,
                              TupleSpace ts3, TupleSpace ts4,
                              TupleSpace ts5, TupleSpace ts6)
{
  return createAnd(ts1,ts2,ts3,ts4,ts5,ts6,null,null);
}

/**
* and
*
* @param ts1 a value of type 'TupleSpace'
* @param ts2 a value of type 'TupleSpace'
* @param ts3 a value of type 'TupleSpace'
* @param ts4 a value of type 'TupleSpace'
* @param ts5 a value of type 'TupleSpace'
* @param ts6 a value of type 'TupleSpace'
* @param ts7 a value of type 'TupleSpace'
* @return a value of type 'TupleSpaces'
*/
public static TupleSpaces and(TupleSpace ts1, TupleSpace ts2,
                              TupleSpace ts3, TupleSpace ts4,
                              TupleSpace ts5, TupleSpace ts6,
                              TupleSpace ts7)
{
  return createAnd(ts1,ts2,ts3,ts4,ts5,ts6,ts7,null);
}

/**
* and
*
* @param ts1 a value of type 'TupleSpace'
* @param ts2 a value of type 'TupleSpace'
* @param ts3 a value of type 'TupleSpace'
* @param ts4 a value of type 'TupleSpace'
* @param ts5 a value of type 'TupleSpace'
* @param ts6 a value of type 'TupleSpace'
* @param ts7 a value of type 'TupleSpace'
* @param ts8 a value of type 'TupleSpace'
* @return a value of type 'TupleSpaces'
*/
public static TupleSpaces and(TupleSpace ts1, TupleSpace ts2,
                              TupleSpace ts3, TupleSpace ts4,
                              TupleSpace ts5, TupleSpace ts6,
                              TupleSpace ts7, TupleSpace ts8)
{
  return createAnd(ts1,ts2,ts3,ts4,ts5,ts6,ts7,ts8);
}


/**
* or
*
* @param ts1 a value of type 'TupleSpace'
* @param ts2 a value of type 'TupleSpace'
* @return a value of type 'TupleSpaces'
*/
public static TupleSpaces or(TupleSpace ts1, TupleSpace ts2)
{
  return createOr(ts1,ts2,null,null,null,null,null,null);
}
```

```
/**
* or
*
* @param ts1 a value of type 'TupleSpace'
* @param ts2 a value of type 'TupleSpace'
* @param ts3 a value of type 'TupleSpace'
* @return a value of type 'TupleSpaces'
*/
public static TupleSpaces or(TupleSpace ts1, TupleSpace ts2,
                             TupleSpace ts3)
{
  return createOr(ts1,ts2,ts3,null,null,null,null,null);
}

/**
* or
*
* @param ts1 a value of type 'TupleSpace'
* @param ts2 a value of type 'TupleSpace'
* @param ts3 a value of type 'TupleSpace'
* @param ts4 a value of type 'TupleSpace'
* @return a value of type 'TupleSpaces'
*/
public static TupleSpaces or(TupleSpace ts1, TupleSpace ts2,
                             TupleSpace ts3, TupleSpace ts4)
{
  return createOr(ts1,ts2,ts3,ts4,null,null,null,null);
}

/**
* or
*
* @param ts1 a value of type 'TupleSpace'
* @param ts2 a value of type 'TupleSpace'
* @param ts3 a value of type 'TupleSpace'
* @param ts4 a value of type 'TupleSpace'
* @param ts5 a value of type 'TupleSpace'
* @return a value of type 'TupleSpaces'
*/
public static TupleSpaces or(TupleSpace ts1, TupleSpace ts2,
                             TupleSpace ts3, TupleSpace ts4,
                             TupleSpace ts5)
{
  return createOr(ts1,ts2,ts3,ts4,ts5,null,null,null);
}

/**
* or
```

```
*
* @param ts1 a value of type 'TupleSpace'
* @param ts2 a value of type 'TupleSpace'
* @param ts3 a value of type 'TupleSpace'
* @param ts4 a value of type 'TupleSpace'
* @param ts5 a value of type 'TupleSpace'
* @param ts6 a value of type 'TupleSpace'
* @return a value of type 'TupleSpaces'
*/
public static TupleSpaces or(TupleSpace ts1, TupleSpace ts2,
                             TupleSpace ts3, TupleSpace ts4,
                             TupleSpace ts5, TupleSpace ts6)
{
  return createOr(ts1,ts2,ts3,ts4,ts5,ts6,null,null);
}

/**
* or
*
* @param ts1 a value of type 'TupleSpace'
* @param ts2 a value of type 'TupleSpace'
* @param ts3 a value of type 'TupleSpace'
* @param ts4 a value of type 'TupleSpace'
* @param ts5 a value of type 'TupleSpace'
* @param ts6 a value of type 'TupleSpace'
* @param ts7 a value of type 'TupleSpace'
* @return a value of type 'TupleSpaces'
*/
public static TupleSpaces or(TupleSpace ts1, TupleSpace ts2,
                             TupleSpace ts3, TupleSpace ts4,
                             TupleSpace ts5, TupleSpace ts6,
                             TupleSpace ts7)
{
  return createOr(ts1,ts2,ts3,ts4,ts5,ts6,ts7,null);
}

/**
* or
*
* @param ts1 a value of type 'TupleSpace'
* @param ts2 a value of type 'TupleSpace'
* @param ts3 a value of type 'TupleSpace'
* @param ts4 a value of type 'TupleSpace'
* @param ts5 a value of type 'TupleSpace'
* @param ts6 a value of type 'TupleSpace'
* @param ts7 a value of type 'TupleSpace'
* @param ts8 a value of type 'TupleSpace'
* @return a value of type 'TupleSpaces'
*/
```

```
public static TupleSpaces or(TupleSpace ts1, TupleSpace ts2,
                             TupleSpace ts3, TupleSpace ts4,
                             TupleSpace ts5, TupleSpace ts6,
                             TupleSpace ts7, TupleSpace ts8)
{
  return createOr(ts1,ts2,ts3,ts4,ts5,ts6,ts7,ts8);
}




private static TupleSpaces createOr(TupleSpace ts1, TupleSpace ts2,
                                    TupleSpace ts3, TupleSpace ts4,
                                    TupleSpace ts5, TupleSpace ts6,
                                    TupleSpace ts7, TupleSpace ts8)
{
  OrLogicalOperator logicalOperator = new OrLogicalOperator();

  logicalOperator.add(ts1,ts2);

  if(ts3 != null) logicalOperator.add(ts3);
  if(ts4 != null) logicalOperator.add(ts4);
  if(ts5 != null) logicalOperator.add(ts5);
  if(ts6 != null) logicalOperator.add(ts6);
  if(ts7 != null) logicalOperator.add(ts7);
  if(ts8 != null) logicalOperator.add(ts8);

  return new TupleSpaces(logicalOperator);
}






private static TupleSpaces createAnd(TupleSpace ts1, TupleSpace ts2,
                                     TupleSpace ts3, TupleSpace ts4,
                                     TupleSpace ts5, TupleSpace ts6,
                                     TupleSpace ts7, TupleSpace ts8)
{
  AndLogicalOperator logicalOperator = new AndLogicalOperator();

  logicalOperator.add(ts1,ts2);

  if(ts3 != null) logicalOperator.add(ts3);
  if(ts4 != null) logicalOperator.add(ts4);
  if(ts5 != null) logicalOperator.add(ts5);
  if(ts6 != null) logicalOperator.add(ts6);
```

```
    if(ts7 != null) logicalOperator.add(ts7);
    if(ts8 != null) logicalOperator.add(ts8);

    return new TupleSpaces(logicalOperator);
  }




  private static TupleSpace []setDifference(Vector tupleSpaceDomain,
                                            TupleSpace []inTupleSpace)
  {
    TupleSpace []outTupleSpace=null;

    Vector tmpOutTupleSpace = new Vector();

    for(int index=0;index<tupleSpaceDomain.size();index++) {

      TupleSpace tmp = (TupleSpace)tupleSpaceDomain.elementAt(index);


      if(tmp != null) {
        boolean found = false;
        int index2 = 0;
        while((found == false) && (index2 < inTupleSpace.length)) {
          found = (tmp.equals(inTupleSpace[index2]));
          index2++;
        }

        if(found == false) {
          tmpOutTupleSpace.add(tmp);
        }
      }
    }

    outTupleSpace = new TupleSpace[tmpOutTupleSpace.size()];
    for(int index=0;index<tmpOutTupleSpace.size();index++) {
      outTupleSpace[index] = (TupleSpace)tmpOutTupleSpace.elementAt(index);
    }

    return outTupleSpace;
  }
```

```
/**
* andSetDiff
*
* @param ts1 a value of type 'TupleSpace'
* @return a value of type 'TupleSpaces'
* @exception EmptySetTupleSpaceException if an error occurs
*/
public static TupleSpaces andSetDiff(TupleSpace ts1)
  throws EmptySetTupleSpaceException
{
  return createAndSetDiff(ts1,null,null,null,null,null,null,null);
}

/**
* andSetDiff
*
* @param ts1 a value of type 'TupleSpace'
* @param ts2 a value of type 'TupleSpace'
* @return a value of type 'TupleSpaces'
* @exception EmptySetTupleSpaceException if an error occurs
*/
public static TupleSpaces andSetDiff(TupleSpace ts1, TupleSpace ts2)
  throws EmptySetTupleSpaceException
{
  return createAndSetDiff(ts1,ts2,null,null,null,null,null,null);
}

/**
* andSetDiff
*
* @param ts1 a value of type 'TupleSpace'
* @param ts2 a value of type 'TupleSpace'
* @param ts3 a value of type 'TupleSpace'
* @return a value of type 'TupleSpaces'
* @exception EmptySetTupleSpaceException if an error occurs
*/
public static TupleSpaces andSetDiff(TupleSpace ts1, TupleSpace ts2,
                                     TupleSpace ts3)
  throws EmptySetTupleSpaceException
{
  return createAndSetDiff(ts1,ts2,ts3,null,null,null,null,null);
}

/**
* andSetDiff
*
* @param ts1 a value of type 'TupleSpace'
* @param ts2 a value of type 'TupleSpace'
* @param ts3 a value of type 'TupleSpace'
* @param ts4 a value of type 'TupleSpace'
* @return a value of type 'TupleSpaces'
* @exception EmptySetTupleSpaceException if an error occurs
*/
public static TupleSpaces andSetDiff(TupleSpace ts1, TupleSpace ts2,
                                     TupleSpace ts3, TupleSpace ts4)
  throws EmptySetTupleSpaceException
{
  return createAndSetDiff(ts1,ts2,ts3,ts4,null,null,null,null);
}

/**
* andSetDiff
*
* @param ts1 a value of type 'TupleSpace'
* @param ts2 a value of type 'TupleSpace'
* @param ts3 a value of type 'TupleSpace'
* @param ts4 a value of type 'TupleSpace'
* @param ts5 a value of type 'TupleSpace'
* @return a value of type 'TupleSpaces'
* @exception EmptySetTupleSpaceException if an error occurs
*/
public static TupleSpaces andSetDiff(TupleSpace ts1, TupleSpace ts2,
                                     TupleSpace ts3, TupleSpace ts4,
                                     TupleSpace ts5)
  throws EmptySetTupleSpaceException
{
  return createAndSetDiff(ts1,ts2,ts3,ts4,ts5,null,null,null);
}

/**
* andSetDiff
*
* @param ts1 a value of type 'TupleSpace'
* @param ts2 a value of type 'TupleSpace'
* @param ts3 a value of type 'TupleSpace'
* @param ts4 a value of type 'TupleSpace'
* @param ts5 a value of type 'TupleSpace'
* @param ts6 a value of type 'TupleSpace'
* @return a value of type 'TupleSpaces'
* @exception EmptySetTupleSpaceException if an error occurs
*/
public static TupleSpaces andSetDiff(TupleSpace ts1, TupleSpace ts2,
                                     TupleSpace ts3, TupleSpace ts4,
                                     TupleSpace ts5, TupleSpace ts6)
  throws EmptySetTupleSpaceException
{
  return createAndSetDiff(ts1,ts2,ts3,ts4,ts5,ts6,null,null);
}

/**
* andSetDiff
*
* @param ts1 a value of type 'TupleSpace'
* @param ts2 a value of type 'TupleSpace'
* @param ts3 a value of type 'TupleSpace'
* @param ts4 a value of type 'TupleSpace'
* @param ts5 a value of type 'TupleSpace'
* @param ts6 a value of type 'TupleSpace'
* @param ts7 a value of type 'TupleSpace'
* @return a value of type 'TupleSpaces'
* @exception EmptySetTupleSpaceException if an error occurs
*/
public static TupleSpaces andSetDiff(TupleSpace ts1, TupleSpace ts2,
                                     TupleSpace ts3, TupleSpace ts4,
                                     TupleSpace ts5, TupleSpace ts6,
                                     TupleSpace ts7)
  throws EmptySetTupleSpaceException
{
  return createAndSetDiff(ts1,ts2,ts3,ts4,ts5,ts6,ts7,null);
}

/**
* andSetDiff
*
* @param ts1 a value of type 'TupleSpace'
* @param ts2 a value of type 'TupleSpace'
* @param ts3 a value of type 'TupleSpace'
* @param ts4 a value of type 'TupleSpace'
* @param ts5 a value of type 'TupleSpace'
* @param ts6 a value of type 'TupleSpace'
* @param ts7 a value of type 'TupleSpace'
* @param ts8 a value of type 'TupleSpace'
* @return a value of type 'TupleSpaces'
* @exception EmptySetTupleSpaceException if an error occurs
*/
public static TupleSpaces andSetDiff(TupleSpace ts1, TupleSpace ts2,
                                     TupleSpace ts3, TupleSpace ts4,
                                     TupleSpace ts5, TupleSpace ts6,
                                     TupleSpace ts7, TupleSpace ts8)
  throws EmptySetTupleSpaceException
{
  return createAndSetDiff(ts1,ts2,ts3,ts4,ts5,ts6,ts7,ts8);
}
```

```
private static TupleSpaces createAndSetDiff(TupleSpace ts1, TupleSpace ts2,
                                            TupleSpace ts3, TupleSpace ts4,
                                            TupleSpace ts5, TupleSpace ts6,
                                            TupleSpace ts7, TupleSpace ts8)
  throws EmptySetTupleSpaceException
{
  TupleSpace []tupleSpaceArray = new TupleSpace[8];
  tupleSpaceArray[0] = ts1;
  tupleSpaceArray[1] = ts2;
  tupleSpaceArray[2] = ts3;
  tupleSpaceArray[3] = ts4;
  tupleSpaceArray[4] = ts5;
  tupleSpaceArray[5] = ts6;
  tupleSpaceArray[6] = ts7;
  tupleSpaceArray[7] = ts8;

  return createAndSetDiff(tupleSpaceArray);
}

/**
* createAndSetDiff
*
* @param []tupleSpaceArray a value of type 'TupleSpace'
* @return a value of type 'TupleSpaces'
* @exception EmptySetTupleSpaceException if an error occurs
*/
public static TupleSpaces createAndSetDiff(TupleSpace []tupleSpaceArray)
  throws EmptySetTupleSpaceException
{
  TupleSpace []resultTupleSpaceArray =
    setDifference(ServerAdapter.getInstance().getTupleSpaceDomain(),
                  tupleSpaceArray);

  AndLogicalOperator andLogicalOperator = new AndLogicalOperator();
  for(int index=0;index<resultTupleSpaceArray.length;index++) {
    andLogicalOperator.add(resultTupleSpaceArray[index]);
  }

  return new TupleSpaces(andLogicalOperator);
}





/**
* orSetDiff
*
```

```
* @param ts1 a value of type 'TupleSpace'
* @return a value of type 'TupleSpaces'
* @exception EmptySetTupleSpaceException if an error occurs
*/
public static TupleSpaces orSetDiff(TupleSpace ts1)
  throws EmptySetTupleSpaceException
{
  return createOrSetDiff(ts1,null,null,null,null,null,null,null);
}

/**
* orSetDiff
*
* @param ts1 a value of type 'TupleSpace'
* @param ts2 a value of type 'TupleSpace'
* @return a value of type 'TupleSpaces'
* @exception EmptySetTupleSpaceException if an error occurs
*/
public static TupleSpaces orSetDiff(TupleSpace ts1, TupleSpace ts2)
  throws EmptySetTupleSpaceException
{
  return createOrSetDiff(ts1,ts2,null,null,null,null,null,null);
}

/**
* orSetDiff
*
* @param ts1 a value of type 'TupleSpace'
* @param ts2 a value of type 'TupleSpace'
* @param ts3 a value of type 'TupleSpace'
* @return a value of type 'TupleSpaces'
* @exception EmptySetTupleSpaceException if an error occurs
*/
public static TupleSpaces orSetDiff(TupleSpace ts1, TupleSpace ts2,
                                    TupleSpace ts3)
  throws EmptySetTupleSpaceException
{
  return createOrSetDiff(ts1,ts2,ts3,null,null,null,null,null);
}

/**
* orSetDiff
*
* @param ts1 a value of type 'TupleSpace'
* @param ts2 a value of type 'TupleSpace'
* @param ts3 a value of type 'TupleSpace'
* @param ts4 a value of type 'TupleSpace'
* @return a value of type 'TupleSpaces'
* @exception EmptySetTupleSpaceException if an error occurs
```

```
*/
public static TupleSpaces orSetDiff(TupleSpace ts1, TupleSpace ts2,
                                    TupleSpace ts3, TupleSpace ts4)
  throws EmptySetTupleSpaceException
{
  return createOrSetDiff(ts1,ts2,ts3,ts4,null,null,null,null);
}

/**
* orSetDiff
*
* @param ts1 a value of type 'TupleSpace'
* @param ts2 a value of type 'TupleSpace'
* @param ts3 a value of type 'TupleSpace'
* @param ts4 a value of type 'TupleSpace'
* @param ts5 a value of type 'TupleSpace'
* @return a value of type 'TupleSpaces'
* @exception EmptySetTupleSpaceException if an error occurs
*/
public static TupleSpaces orSetDiff(TupleSpace ts1, TupleSpace ts2,
                                    TupleSpace ts3, TupleSpace ts4,
                                    TupleSpace ts5)
  throws EmptySetTupleSpaceException
{
  return createOrSetDiff(ts1,ts2,ts3,ts4,ts5,null,null,null);
}

/**
* orSetDiff
*
* @param ts1 a value of type 'TupleSpace'
* @param ts2 a value of type 'TupleSpace'
* @param ts3 a value of type 'TupleSpace'
* @param ts4 a value of type 'TupleSpace'
* @param ts5 a value of type 'TupleSpace'
* @param ts6 a value of type 'TupleSpace'
* @return a value of type 'TupleSpaces'
* @exception EmptySetTupleSpaceException if an error occurs
*/
public static TupleSpaces orSetDiff(TupleSpace ts1, TupleSpace ts2,
                                    TupleSpace ts3, TupleSpace ts4,
                                    TupleSpace ts5, TupleSpace ts6)
  throws EmptySetTupleSpaceException
{
  return createOrSetDiff(ts1,ts2,ts3,ts4,ts5,ts6,null,null);
}

/**
* orSetDiff
```

```
*
* @param ts1 a value of type 'TupleSpace'
* @param ts2 a value of type 'TupleSpace'
* @param ts3 a value of type 'TupleSpace'
* @param ts4 a value of type 'TupleSpace'
* @param ts5 a value of type 'TupleSpace'
* @param ts6 a value of type 'TupleSpace'
* @param ts7 a value of type 'TupleSpace'
* @return a value of type 'TupleSpaces'
* @exception EmptySetTupleSpaceException if an error occurs
*/
public static TupleSpaces orSetDiff(TupleSpace ts1, TupleSpace ts2,
                                    TupleSpace ts3, TupleSpace ts4,
                                    TupleSpace ts5, TupleSpace ts6,
                                    TupleSpace ts7)
  throws EmptySetTupleSpaceException
{
  return createOrSetDiff(ts1,ts2,ts3,ts4,ts5,ts6,ts7,null);
}

/**
* orSetDiff
*
* @param ts1 a value of type 'TupleSpace'
* @param ts2 a value of type 'TupleSpace'
* @param ts3 a value of type 'TupleSpace'
* @param ts4 a value of type 'TupleSpace'
* @param ts5 a value of type 'TupleSpace'
* @param ts6 a value of type 'TupleSpace'
* @param ts7 a value of type 'TupleSpace'
* @param ts8 a value of type 'TupleSpace'
* @return a value of type 'TupleSpaces'
* @exception EmptySetTupleSpaceException if an error occurs
*/
public static TupleSpaces orSetDiff(TupleSpace ts1, TupleSpace ts2,
                                    TupleSpace ts3, TupleSpace ts4,
                                    TupleSpace ts5, TupleSpace ts6,
                                    TupleSpace ts7, TupleSpace ts8)
  throws EmptySetTupleSpaceException
{
  return createOrSetDiff(ts1,ts2,ts3,ts4,ts5,ts6,ts7,ts8);
}




private static TupleSpaces createOrSetDiff(TupleSpace ts1, TupleSpace ts2,
                                           TupleSpace ts3, TupleSpace ts4,
                                           TupleSpace ts5, TupleSpace ts6,
                                                  TupleSpace ts7, TupleSpace ts8)
    throws EmptySetTupleSpaceException
  {
    TupleSpace []tupleSpaceArray = new TupleSpace[8];
    tupleSpaceArray[0] = ts1;
    tupleSpaceArray[1] = ts2;
    tupleSpaceArray[2] = ts3;
    tupleSpaceArray[3] = ts4;
    tupleSpaceArray[4] = ts5;
    tupleSpaceArray[5] = ts6;
    tupleSpaceArray[6] = ts7;
    tupleSpaceArray[7] = ts8;

    return createOrSetDiff(tupleSpaceArray);
  }

  /**
  * createOrSetDiff
  *
  * @param []tupleSpaceArray a value of type 'TupleSpace'
  * @return a value of type 'TupleSpaces'
  * @exception EmptySetTupleSpaceException if an error occurs
  */
  public static TupleSpaces createOrSetDiff(TupleSpace []tupleSpaceArray)
    throws EmptySetTupleSpaceException
  {
    TupleSpace []resultTupleSpaceArray =
      setDifference(ServerAdapter.getInstance().getTupleSpaceDomain(),
                    tupleSpaceArray);

    OrLogicalOperator orLogicalOperator = new OrLogicalOperator();
    for(int index=0;index<resultTupleSpaceArray.length;index++) {
      orLogicalOperator.add(resultTupleSpaceArray[index]);
    }

    return new TupleSpaces(orLogicalOperator);
  }

}// LogOp
```

Appendix C

LogOp Linda Primitive IDs

```
public interface LogOpPrimitiveID
{

  public static final int LOGOP_LINDA_OUT  = 9;
  public static final int LOGOP_LINDA_IN   = 10;
  public static final int LOGOP_LINDA_INP  = 11;
  public static final int LOGOP_LINDA_RD   = 12;
  public static final int LOGOP_LINDA_RDP  = 13;
  public static final int LOGOP_LINDA_EVAL = 14;

}// LogOpPrimitiveID
```

Appendix D

Tuple class

```
public class Tuple
  implements Serializable
{

  public final static int IS_FORMAL    = 2;
  public final static int IS_OBJECT    = 1;
  public final static int IS_PRIMITIVE = 0;

  private LinkedList items_;
  private String hashKey_=null;
  private int []objTypes_=null;
  private TupleSpace ts_;

  /**
  * Constructor
  *
  */
  public Tuple()
  {
    items_ = new LinkedList();
  }


  /**
  * Constructor
  *
  * @param o1 a value of type 'Object'
  */
  public Tuple(Object o1)
  {
    this();
    add(o1);
  }
```

```
/**
* Constructor
*
* @param o1 a value of type 'Object'
* @param o2 a value of type 'Object'
*/
public Tuple(Object o1, Object o2)
{
  this();
  add(o1);
  add(o2);
}

/**
* Constructor
*
* @param o1 a value of type 'Object'
* @param o2 a value of type 'Object'
* @param o3 a value of type 'Object'
*/
public Tuple(Object o1, Object o2, Object o3)
{
  this();
  add(o1);
  add(o2);
  add(o3);
}

/**
* Constructor
*
* @param o1 a value of type 'Object'
* @param o2 a value of type 'Object'
* @param o3 a value of type 'Object'
* @param o4 a value of type 'Object'
*/
public Tuple(Object o1, Object o2, Object o3, Object o4)
{
  this();
  add(o1);
  add(o2);
  add(o3);
  add(o4);
}

/**
* Constructor
*
```

```
* @param o1 a value of type 'Object'
* @param o2 a value of type 'Object'
* @param o3 a value of type 'Object'
* @param o4 a value of type 'Object'
* @param o5 a value of type 'Object'
*/
public Tuple(Object o1, Object o2, Object o3, Object o4,
             Object o5)
{
  this();
  add(o1);
  add(o2);
  add(o3);
  add(o4);
  add(o5);
}

/**
* Constructor
*
* @param o1 a value of type 'Object'
* @param o2 a value of type 'Object'
* @param o3 a value of type 'Object'
* @param o4 a value of type 'Object'
* @param o5 a value of type 'Object'
* @param o6 a value of type 'Object'
*/
public Tuple(Object o1, Object o2, Object o3, Object o4,
             Object o5, Object o6)
{
  this();
  add(o1);
  add(o2);
  add(o3);
  add(o4);
  add(o5);
  add(o6);
}

/**
* Constructor
*
* @param o1 a value of type 'Object'
* @param o2 a value of type 'Object'
* @param o3 a value of type 'Object'
* @param o4 a value of type 'Object'
* @param o5 a value of type 'Object'
* @param o6 a value of type 'Object'
* @param o7 a value of type 'Object'
```

```
*/
public Tuple(Object o1, Object o2, Object o3, Object o4,
             Object o5, Object o6, Object o7)
{
  this();
  add(o1);
  add(o2);
  add(o3);
  add(o4);
  add(o5);
  add(o6);
  add(o7);
}


/**
* Constructor
*
* @param o1 a value of type 'Object'
* @param o2 a value of type 'Object'
* @param o3 a value of type 'Object'
* @param o4 a value of type 'Object'
* @param o5 a value of type 'Object'
* @param o6 a value of type 'Object'
* @param o7 a value of type 'Object'
* @param o8 a value of type 'Object'
*/
public Tuple(Object o1, Object o2, Object o3, Object o4,
             Object o5, Object o6, Object o7, Object o8)
{
  this();
  add(o1);
  add(o2);
  add(o3);
  add(o4);
  add(o5);
  add(o6);
  add(o7);
  add(o8);
}

/**
* add
*
* add the Object o to the linked list and remember
* if it was an Object or a primitive.
*
* @param o a value of type 'Object'
* @param objType a value of type 'int'
```

```
*/
protected void add(Object o, int objType)
{
  if(o != null) {
    items_.add(o);
    int size = ((objTypes_==null) ? 0 : objTypes_.length);
    int []tmpArray = new int[size+1];
    for(int index=0;index<size;index++) {
      tmpArray[index] = objTypes_[index];
    }
    tmpArray[size] = objType;
    objTypes_ = tmpArray;
  }
}

public void add(Object object) { add(object,IS_OBJECT); }

public void add(int i) { add(new Integer(i),IS_PRIMITIVE); }

public void add(float f) { add(new Float(f),IS_PRIMITIVE); }

public void add(double d) { add(new Double(d),IS_PRIMITIVE); }

public void add(boolean b) { add(((b==true)
                                  ? Boolean.TRUE
                                  : Boolean.FALSE),
                                 IS_PRIMITIVE); }

public int size()
{
  return items_.size();
}

public Object get(int index)
{
  Object object = items_.get(index);
  return object;
}


/**
* getTS
*
* @return a value of type 'TupleSpace'
*/
public TupleSpace getTS()
{
  return ts_;
}
```

```
/**
* setTS
*
* @param ts a value of type 'TupleSpace'
*/
public void setTS(TupleSpace ts)
{
  ts_ = ts;
}


//public Integer getHashkey()
public String getHashkey()
{
  if(hashKey_==null) {
    generateHashkey();
  }
  return hashKey_;
}



/**
* generateHashkey
*
* must be done on server side due to the sb.toString generating
* a unique hash code in this VM.
*
*/
private void generateHashkey()
{
  if(hashKey_==null) {
    StringBuffer sb = new StringBuffer();
    ListIterator listIterator = items_.listIterator(0);
    int index=0;
    while(listIterator.hasNext()) {
      if(objTypes_[index]==IS_FORMAL) {
        sb.append(listIterator.next());
      }
      else {
        sb.append(((Class)(listIterator.next().getClass())).getName());
      }
      index++;
    }
    hashKey_ = sb.toString();
  }
  }

  public String toString()
  {
    StringBuffer sb = new StringBuffer();
    int max = items_.size();

    for(int index=0;index<max;index++) {
      Object item = items_.get(index);
      if(index==0) {
        sb.append(item.getClass().getName()).append(":")
          .append(item.toString());
      }
      else {
        sb.append(" ").append(item.getClass().getName()).append(":")
          .append(item.toString());
      }
    }
    return sb.toString();
  }


  public Template toTemplate()
  {
    Template template = new Template();

    int max = items_.size();
    for(int index=0;index<max;index++) {
      Object object = items_.get(index);
      template.addFormal(object.getClass().getName());
    }

    return template;
  }

}// Tuple
```

Appendix E

Template class

```
public class Template
  extends Tuple
{
  public Template()
  {
  }


  /**
  * Constructor
  *
  * @param s1 a value of type 'String'
  */
  public Template(String s1)
  {
    this();
    addFormal(s1);
  }

  /**
  * Constructor
  *
  * @param s1 a value of type 'String'
  * @param s2 a value of type 'String'
  */
  public Template(String s1, String s2)
  {
    this();
    addFormal(s1);
    addFormal(s2);
  }

  /**
```

```
* Constructor
*
* @param s1 a value of type 'String'
* @param s2 a value of type 'String'
* @param s3 a value of type 'String'
*/
public Template(String s1, String s2, String s3)
{
  this();
  addFormal(s1);
  addFormal(s2);
  addFormal(s3);
}

/**
* Constructor
*
* @param s1 a value of type 'String'
* @param s2 a value of type 'String'
* @param s3 a value of type 'String'
* @param s4 a value of type 'String'
*/
public Template(String s1, String s2, String s3, String s4)
{
  this();
  addFormal(s1);
  addFormal(s2);
  addFormal(s3);
  addFormal(s4);
}

/**
* Constructor
*
* @param s1 a value of type 'String'
* @param s2 a value of type 'String'
* @param s3 a value of type 'String'
* @param s4 a value of type 'String'
* @param s5 a value of type 'String'
*/
public Template(String s1, String s2, String s3, String s4,
                String s5)
{
  this();
  addFormal(s1);
  addFormal(s2);
  addFormal(s3);
  addFormal(s4);
  addFormal(s5);
}

/**
* Constructor
*
* @param s1 a value of type 'String'
* @param s2 a value of type 'String'
* @param s3 a value of type 'String'
* @param s4 a value of type 'String'
* @param s5 a value of type 'String'
* @param s6 a value of type 'String'
*/
public Template(String s1, String s2, String s3, String s4,
                String s5, String s6)
{
  this();
  addFormal(s1);
  addFormal(s2);
  addFormal(s3);
  addFormal(s4);
  addFormal(s5);
  addFormal(s6);
}

/**
* Constructor
*
* @param s1 a value of type 'String'
* @param s2 a value of type 'String'
* @param s3 a value of type 'String'
* @param s4 a value of type 'String'
* @param s5 a value of type 'String'
* @param s6 a value of type 'String'
* @param s7 a value of type 'String'
*/
public Template(String s1, String s2, String s3, String s4,
                String s5, String s6, String s7)
{
  this();
  addFormal(s1);
  addFormal(s2);
  addFormal(s3);
  addFormal(s4);
  addFormal(s5);
  addFormal(s6);
  addFormal(s7);
}
```

```
  /**
  * Constructor
  *
  * @param s1 a value of type 'String'
  * @param s2 a value of type 'String'
  * @param s3 a value of type 'String'
  * @param s4 a value of type 'String'
  * @param s5 a value of type 'String'
  * @param s6 a value of type 'String'
  * @param s7 a value of type 'String'
  * @param s8 a value of type 'String'
  */
  public Template(String s1, String s2, String s3, String s4,
                  String s5, String s6, String s7, String s8)
  {
    this();
    addFormal(s1);
    addFormal(s2);
    addFormal(s3);
    addFormal(s4);
    addFormal(s5);
    addFormal(s6);
    addFormal(s7);
    addFormal(s8);
  }



  public void addFormal(String formal)
  {
    super.add(formal,Tuple.IS_FORMAL);
  }

}// Template
```

Appendix F

Tuples class

```
public class Tuples
  implements Serializable
{
  private final Pair dummyPair_ = new Pair(null,null);
  private LinkedList pairs_;

  private int tsIndex_;

  public Tuples(int numberOfPairs)
  {
    pairs_ = new LinkedList();
    for(int index=0;index<numberOfPairs;index++) {
      pairs_.add(dummyPair_);
    }
    tsIndex_ = 0;
  }



  /**
  * strip
  *
  * removes dummyPair object from pairs_.
  *
  */
  public void strip()
  {
    int max = getCount();

    for(int index=max;index>0;index--) {
      Pair pair = (Pair)pairs_.get(index-1);
      if(pair == dummyPair_) {
        pairs_.remove(index-1);
      }
    }

    max = getCount();
  }


  public void add(TupleSpace ts, Tuple tuple)
  {
    if((ts != null) && (tuple!=null)) {
      Pair pair = new Pair(ts,tuple);
      pairs_.set(tsIndex_,pair);
    }
    tsIndex_++;
  }

  public int getCount()
  {
    return pairs_.size();
  }

  public boolean foundAllMatched()
  {
    boolean foundAllMatchedTuples = (tsIndex_==pairs_.size());
    return foundAllMatchedTuples;
  }



  /**
  * getTupleSpace
  *
  * @param index a value of type 'int'
  * @return a value of type 'TupleSpace'
  */
  public TupleSpace getTupleSpace(int index)
  {
    return ((Pair)pairs_.get(index)).getTupleSpace();
  }

  /**
  * getTuple
  *
  * @param index a value of type 'int'
  * @return a value of type 'Tuple'
  */
  public Tuple getTuple(int index)
  {
    return ((Pair)pairs_.get(index)).getTuple();
  }



  private class Pair
    implements Serializable
  {
    private TupleSpace ts_;
    private Tuple tuple_;

    public Pair(TupleSpace ts, Tuple tuple)
    {
      ts_ = ts;
      tuple_ = tuple;
    }

    private TupleSpace getTupleSpace()
    {
      return ts_;
    }
    private Tuple getTuple()
    {
      tuple_.setTS(ts_);
      return tuple_;
    }

    public String toString()
    {
      return "ts_:'"+ts_+"'   tuple_:[@"+
        ((tuple_==null)?"null":
         Integer.toHexString(tuple_.hashCode()))+"]:'"+tuple_+"'";
    }
  }
}// Tuples
```

Appendix G

OrDeterminism Interface Implementation

```
public interface OrDeterminism
{

  //
  // computes the resulting tuple spaces
  // to be used for an Or operator.
  //
  public TupleSpace []compute(TupleSpace []tss,
                            int logOpPrimitiveID);

}// OrDeterminism
```

```
public class OrOperator
  implements OrDeterminism
{
  public OrOperator()
  {
  }

  public TupleSpace []compute(TupleSpace []tupleSpaceArray,
                            int logOpPrimitiveID)
  {
    TupleSpace []newArray;

    if(duringWorkHours()) {
      switch(logOpPrimitiveID) {
        case LogOpPrimitiveID.LOGOP_LINDA_OUT:
        case LogOpPrimitiveID.LOGOP_LINDA_IN:
        case LogOpPrimitiveID.LOGOP_LINDA_INP:
        case LogOpPrimitiveID.LOGOP_LINDA_RD:
        case LogOpPrimitiveID.LOGOP_LINDA_RDP:
        case LogOpPrimitiveID.LOGOP_LINDA_EVAL:
          newArray = calculateNewTupleSpacesForWorkHours(workHourIPRanges,
                                                         tupleSpaceArray);
          break;
      }
    }
    else {
      switch(logOpPrimitiveID) {
        case LogOpPrimitiveID.LOGOP_LINDA_OUT:
        case LogOpPrimitiveID.LOGOP_LINDA_IN:
        case LogOpPrimitiveID.LOGOP_LINDA_INP:
        case LogOpPrimitiveID.LOGOP_LINDA_RD:
        case LogOpPrimitiveID.LOGOP_LINDA_RDP:
        case LogOpPrimitiveID.LOGOP_LINDA_EVAL:
          newArray = calculateNewTupleSpacesForNonWorkHours(nonWorkHourIPRanges,
                                                            tupleSpaceArray);
          break;
      }
    }

    return newArray;
  }
}// OrOperator
```

Appendix H

LogOpOutputStream and LogOpInputStream classes

```
public class LogOpOutputStream
  extends ObjectOutputStream
{
  public LogOpOutputStream(OutputStream os)
    throws IOException
  {
    super(new BufferedOutputStream(os));
    super.flush();
  }
}

public class LogOpInputStream
  extends ObjectInputStream
{
  public LogOpInputStream(InputStream is)
    throws IOException
  {
    super(new BufferedInputStream(is));
  }
}
```

Appendix I

LogOp.properties

```
#
# the initial capacity of hash tables in the
# kernel.
#
initialCapacity=100001

defaultServerBroadCastPort=9000

utsHost=machineNameC

servers=machineNameA, machineNameB, machineNameC, machineNameD



#
# ranges (23-50) are inclusive.
# * implies (0-255, inclusive).
#
#
# NOTE: range below does not include REMORA.
#
# NOTE: THERE IS NO PARSE CHECKING. YOU PUT SOMETHING
#       IN THAT SHOULD NOT BELONG, CODE WILL DIE!!!!!
#
#
OrLocale=22.50.74.130-140:22.50.74.150

OrDeterminismClass=test.OrOperator
```

Appendix J

Test case: FDRd.java

```
public class FDRd
  extends AllTests
{
  // tscount: tuple space count
  public FDRd(int tscount, String clientType)
  {
    TupleSpace []tupleSpaceArray = new TupleSpace[tscount];

    ServerAdapter server = ServerAdapter.connect();
    TupleSpace uts = server.getUTS();


    int doItIndex = 0;

    if(clientType.equals("p")) {

      //
      // producer
      //

      for(int index=0;index<tscount;index++) {
        tupleSpaceArray[index] =
          server.createTupleSpace(getHostname()+"-"+index);
        System.out.println("creating TupleSpace:'"+tupleSpaceArray[index]+"'");
      }

      AndLogicalOperator andLO = new AndLogicalOperator();
      for(int index=0;index<tscount;index++) {
        andLO.add(tupleSpaceArray[index]);
      }

      TupleSpaces tupleSpaces = new TupleSpaces(andLO);
      for(int index=0;index<tscount;index++) {
        server.out(uts,new Tuple(tupleSpaceArray[index]));
      }

      int []tupleCount = {1};

      for(int index=0;index<tupleCount.length;index++) {

        int MAX = tupleCount[index];
        System.out.println("doing: "+MAX+":   "+new java.util.Date());
        for(int outIndex=0;outIndex<MAX;outIndex++) {
          Integer i = new Integer(outIndex);
          server.out(tupleSpaces, new Tuple(i));
        }
      }

    }
    else {

      //
      // consumer
      //
      AndLogicalOperator andLO = new AndLogicalOperator();
      for(int index=0;index<tscount;index++) {
        andLO.add(uts);
      }
      TupleSpaces utss = new TupleSpaces(andLO);



      Template tsTemplate = new Template("edu.fit.linda.client.TupleSpace");
      Tuples tuples = server.in(utss,tsTemplate);

      andLO = new AndLogicalOperator();
      for(int index=0;index<tuples.getCount();index++) {
        TupleSpace ts = (TupleSpace)tuples.getTuple(index).get(0);
        System.out.println("["+index+"]:got TS:'"+ts+"'");
        //
        // load up LINDA array
        //
        tupleSpaceArray[index] = ts;

        //
        // load up AndLogicalOperator.
        //
        andLO.add(ts);
      }
```

```
TupleSpaces tupleSpaces = new TupleSpaces(andL0);

int []sample =
   {
     10,
     100,
     250,
     500,
     750,
     1000};
Template template = new Template("java.lang.Integer");

for(int countIndex=0;countIndex<5;countIndex++) {

  System.out.println("countIndex='"+countIndex+"'");

  for(int sampleIndex=0;sampleIndex<sample.length;sampleIndex++) {

    int MAX = sample[sampleIndex];

    {
      System.out.println("["+countIndex+"]:doing: "+MAX+":   "+
                         new java.util.Date());
      long startTime = System.currentTimeMillis();
      for(int index=0;index<MAX;index++) {
        tuples = server.rd(tupleSpaces,template);
      }
      long stopTime = System.currentTimeMillis();
      System.out.println("LOGOP:DJKS:["+doItIndex+"]--RD--[ "+
                         MAX+" ]: TOTAL_TIME: "+
                         (stopTime-startTime)+"(ms)");
    }


    {
      System.out.println("["+countIndex+"]:doing: "+MAX+
                         ":   "+new java.util.Date());
      long startTime = System.currentTimeMillis();
      for(int index=0;index < MAX; index++) {
        for(int tsindex=0;tsindex<tscount;tsindex++) {
          tuple = server.rd(tupleSpaceArray[tsindex],template);
        }
      }
      long stopTime = System.currentTimeMillis();
      System.out.println("LINDA:DJKS:["+doItIndex+
                         "]--RD--[ "+MAX+" ]: TOTAL_TIME: "+
                         (stopTime-startTime)+"(ms)");
    }
```

```
          System.out.println("");

        }

      }

    }

    server.disconnect();

    alldone(false);
  }


  public static void main (String[] args)
  {
    new FDRd(new Integer(args[0]).intValue(), args[1]);
  }

}// FDRd
```

Appendix K

Total tuples retrieved

Figure K.1 Total number of tuples retrieve from LogOp Linda and Linda between two servers. LogOp Linda and Linda calling *reset()* every 5000 object writes

Total Number of Tuples	Total Number of Tuple Spaces	LogOp Linda time per Iteration (millisecond)	Linda time per Iteration (millisecond)
62640	8	4	19
125280	16	5	39
187920	24	101	58
250560	32	100	78
313200	40	100	98
375840	48	100	117
501120	64	100	155
626400	80	101	195
939600	120	111	291
1252800	160	112	407
1566000	200	120	516
1879200	240	50	714
3132000	400	152	1145
4698000	600	215	1936
6264000	800	471	2602

Figure K.2 Total number of tuples retrieve from LogOp Linda and Linda between four servers. LogOp Linda and Linda calling *reset()* every 5000 object writes

Total Number of Tuples	Total Number of Tuple Spaces	LogOp Linda time per Iteration (millisecond)	Linda time per Iteration (millisecond)
62640	8	5	21
125280	16	6	40
187920	24	100	60
250560	32	100	79
313200	40	100	100
375840	48	100	120
501120	64	101	163
626400	80	102	198
939600	120	111	282
1252800	160	113	382
1566000	200	113	502
1879200	240	33	609
3132000	400	138	1010
4698000	600	160	1460
6264000	800	187	1983

Figure K.3 Total number of tuples retrieve from LogOp Linda and Linda between eight servers. LogOp Linda and Linda calling *reset()* every 5000 object writes

Total Number of Tuples	Total Number of Tuple Spaces	LogOp Linda time per Iteration (millisecond)	Linda time per Iteration (millisecond)
62640	8	8	33
125280	16	10	65
187920	24	169	96
250560	32	169	128
313200	40	101	96
501120	64	101	151
626400	80	101	192
939600	120	111	281
1252800	160	112	377
1566000	200	114	473
1879200	240	31	583
3132000	400	133	991
4698000	600	154	1481
6264000	800	176	1979

Figure K.4 Total number of tuples retrieve from LogOp Linda and Linda between two servers. LogOp Linda calling *reset()* and Linda not calling *reset()*

Total Number of Tuples	Total Number of Tuple Spaces	LogOp Linda time per Iteration (millisecond)	Linda time per Iteration (millisecond)
62640	8	110	19
125280	16	25	39
187920	24	120	58
250560	32	120	78
313200	40	36	98
375840	48	132	117
501120	64	52	155
626400	80	223	195
939600	120	183	291
1252800	160	192	407
1566000	200	209	516
1879200	240	232	714
3132000	400	324	1145
4698000	600	395	1936
6264000	800	563	2602

Figure K.5 Total number of tuples retrieve from LogOp Linda and Linda between four servers. LogOp Linda calling *reset()* and Linda not calling *reset()*

Total Number of Tuples	Total Number of Tuple Spaces	LogOp Linda time per Iteration (millisecond)	Linda time per Iteration (millisecond)
62640	8	111	21
125280	16	28	40
187920	24	121	60
250560	32	120	79
313200	40	38	100
375840	48	130	120
501120	64	79	163
626400	80	142	198
939600	120	165	282
1252800	160	182	382
1566000	200	180	502
1879200	240	221	609
3132000	400	303	1010
4698000	600	357	1460
6264000	800	455	1983

Figure K.6 Total number of tuples retrieve from LogOp Linda and Linda between eight servers. LogOp Linda calling *reset()* and Linda not calling *reset()*

Total Number of Tuples	Total Number of Tuple Spaces	LogOp Linda time per Iteration (millisecond)	Linda time per Iteration (millisecond)
62640	8	114	33
125280	16	34	65
187920	24	120	96
250560	32	130	128
313200	40	41	96
501120	64	141	151
626400	80	149	192
939600	120	168	281
1252800	160	185	377
1566000	200	191	473
1879200	240	222	583
3132000	400	301	991
4698000	600	352	1481
6264000	800	505	1979

Bibliography

[And81] Gregory R. Andrews. Synchronizing Resources. In *ACM Transactions on Programming Languages and Systems*, volume 3, pages 405–430, Como, Italy, 1981. ACM.

[BWA94] Paul Butcher, Alan Wood, and Martin Atkins. Global Synchronisation in Linda. *Concurrency: Practice and Experience*, 6(6):505–516, 1994.

[Cia97] Paolo Ciancarini. Coordination and Software Engineering. Tutorial Presented at Coordination '97, Berlin, Germany, September 1997.

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In *ACM Symposium on Theory of Computing*, pages 151–158, 1971.

[GC92] David Gelernter and Nicholas Carriero. Coordination Languages and their Significance. *Communications of the ACM*, 35(2):96–107, February 1992.

[Gel85] David Gelernter. Generative Communication in LINDA. *ACM transactions in programming languages and systems*, 1:80–112, 1985.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional Computing Series. Addison-Wesley, 1994.

[GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Second Edition.* Addison-Wesley, Boston, Mass., 2000.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, Englewood Cliffs, 1985.

[IBM98] IBM Corporation. *T Spaces Programmer's Guide*, 1998. Eletronic version only. http://www.almaden.ibm.com/cs/TSpaces/.

[Mah01] Qusay H. Mahmoud. *Transporting Objects over Sockets.* Sun Microsystems, Inc, 2001. Eletronic version only. http://developer.java.sun.com/developer/technicalArticles/ALT/sockets/.

[Men98] Ronaldo Menezes. Ligia: Incorporating Garbage Collection in a Java based Linda-like Run-Time System. In Raimundo J. Macedo, Alcides Calsavara, and Robert C. Burnett, editors, *Proc. of the 2nd Workshop on Distributed Systems (WOSID'98)*, pages 81–88, Curitiba, Paraná, Brazil, June 1998.

[MW00] I. Merrick and A. Wood. Coordination with scopes. In *Proceedings of the 2000 ACM symposium on Applied computing 2000, volume 1*, pages 210–217, Como, Italy, March 2000. ACM.

[Pin91] James Pinakis. Providing Directed Communication in Linda. Technical report, University of Western Australia, 1991.

[PMR99] Gian Pietro Picco, Amy L. Murphy, and Gruia-Catalin Roman. LIME: Linda Meets Mobility. In D. Garlan, editor, *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*, pages 368–377, Los Angeles, CA, USA, May 1999. ACM Press.

[RDW96] A. Rowstron, A. Douglas, and A. Wood. Copy-collect: A new primitive for linda. Technical Report YCS-268, Department of Computer Science, University of York, 1996.

[RW96] Antony Rowstron and Alan Wood. An Efficient Distributed Tuple Space Implementation for Networks of Workstations. In L. Bougé, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *Euro-Par'96*, volume 1123 of *Lecture Notes in Computer Science*, pages 510–513. Springer-Verlag, 1996.

[RW97] Antony Rowstron and Alan Wood. BONITA: A Set of Tuple Space Primitives for Distributed Coordination. In Hesham El-Rewini and Yale N. Patt, editors, *Proc. of the 30th Hawaii International Conference on System Sciences*, volume 1, pages 379–388. IEEE Computer Society Press, January 1997.

[SM02] Jim Snyder and Ronaldo Menezes. Using Logical Operators as an Extended Coordination Mechanism in Linda. In *Proceedings of COORDINATION '02*, LNCS, York, England, 2002. Springer-Verlag.

[Som92] Ian Sommerville. *Software Engineering.* International Computer Sciences Series. Addison-Wesley, Harlow, UK, 4th edition, 1992.

[Sud88] Thomas Sudkamp. *Languages and Machines.* Addison-Wesley Publishing Company, Inc., 1988.

[Sun97] Sun Microsystems, Inc. *JavaSpace Specification*, June 1997. Revision 0.4.