A Programmable Matching Engine for Application Development in Linda

George Clifford Wells

July 2001

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Doctor of Philosophy (PhD) in the Faculty of Engineering, Department of Computer Science.

32 200 words

Abstract

This thesis describes the novel features of an extended Linda system, called *eLinda*. The main contribution of eLinda is the introduction of a powerful and flexible mechanism for expressing the queries used for data retrieval in Linda systems. This provides simplicity for applications that would otherwise need to handle complex retrieval operations explicitly, and enhances efficiency, particularly where the data is distributed across a network.

Other extensions introduce support for distributed multimedia resources, and provide additional support for efficient data distribution across a communication network.

The enhanced functionality of the eLinda system is described in detail and compared with existing Linda systems, both commercial products and research projects. This is done primarily with an emphasis on the expressiveness and functionality of the new features, but also takes into consideration performance and efficiency issues, including the scalability of eLinda applications.

The use of these novel features is demonstrated by their application to a number of problems, including a video-on-demand system and a parser for visual programming languages. The latter application particularly shows the benefit of the new data retrieval mechanisms present in the eLinda system.

Acknowledgements

An undertaking like this, which has spanned some seven years and two continents, inevitably involves many people. While it would be impossible to mention them all individually, it is a pleasant task to pay tribute to those who have had a particularly large part to play.

First, and foremost, I would like to thank Alan Chalmers, firstly for agreeing to supervise this project and then putting up with the challenges of supervision over a great distance, under sometimes trying circumstances. Thanks to him the "story" has finally been told!

To my faithful team of proof readers, Alan Chalmers, Peter Clayton, Theo Groeneveld, Andrew Turpin and Madeleine Wright, thank you all for doing an excellent job, under a great deal of time pressure. I hope that I have not undermined your good work too drastically!

The work commenced during a period of sabbatical leave, spent at the University of Bristol. I wish to thank the staff and students of the Department of Computer Science, who provided a stimulating environment for my year with them. I must particularly mention Mike Rogers, for his help in making both the sabbatical year and the PhD study possible.

My "family" in the Department of Computer Science at Rhodes University, who are the best group of people I think I could ever hope to work with, have been an incredible source of encouragement and support. While it is invidious to single out any individuals, I feel I must particularly thank Peter Wentworth for being a constant source of good ideas and inspiration. I would also like to thank Jody Balarin and Billy Morgan for cheerfully helping with my requests for technical support and access to equipment. Thanks to Shaun Bangay for the background information about ray-tracing. Most of all I would like to thank Peter Clayton for his concern, assistance, guidance and support—it is a privilege to work with you Pete.

Pareen Daya did a great job of implementing the full support for the multimedia extensions in eLinda, under my supervision. He also produced the results for this section at very short notice, for which I am very grateful.

During the years that this project has taken I have been very lucky to be surrounded by a caring and supportive group of friends. While more grateful to all of them than they will ever know, I must again make a few "special mentions". To Kevin and Adele: thank you for putting up with me during the final stretch, and keeping the daily distractions and diversions at bay. Thank you to my friends in the River of Life Christian Assembly, who so gladly supported me, prayed for me, and understood and filled in for me when I wasn't around.

And most of all, Theo, Brenda, Caleb and Tiger, without whom I don't think this project would ever have been completed, you cannot know just how grateful I am to you. Thank you for providing an incredibly productive work environment in my "home from home" in Pretoria; thank you for your encouragement and your prayers, and for "cracking the whip" (in the nicest possible way!).

Last, but definitely not least, my thanks go to my family who for years have supported and encouraged me in everything that I have done. Thank you for giving me the start in life, the opportunities and the confidence to achieve what I have done.

Funding

I am very grateful to the following sources of funding for this work:

- The Distributed Multimedia Centre of Excellence in the Departments of Computer Science at Rhodes University and the University of Fort Hare, supported by Telkom SA, Lucent Technologies, Dimension Data and the Technology and Human Resources for Industry Programme (THRIP).
- The Joint Research Committee of Rhodes University, who provided funding support for conferences and travel.
- The National Research Foundation (NRF) of South Africa, who also provided conference and travel funds.

Trademarks and Copyright Notices

To him who is able to keep me from falling and present me before his glorious presence without fault and with great joy—to the only wise God my saviour be glory, majesty, power and authority, now and for evermore! Amen.

Adapted from Jude 24–25

Author's Declaration

I declare that the work in this dissertation was carried out in accordance with the Regulations of the University of Bristol. The work is original except where indicated by special reference in the text and no part of the dissertation has been submitted for any other degree. Any views expressed in the dissertation are those of the author and in no way represent those of the University of Bristol. The dissertation has not been presented to any other university for examination either in the United Kingdom or overseas.

SIGNED: _____

DATE: October 5, 2001

Contents

List of Figures

List of Tables

List of Program Segments

Chapter 1

Introduction

This thesis describes an extended version of Linda called *eLinda*. Linda is a language for coordination and communication in parallel and distributed programming that is based on a shared memory paradigm with a small set of simple operations that can be used by processes to access the shared data. This inherent simplicity offers a number of advantages for parallel and distributed programming. However, there are a number of problems associated with the Linda programming model. The novel features of eLinda address some of these weaknesses, particularly in the area of retrieving shared data. High-level support for distributed multimedia applications has also been integrated into the eLinda system.

This chapter presents a general introduction to the subject of Linda systems and the eLinda project, and provides an overview of the structure of the rest of the dissertation.

1.1 The eLinda System in Context

The Linda model for parallel and distributed programming was developed at Yale University in the mid-1980's[56]. In essence, it provides a shared memory model of a parallel or distributed computing system that can be accessed by the participating processing nodes. The logically shared memory space may be implemented using physically shared memory hardware, or, more usually, as some form of virtual shared memory.

The shared memory approach of Linda provides a high-level abstraction that can simplify the task of programming distributed or parallel systems, as the shared data space effectively decouples the communication between processes. This is the case both with respect to time (communication is asynchronous) and location (communicating

processes do not need to be aware of each other's identity or location in a multiprocessor or networked system). The inherent simplicity of the programming model and the decoupling of processes offer a number of benefits over systems based on message-passing, remote procedure call, etc.

However, Linda has been criticised for poor performance. Furthermore, there are many applications with data retrieval requirements that are difficult to express efficiently in Linda. The extensions introduced in eLinda have been designed with a view to making some of the underlying data retrieval and communication issues more explicit, thus providing the programmer with a greater level of control of the underlying system. A key feature of this is the distribution of complex matching (or searching) operations, which are central to the retrieval of data in the Linda programming model. Additionally, as distributed multimedia applications are becoming increasingly important, high-level support for multimedia data types and their efficient distribution across a communication network has also been added to eLinda.

1.1.1 The History of eLinda

The eLinda project began in 1995, and its current form may be better understood if the original motivations and subsequent changes in direction are made known from the outset.

The original intention was to develop the eLinda system for use with multiprocessor Transputer systems[158], making use of minimum path communication network configurations for efficient broadcast communication[34]. In preparation for this, a prototype system was initially developed to test the new concepts. This was implemented for networks of UNIX workstations, using a message-passing library (PVM[144]) to handle the communication issues[156]. The initial application area of interest was high performance graphics rendering[157]. However, in 1996 SGS-Thompson, the manufacturers of the Transputer, announced their intention of discontinuing production and development of the Transputer family of processors. At about the same time the Java programming language was beginning to increase in popularity, and provided some interesting features, specifically program portability and simple, but powerful, networking support. Accordingly the eLinda project was redirected to make use of Java as the development language, and targeted at distributed, "network of workstation" (NOW) systems rather than multiprocessor parallel processing systems. However, despite these changes, the unique features of eLinda remained essentially unchanged.

1.2 Structure of the Thesis

The following chapters expand on the themes outlined above as follows:

Chapter 2 presents a discussion of techniques for distributed and parallel programming, focussing on shared memory abstractions, and other mechanisms available for distributed programming in Java. This includes a detailed description of the original Linda programming model developed at Yale University, and two recent implementations of Linda in Java by large organisations.

Chapter 3 introduces the eLinda system and its extensions to the original Yale Linda model. Three implementations of eLinda have been developed to demonstrate the new concepts, and are also described in this chapter.

Chapter 4 expands on Chapter 3, describing the central extension in eLinda, the *Programmable Matching Engine*, in greater detail.

Chapter 5 describes a number of example applications for which eLinda has been used. These include simple benchmark programs and larger applications, including the parallel implementation of a parser for graphical programming languages.

Chapter 6 presents the results, both qualitative and quantitative, of implementing and running the applications described in Chapter 5. Comparisons are made with the commercial Linda implementations and with other research projects.

Chapter 7 presents an analysis of the results, and outlines possible directions for future research in this area.

There are two appendices that provide additional details of some of the features of eLinda, and other supporting material:

Appendix A provides an overview of the process for developing new matching algorithms, using the Programmable Matching Engine.

Appendix B outlines the support that would be required for a preprocessor for eLinda.

Chapter 2

Background

This chapter is divided into two main sections. The first introduces the area of concurrency, focussing on the Linda programming model as originally developed at Yale University. The second section concentrates specifically on the mechanisms that are available in Java for parallel and distributed programming, including a number of Linda systems and related projects. In particular, detailed descriptions are given of two commercial Linda implementations in Java: TSpaces from IBM, and JavaSpaces from Sun Microsystems.

2.1 Parallel and Distributed Programming

Cleaveland *et al* provide a good definition of the related concepts of concurrency, parallel and distributed systems as follows:

> *Concurrency* is concerned with the fundamental aspects of systems of multiple, simultaneously active computing agents that interact with each other. This notion is intended to cover a wide range of system architectures, from tightly coupled, mostly synchronous *parallel* systems, to loosely coupled, largely asynchronous *distributed* systems. [42]

As is implied by this definition, the field of concurrency can be viewed as a spectrum ranging from parallel systems to distributed systems. Along this continuum there are many different languages, models and systems available for programming concurrent applications. A very useful organisation of these different approaches is their classification into a hierarchy of differing levels of abstraction, as shown in Figure 2.1[92].

Tuple/Object Space	Highest level of abstraction
Network Services, Object Request	↑
Brokers, Mobile Agents	↑
Remote Procedure Call/	↑
Remote Method Invocation	↑
Client/Server, Peer to Peer	↑
Message Passing	Lowest level of abstraction

Figure 2.1: Abstraction Levels for Parallel and Distributed Programming

Linda falls into the category of tuple or object space systems, at the highest level of abstraction in this hierarchy.

A useful analogy can be made with the traditional description of levels of abstraction in programming language systems, from the low level view of assembly language programming through to the higher levels of abstraction provided by fourth generation programming languages. This view is expressed well by Shires *et al*, who describe a particular message passing system as "the assembly-language of parallelism with its required attention to detail"[137].

Before considering various languages and models that have been proposed for concurrent programming we will discuss briefly some of the major factors influencing the hardware platforms for supporting concurrent applications.

2.1.1 Classifying Computer Architectures for Concurrency

The major classification in this area is due to Flynn[48], whose well-known taxonomy of computer architectures is the starting point for any discussion of this topic. He classified computer architectures into four categories, based on the way in which they handle *instructions* and *data*, either as a single stream or as multiple streams. This can be pictured as shown in Figure 2.2. Each of these categories will be considered briefly.

SISD This is the classical von Neumann architecture of a sequential processor, executing one instruction at a time, and processing a single datum at a time. As such it is of little interest to us, and will not be considered further.

MISD This organisation consists of several instructions operating on a single datum. Such an architecture has no obvious applications, although sometimes vector

Single Instruction Stream Single Data Stream (SISD)	Multiple Instruction Stream Single Data Stream (MISD)
Single Instruction Stream Multiple Data Stream (SIMD)	Multiple Instruction Stream Multiple Data Stream (MIMD)

Figure 2.2: Flynn's Taxonomy of Computer Architectures

processors are categorised as MISD architectures (this arises from viewing an entire vector as a single datum).

SIMD This architecture applies a single instruction to multiple data elements. Such architectures (*array processors*) have been developed, but are not common.

MIMD These are the most complex architectures in Flynn's classification, where multiple operations are simultaneously applied to multiple data values. The machines in this category can be further classified into four subdivisions[171]:

1. *Multiprocessors*, in which a number of processors are connected to a number (possibly only one) of shared memory modules through an interconnection network.
2. *Multicomputers*, in which a number of processing nodes (each containing both processor and memory) are connected by a communication network.
3. *Multi-multiprocessors*, a combination of the two approaches above, where a number of distinct processing elements are connected by a communication network, and each processing element comprises a number of processors with shared local memory.
4. *Data flow machines*, where the movement of data through the machine triggers the execution of instructions.

Hybrid approaches, combining the features of two (or more) of the categories identified above are also possible (for example, a MIMD machine, where the processing nodes are each based on the SIMD approach).

The rest of this thesis will concentrate on the use of MIMD architectures, and multicomputers in particular. The hardware platform that was used for the implementation and testing of the application of eLinda was a collection of single-processor workstations, connected by a local area network. More details about the exact specifications of the hardware used are given in Chapter 6.

2.1.2 Linda

Linda is a *coordination language* for parallel and distributed processing, providing a communication mechanism based on a logically shared memory space called *tuple space*. On a shared memory multiprocessor the tuple space may actually be physically shared. The close match in this case between the hardware platform being used and the logical programming model embodied in the Linda approach will generally make the implementation of a Linda system (or any virtual shared memory system) extremely simple. For this reason most of the research on the Linda approach to parallel and distributed programming has been focussed on distributed memory systems where the problems are more challenging and interesting[2, 26, 46, 56]. On distributed memory systems (such as a network of workstations) the tuple space may either be centralised on a single processing node or else distributed among the processing nodes in some way.

An entry in the tuple space is stored as a *tuple*, made up of a collection of fields. A simple example of a tuple with three fields is `("point", 12, 67)`, where 12 and 67 might be the x and y coordinates of the point represented by this tuple. The combination of the number and type of the fields in a tuple (in this case: "string, integer, integer") is referred to as the *type signature* of the tuple. The tuples in the tuple space are always accessed using *associative addressing* to specify the required tuple that is to be retrieved.

As a coordination language, Linda is designed to be coupled with a sequential programming language, called the *host language* (the host language used in this work is Java). The rationale for separating the coordination language and the host language is that existing sequential programming languages already provide all the features required for computation, input/output, etc.—all that is then required to support distributed programming is the addition of mechanisms for communication and coordination. These may be provided by a coordination language, such as Linda, as an orthogonal addition to the host language. An advantage of this approach is that programmers who are migrating to distributed programming do not need to learn a completely new language if they are already familiar with the host language. All that is required is for them to gain familiarity with the additional features provided by the coordination language. While this may appear straightforward, it is important to note that the coordination language is usually tightly coupled with the host language, and is not simply an additional programming library. This is most apparent in the way

that a preprocessor is required to handle the inclusion of the coordination language features in the host language programs.

The Linda Programming Model

From a programmer's perspective, Linda effectively provides a small set of operations that allow tuples to be placed into the tuple space and retrieved from it. These operations are:

`out` Places a tuple in the tuple space.

`in` Removes a tuple from the tuple space, returning it to the application. If a suitable tuple is not found in the tuple space then this operation blocks until such a tuple is placed into the tuple space by another process.

`rd` Returns a copy of a tuple, leaving the original tuple in the tuple space, and blocking if a suitable tuple cannot be found.

`inp` The predicate form of `in`. This operation does not block if a suitable tuple cannot be found, but returns immediately with an indication of failure.

`rdp` The predicate form of `rd`.

For the input operations (i.e. `in`, `rd`, and their predicate forms) the specification of the tuple to be retrieved makes use of an associative matching technique. In this case a subset of the fields in the tuple have their values specified, and these are used to locate a matching tuple in the tuple space. For example, if a tuple representing a point (such as that in the example above) were required then the following operation might be used to retrieve it:

```
in("point", ?x, ?y)
```

The tuple specification used in an input operation (`("point", ?x, ?y)` in this example), is referred to as an *anti-tuple*. The unspecified fields in the anti-tuple (the x and y values in this example) are referred to as *formals* (or sometimes as *wildcards*).

The input operation uses the given anti-tuple to attempt to locate a suitable tuple from among those currently found in the tuple space. Any tuple that has the same type signature and has fields with values that exactly match the specified fields in the anti-tuple (i.e. the string `"point"` in the first field for the example above) will be a suitable match. If such a tuple is found in the tuple space, then any formal fields are

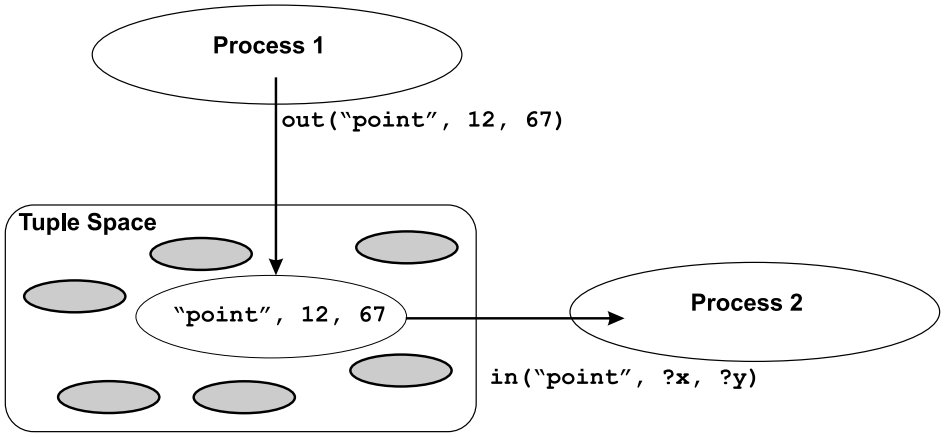


Figure 2.3: A Simple Communication Pattern

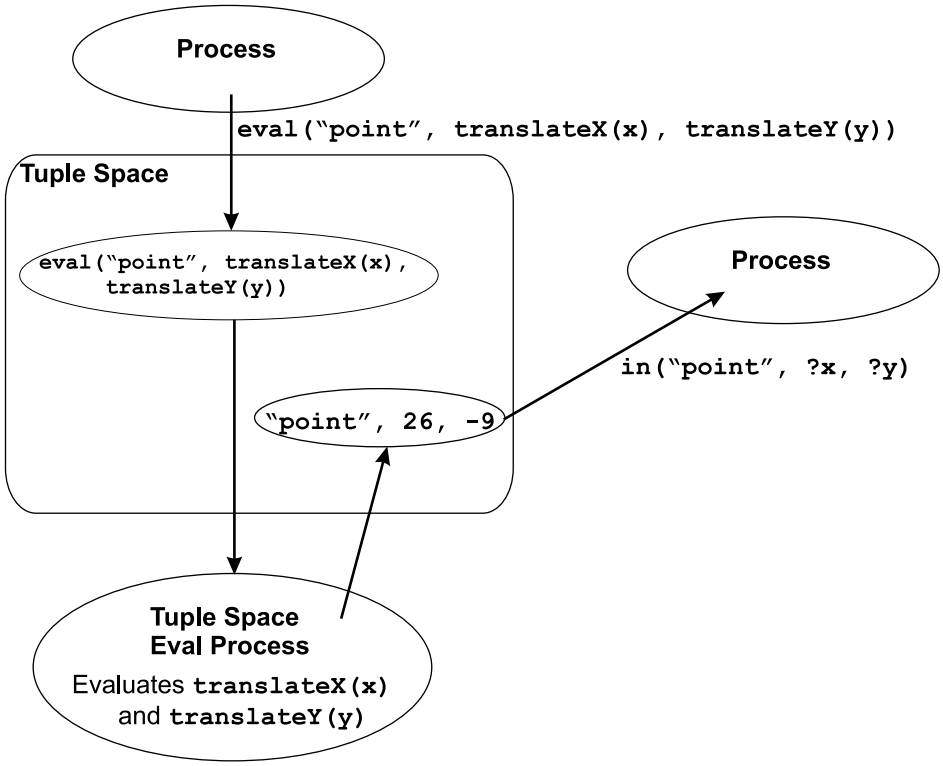
assigned the values of the corresponding fields of the matching tuple. If more than one matching tuple is found in the tuple space then a Linda system is free to return any one of them—there is no requirement for first-in-first-out behaviour (or any other specific semantics) in this case. This simple use of Linda for direct communication is illustrated in Figure 2.3.

The original Linda model also specified a mechanism for dynamic process creation. This was provided by means of a further operation: `eval`. The `eval` operation places a tuple in the tuple space, in the same way that the `out` operation does. The difference is that the tuple may include unevaluated function calls. For example:

```
eval("point", translateX(x), translateY(y))
```

In this case the functions `translateX` and `translateY` are not evaluated in the same way as parameters. An *active tuple* such as this would be retrieved and the functions executed by the Linda system on any available processing node. The results of the function evaluations are then used to create a normal (or *passive*) tuple that is placed into the tuple space. In this example, the eventual result would be a three-tuple with the new, translated x and y coordinate values. This is illustrated in Figure 2.4.

In this way, the `eval` mechanism provides simple, powerful support for the creation of new processes to compute results, which may then be retrieved from the tuple space. However, the `eval` operation is not an essential part of the Linda programming model, especially where the host language provides mechanisms for process creation. It has also been shown that the `eval` operation may be implemented in terms of the other Linda operations with some support from a preprocessor[74]. As a result it will not be considered any further here.



Note: While two application processes are shown here, there may only be one involved if the original process retrieves the result of the tuple that it created with the `eval` operator.

Figure 2.4: The Use of the `eval` Operation

Full details of the Linda programming model, together with detailed case studies of some of its applications, may be found in [30].

Decoupling of Processes

An important aspect of the Linda programming model is that the processes are highly independent of each other. This has two main aspects: *spatial decoupling* and *temporal decoupling*.

Spatial decoupling refers to the fact that processes communicating in a Linda system need not be aware of the processor network topology or of each other's location in this network. The logically shared tuple space removes any requirement for addressing tuples or processes.

Temporal decoupling arises from the fact that the transmission of tuples between processes through the medium of the tuple space is asynchronous. A process may place a tuple in the tuple space and then continue with other activities (or even terminate), completely independently of the execution of the process that may retrieve the tuple at some future time. If the tuple space is made persistent then the output and input operations that comprise the communication between two processes may even span time periods covering system restarts[4].

This high degree of decoupling is evidence of the high level of abstraction of Linda in comparison with other mechanisms for parallel and distributed programming.

Tuple Space Implementation Strategies

Grune *et al* give a very good overview of possible implementation strategies for Linda systems[62]. Particularly on distributed memory systems, there are many possible strategies for implementing the shared tuple space (of course, such issues are usually transparent to the programmers and users of the system)[2, 26, 56]. The most basic classification is into centralised and distributed approaches. A centralised tuple space implementation makes use of a single server node to store all of the tuples for the entire system. This has the obvious advantage of simplicity, but the central server may become a performance bottleneck.

Distributing the tuple space across more than one server may be done in many ways. Two common approaches are *hashing* and *partitioning*. In a hashing system the contents of a tuple are hashed, and this value is used to allocate the tuple to a specific processor[12]. This approach either requires the use of a preprocessor to perform a

usage analysis to determine which of the fields may be used as a key (or keys) for the hashing function, or the tuple structure must be fixed (for example, the first field must always be specified so that it may be used as the hash key).

In partitioned systems, tuples with a common *structure* are allocated to a specific processor[62]. The structure is generally determined from the type signature of the tuples, but may also include information supplied by the programmer, such as naming groups of tuples.

Specific system architectures may also provide opportunities for unique methods of distribution. For example, regular mesh networks may be constructed such that Linda output operations distribute tuples along rows (referred to as the *out-set*) and the input operations search for tuples along columns (referred to as the *in-set*)[46].

The most general approach is to have the tuple space fully distributed, such that any tuple may reside on any processing node. This technique may offer advantages in areas such as fault tolerance. It will generally require efficient broadcast communication mechanisms, as searching for tuples will often require interrogating all the distributed sections of the tuple space.

Problems with Linda

As discussed at the start of this chapter, the Linda approach to parallel and distributed programming is at a high level of abstraction. This can be seen in the provision of a small set of operations, the powerful associative retrieval mechanism, the logically shared tuple space and the temporal and spatial decoupling of processes. In combination these features provide a very useful simplicity and flexibility for constructing parallel and distributed applications.

However, Linda has been subject to criticism for inefficient and unpredictable performance[173], as the high level of abstraction of the Linda programming model hides the underlying complexity of the data sharing and communication required to implement the tuple space operations. The analogy with traditional programming language abstraction levels can be seen here again: the situation is similar to the performance disparities between low level assembly languages and higher level, third or fourth generation programming languages.

Furthermore, some applications may be very difficult to implement efficiently using the standard Linda communication mechanisms. The extensions in eLinda have been developed to overcome some of these problems, while retaining as much of the essential

simplicity of the original Linda approach as possible. These problems and the solutions provided by eLinda will be discussed in more detail in the next chapter.

Subsequent Research Directions at Yale

After the initial development of the model and research into efficient implementation techniques, the Linda research group at Yale University moved on to focus more on the applications of Linda, or on adaptations of the model for specific purposes.

One of the main projects that arose out of the work on Linda was Piranha[85, 117]. This used a variation of Linda to support *adaptive parallelism*. Adaptive parallelism refers to systems where processors can dynamically join and leave the group of processors executing a parallel program. This is intended to allow personal computers or workstations to be used for parallel processing while idle. If a user commences (or resumes) work on a computer then the Piranha system will cease its activity on that processing node (or *retreat* in Piranha terminology). The Piranha project used a variation of the Linda programming model for communication and coordination. This was adapted to support the features required for distributing tasks, allowing them to retreat and then to resume on another processing node. The spatial decoupling of processes that is provided by the Linda model is extremely useful in these circumstances.

Other Linda Systems

This section considers other Linda systems that have been developed, especially those that have proposed extensions related to the new features of eLinda. It excludes the many such systems that have been developed using Java as the host language, which are discussed in Section 2.2.3.

Objective Linda Objective Linda is a model for object-oriented implementations of Linda[86, 87]. All aspects of an application (i.e. data, active agents and the tuple spaces) are modelled as objects. A language-independent notation (called the *Object Interchange Language*, or *OIL*) is then used to describe the classes comprising an application. The tuple spaces are called *object spaces* in Objective Linda. They form a strongly encapsulated hierarchy of objects, containing passive objects (i.e. tuples), active objects (i.e. agents) and other object spaces.

Agents in Objective Linda may move from one object space to another. This is controlled by the object space itself, which makes available an *object space logical* (or

simply *logical*—this can be thought of as a "key") to permit agents to move into the object space. This is supported by two new operations: `join` and `enter`, which are analogous to `rd` and `in` respectively. These new operations also have predicate forms, `joinp` and `enterp`.

Of particular relevance for comparison with eLinda is the fact that the matching mechanism in Objective Linda differs from the usual Linda approach. The objects that are to be used as tuples in Objective Linda are required to provide a `match` method. This method is then called when performing an input operation. This means that the programmer writing the classes to be used as tuples in an application can define the precise meaning of a "match". This is similar to the way in which Java programmers can override the `equals` method for objects to define the precise meaning of equality for their own classes. This feature of Objective Linda provides a restricted form of extended matching similar to that in eLinda.

An interesting feature of Objective Linda is the support for the `eval` operation. This is provided in a similar way to the implementation of matching described above in that tuples that are to be used with `eval` are required to provide an `evaluate` method. This method is used by the Objective Linda system to perform the evaluation of the necessary result(s).

I-Tuples I-Tuples (*Intelligent Tuples*) is an extended Linda system developed in C[49]. The I-Tuples proposal is a simple and elegant one. Many programs may need to perform minor updates to shared data values, for example incrementing a counter. A simple example of this for a conventional Linda system is as follows:

```
in(?someValue);
someValue++;
out(someValue);
```

If the required tuple is stored on another processing node (as is likely to be the case in most implementations), then this involves a considerable amount of network traffic:

1. The anti-tuple is sent to the server.
2. The result of the `in` operation is returned.
3. After the addition, the new tuple is sent back to the server.
4. Depending on the implementation, there may be an acknowledgment returned by the server for the `out` operation.

This amounts to three or four network messages to perform a very simple task. In any system this is likely to exhibit a highly unfavourable communication:computation ratio.

The I-Tuples system allows the programmer to use a new operation (`tmexec`) to get the *server* to perform the desired task (i.e. incrementing the value contained in a tuple in the example above). In this way the number of network messages is reduced: the command is sent to the server, which does the necessary update and then an acknowledgement message is returned to the application. The command sent to the server is programmed using the standard Linda operations (`in`, `out`, etc.).

The results reported for this system show a substantial decrease in execution times when the new features of I-Tuples are used. However, the current implementation suffers from some disadvantages. The most important of these is that it requires all the processing nodes to have identical processor architectures[1]. It is also likely that extensive use of this feature will overload the server.

ELLIS ELLIS (a EuLisp LInda System) is a Linda system developed in EuLisp, using PVM as the communication medium[14]. It was intended as a platform for experimenting with object-oriented concepts in Linda. Of particular interest is that matching is performed by a method in the tuple space class (*pool objects* as they are known in ELLIS). This allows the matching method to be overridden, but the mechanism seems clumsy: new classes of tuple spaces must be created to support new matching algorithms. Few details of this process are given in the description of ELLIS, but the programming interfaces for new matchers appear to be complex and to involve dealing with the pool of tuples at a very low level of abstraction (for example, the writer of a new matching algorithm must concern himself with the details of tuple distribution, etc.).

The York Coordination Group The coordination research group at the University of York has been actively researching in the area of Linda systems for some time. One of their major projects has been to extend the Linda operations with `collect` and `copy-collect`[17, 127]. These *bulk operations* may be used to move or copy multiple tuples from one tuple space to another. The work at York suggests that these operations may be suitable replacements for the predicate operations in Linda. Other

[1]This is due to the fact that the code to be executed by the server is passed as a memory pointer, requiring identical code memory images on all the processing nodes.

aspects of their Linda research are the provision of security, garbage collection facilities, distributed I/O support, and transaction processing[100, 101, 165].

Of particular interest is the extension of Linda to support constraint matching for case base reasoning systems[22]. This project made use of an implementation of Linda, based on the *Chemical Abstract Machine*, or *CHAM*. This is an unusual programming model, in which systems are expressed as "solutions of molecules" (multisets, describing the state of the system), and subjected to "chemical reactions" (rewriting of the multisets, subject to "reaction rules"). Programs in this model consist of sets of reaction pairs, composed of a condition, specifying when the rule may be applied, and an action, which is a function that produces new molecules from the reactants. This model is ideally suited to parallel implementation, as independent reactions may take place simultaneously.

The Chemical Abstract Machine has been used to implement a Linda system, called Liam. This system allows the matching algorithm for tuples to be provided in CHAM form. This has the drawback that programmers must become familiar with the syntax used by the Chemical Abstract Machine. As an example, the following line of CHAM code is taken from a simple matcher that specifies the retrieval of tuples containing two fields with equal values. The total specification is nine lines of CHAM code, of which this one is typical.

$$\text{cm2eq)}\ \text{op}'_{\mathbf{cm2eq}}(1, a:\tau), (1, a:\tau) \leftrightarrow \text{op}(1, a:\tau/a:\tau, \mathbf{cm2eq}), (1, a:\tau)$$

This is clearly not a simple notation for the average application programmer to learn and use.

2.1.3 Other Concurrent Programming Models

The Linda model of parallel and distributed processing falls into the general category of *virtual shared memory* systems. There are a number of other implementations of virtual shared memory. Additionally there are a number of popular approaches to the programming of concurrent systems based on forms of message passing. There are also other parallel programming models, such as the Bulk Synchronous Parallel (BSP) model. These various approaches are considered and contrasted with the Linda model in the following sections.

Shared Address Space Shared Memory (SASM)	Disjoint Address Space Shared Memory (DASM)
Shared Address Space Distributed Memory (SADM)	Disjoint Address Space Distributed Memory (DADM)

Figure 2.5: Raina's Taxonomy of Shared Memory Architectures

Virtual Shared Memory Systems

Virtual shared memory is a rather loosely used term. In this section we will consider the definition of the term, and also the possible implementations of this concept. The first point to note is that we are not concerned here with architectures that provide shared access to common memory modules (i.e. physically shared memory systems). Rather the focus is on systems that are implemented with distributed memory modules, but which provide the appearance of a common, shared memory in some way. This may be done either in hardware or in software.

Raina provides a very useful overview and organisation of the field, with an emphasis on hardware-supported virtual shared memory[114].

Classification of Virtual Shared Memory Approaches Raina's classification is similar to that used by Flynn for computer architectures: he identifies two important characteristics and uses these to classify systems into four distinct quadrants. The characteristics used to distinguish virtual shared memory systems are the *address space* and the *physical memory*, both of which may either be *shared* or *disjoint*. This classification scheme gives rise to the categories illustrated in Figure 2.5, and discussed below.

SASM This category comprises conventional shared memory systems (i.e. *multiprocessors*, as defined in Section 2.1.1).

DADM These systems are conventional distributed memory architectures (i.e. *multicomputers*).

DASM Machines supporting a disjoint address space, but having physically shared memory would probably not be that useful, and so are not considered any further.

SADM These are the most interesting category in the classification, comprising architectures where the address space is shared, but the memory is physically dis-

tributed among the processing nodes. In other words these systems are *multicomputers*, but give the appearance of being *multiprocessors* (in the terminology of Section 2.1.1).

These systems may be further classified according to how the appearance of shared memory is maintained.

SADM-NUMA These systems provide transparent access to those parts of the shared address space held in remote phyical memories (giving rise to a Non-Uniform Memory Access time—NUMA). The data is always stored in a fixed location in this model.

SADM-OS In this case the operating system provides the abstraction of shared memory. This is usually implemented by the distribution of pages of conventional virtual memory across the processor network. Page faults then require the processor to identify the location of the required memory page and fetch it from the processing node that currently holds it.

SADM-CC These systems generally use hardware assistance to provide access to small units of memory (the size of a cache line). Standard Cache Coherency (CC) protocols are used to ensure the consistency of data being accessed by the processing nodes.

SADM-COMA This is similar to the SADM-CC approach, but there is no concept of "main memory" in these systems. The local memory of each processing node is viewed purely as a *cache* (hence, Cache Only Memory Architecture—COMA). Data in remote caches can be moved to the local cache for processing (distinguishing this from the NUMA approach where a data element has a fixed memory location).

The term *virtual shared memory* is used generally of all of the SADM approaches described above.

The Linda approach might be viewed as a variation of the SADM-OS model described by Raina, as the support for the virtual shared memory in both cases is provided by software. The major difference is that the data in the case of an SADM-OS system is handled in units of virtual memory *pages*, whereas in Linda *tuples* are used as the unit of storage. Furthermore, the addressing of memory in an SADM-OS system uses conventional memory addressing, rather than the associative addressing used by Linda.

Message Passing

Message passing is one of the most common and popular methods for communication in parallel and distributed systems. In terms of the analogy with traditional programming languages, it can be considered the "assembly language" of parallel and distributed programming[137]. Zenith also identifies problems with message passing related to the low level of abstraction[172].

There are many different systems that implement some form of message passing, and it is often used as the foundation for other, higher levels of abstraction. The major distinction that can be drawn between message passing systems is whether the message passing is *synchronous* or *asynchronous*. This section will use this classification as a framework for the discussion of some of the more popular message passing systems available and their main distinguishing features.

Synchronous Message Passing In this form of message passing, when two processes wish to communicate, the first one that is ready to do so (whether it is reading or writing) must wait for the second one to join in the communication operation before it can proceed. In this way the communication operation is also a synchronisation operation. One of the advantages of this approach is that there is no requirement for buffering messages that have been sent but not yet received—the message is simply transferred directly from the sending process to the receiving process at the moment of communication.

One of the best known examples of the synchronous message passing approach is Hoare's CSP (Communicating Sequential Processes).

CSP CSP is a theoretical model of parallel programming that has a sound mathematical underpinning[70, 71]. This allows formal reasoning and proofs to be carried out on systems designed using CSP. In particular a system designed using CSP can be analysed to determine its *liveness* and *safety* properties. These are defined by Owicki and Lamport[107] as follows:

safety "something bad will not happen"

liveness "something good will happen"

CSP describes a concurrent system in terms of *processes*. The only interaction allowed between two processes, and between a process and its environment is when both

$$\alpha \text{ ACC} = \{ \text{ deposit, withdraw } \}$$

$$\begin{aligned} \text{ACC} = (\ & \text{deposit} \rightarrow \text{ACC} \\ & \mid \text{withdraw} \rightarrow \text{ACC}) \end{aligned}$$

Program Segment 2.1: A Simple CSP Example

participate in a common *event*. In CSP a process can be described by its *interface*, i.e. the set of events in which it may participate. Communication is handled as a special case of an event in CSP, and is synchronous. In the original model both processes were explicitly named[70], giving rise to tight spatial coupling, but this restriction was later removed with the introduction of *channels*[71]. Furthermore, while the synchronous communication in CSP embodies tight temporal coupling, this can be alleviated very easily by using buffering.

The CSP model is also *compositional*: if two processes are combined to form a larger system, then the combination is also a process. A group of processes may be described by a trace of the events in which they have participated.

A simple CSP example is shown in Program Segment 2.1. This describes a process (called ACC) representing a bank account. The interface of the process (denoted α ACC) is the set of events { deposit, withdraw }. The description that follows this states that the ACC process may participate in either the deposit or withdraw events, after which it returns to its previous state.

A "bank client" process might be modelled similarly, and would participate in the same events. In that case the presence of the same event in the description of the two processes implies that they are synchronised at that point.

While CSP is a theoretical model intended for the description of concurrent systems, and reasoning about their behaviour, it was the basis of the development of the occam programming language for Transputers.

Transputers and occam The occam programming language[75, 84, 96, 97] was developed by Inmos as the programming language for the Transputer processor[152]. It is based closely on the principles of CSP. One of the main features of the occam language is its strong support for parallel programming. Creating a process in occam is as easy as executing sequential code (one simply uses `PAR` rather than `SEQ` to denote a sequence of statements as processes to be executed in parallel). These processes may

be time-sliced on a single Transputer, or may be executed on separate processors in a multiprocessor system.

Processes in occam may only communicate through *channels* (no shared variables are permitted, even on a single processor). Channels are supported by the Transputer hardware in the form of four high-speed bidirectional serial links allowing multiple Transputers to be connected together into a communication network. As in CSP, channel communication in occam is strictly synchronous: the first process that is ready to communicate must wait until the other is prepared to communicate.

The occam language also supports nondeterministic process selection, depending on the status of a *guard*. A guard may be a channel that is ready for communication, a timer, or one of these with an additional boolean condition that must also be satisfied.

Asynchronous Message Passing In asynchronous message passing the two participating processes are not required to be synchronised on the communication operation. The sending process may send a message, and then continue with its execution regardless of the state of the receiving process. In general, the receiving process will block on the receive operation, if there is no message waiting for it.

There are many examples of this form of communication ranging from the message passing interprocess communication primitives built into the UNIX operating system[146] to popular libraries built on the Internet protocols, such as PVM (the Parallel Virtual Machine)[54, 144] and MPI (the Message Passing Interface)[67].

PVM The *Parallel Virtual Machine* (or simply *PVM*) project began in 1989 at Oak Ridge National Laboratory for internal use. In 1991 version 2 was released for general use, and version 3 followed in 1993. PVM is freely available for use on many hardware platforms: UNIX workstations and servers, specialised parallel processors (such as those manufactured by Sequent, Cray and Convex) and PC's (Macintosh and Intel, with UNIX and Windows operating systems). One of the strengths of PVM is the fact that it is designed to work with heterogenous networks of processors. It effectively organises such a collection of machines into a virtual parallel computer (hence the name, *Parallel Virtual Machine*). PVM is responsible for all communication, data conversion and process scheduling in such a system. PVM interfaces are available for FORTRAN, C, and C++, and work is being done to provide Java support[47].

PVM provides a library of operations for creating new processes and enabling communication between them. Processes are identified by means of a system-defined "task

identifier" (or *TID*). Processes may also be grouped together. Communication is asynchronous[2], and may make use of the following operations:

- send
- blocking receive (with optional timeout)
- nonblocking receive
- multicast send (to specified recipients)
- broadcast send (to a recipient group)

Messages may be received in a number of ways (for example, input may be restricted to messages from a specific process). A process can also check whether a message is available without receiving it. Message ordering is guaranteed by PVM.

PVM also supports a number of high-level operations, such as barrier synchronisation, and global maximum and sum operations.

MPI The *Message Passing Interface* (*MPI*) is an international project to provide an agreed standard for writing message passing programs. The standard is maintained by the *MPI Forum*, an open group with representatives from many companies and organisations[105]. Language bindings are provided for FORTRAN, C, and C++, and work is being done on versions for Java[25]. The MPI-1 standard was released in February 1994, and the MPI-2 standard in April 1997.

MPI provides very flexible message passing facilities that are asynchronous by default, but also support synchronous modes. Processes are simply identified by non-negative integer values. The sending process must explicitly identify the recipient. However, a receiving process may choose either to specify a particular sender, or use a "wildcard" to accept any incoming message. The message space can also be partitioned using "tags" attached to messages (the recipient can receive messages with a specified tag if this is so desired), or using "communicators" (groups of processes that can communicate with each other in various specified ways). Broadcast communication is also possible, using the communicator concept.

[2] An option is provided to support non-buffered message transmission, in which case a send operation may block.

MPI-2 added a number of more advanced features, including process creation and management, "one-sided communications" (where one process specifies all of the communication parameters—both for the sender and the recipient), extended collective operations based on communicators (e.g. "all-to-all" communication), external interfaces (for building new abstractions on top of MPI), and powerful parallel I/O mechanisms.

Geist *et al* provide a good summary and comparison of PVM and MPI in [55]. Essentially MPI provides a more flexible and richer set of communication mechanisms. In general, these are more efficient than the equivalent features of PVM. The strength of PVM lies in its support for heterogenous systems. PVM also provides better facilities for resource and process management.

Other Models

This section discusses the BSP and RPC models of concurrency.

The Bulk Synchronous Parallel (BSP) Model The Bulk Synchronous Parallel (BSP)[150] approach to parallel processing can be viewed as a variation on message passing. Processes in a BSP program work independently on local data during the course of a "superstep". During this processing stage they may initiate requests for data, or updates to data held on other processors. At the end of a superstep all the processes synchronise their activities using a barrier synchronisation mechanism. At this point the requests for data and updates to data stored on other processors are handled. Once the data exchange is completed, the next superstep commences.

This model is similar to Linda in that there is a high degree of decoupling. In this case it is the communication and synchronisation that are decoupled, preventing deadlock and related problems. Furthermore, the barrier synchronisation point at the end of each superstep provides a natural checkpoint which simplifies debugging (a notoriously difficult problem in parallel and distributed programming). One of the advantages cited for the BSP model is that it helps the programmer to avoid the burdens of handling memory management, communication and synchronisation issues[150]. Of course, this is also an advantage of the Linda model.

One of the notable features of the BSP model is that it is not tied to a specific hardware or network architecture, but could be implemented on any system ranging

from a shared memory multiprocessor to a network of workstations. Similarly, BSP systems can be implemented on top of other communication and synchronisation methods such as message passing. As decoupled communication is one of the main features of Linda, and barrier synchronisation is easily implemented using the Linda primitives, it should, in principle, be a simple task to implement a BSP system using Linda as the underlying mechanism.

Remote Procedure Call (RPC) Remote procedure call allows one process to execute a procedure on a remote machine. This requires transmitting any parameters required to the remote host, then waiting for the completion of the computation and the return transmission of the results of the computation. This usually requires the cooperation of a remote process that must be prepared to accept the remote procedure call. Most languages that support this form of communication and coordination do so in way that syntactically resembles the usual function or procedure calling mechanisms for transfer of control within a process.

Remote procedure call can be seen as a form of message passing. In this view, a message is sent, and the sender may only continue once a reply is returned from the recipient. This view, while it may be helpful in some situations, tends to obscure the value of RPC as a more abstract form of communication.

Ada One of the best known examples of RPC is the *extended rendezvous* mechanism in the Ada programming language[15]. In Ada a *task* (i.e. a process) may declare that it accepts remote *entries*. This allows another concurrent task to call the entry, passing parameters to it (which in Ada may be *in*, *out* or *in/out* parameters). This model is synchronous, so processes are temporally coupled. However, the naming of the processes is asymmetrical (the calling process must identify the callee, but the callee requires no knowledge of the caller), so the processes are only partly spatially coupled.

CORBA With the advent of object-oriented languages, the idea of remote procedure call has been generalised to remote *method* calling. The best known example of this is probably CORBA (Common Object Request Broker Architecture)[106]. CORBA encompasses a number of standards produced by the Object Management Group (OMG), with the aim of supporting interoperability between distributed enterprise applications. The claim is made that:

> Using the standard protocol IIOP, a CORBA-based program from any vendor, on almost any computer, operating system, programming language, and network, can interoperate with a CORBA-based program from the same or another vendor, on almost any other computer, operating system, programming language, and network. [106]

As with many such standards produced by large committees, the OMG has been responsible for the production of several new acronyms to describe aspects of the CORBA system:

IIOP is the *Internet Inter-ORB Protocol*, a network protocol used by CORBA to support the communication requirements over TCP/IP.

ORB: an *Object Request Broker*. This is the component of CORBA that acts as the mediator (or *broker*) between a client application and the remote service which it wishes to access. It effectively isolates the client program from the details required to perform the communication.

IDL: the *Interface Definition Language* used to specify the interface for a service which may be used by remote clients.

Naming and trading services are supported to allow remote services to be located and contacted. All of this has been provided with a view to the importance of issues such as load balancing, resource control, and fault tolerance. CORBA implementations are available for many programming languages—C, C++, Java, COBOL, Smalltalk, Ada, Lisp and Python, to name a few.

2.2 Concurrent Programming in Java

The Java language offers, as standard features, a number of mechanisms for concurrent programming. In addition there are a number of libraries that provide alternative mechanisms. Several of these are considered in this section.

2.2.1 Concurrency in Java

The Java language includes support for multi-threaded applications and for communication and synchronisation between threads. This is based on the "monitor"

concept[63]. This provides protection for any Java object that may be accessed by multiple threads. In order to prevent simultaneous access to critical sections of code it is possible to enclose them in "synchronised blocks", using the `synchronized` keyword. Alternatively, an entire method may be marked as synchronised, thus allowing only one thread at a time to execute the code in the body of the method. In order to provide for the efficient coordination of cooperating threads, this basic synchronisation mechanism is extended to allow a thread to be suspended in a synchronised block or method by using the `wait` method. A thread that has been suspended in this way can later be resumed by another thread executing the `notify` method (or the `notifyAll` method) in a synchronised block or method belonging to the same object. The semantics of these operations are rather loosely defined, and implementations may thus be prone to problems such as starvation[153].

As an example of the use of these features, consider the class shown in Program Segment 2.2. This shows the use of the monitor facilities in Java to provide a shared data structure (a queue of messages for transmission across a network connection), which is accessed using a multiple writer/single reader mechanism. This is a slightly simplified version of one of the classes in the eLinda system. The `IOQueueEntry` class (not shown in the code) defines a simple data structure with two public fields:

`next` a reference to the next entry in the queue

`msg` a reference to the network message to be transferred

Full details of the concurrency mechanisms in Java, together with examples of their use can be found in [91].

2.2.2 Networking, Message Passing and RPC Mechanisms

This section first considers the standard communication mechanisms that are provided with the Java system by Sun Microsystems, then considers additional libraries that have been developed by third parties.

Standard Communication Mechanisms

There are two levels of standard communication available in Java. At a lower level of abstraction there is support for common network protocols and facilities for transmitting data and objects across such networks. At higher levels of abstraction there are

```
class IOQueue
  { // Head and tail of the queue of messages.
    private IOQueueEntry head, tail;

    // Add a message to the queue.
    public synchronized void addMessage (Message m)
      { if (tail != null)
          { tail.next = new IOQueueEntry(m);
            tail = tail.next;
          }
        else // First entry
          { tail = new IOQueueEntry(m);
            head = tail;
          }
        notify();
      } // addMessage

    // Remove a message from the queue.
    public synchronized Message getMessage ()
      { while (head == null) // No messages
          try
            { wait();
            }
          catch (InterruptedException e)
            {}
        Message tmp = head.msg;
        head = head.next;
        if (head == null) // Last Entry
          tail = null;
        return tmp;
      } // getMessage

  } // class IOQueue
```

Program Segment 2.2: An Example of a Shared Data Structure in Java

remote object access facilities that provide a form of object-oriented remote procedure call.

Networking and Serialisation The most basic form of communication mechanism provided by the Java language is the provision of classes for Internet Protocol (IP) networking. These classes are part of the `java.lang.net` and `java.lang.io` packages. They provide a Java programmer with the ability to create TCP/IP socket connections and to transmit information freely between any two participating processing nodes. The data transmission itself is handled by the same stream mechanisms that are used for other forms of input and output in Java.

Additionally, there is the ability to make use of UDP datagrams for communication, but this is not a reliable form of communication and is less commonly used. One useful facility of the UDP mechanisms is the ability to send broadcast messages across a network using the *multicast* feature. Java also provides classes supporting other, higher level Internet protocols, such as HTTP (the Hypertext Transfer Protocol), but these are typically not used for distributed processing applications due to their high degree of specialisation for other purposes.

An important feature of the language is the ability to transmit objects across a network connection, using the *serialisation* facilities. This provides for a very powerful and flexible form of communication between Java processes. Serialisation allows the sender to convert an object into a form that can be transmitted as a stream of bytes. When the receiver reads the stream of bytes it can be *deserialised* to recreate the original object.

There are some restrictions on serialisation, due mainly to the fact that serialising an object involves the recursive serialisation of the entire graph of objects to which the original object has references, either directly or indirectly. Certain classes of objects cannot sensibly be serialised. Generally, these are objects that involve some form of dynamic behaviour, such as a thread of execution, or an input/output stream. The presence of such an object in the object reference graph will cause the entire serialisation process to fail. This can be prevented by marking such references as `transient`, in which case the serialisation mechanisms will not attempt to serialise the transient object (with the obvious side-effect that no information is then communicated across the network for that object).

RMI and CORBA The Remote Method Invocation (RMI) mechanism provided in Java[6] is built on the network communication and serialisation facilities. It provides a form of remote procedure call, specifically implemented in an object-oriented manner. RMI provides a library of classes and supporting software that permit a Java program to transparently call methods of a remote object across a network. This is implemented using the TCP/IP networking mechanisms and serialisation to handle parameter passing and the returning of results. However, all of the details of setting up the network connections and handling the communication are abstracted away from the programmer by the RMI system. RMI also provides a directory facility (the *registry*), allowing processes that wish to make use of a remote service to locate the service. This provides an element of spatial decoupling of the processes.

An alternative form of remote procedure call is also supported by Java, through the provision of CORBA facilities (see Section 2.1.3). Like RMI, CORBA allows a Java program to invoke the methods of remote objects. However, the CORBA standard was designed to provide language-independent method calling, and so in this case the remote object need not be a Java object, but could be implemented in any language that provided a CORBA interface (indeed, it may not even be a true object, but some form of procedure implemented in a non-object-oriented language).

Additional Libraries and Other Approaches

The increasing popularity of the Java language has led to the development of many libraries supporting various forms of communication and synchronisation mechanisms for parallel and distributed programming. This section discusses a number of these libraries and other approches to concurrent programming in Java.

CSP Libraries Two similar libraries of communication and concurrency mechanisms have been developed, based on the CSP model discussed in Section 2.1.3. These are Communicating Sequential Processes for Java (JCSP) developed at the University of Kent at Canterbury[154], and Communicating Threads for Java (CTJ) developed at the University of Twente[68].

JCSP was developed by members of the occam/CSP community arising from their disquiet with the monitor concept in Java, which is demonstrably unsafe[153]. JCSP provides Java programmers with the advantages of the CSP model of concurrency, namely simplicity, scalablility and verifiability. It supports the CSP synchronous chan-

nel communication mechanisms, and also provides for process creation and management in the CSP style. An interesting aspect of this work is the use of CSP to model the Java monitor mechanism, leading to a formal proof of the equivalence of the CSP and JCSP channel mechanisms[155].

CTJ is intended to provide reliable and efficient mechanisms specifically for real-time applications in Java. It is very similar to JCSP and also supports the CSP synchronous channel communication, process creation and management mechanisms.

Other Message Passing Libraries PVM and MPI were introduced earlier in this chapter (see Section 2.1.3) as two very widely used message passing libraries for networks of workstations. Both of these systems have been ported to Java.

The Java implementation of PVM (called *JPVM*) was developed as experimental prototype[47]. It is not interoperable with other implementations of PVM. However, it does offer a number of extra or improved features, such as thread-safety and efficient, direct message transmission.

On the other hand, the Java implementation of MPI, *mpiJava*, is a Java library providing access to the native MPI communication mechanisms[25]. This makes use of the Java Native Interface (JNI) to bridge to the native code MPI library on platforms that support both MPI and Java, such as Solaris, Linux and Windows NT.

Hyperion The developers of Hyperion have taken a very interesting approach to supporting distributed applications in Java[5]. They have developed a Java environment that treats an entire collection of processors as a single JVM. This involves distributing the threads in the Java program across the processing nodes for execution, and also providing an emulation of the Java memory model. Hyperion is implemented on a run-time system called PM2, which supports distributed threads and provides a tailorable "distributed-shared memory". The system also makes use of a native code compiler. This converts Java bytecodes to C code, which is then compiled to machine language (however, currently only a subset of the Java 1.1 libraries is supported). The bytecode-to-C converter also applies some optimisations to the code as it performs the translation. Their preliminary results show promise for the potential of this approach.

2.2.3 Linda Implementations in Java

Recently, two implementations of Linda in Java have been developed by major computer companies. The first of these is JavaSpaces[51], developed by Sun Microsystems as part of the Jini project. The second is TSpaces[76], developed by IBM's alphaWorks research division. Each of these systems is introduced in the following sections, and then covered in more detail in later chapters where they are compared with eLinda.

In addition to these commercial products there are other research projects that have developed Java implementations of Linda. These are also summarised below.

JavaSpaces

JavaSpaces[51] is a complex product and relies heavily on a number of other technologies from Sun. It forms part of the Jini system for networking heterogeneous systems[141] and so makes extensive use of the Jini API. Network support is provided by the Java RMI (Remote Method Invocation) protocol[6]. Furthermore, distribution of classes to clients is handled by the standard Internet hypertext protocol (HTTP). This means that before a JavaSpaces application can be started the following set of services must be running:

- a web (HTTP) server (a minimal one is provided with the Jini/JavaSpaces release)
- an RMI activation server (part of the standard RMI software bundled with Java)
- a Jini lookup service (alternatively the RMI registry service can be used, but this is discouraged as support for this option may be discontinued by Sun in the future)
- a Jini transaction manager
- a JavaSpaces server

Most of these services (and any application programs) also require extensive setting of command line parameters, further adding to the overall complexity of using JavaSpaces. Applications are also required to run a security manager, whether security checking is required or not. A typical command line required to run a JavaSpaces application is as follows:

```
java -Djava.security.policy=D:\JavaProgs\policy.all
  -Doutrigger.spacename=JavaSpaces
  -Dcom.sun.jini.lookup.groups=public
  -Djava.rmi.server.codebase=http://host/space-examples-dl.jar
  -cp D:\JavaProgs\space-examples.jar;D:\JavaProgs\classes
  sun.applet.AppletViewer worker.html
```

JavaSpaces supports the following operations (the names differ from the original names used by Yale, but essentially the same set of functions is provided): `write` (output), `read` (non-destructive input) and `take` (destructive input), and also predicate input forms: `readIfExists` and `takeIfExists`. Tuples can be created by the programmer from any classes that implement the Jini `Entry` interface. Only the public fields of these classes that refer to objects are considered for matching purposes (i.e. private fields are ignored, as are fields of the primitive data types).

Tuples are transmitted across the network using serialisation. However, JavaSpaces uses a non-standard method of serialisation in that only public fields of classes are serialised. Furthermore, multiple references to the same object cause multiple copies to be serialised[3]. Matching of tuples with anti-tuples (called *templates* in JavaSpaces) is done using byte-level comparisons of the data, not the conventional `equals` method. Matching can make use of object-oriented polymorphism for matching sub-types of a class.

The Linda programming model is extended in JavaSpaces to provide support for commercial applications in two ways:

Transactions A number of tuple space operations can be grouped into a single transaction. A transaction can be "rolled back" if any one step cannot be completed successfully.

Leases Tuples can be given an expiry time after which they will automatically be removed from the tuple space.

While both of these extensions are relevant to commercial software, they do not address any of the performance issues or other problems inherent in the original Linda model.

A centralised tuple storage approach is used and this may become a performance bottleneck in large systems. JavaSpaces does not provide a preprocessor. The tuple

[3]The standard serialisation mechanisms detect this situation and serialise the object only once.

Feature	JavaSpaces	Yale Linda
"Rich typing" (tuples as classes)	Yes	No
Objects (with methods)	Yes	No
Matching subtypes	Yes	No
Fields must be objects	Yes	No
More than one tuple space	Yes	No
Leases	Yes	No
Transactions	Yes	No
Has `eval`	No	Yes

Table 2.1: Comparison of JavaSpaces and Yale Linda

space operations are simply implemented as methods using the standard parameter-passing mechanisms and object-oriented features of Java (i.e. inheritance, polymorphism and interfaces).

Table 2.1 summarises the differences between the original Yale Linda model and JavaSpaces. As can be seen from this table, JavaSpaces provides almost all the functionality of the original Linda model, with the exception of the `eval` operation, and, as has already been noted, this is not an essential part of Linda. Furthermore, JavaSpaces provides considerably extended functionality, especially in areas such as transaction support and leases, which are important for commercial applications. The other differences arise mainly from the object-oriented nature of JavaSpaces, which was not an important consideration when the original Yale model was proposed.

TSpaces

In IBM's words TSpaces is intended as "the common platform on which we build links to all system and application services"[79]. Within this grand vision they identify "Tier 0 devices" (i.e. systems smaller than traditional desktop or laptop machines, such as PDA's, embedded processors, etc.) as a particular area of interest[170].

The implementation of TSpaces is simple and elegant, particularly in comparison with JavaSpaces—all that is required is that a single server process be running on the network. The server makes use of a textual configuration file, and provides a useful web interface for monitoring and configuration purposes. Applications wishing to make use of the TSpaces service need only know the network hostname of the computer running the server. The following example shows the command line equivalent to the previous

JavaSpaces example on page 31[4]:

```
java -Djava.security.policy=D:\JavaProgs\policy.all
  -cp D:\JavaProgs\tspaces_client.jar;D:\JavaProgs\classes
  sun.applet.AppletViewer worker.html
```

TSpaces supports a large number of operations. The basic Linda operations are provided (again different names are used): `write` (output), `read` (non-destructive, predicate input), `take` (destructive, predicate input), and non-predicate input forms (`waitToRead` and `waitToTake`). Note the rather confusing way in which the basic forms of the input operations are predicates and the alternatives are blocking. There is a `delete` operation that will simply delete a matching tuple from the tuple space without returning it to the application. There are also operations for the input and output of multiple tuples: `scan`, `countN`, `consumingScan`, `deleteAll`, `multiWrite` and `multiUpdate`. There are a number of operations that specify tuples by means of a "tuple ID" rather than the usual associative matching mechanisms: `update`, `readTupleById` and `deleteTupleById`. There is also the `rhonda` operator, which performs an atomic synchronisation and data exchange operation between two processes. Lastly there is an "event registration" mechanism. This allows a process to request notification when a certain tuple is written to the tuple space or deleted from it.

TSpaces transports tuples across the network using the standard Java object serialisation mechanisms and TCP/IP sockets. Tuples are simply objects that consist of a number of `Field` objects (or `FieldPS` objects which preserialise to a byte array to allow the server to work with unknown classes). Wildcard or formal values for anti-tuples are specified by `Field` objects containing a class type (e.g. `String.class`), rather than a data value. TSpaces thus restricts tuples to containing objects, not primitive values, for matching purposes. Matching is performed using the standard `equals` method, and, in some cases, the `compareTo` method, specified by the `Comparable` interface. Matching can be done using so-called "indexed tuples". In this case the fields may be named, ranges of values may be included in the matching process, and AND and OR operations may be specified. These features may all be used in combination. It is also possible to perform matching on XML[5] data contained in tuples.

[4]The setting of the security policy file here is only required due to the use of the Java AppletViewer. Unlike JavaSpaces, this is not a general requirement.

[5]Extensible Markup Language, a specification for structured documents produced by the World Wide Web Consortium[167].

Feature	**TSpaces**	**Yale Linda**
Subclassable tuples	Yes	No
Matching subtypes	Yes	No
Fields *must* be objects	Yes	No
More than one tuple space	Yes	No
Transaction support	Yes	No
Event notification	Yes	No[6]
Extensible matching	Yes	No
Tuple expiration	Yes	No
Has `eval`	No	Yes

Table 2.2: Comparison of TSpaces and Yale Linda

Tuples may have an expiration time set, providing similar functionality to the lease mechanism in JavaSpaces, and there is support for transactions. Furthermore, access control is provided for tuple spaces. This is based on user names, passwords and groups, and provides a level of control similar to that of the UNIX file access control mechanisms. TSpaces also provides persistence for the tuple spaces.

Like JavaSpaces, TSpaces does not provide a preprocessor. The current implementation of TSpaces also makes use of a centralised server model, which may become a performance bottleneck.

Table 2.2 summarises the differences between the original Yale Linda model and TSpaces. In general it can be seen that TSpaces provides considerably more functionality than the original Yale Linda proposal, again with the exception of the `eval` operation.

Adding New Commands New commands can be added to the TSpaces system relatively easily. This ability, together with the rich set of operations supported and complex matching criteria described above, provides a facility similar to the extended features of eLinda. These features will be compared and discussed in more detail in Chapter 6. This section presents a brief overview of the mechanisms for adding new commands to TSpaces.

The implementation of TSpaces makes use of a number of layers of software (a simplified view of this is shown in Figure 2.6[78]). At the lowest level the tuples

[6]While the original Linda model does not provide any form of event notification, it is not difficult to emulate for writes, using threads and the standard `rd` operation.

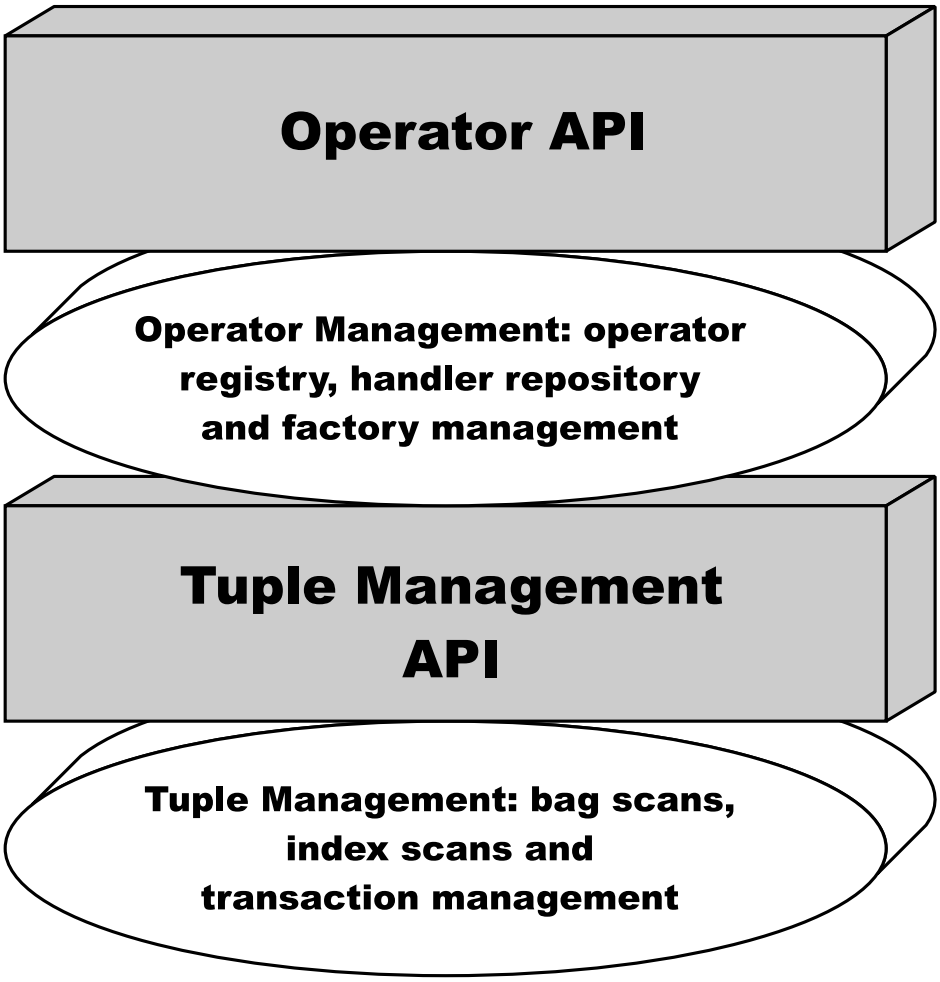


Figure 2.6: TSpaces Internal Structure

themselves are stored in a form of database. This may be an actual database product (such as IBM's DB2), or simply some form of data structure in the computer's main memory. Above this is the tuple management layer, which handles the retrieval of tuples from the tuple space database. Above this layer (and accessed through a well-defined API) is the operator management level. This is comprised of a number of "factory" objects arranged in a list. The factories are responsible for creating "tuple handlers" for each command that is passed to the tuple space. If a factory does not recognise a particular command then it is passed down to the next factory in the list.

Users with appropriate permission levels can add new factories and handlers to the system dynamically, providing a great deal of flexibility. However, this is a complex process from a programmer's perspective, as has also been noted by Foster *et al*[49]. Some of the complexity could perhaps be handled by providing classes with methods to

Operation	Yale Linda	JavaSpaces	TSpaces
Output	`out`	`write`	`write`
Input	`in`	`take`	`waitToTake`
Input (copying)	`rd`	`read`	`waitToRead`
Predicate input	`inp`	`takeIfExists`	`take`
Predicate input (copying)	`rdp`	`readIfExists`	`read`

Table 2.3: Summary of Names for Linda Operations

automate the installation and initialisation of the new tuple handlers, but this would have to be done by the writers of the new factories and command handlers.

Comparison of JavaSpaces and TSpaces

Both Sun and IBM have based their systems closely on the original Linda system from Yale and so there are obvious common characteristics. Unfortunately, there seems to be little agreement on naming conventions for the basic Linda operations, as can be seen from the summary in Table 2.3.

Since the intended market for both JavaSpaces and TSpaces is the same, they share some common characteristics, such as support for transactions and leases/expiration. However, while Sun has otherwise followed the original Yale Linda programming model very closely, IBM has chosen to extend the model considerably.

Other Research Projects

This section provides a brief overview of a number of research projects that have developed implementations of Linda in Java.

XMLSpaces XMLSpaces is designed to support the use of XML data in tuples[149]. It is based on TSpaces, and considerably extends the XML support already provided by TSpaces. The `Field` class used by TSpaces for the fields in tuples is subclassed to create a class called `XMLDocField`. This new class overrides the matching method used by TSpaces to provide matching on the basis of the XML content of the field. The matching is performed by a method of the anti-tuple that can be provided by the application programmer. This results in a great deal of flexibility for XML matching operations. A number of matching operations are currently supported, including the

use of XML query languages. Of the many query languages available for XML[169], the two currently supported by XMLSpaces are XQL[121] and XPath[168].

XMLSpaces further extends TSpaces by supporting a distributed tuple space model, rather than the centralised model used by TSpaces. The distributed tuple space support is provided in a flexible and tailorable way, allowing different methods for the distribution of tuples to be used, and selected dynamically when an application starts. Currently only centralised and partial replication[7] strategies are supported. A subset of the basic TSpaces operations is augmented with distributed versions that take into acccount the distribution of the tuple space (e.g. `distributedWrite` and `distributedWaitToTake`).

CO^{3}PS CO^{3}PS stands for "Computation, Coordination and Composition with Petri net Specifications"[72, 73]. This builds on the usual coordination model of a concurrent system, where the responsibilities for computation are handled by the host language, and for coordination by the coordination language, as exemplified by Linda. Petri nets are used in CO^{3}PS for the specification of the computational aspect of a system. CO^{3}PS is implemented in Java, and makes use of agents[43, 65].

The coordination model used in CO^{3}PS is based closely on that of Objective Linda[86, 87], discussed in Section 2.1.2. As such it shares the approach of Objective Linda in allowing the method for the matching of tuples to be overridden. The main application of this in CO^{3}PS is to support the introduction of *non-functional requirements*. The developers of CO^{3}PS distinguish two phases of application design:

1. The *logical phase*, which concentrates on the programming logic—the functional requirements of the system.

2. The *non-functional phase*, in which issues such as efficiency, load-balancing, security, etc. are taken into account.

This approach might be summarised as: "first get it working, then get it working well". In order to support this technique, they make use of a *reflective architecture*, i.e. an architecture that permits the designer to reflect on the behaviour of the system, and to adapt it, without affecting the interaction with clients. The developers of CO^{3}PS go

[7]Where subsets of servers hold consistent copies of subsets of the tuple space—similar to the strategy described in [46].

to great lengths to explain that this should be done without impacting on the semantics of the coordination operations.

A further unique feature of CO^{3}PS is the introduction of *composition* as a third aspect of the behaviour of a concurrent system, orthogonal to computation and coordination. Composition refers to the view of an application as a configuration composed of a collection of agents. These agents may recompose themselves into new configurations as needed during the execution of the application. The following set of operations is provided to support this dynamic reconfiguration:

- Creating agents.
- Terminating agents.
- Creating *object spaces* (i.e. tuple spaces).
- Deleting object spaces.
- Allowing agents to *expose* object spaces, making them available to other agents.
- Allowing agents to attach to and detach from object spaces.

Java-Linda Java-Linda is a student project at Yale University intended as the first step in developing a Java version of Piranha[138]. As such it provides a subset of the features of the original Linda system (for example, there is only partial support for the `eval` operation, no preprocessor, etc.). TCP/IP is used for communication in Java-Linda, with a simple, centralised server.

As there are no extensions to the original Linda model present in Java-Linda, it is of little interest, except for the novel way in which it implements the associative matching mechanism. Any object can be used as a tuple in Java- Linda. This would seem to pose some difficulties for the Java-Linda system that needs to perform matching operations on these objects. The solution to this problem that has been adopted is that the Java-Linda system interrogates the structure of the objects in order to extract the fields and perform matching. This has been done using the serialised version of the objects, which includes a considerable amount of information about the structure of the serialised object. This metadata is parsed to extract the information about the structure of the object and this is then used to guide the matching process. The matching itself is done on the bytes in the serialised form of the object. This process is described as "tedious and time-consuming"[138], but is convenient and elegant for

application programmers. This should be contrasted with JavaSpaces, which appears to do very simple byte-level matching on objects.

An alternative approach that would provide a similar level of convenience for programmers would be to use the *Reflection API*[8] facilities provided in Java. This is a set of classes and methods that can be used to determine the structure and contents of a object. The use of the reflection mechanism would in some respects be a far better approach, as it is not reliant on the structure of the serialised form of an object, which may be liable to change in the future (indeed, the serialisation mechanisms have been the subject of ongoing development by Sun[142]). Accordingly, the use of the Reflection API is likely to be more resilient in the future. However, the reflection mechanisms may be less efficient than the approach adopted by Java-Linda, which may also explain why it was not used by Sun for JavaSpaces.

Mobile Coordination The work on mobile coordination performed at the University of Cambridge is very similar in many respects to I-Tuples, which was discussed previously, in Section 2.1.2. However, the motivation behind the unique features of this project are very different[124]. Like I-Tuples the idea of mobile coordination is to move the processing of tuple space operations from the application processing node to the server. However, in mobile coordination this was done with a view to enhancing fault tolerance, rather than increasing performance. In fact, the mobile coordination mechanism is presented as an alternative to the transaction mechanisms found in the current commercial implementations of Linda. Despite this, the performance results reported for the Java implementation of mobile coordination show that in many cases it can lead to increased performance.

2.3 Summary

This chapter has discussed concurrent programming, focussing specifically on the Linda approach, and on the concurrency mechanisms available in Java. Of particular interest have been the various attempts to extend the matching facilities of Linda. This subject is addressed directly by eLinda, which includes a very powerful and flexible approach to solving the problems in this area. This is covered in Section 3.2.1, and in greater detail in Chapter 4.

[8] Application Programming Interface.

The following chapter introduces the eLinda system, describing the extensions to the original Linda model.

Chapter 3

eLinda

This chapter introduces the eLinda system and its novel features. In essence, eLinda provides an application programmer with a greater degree of control over the matching and communication mechanisms than the original Yale Linda model. This enhances the functionality of the original Linda model, and makes the performance issues more explicit, with a view to aiding predictability and improving performance. Support is also provided in eLinda for multimedia applications, simplifying application development in this important area.

This chapter begins with a discussion of the implementations of the eLinda system that were developed for testing these concepts. The extensions themselves are then described in the second part of the chapter.

3.1 Implementation Issues and Rationale

Three different implementations of the eLinda system were developed to allow for the exploration of various communication and system configuration issues. The first version (referred to either as eLinda, or as eLinda1 where necessary to prevent ambiguity) makes use of a fully distributed tuple space model, where any tuple may reside on any processing node. This implementation was the main focus of much of the research, as the difficulties presented by a fully distributed approach ensured that all possibilities were considered in the development of the extensions. Unless otherwise noted, references in this document to eLinda are to this version.

The second implementation of eLinda, called eLinda2, uses a centralised tuple space model (as is used in TSpaces and JavaSpaces, and many of the current research projects). This allowed for a comparison to be made between the fully distributed ap-

proach and a centralised approach. This version does not exploit all of the distinctive features of eLinda.

Finally, the third implementation (eLinda3) uses a centralised tuple space model, but with local caching of certain tuples at the processing nodes. The reason for this and the nature of the caching will become more clear in Section 3.2.3. This version can be considered as a variation of eLinda2, rather than a completely different implementation.

All three of these implementations also make use of internal partitioning of the tuple space. This means that tuples with a distinct structure (i.e. where the type signature can be used to categorise tuples into disjoint sets) are stored separately to improve the efficiency of searching the tuple space. In all three versions the programmer is also provided with mechanisms to use multiple, named tuple spaces. These are again stored separately by the system.

3.1.1 The System Architecture

The overall structure of the base eLinda system is shown diagrammatically in Figure 3.1. This diagram illustrates a relatively small configuration with only three distributed application components (labelled "App" in the diagram).

The diagram illustrates the system in terms of Java Virtual Machines (JVM's). These may be running on separate network hosts or may be allocated to common hosts in any desired configuration, allowing for simple load-balancing between hosts. The communication between virtual machines makes use of the TCP/IP network protocol, implemented by the Java `Socket` class.

Within a single virtual machine the communication between separate threads of execution is implemented using shared data structures. In this case, the standard Java monitor synchronisation mechanisms are used to protect access to these data structures.

As indicated in the diagram, the applications may be multithreaded (one virtual machine is excuting two application threads), and in general there may be many more application threads and processes than shown in this diagram. The Tuple Space Managers (TSM's) are implemented by a Java class, which is responsible for controlling access to the tuple space. The "Comm" components shown in the diagram are lightweight threads that are responsible for handling the communication and buffering requirements.

The "Directory" process is used to direct network messages between the cooperat-

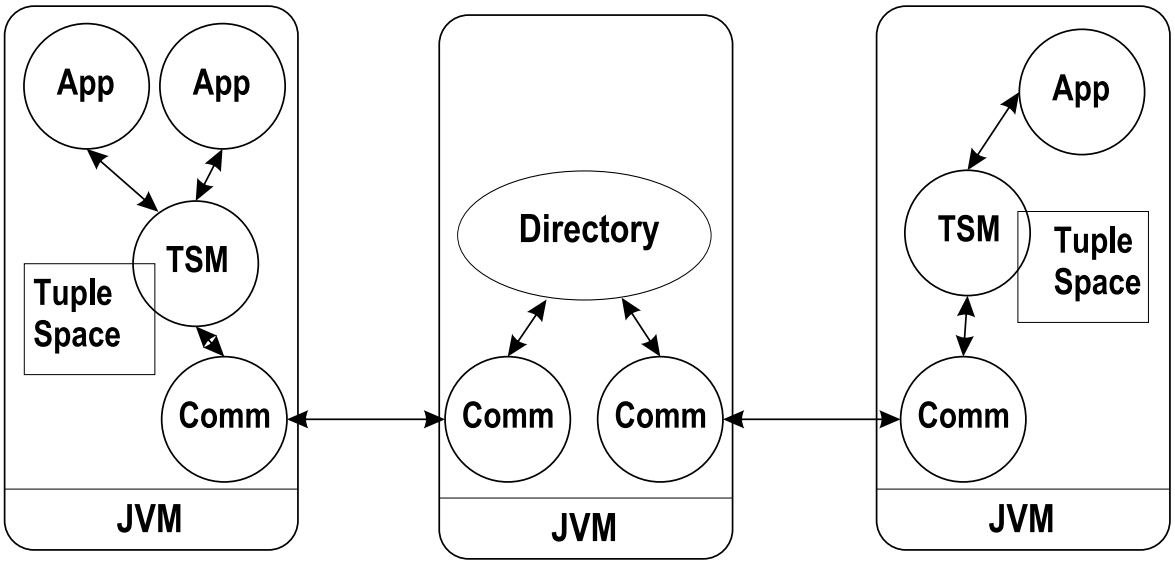


Figure 3.1: The Structure of the eLinda System

ing processes. This centralises some of the communication and network configuration issues—each Tuple Space Manager needs to connect to just one Directory handler (in general there will be more than one). The management of the Directory handlers is undertaken by a global master process (not shown in the diagram). This allocates new Directory handlers as they are required by the system and manages load balancing between them. The communication configuration is discussed in greater detail below.

The centralised approach in eLinda2 allows for a design that is considerably more simple, as shown in Figure 3.2. As already mentioned, the structure of eLinda3 can be considered as a simple variation on this and is illustrated in Figure 3.3.

Communication

The decision to use the TCP/IP protocol for communication between the Java Virtual Machines was taken for efficiency reasons. Several other higher level communication mechanisms exist, notably RMI (Remote Method Invocation) and CORBA, both of which support remote objects and method calls. While the use of these higher level protocols would simplify the design and implementation of eLinda, they both rely on TCP/IP for their underlying communication needs, and introduce their own additional processing overheads.

In designing the communication mechanisms for eLinda there were a number of factors to be taken into account. Firstly, the number of TCP/IP connections that are

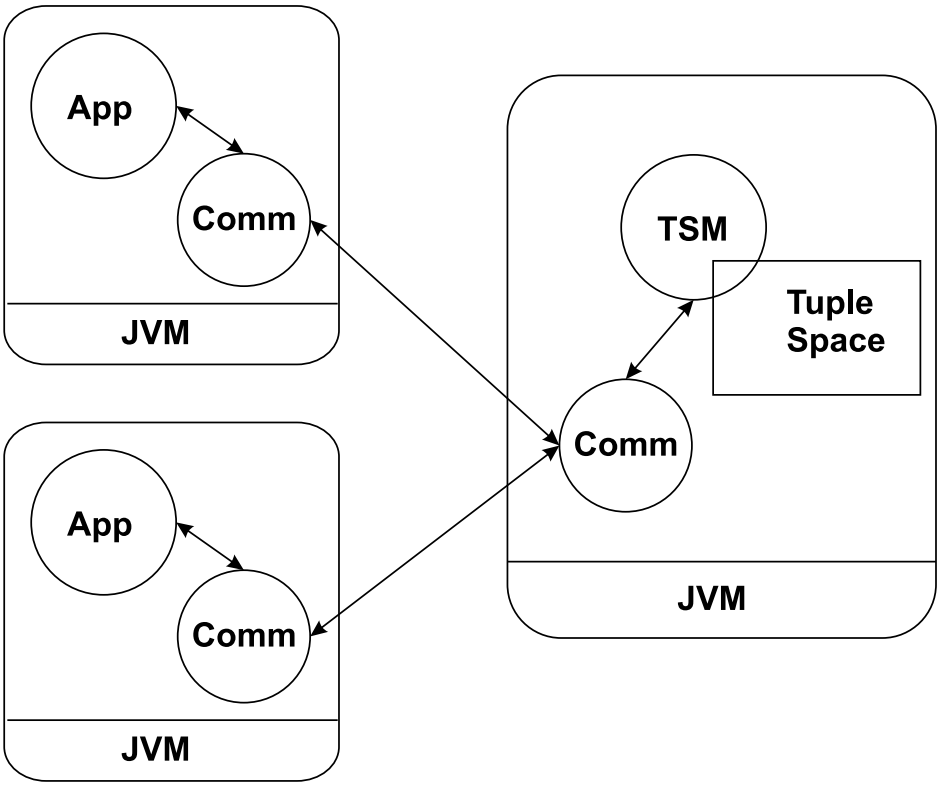


Figure 3.2: The Structure of the eLinda2 System

open at any time should be minimised to prevent excessive use of system resources. At the same time, the desire to provide optimal performance leads towards increasing the number of interconnections to minimise the number of "network hops" that a message must take, and hence minimise the communication time. Accordingly a number of different communication configurations were considered. A theoretical analysis of the options that were contemplated is shown in Table 3.1, which describes each option and shows the number of network connections required, and the number of hops or steps that must be taken by a message sent from one processing node to another. The relative advantages and disadvantages of the different schemes are also shown.

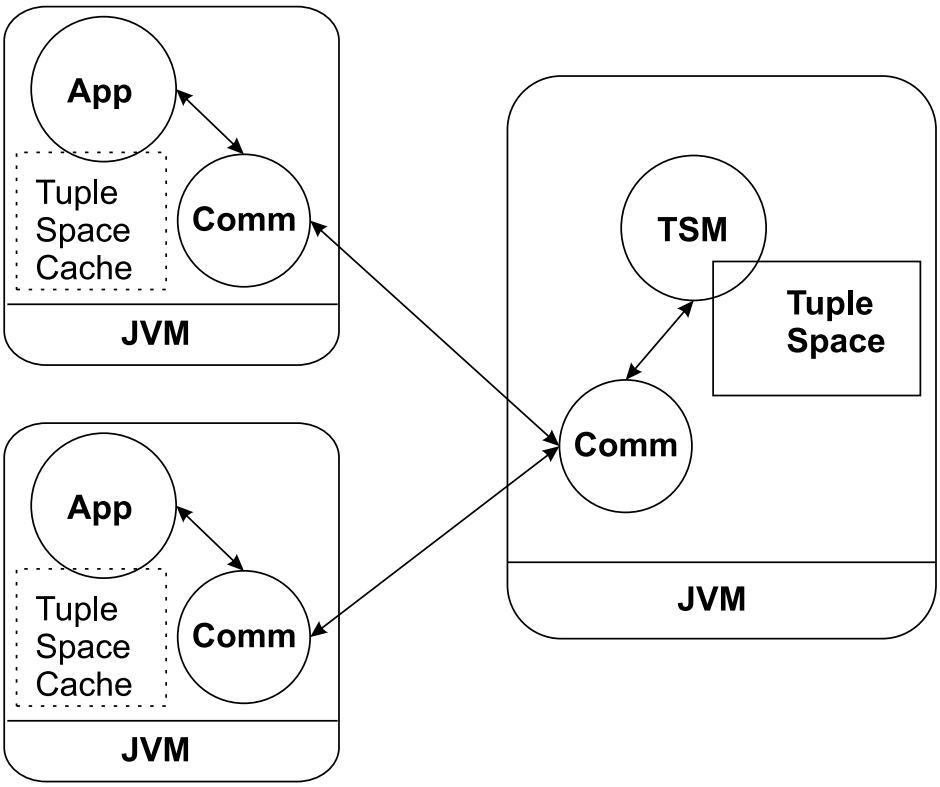


Figure 3.3: The Structure of the eLinda3 System

Type	**Description**	**Number of network connections**	**Number of hops**	**Advantages**	**Disadvantages**
Full Interconnect	All TSM's connected to each other	$\frac{n}{2}(n-1)$	1	Short paths	High number of connections
Central Directory	All TSM's connected to one Directory	n	2	Short paths, simple	Central bottleneck
Directories connected in a ring	A closed ring of m Directories	$(n+m)$	$2 \ldots m+1$	Fairly simple	Long paths when m is large
Hierarchy of Directories	One "Super Directory" connected to m Directories	$(n+m)$	2 or 4	Fairly simple	Partial central bottleneck and longer paths
Fully Connected Directories	All Directories connected to each other	$n + \frac{m}{2}(m-1)$	2 or 3	Short paths	Fairly high number of connections

Where: n = number of processing nodes
m = number of Directories

Table 3.1: Communication Options for eLinda

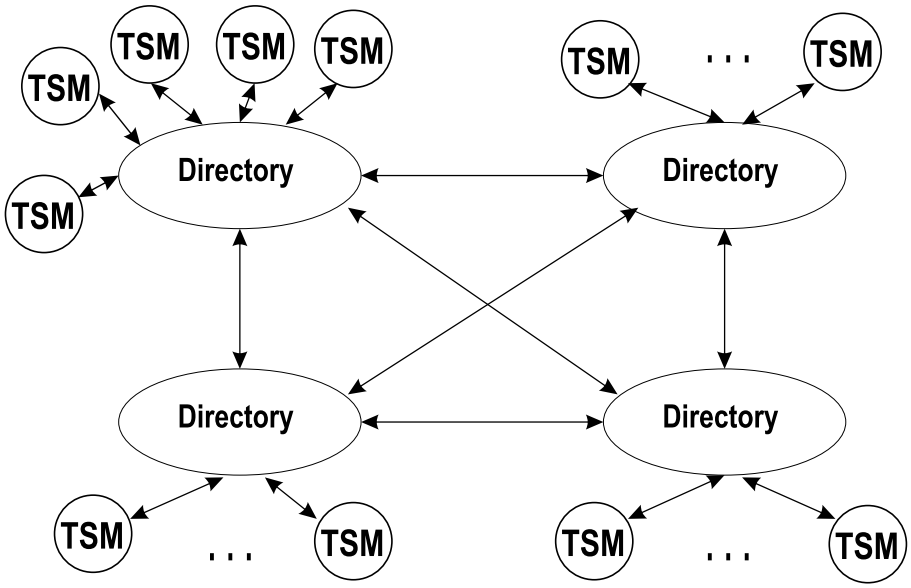


Figure 3.4: The Communication Structures in eLinda

Communication in the Base Version of eLinda The approach that was adopted for eLinda1 is the last one in Table 3.1, that is, using fully connected directories. This means that each processing node is connected to one directory handler, while all the directory handlers are directly connected to each other. This is illustrated in Figure 3.4 for four directory handlers. This communication configuration gives a good balance between minimising the number of network connections used, while at the same time minimising the number of steps that a message must take. The number of network connections is given by: $n + \frac{m}{2}(m-1)$, where n is the number of processing nodes, and m is the number of directories. The number of hops a message must take is either 2 (for messages between nodes connected to the same directory) or 3 (for messages between nodes connected to different directories). The crucial parameter in determining how efficiently this configuration works is the ratio of processing nodes to directories. Clearly, in order to minimise the number of network connections, the number of directory processes should be kept low. This will also reduce the average number of hops per message, as it increases the number of processing nodes per directory, and hence the probability that communicating processing nodes are connected to a common directory.

Communication in the Other Versions of eLinda Both eLinda2 and eLinda3 make use of a centralised tuple space server. Communication between client application processes and this server is handled by means of TCP/IP connections, giving rise to a

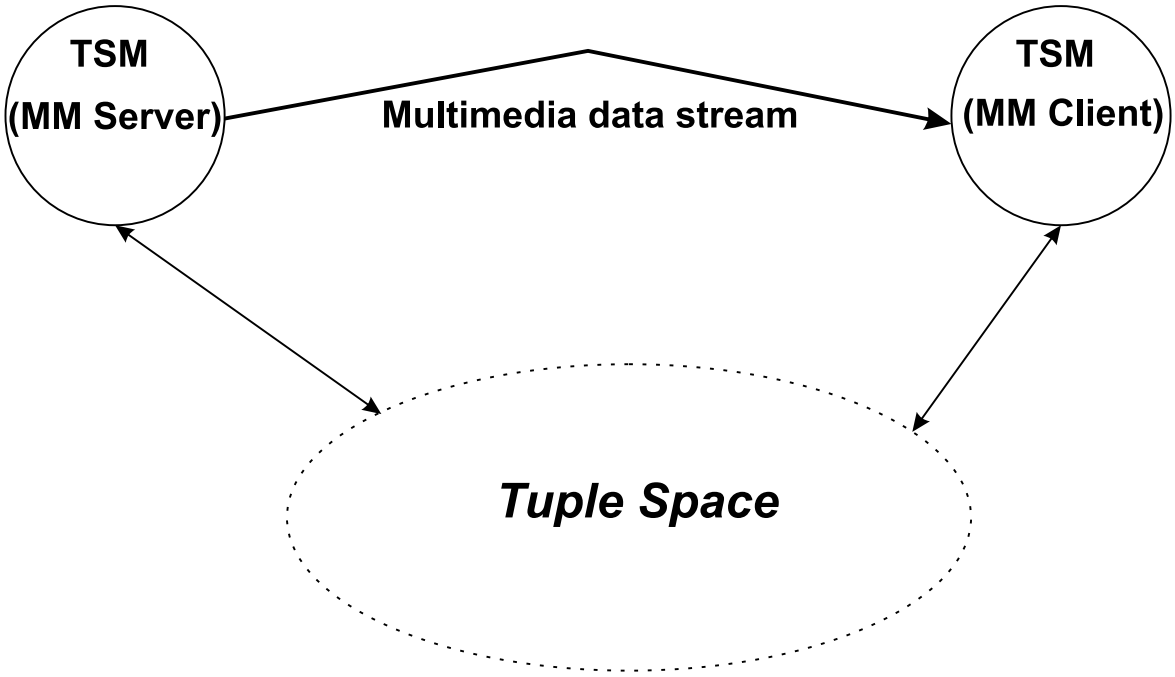
simple star network topology.

UDP multicast communication is used in eLinda3 to broadcast tuples directly to all participating nodes efficiently. The broadcasting of tuples is handled by the central tuple space server, as is control of the deletion of broadcast tuples. The UDP protocol is connectionless and does not provide guaranteed delivery of messages. However, it does provide the "multicast" feature for efficient broadcasting of information to many network nodes at once. In eLinda3 the possibility of broadcast communication failure is not important as the data is still stored centrally and thus any node that does not receive a broadcast message will find the correct response when it queries the centralised tuple space. All that is lost in this case is the potential performance benefit of using a locally cached tuple.

3.1.2 Multimedia Support

The multimedia support in eLinda is provided by a client-server subsystem, independently of the tuple space communication mechanisms. When a multimedia resource is created by an eLinda process a multimedia server thread is started locally on the processing node (if it is not already running). The multimedia resource object can then be added to the eLinda tuple space and may be picked up by any other process that is part of the eLinda system—this is then the *client* from the perspective of the multimedia subsystem. When the client requests that the multimedia resource be presented to the user, the eLinda system transparently connects to the multimedia server running on the original processing node and starts to transfer the actual multimedia data across the network. This data is then passed into the usual Java Media Framework presentation facilities. This is possible largely due to the open, extensible architecture of the Java Media Framework, which allows new forms of data sources to be incorporated into the Java Media Framework very easily. The process of handling a multimedia data stream can be illustrated as shown in Figure 3.5.

On the multimedia server side, when a multimedia resource is requested by a client process, a check is made to see if the resource is a local disk file or is some other form of Java Media Framework datasource. In the former case it is opened as a normal random access file for efficiency, otherwise the Java Media Framework data access mechanisms are used by the multimedia server to read the data for transmission to the client. In either case, the multimedia server and the eLinda multimedia client software attempt to presend data and buffer it on the client machine to minimise the problems associated



Note: The Tuple Space is shown here as an abstract entity, representing any of the three implementations.

Figure 3.5: The Configuration of the Multimedia Subsystem

with network latency and bandwidth limitations.

The network communication for the multimedia client-server system is performed either using the TCP/IP connection-based network mechanisms, or using the support for RTP (Real-time Transport Protocol)[133, 134] provided by the Java Media Framework.

3.1.3 Comparison with Other Linda Systems

The major distinguishing factor between the implementations of eLinda, TSpaces and JavaSpaces is that eLinda1 makes use of a fully distributed tuple space, whereas the other two systems use a centralised approach (of course, eLinda2 and eLinda3 also make use of a centralised approach). While distributing the storage of the tuples introduces a certain amount of overhead, it also provides for a degree of fault-tolerance, and may contribute to efficiency, especially when dealing with very large networks of processing elements with high communication latencies. In such situations a centralised storage model may become a bottleneck for the system.

There is not a lot of further detail available about the implementation of JavaSpaces. However the internal architecture of TSpaces is discussed at some length in the TSpaces Programmer's Guide[77], particularly the way in which new command handlers are managed (this was described in Section 2.2.3). A detailed comparison of this feature with the extended matching facilities in eLinda is given in Chapter 6.

3.2 The Extensions

This section discusses the three extensions made to the original Yale Linda model in eLinda. These are the Programmable Matching Engine (PME), the support for distributed multimedia applications, and the provision of explicit broadcast communication. As the Programmable Matching Engine is the main feature of eLinda it is discussed in greater detail in Chapter 4—only a brief overview is given here.

Usually a program source preprocessor is used with Linda systems to translate the Linda operations into the actual forms used by the host language. A preprocessor is not provided for eLinda, and so all interaction with the system takes place using the standard Java method calling and parameter passing mechanisms. However, the examples given in this chapter all make use of a simplified syntax (referred to as the *ideal syntax*), such as might be supported by a preprocessor. The features required of

a preprocessor are discussed in Appendix B.

3.2.1 The Programmable Matching Engine

A programmable matching mechanism (referred to as the *Programmable Matching Engine*, or simply the *PME*) is provided for use with the Linda input operations, allowing the use of more flexible criteria for the associative addressing of tuples. For example, in dealing with numeric data one might require a tuple that has a value which is "close to" some specified value (possibly using fuzzy set membership functions). As another example, in a graphical context, where the tuples represent the objects in an image, one might require a tuple corresponding to an object located within a specified area of the image. These operations cannot be performed directly in conventional Linda systems as they use only exact equality of field values for matching.

It may be possible to express such queries using the standard Linda associative matching methods, but such solutions will generally be very inefficient. For example, the application might have to retrieve *all* tuples of the required type, select one of interest and then return the rest to tuple space. This form of solution has a number of serious problems. Firstly, there is the communication overhead for retrieving and returning all the unwanted tuples. This is associated with the second disadvantage: the potential loss of parallelism as other processes are prevented from retrieving these tuples while they are held by the querying process. In order to manage the access to all of the tuples, it may even be necessary to provide barrier synchronisation points, further impacting the degree of parallelism that may be obtained from the system.

Depending on the design of the Linda system there may be further problems. For example, if the tuple space is not centralised, searching for a tuple will require accessing the sections held on a number of processors, further increasing the communication cost.

These situations are handled in eLinda by allowing a programmer to specify a non-standard matching algorithm to be used together with the anti-tuple for any of the Linda input operations (i.e. `in`, `rd`, and their predicate forms). In the Java implementation this is simply done by providing an object that conforms to a specific interface for matchers. This matching algorithm is then able to search through the tuples in tuple space in order to locate a suitable result tuple to return to the requesting process.

The syntax used to specify the matcher that is to be used is `op.matcher`, where `op` is one of the eLinda input operations and `matcher` specifies the customised matching

routine to be used[1]. In addition, the extended syntax `?=` is used to identify which field (or fields) is to be used by the Programmable Matching Engine matching routine. For example, the eLinda statement `in.maximum("point", ?=x, ?y)` specifies an `in` operation using a matcher that will search for the tuple with the maximum value of the x coordinate. Omitting the matcher specification causes the system to use the usual Linda technique of matching for strict equality.

Further details of the Programmable Matching Engine and its implementation in eLinda can be found in Chapter 4.

3.2.2 Multimedia Support

There is an increasing demand for distributed multimedia applications, and so support for multimedia data types was included in eLinda[159]. This was done by building on the facilities provided by the Java Media Framework (JMF)[140].

Tuples in eLinda may contain any of the primitive data types supported by Java (i.e. `int`, `char`, `double`, `float`, `byte`, `short`, `long` and `boolean`) as well as standard Java `String` objects. Furthermore, almost any other Java object[2] may be added to a tuple, although this limits the type checking that can be performed by the eLinda system. In this way the eLinda system attempts to provide the maximum possible functionality for general purpose applications.

A further, new type, `MultiMediaResource`, has been added to the set of data types supported by eLinda. This class acts as a wrapper to the underlying Java Media Framework multimedia resource. In particular, the implementation of the `MultiMediaResource` class provides the support (transparent to the application programmer) for any necessary buffering of data, fetching or streaming of multimedia data across the network, etc. (as described in Section 3.1.2).

As a simple example, in order to search for and present a multimedia resource the following code might be written using the eLinda multimedia extensions:

```
MultiMediaResource m;
if (inp("Movies", "Chicken Run", ?m))
  m.play();
else
  System.out.println("\"Chicken Run\" is unavailable");
```

[1] Again, note that this is the *ideal syntax*, not the actual Java code.

[2] The only restriction is that the object *must* be serialisable.

Support is currently provided for both stored multimedia resources and "live" (or streaming) resources. The design of this part of the system relies heavily on the multimedia facilities implemented in the Java Media Framework. This is a Java library that defines support for various differing types of audiovisual media, including stored and streamed media (for example, live video-conference feeds).

3.2.3 Broadcast Communication

The original Linda model provides a single form of output operation: `out`. In e-Linda, two types of output operation are provided to reflect explicitly a choice of optimised communication strategies. These are a "point-to-point" mechanism (using non-replicated data) and a "broadcast" mechanism (using replicated data). This contrasts with the existing Linda mechanism where data is written to tuple space using `out`, but is then read using one of the two basic input methods: `in` or `rd` (or their equivalent predicate forms). In effect, the use of `in` implies a form of exclusive point-to-point communication, in that one process places a tuple into tuple space, which is then removed by another. Similarly, the use of `rd` suggests a form of shared, or broadcast (read-only), communication, as several processes may obtain copies of the tuple in this case.

To allow the programmer to take advantage of this behaviour, a new output operation, called `wr`, has been added in eLinda. In a distributed tuple space implementation the `wr` operation will broadcast the tuple throughout the processor network, whereas `out` will place only a single tuple in the tuple space. These mechanisms provide the programmer with the necessary facilities to express shared, read-only access to data (`wr-rd`), or exclusive, delete/modify access (`out-in`).

It should be noted that this usage is *not* enforced by the system. For example, it may occasionally be necessary to update data that is otherwise shared in a read-only fashion. In such a case a tuple would be broadcast using `wr`, accessed using `rd`, and then removed for updating using `in`. This would result in a performance penalty as all the duplicated broadcast tuples would have to be deleted. Similarly, `rd` may be used to retrieve a tuple placed in tuple space using `out`, but a search of all the processors with stored tuples may be required to locate it. The overall effect of this behaviour is that the semantics of the `wr` operation are the same as those of the `out` operation—the only differences are in terms of performance and network load.

The `wr` operation works exactly the same as `out` in the case of eLinda2. In eLin-

da3, the tuples that are written to tuple space using `wr` are those that are broadcast to all processing nodes and cached locally. This allows processing nodes to access such tuples without the need for any network communication. If such a tuple is removed from the tuple space (using `in` or `inp`) then the central server in eLinda3 adjudicates the deletion.

3.3 Summary

This chapter has described the implementations of the eLinda system, and introduced the novel features which it supports. The next chapter expands on the brief overview of the Programmable Matching Engine that was given in Section 3.2.1.

Chapter 4

The Programmable Matching Engine

As the provision of flexible, distributed matching operations is the main feature of eLinda, this chapter provides a detailed description of the Programmable Matching Engine (PME), together with strategies for its implementation and use. The different types of matchers that might be used in practice are discussed, as are limitations on the use of the Programmable Matching Engine. Finally, the Programmable Matching Engine is compared with mobile agent technologies.

While the eLinda system provides the support for using programmable matching algorithms and currently includes a number of examples of such matchers, it is unrealistic to expect that all possible matchers could be provided with such a system. It is envisaged that any practical or commercial implementation of the Programmable Matching Engine concept would include a library of commonly required matchers, written in such a way as to provide a useful set of generic matching facilities. More specialised matchers would have to be written as part of the development of the application for which they were required. Such matchers could then be added to the library of existing matchers for future use. It is also possible that writing specialised matchers could become a service provided by an entity separate from the application development team. The subject of writing matchers is discussed in more detail in Section 4.2.

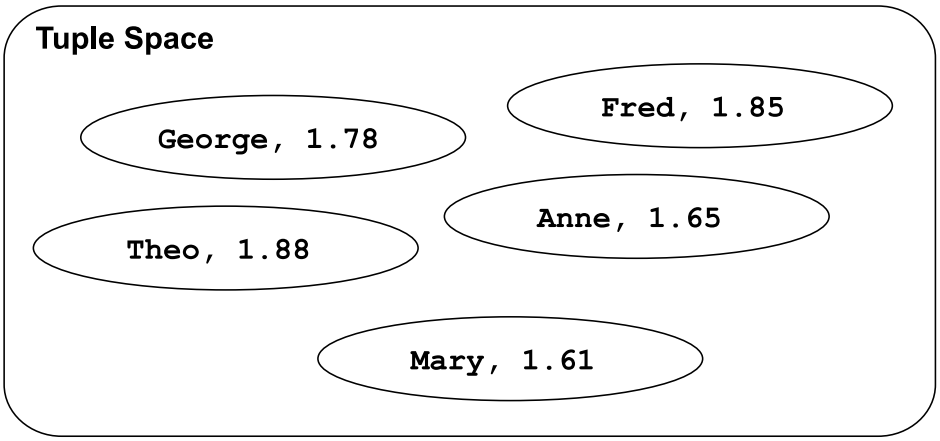


Figure 4.1: Example of People and Heights

4.1 The Use of the Programmable Matching Engine

The Programmable Matching Engine was briefly introduced in Chapter 3. Before discussing it in more detail we will consider a simple example of its use: the provision of a matcher to locate the tuple with the maximum value of a particular field. This is only one of the simplest of the many possible applications of the Programmable Matching Engine, and serves as a useful introductory example.

4.1.1 An Example of the Use of the Programmable Matching Engine

To illustrate the problems associated with the standard approach to matching in Linda systems, we will assume that we have a set of tuples with a type signature of *(string, double)*, corresponding to people's names and heights, as illustrated in Figure 4.1. We will assume that a particular application needs to determine the name of the tallest person.

Before considering the way in which this problem is solved with the use of the Programmable Matching Engine we will consider how it could be done in a conventional Linda system. In such a system the matching of tuples is restricted to exact equality for specified values. If we knew the exact value of the maximum height we could retrieve the corresponding name as follows:

```
in(?name, 1.88)
```

```
double maxHeight = 0.0
String maxName = null
Vector list = new Vector()
while (inp(?name, ?height)) // Retrieve all tuples
  if (height > maxHeight) // Biggest so far
    if (maxName != null)
      add maxName, maxHeight to list
    maxName = name
    maxHeight = height
  else // Shorter than current max
    add name, height to list
while (! list.isEmpty())
  remove name, height from list
  out(name, height)
```

Program Segment 4.1: Finding a Maximum Value Using Linda

However, in many practical situations it is extremely unlikely that the exact value of the required field will be known. If this is the case then the problem is clearly more difficult to solve. A possible solution is shown in Program Segment 4.1. This works by removing *all* of the tuples from the tuple space, locating the maximum value and then returning all the remaining tuples back to the tuple space, as outlined in the previous chapter.

In eLinda, using a programmable matcher, this problem can simply be solved as follows:

```
in.maximum(?name, ?=height)
```

From the application programmer's perspective, the use of the programmable matcher has simplified the program considerably. The logic of the application is also far more clearly discernible in the eLinda form. Even more importantly, as the next section explains, the network load is considerably reduced through the use of the Programmable Matching Engine.

Obviously this example assumes that the eLinda system already has a suitable "maximum matcher" available. As discussed in the introduction to this chapter, it is likely that any production system employing the Programmable Matching Engine concept would be supplied with a library of matchers for common problems such as finding maxima and minima, performing case-insensitive string matches, etc.

4.1.2 The Interaction between Matchers and the Tuple Space

The specific problem considered above (i.e. finding the tuple with a maximum value) is handled efficiently in eLinda1 by distributing the matching engine so that network traffic is minimised. For example, in searching for the maximum value, each section of the tuple space is searched locally for the largest tuple and only that tuple is then returned to the originating process. The originating process then selects the largest of all the replies received, effectively selecting the global maximum from among the local maxima found by each of the processing nodes. In this way the total network traffic is minimised in comparison to the solution required for a conventional Linda system.

In the other two eLinda implementations, which use a centralised tuple space, the programmable matching approach also dramatically reduces the volume of network traffic. In these cases, the matcher is sent to the server node that holds the centralised tuple space and the search is performed locally there, with only the result tuple being transmitted back to the processing node that originated the query.

More generally, in a distributed implementation (such as eLinda1), when a Programmable Matching Engine matcher is invoked, it will usually need to broadcast the request to the other participating processing nodes in the network and then commence a search of the locally held tuples. The remote matchers will simply search their own local sections of the tuple space and then return the results to the matcher on the originating node where the overall result can be determined. During this process other operations on the tuple space may be taking place that may affect the result.

Due to the temporal decoupling of distributed processes interacting through tuple space, the Programmable Matching Engine approach may result in outcomes that are apparently inconsistent. Locally, each matcher has exclusive access to the local tuple space until the matching process is completed. This means that there is a locally well-defined "snapshot" view of the tuple space taken at the moment when the local matcher commences execution. However the global view of the tuple space is not so well defined, as the network transmission time for the distribution of the request means that the various local tuple spaces are searched without any form of global locking. This leads to what is, at best, a "fuzzy snapshot" view of the global tuple space. This behaviour is intentional and underscores the temporal decoupling of the underlying Linda paradigm. It is also important to note that it is no different from the situations that might arise for the possible solutions using conventional Linda systems (such as that given in Program Segment 4.1).

In circumstances where this behaviour is unacceptable, the application can provide any required synchronisation of the participating processes to ensure that the tuple space is in a consistent and unchanging state for the duration of a query. Semaphore or barrier synchronisation algorithms are easily implemented in Linda and can be used for this purpose[51]. Of course, this assumes that the application has control over all processes acting on the tuple space. If there are other processes that do not form part of a common application accessing the tuple space, then the job may be much harder. However this is not the case in general, especially as eLinda provides for explicitly named, separate tuple spaces, and further distinctions between sections of tuple spaces are enforced using the type signatures of tuples.

It should be noted that there are circumstances in which the use of programmable matchers may result in synchronisation problems, such as deadlock. This is due to the fact that the local tuple spaces are locked while programmable matchers are active in them. Writers of new matchers and application programmers using them should be aware of this possiblity and take steps to control the synchronisation if necessary.

4.1.3 Simple Matchers

The example presented in Section 4.1.1 of a matcher to locate a tuple with the maximum numeric value in some field is one of the simplest. Other related matchers might provide the ability to locate the minimum value of some field, or that with the value closest to some desired value. In each of these cases the procedure required is much the same as was outlined previously. The originating matcher would first distribute the query to all other participating processing nodes and then commence a search of the local tuple space. On completion of the local search the originating matcher would wait to receive results from the remote matchers and then select a tuple from all the candidates as the overall result. If the operation is a destructive one then the originating matcher is also responsible for returning the unwanted remote tuples to the tuple space as they will have been removed from the other sections of the tuple space by the remote matchers.

Some queries may be even more simple than those discussed above. For example, an application may require a tuple with a field that is in some way "close to" to a specified value (i.e. within a specified range). If such a request can be satisfied by *any* tuple that has a field sufficiently close to the required value (i.e. not necessarily the *closest* value) then it may not be necessary to distribute the query if the local section of the tuple space contains a suitable result. In this case the matcher could be developed

such that it first searches the local tuple space, and then distributes the query only if no result is found locally. Furthermore, the first successful reply from any remote matcher can be accepted as the result without waiting for any further responses from other remote matchers. The Programmable Matching Engine easily supports this form of matching in addition to that described previously.

4.1.4 Aggregate Matchers

In addition to simple matchers where the result is a single tuple selected from the set of available tuples, the Programmable Matching Engine also allows for matchers that provide an application with the ability to retrieve result tuples that provide an aggregated or summarised view of a set of tuples. Simple examples of this kind of operation would be matchers that provided the total of all the values of some field of a number of tuples, or the average value, or a simple count of the number of tuples meeting some criteria. In each of these cases the result is not a tuple that was originally present in the tuple space, and is referred to as a *pseudo-tuple* to distinguish it as such.

This can be taken even further, as a programmable matcher can return a pseudo-tuple that includes an entire collection of tuples. The application may need to take care in such cases as the type signature of the result may not be the same as that of the anti-tuple. Commonly, the resulting pseudo-tuple in this situation simply contains a single object: a Java `Vector`, array, or other suitable compound data structure.

Aggregate matchers can be used with either destructive or non-destructive input operations (i.e. `in` or `rd` respectively). The only difference is whether the production of the resulting pseudo-tuple has the side-effect of removing the actual tuples that are being aggregated from the tuple space. The predicate forms of the input operations (i.e. `inp` and `rdp`) can also be used with these matchers. In such cases, the failure of the predicate would indicate that there were no suitable tuples available for aggregation.

The development of such matchers would be similar to that of simple matchers that require a global view of the tuple space (such as the matcher described above to find the maximum value of some field). In a distributed tuple space environment (such as eLinda1) they would need to broadcast the query to all participating processing nodes, search the locally held tuples, and then integrate the local and remote results to form the overall result.

4.1.5 Limitations of the Programmable Matching Engine

There are some limitations to the kinds of matching operations that are supported by the Programmable Matching Engine. Notably, some matchers may require a complete global view of the tuple space. An example of such a matcher would be one that was required to return the tuple with the *median* value of some field. This example will be used to explore a number of aspects of this problem.

Firstly, we need to consider how a conventional Linda application would solve this problem, without the use of the Programmable Matching Engine. As in the earlier example, this would require the application to retrieve *all* the tuples meeting the specification, using `inp`. Once this was done the application could sort the tuples and locate the one with the median value. All the other tuples would then need to be returned to the tuple space (including the result tuple if the overall effect is to be non-destructive, i.e. a `rd` or `rdp` operation).

When using the Programmable Matching Engine a very similar approach could be adopted. In this case, the originating matcher could distribute the request, retrieve the local tuples and then collect the remote tuples returned by the remote matchers (i.e. *all* of the tuples found by the remote matchers). It could then sort the tuples, extract the median and return the unwanted tuples to the tuple space. The only advantage in this case, but a potentially significant one, is that the application itself is simplified by moving the complexity into the matcher.

However, another approach can be adopted that may be beneficial where the tuples under consideration are large in size. In this case, the originating Programmable Matching Engine matcher could distribute the request and then extract the local tuples. The remote matchers could also retrieve their locally held tuples. However, rather than returning the complete tuples to the originating matcher, the remote matchers could extract *only* the field required for the determination of the median value. A list of these values could be returned to the originating matcher for use in determining the median value. Once this was established, then the necessary result tuple could be fetched from the processing node that held it. All other tuples would then be returned to tuple space by the local matchers. In this way the network traffic between the processing nodes is minimised: only the essential "summary" information is transmitted to the originating matcher and then only the one required result tuple. If the tuples are large in relation to the size of the field being used to determine the median value then this saving in network bandwidth could be considerable.

To summarise, while there are situations where the use of the Programmable Matching Engine might not be ideal, they are handled no less efficiently than if the application handled them directly. Furthermore, there may still be opportunities to minimise the network bandwidth required, which would not be possible in conventional Linda systems. In any case, the use of the Programmable Matching Engine will simplify the development of the application, where a pre-written matcher is available.

4.1.6 Other Uses of the Programmable Matching Engine

The examples given in this section have focussed mainly on numeric examples. These have been used for the discussion as they are simple, easily explained and easily understood. However, it would be incorrect to believe that the Programmable Matching Engine was only useful for numeric problems—it is just as applicable to textual or other problems. Some examples that capture the flavour of the previous ones, but emphasise the generic nature of the Programmable Matching Engine are:

- A string matcher could match string fields using some alphabetic measure of "closeness", or even approximate homophonic matching.
- A spatial matcher could compare two fields, taken to be x and y coordinates to locate a tuple corresponding to a point in some two-dimensional space (or, equivalently, in three or more dimensions).
- A matcher could be written to locate tuples with fields corresponding to a date or time in some range of temporal values.
- A matcher could make use of "fuzzy logic" to locate a tuple with some associated degree of certainty of its suitability.
- A matcher could be written to select a tuple at random from some subset of the available tuples[1].
- A matcher could be written to extract XML data from a tuple and perform complex matching operations based on this (providing an equivalent to the XML support in TSpaces and XMLSpaces).

[1]While it is not required of a Linda system, the eLinda system stores tuples using a FIFO queuing technique for fairness.

4.2 Writing Distributed Matchers

Several aspects of the writing of Programmable Matching Engine matchers have already been discussed above, such as the outlines of the algorithms to be followed by various types of matchers in sections 4.1.3 and 4.1.4. This section explains in some detail what is required of a matcher, and what support is provided by the eLinda system. A complete treatment of this subject is given in Appendix A. The purpose of the following overview is to highlight the simplicity of the task of developing new matchers. If a programming technique like use of the Programmable Matching Engine is to become widely employed, it should be as simple as possible for application programmers to use.

4.2.1 Requirements for Matchers

The eLinda system requires that a programmable matcher provides certain methods. A particularly useful feature of Java for the specification of these methods is the *interface* mechanism provided by the language. This allows the eLinda system to specify the methods that must be provided by the matcher in order to implement the matching process, without dictating the implementation details. The interface for the programmable matching engine is shown in Program Segment 4.2.

In essence this means that the writer of a new matcher must provide the two methods specified in the `ProgrammableMatcher` interface (i.e. `matchList` and `match`) in some class. This class may or may not be an integral part of a particular application. An object of this class can then be passed to the eLinda system and be used by it to perform the necessary customised matching operation.

The two methods are used by the eLinda system in the following ways: the `matchList` method is always called first, as soon as the input operation using the matcher is executed. It is given an iterator, which allows it to work through all of the potentially matching tuples currently in the tuple space. If no suitable tuple is found the method can return `null`, and the eLinda system will automatically handle any blocking that is required (i.e. for non-predicate forms of input). If the input operation is blocked, then the other method required by the interface (i.e. `match`) is used to check each new tuple that is subsequently added to the tuple space, to see if it is a possible match.

```
public interface ProgrammableMatcher
  { /** This method compares one anti-tuple with a list of
      * tuples.
      * This is needed for all operations, but particularly
      * for non-blocking operations (i.e. rdp and inp).
      */
    public Tuple matchList (AntiTuple a, TupleIterator t)
        throws MatcherException;

    /** This method compares one anti-tuple with one tuple.
      * This is needed only for blocking operations (i.e. in
      * and rd) where tuples may come in one at a time (of
      * course, it can be used by the matchList function).
      * If a matcher is never to be used in a blocking
      * operation this can simply return false.
      */
    public boolean match (AntiTuple a, Tuple t)
        throws MatcherException;

  } // interface ProgrammableMatcher
```

Program Segment 4.2: The `ProgrammableMatcher` Interface

4.2.2 Support Functions

Writing a matcher is not a trivial operation, and a number of functions are provided by the eLinda system to support the development of new matchers. These may be divided into two categories: the methods provided by the tuple and anti-tuple objects themselves, and the methods provided by the tuple space manager (TSM) for communication between processing nodes and manipulation of the tuple space.

In addition, there is an exception class, `MatcherException`, which may be used by a matcher to indicate failure of the matching process (for example, if an attempt is made to perform some arithmetic matching operation on non-numeric fields). As can be seen in the `ProgrammableMatcher` interface in Program Segment 4.2, both of the methods required by the interface may throw this type of exception.

Tuple and Anti-Tuple Operations

There are two methods of the `Tuple` class that are of interest when writing new matchers. The first of these allows a matcher to tell if the tuple "belongs" to the local processing node. This may be necessary if a matcher needs to exclude non-local tuples (for example, to prevent double counting of broadcast tuples). The second method returns the type signature of the tuple. This can then be used to determine the type of each field, which may be useful to a matcher (for example, to detect attempts to perform arithmetic matching operations on non-numeric fields).

In the eLinda system the `AntiTuple` class is a subclass of the `Tuple` class, and so inherits both of the above methods, while adding a number of others. These additional methods allow a matcher to determine the exact nature of the matching operation in various ways. Firstly, the matcher can distinguish between the four different input operations (i.e. `in`, `rd`, and the corresponding predicate forms). It can also determine which fields are wildcards, and which are the specified "matching fields" (i.e. those marked with `?=` in the ideal syntax). There are also a number of methods that allow a programmable matcher to apply the standard matching algorithm in various ways (e.g. using only certain fields of the tuple, specified by a bit mask).

Communication and Tuple Space Access Operations

Various methods are provided to allow matchers to interact directly with the eLinda system (e.g. retrieving tuples from tuple space, replacing unwanted tuples, deleting local and remote tuples, broadcasting requests to other processors and subsequently

retrieving the results of such requests, etc.). Of course, this interaction allows the programmer to access the tuples in tuple space at a lower level of abstraction than usual, and care needs to be taken to preserve the semantics of the Linda tuple retrieval operations.

The methods available to a matcher for communication and tuple space access are all static methods of the `TSManager` class. They provide the following set of services:

Scatter/gather operations These allow a matcher to broadcast an anti-tuple across the network to remote matchers, and then retrieve the results from the remote matchers. The eLinda system handles all the communication, ensuring that all the results are received, etc.

Broadcasting delete messages This specifies that a tuple or anti-tuple should be deleted from all the distributed sections of the tuple space in which it appears.

Replacing tuples These methods allow matchers to return unneeded tuples to the tuple space. Two variations are provided allowing a single tuple or an entire array of tuples to be returned.

4.2.3 The Complexity of Writing New Matchers

There is no obvious, simple method for determining the level of difficulty of writing a new matcher. The previous sections have outlined some of the steps involved and the support features of the eLinda system to give an indication of the simplicity of this task.

This section presents counts of the numbers of lines of Java programming code for three programmable matchers. As a metric, lines of code are notoriously unreliable. Factors such as the ability of the programmer writing the code, layout and formatting, density of commenting, etc. are liable to vary widely. Accordingly, the measurements below are not presented as proof of the simplicity of writing new matchers, but simply as an approximate indication of the ease of this task.

It should be noted that the code measured here is extensively commented. This was done so that these matchers could serve as examples for the writers of future matchers. The general programming style adopted was a single Java statement per line, with generous use of blank lines to indicate the program structure.

Given these reservations about the use of this metric, the results are shown in Table 4.1. The `TotalMatcher` is an aggregating matcher that returns a total of numeric

Matcher	Lines of Code
`TotalMatcher`	175
`ClosestMatcher`	190
`MinimumMatcher`	193

Table 4.1: Measurements of Code Length for Three Example Matchers

tuple fields. The `ClosestMatcher` finds a tuple with the nearest numeric value for a particular field (or fields) to a specified value (this is the example shown in full in Appendix A). The `MinimumMatcher` finds the tuple with the minimum value of some field (numeric or string).

Further examples of programmable matchers and indications of their complexity are discussed in Chapter 5.

4.3 The Implementation of the Programmable Matching Engine

The support for the Programmable Matching Engine in the eLinda system is easily integrated into the implementation of the standard Linda operations. The various input methods (i.e. `in`, `rdp`, etc.) of the `TupleSpace` class check at the outset of the operation whether a Programmable Matching Engine matcher has been specified with the anti-tuple. If a Programmable Matching Engine matcher is specified then the `matchList` method is called with the anti-tuple and the list of tuples as parameters. If this returns a result tuple then that is returned to the application.

If the input operation is blocking (i.e. `in` or `rd`) and no result tuple is found by the `matchList` method then the anti-tuple is broadcast through the network. In this case, as tuples are added to the tuple space, they will be checked against the anti-tuple by calling the `match` method of the associated matcher.

4.4 The Programmable Matching Engine and Mobile Agents

The concept of *mobile agents* has recently been popularised as a mechanism for handling distributed data processing problems[43, 65, 80, 104]. The programmable match-

ing engine concept has some features in common with such mobile agents. Effectively, a customised matcher written for the eLinda Programmable Matching Engine is a form of mobile entity that is distributed on a network to find a matching tuple (or tuples). This provides the same performance advantage as for mobile agents, namely that the need to move large amounts of data across the network is minimised by doing the processing where the data is to be found rather than centrally. In the mobile agent scenario this is done by sending an agent out on the network to locate and/or process the data. In the Programmable Matching Engine the matching and retrieval operations of Linda are distributed to all the elements of the distributed tuple space.

The main features of mobile agents can be summarised as follows (this list of features is taken from Conde[43]):

Mobile Mobile agents are capable of moving from one processing node to another, and as such, the code, data, execution state, and travel itinerary that comprise the agent move together.

Autonomous Mobile agents can act independently of each other, and do not require any centralised control structures.

Disconnected Operation Mobile agents can still function in the face of network failures. In this case, an agent that needs to move will suspend its execution until the network service is restored and then continue.

Asynchronous Mobile agents effectively form independent execution threads in the system.

Local Interaction Mobile agents can interact with local data (possibly other mobile agents, or "stationary" objects, or simply static data).

Parallel Execution Many mobile agents may be active in the system at any time (including multiple agents active on a single processing node).

If this list is compared with the features of eLinda, and especially the Programmable Matching Engine, the differences and similarities become apparent. Firstly, while a mobile agent generally has a single instance active at any one time, and this instance moves around the network of processing nodes, in eLinda a distributed matcher may be simultaneously active on all of the processing nodes. Furthermore, Programmable Matching Engine matchers are generally not completely autonomous, but centralised, with

the originating matcher controlling the operation (broadcasting the request, collating the results, etc.). Programmable Matching Engine matchers will also be affected by network failures.

Turning to the similarities, the execution of Programmable Matching Engine matchers is based on the temporal decoupling of Linda, and so is asynchronous. Programmable Matching Engine matchers also provide local data interaction (indeed this is one of the motivating factors for their inclusion in eLinda), and parallel execution.

Consequently, a programmable matcher might be viewed as a highly specialised form of mobile agent, sharing a number of the advantages of mobile agent technologies, but executing in a more constrained and specific environment.

4.5 Summary

This chapter has presented the Programmable Matching Engine in detail. To demonstrate the simplicity of the concept while highlighting its power and flexibility, a number of examples of the possible uses of the Programmable Matching Engine have been presented. The interactions between a matcher and the eLinda system have also been discussed in order to show the relatively small and simple interface that a programmer has to master in order to write a new matcher. Lastly, programmable matchers have been compared with mobile agents, demonstrating some common ground with that concept.

The next chapter shows how the extensions in eLinda can be applied.

Chapter 5

Applications of eLinda

This chapter presents a number of example applications that have been developed using eLinda. These applications highlight the power of the eLinda system, and the simplicity that it brings to the development of parallel and distributed applications. As a particular example of the application of eLinda, a parsing algorithm for visual programming languages was parallelised and implemented in eLinda.

In addition to the larger applications presented in this chapter, a number of smaller benchmark and demonstration programs have been developed in eLinda, such as a Mandelbrot fractal generator, simple communication benchmarks, etc.

5.1 A Video-on-Demand Application

In order to illustrate the power of the multimedia support provided by eLinda a demonstration video-on-demand system was developed. This consists of a server application that is used by the supplier of video resources, and a client application that is used by a customer wishing to view this material. As the main focus was on the support for distributed multimedia resources, a number of practical issues such as security, payment verification, etc. were omitted from this application.

5.1.1 The Video Server Application

This program reads in the details of the available videos from a file and then places these details into a tuple space called "videos". The tuples contain the name of the video supplier, the title of the video, a unique key, and the cost of viewing the video. The server then waits for a tuple to be placed into a tuple space called "requests"

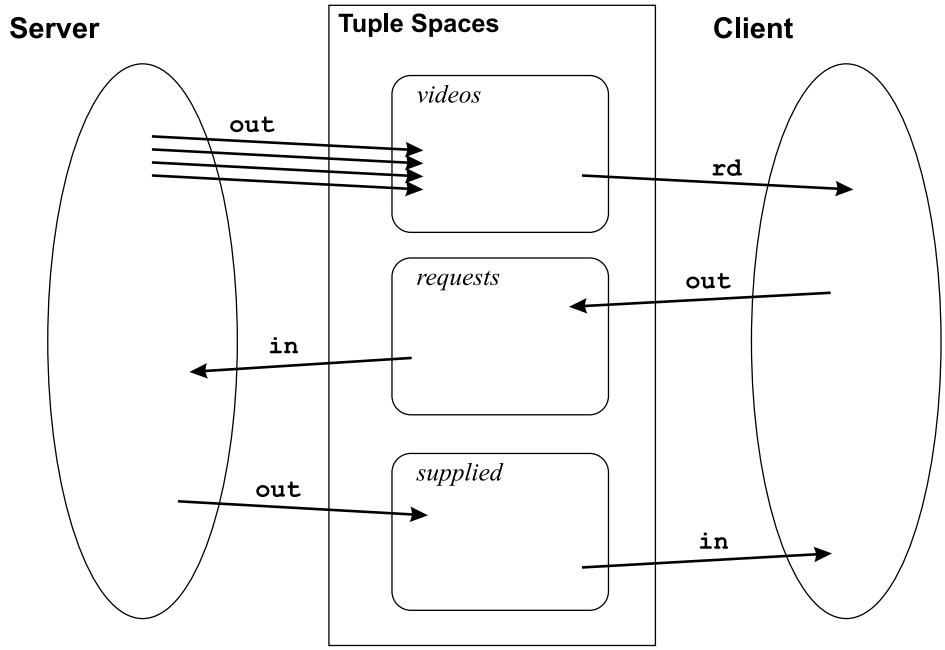


Figure 5.1: Control Flow in the Video Server Application

with a matching supplier name. These request tuples specify the title and key of the video required and also contain payment details. The payment details are verified (the current version simply accepts any payment information as valid), and if the verification succeeds, a tuple is placed into a third tuple space, called "supplied". This tuple contains the key value and a `MultiMediaResource` object that the client can retrieve in order to view the video. This process is shown diagrammatically in Figure 5.1.

Of particular interest is the fact that the server application does not need to handle the multimedia data source at all. The program simply creates a new eLinda `MultiMediaResource` object, specifying the filename for the video resource. The eLinda system then handles the connection to the client program, the transmission of the data to the client, etc. automatically.

5.1.2 The Video Client Application

This program is a GUI, event-driven Java application that allows a user to select a video and then view it. The user is first required to enter the title of a video resource. The "videos" tuple space is then searched for a tuple with a matching title. This makes use of a Programmable Matching Engine matcher that retrieves the tuple with the minimum value in the cost field (i.e. the `MinimumMatcher` discussed in the previous chapter). Alternative matchers could also be provided for this purpose that might take

```
Get videoName
if videos.rdp.minimum(?supplier, videoName, ?key, ?=cost) then
   Display video information
   if video is requested then
      requests.out(supplier, videoName, key, paymentDetails)
      supplied.in(supplier, videoName, key, ?video)
      video.play()
else
   Display "Video is not available"
```

Note: The variable *video* is an eLinda `MultiMediaResource` object; *videos*, *requests* and *supplied* are tuple spaces.

Program Segment 5.1: The Video Client Application

Figure 5.2: Screenshot of the Video Client Application

into account other issues, such as the quality of the video and the network bandwidth requirements. If a matching tuple is found, the details are presented to the user and they are asked if they wish to view the video. If they choose to do so a request tuple containing the payment information is placed in the "requests" tuple space. After this the tuple containing the actual video resource is retrieved from the "supplied" tuple space, and then the video is presented to the user.

The outline of the client program is shown in Program Segment 5.1. It should be noted that, for clarity, this has been rewritten using a procedural programming style, rather than the actual event-driven style used by the application. A screen shot of the simple user interface provided is shown in Figure 5.2.

While this is a simple illustration of the principles involved in such an application, and particularly of the use of the multimedia features present in eLinda, it does provide a convincing demonstration of these facilities. In particular, it shows how the unique features of eLinda greatly simplify the development of such applications.

5.2 Ray-Tracing

One of the demonstration programs provided with JavaSpaces is a simple ray-tracing application, written using a replicated-worker pattern[51]. This application was extended to produce timing results, and then ported to eLinda and to TSpaces. It was used to obtain some preliminary performance results, particularly for the purpose of comparing these three Linda systems.

5.2.1 The Ray-Tracing Algorithm

Ray-tracing is a well established technique for rendering realistic images[59, 89, 162]. It works by following a ray from the viewpoint, through a pixel on the view-plane to the scene being rendered. This is called the *primary ray* (or *eye ray*). The first object that is encountered is then visible at that pixel. The illumination of the pixel is determined by following rays from the point of contact with the object back to all of the light sources illuminating the scene.

It is also important to detect the presence of objects obscuring the light sources, in order to generate realistic shadows. This is done by casting *shadow rays* from the original point of intersection to the light sources and testing to see if any object in the image is intersected by the shadow rays.

If surface of an object is shiny, or transparent, then further rays (called *secondary rays*) need to be cast from the initial point of contact to cater for reflected or refracted light. This is a recursive process, as the secondary rays need to be traced back in exactly the same way as the primary ray from the viewpoint. In practice the depth of the recursion can be limited by calculating the attentuation of the reflected or refracted light and only performing the operation if the contribution to the lighting of the pixel is above some threshold value.

Lastly, the colour and texture of the surface itself must be taken into account. This is usually done by applying a shading technique, such as Phong shading[113], at the intersection point.

This process is repeated for rays through all of the pixels in the view-plane. These pixels are then be mapped onto a graphics display.

As can be seen from this brief description, there is a considerable amount of computation that is required for each pixel in the viewing area. Fortunately the calculations for individual pixels are independent of the others and so it becomes an easy problem to

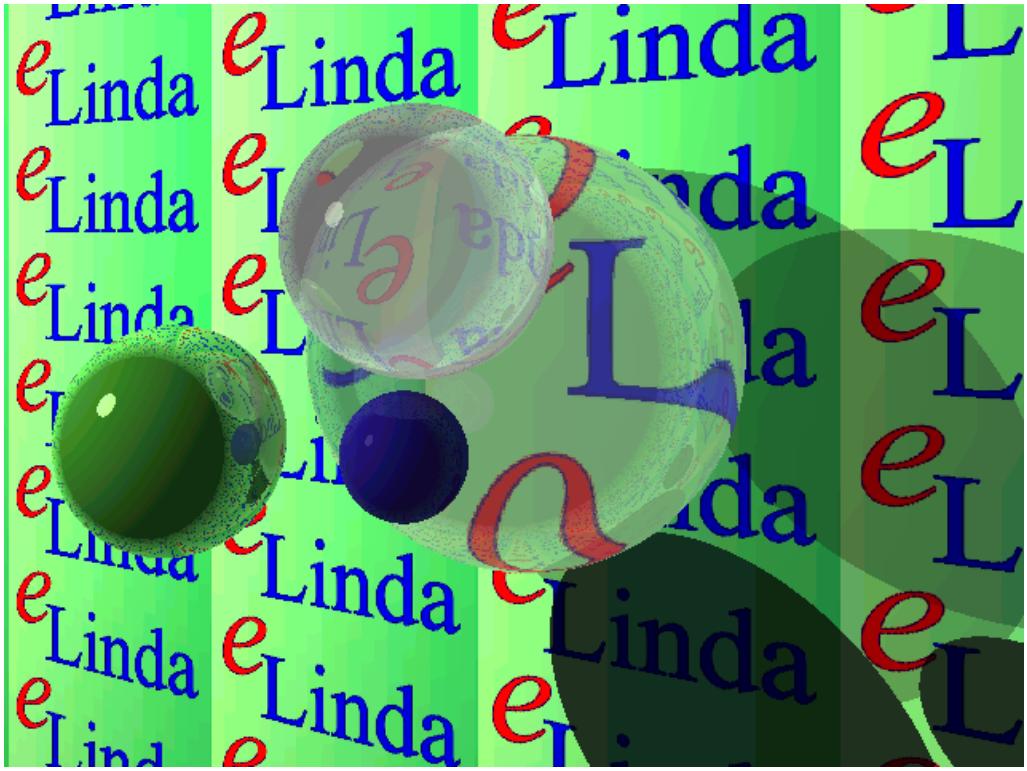


Figure 5.3: The Scene used in the Ray-Tracing Application

parallelise by dividing the viewing area into segments for separate processes to render.

5.2.2 Implementation

The application was ported directly from JavaSpaces to eLinda and TSpaces. While the overall performance of this application is not as good as other ray-tracing systems, it serves as a useful platform for assessing the relative performance of these three Linda implementations. It should also be noted that this application gave no scope for the use of the extended functionality of either eLinda or TSpaces. Additionally, no attempt was made to optimise the original program as supplied with JavaSpaces—it was simply used as a means of measuring the relative efficiency of the three Linda systems.

The image that was rendered was a simple scene with four spheres of varying transparency, a textured background plane and a single light source. The size of the image was 640 × 480 pixels. Figure 5.3 shows the rendered image.

This application was parallelised using a replicated-worker pattern[51]. The outline of the worker process is shown in Program Segment 5.2. This simply inputs a tuple specifying the task to be performed: a `long` value that is used to identify a particular

```
RenderTask task = null;
long key = 0L;
Tuple taskTuple = taskSpace.in(?key, ?task);  // Get a task to do
int[] result = task.execute();  // Execute the task
if (result != null)  // Write the result back
  resultSpace.out(key, task.startX, task.startY, result);
```

Program Segment 5.2: Generic Worker Process used for Ray-Tracing

image (thus allowing many scenes to be rendered simultaneously), and an object containing the task. The task object contains the full details of the image to be rendered. The worker process then uses the `execute` method of the task object to do the work. The result of this method (an array of pixel values) is then placed in the tuple space, together with the unique identifier for this image and parameters that identify the segment of the image which was rendered. This sequence of steps is repeated endlessly by each worker.

The master process divides the image area up into segments of a given size (selected by the user of the program) and places the corresponding task tuples into the tuple space. It then collects the result tuples as they become available from the workers, and displays the given pixels on the screen.

It should be noted that the simple allocation of work among the worker processes, on the basis of dividing the image up into equal-size segments is not ideal. Certain areas of the image are more complex than others and thus require more processing time per pixel. If a complex segment is picked up late by a particular worker, it can happen that all the other workers have completed their tasks well before it completes, leading to a lengthy delay to finalise the rendering of the full image. To minimise this effect, it would be useful to implement some form of load balancing that used information about the relative complexity of the different segments of the image, and subdivided segments with a high degree of complexity.

Comparative results for the performance of this application using eLinda, TSpaces and JavaSpaces are presented in Section 6.2.2.

5.3 Visual Language Parsing

As a particular example, which highlights the flexibility and power of the Programmable Matching Engine, we will consider the problem of parsing visual languages in more

detail. Visual languages are used in many areas to depict situations or activities in a pictorial or diagrammatic form, which is often easier for human beings to comprehend than equivalent textual forms. Examples of such languages abound, not least in the field of Computer Science, where notations such as flowcharts, state transition diagrams, entity-relationship diagrams, etc. are widely used.

If such graphical models are to be used and processed by computer systems there is a requirement for parsing them in order to analyse their structure. This process is directly analogous to the parsing of textual computer programming languages. What sets the parsing of visual languages apart is the increased complexity of the relationships between the components. In a textual language there is a simple, positional sequence relating the components of the language (i.e. the keywords and other tokens). In the case of a visual language there is far more scope for different relationships to exist between tokens in two dimensions (or, more generally, in three or even more dimensions). For example, tokens may be related by inclusion, by contact, by position (e.g. one above another), and so on.

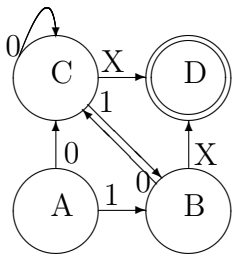
There are many different methods that may be used for specifying and for parsing visual languages. A classification of visual languages that highlights some of these differences can be found in [94]. The method that we will consider here is the use of *picture layout grammars* (a variation on *attributed multiset grammars*), as developed by Golin[60, 120]. Picture layout grammars provide a particularly flexible and powerful means of expressing the syntax of visual languages.

The example of a visual language that will be used in this discussion is the common notation for state transition diagrams (STD's). A simple example of a state transition diagram is shown in Figure 5.4, together with its textual representation.

5.3.1 Picture Layout Grammars

In this formalism, a visual program is represented as an attributed multiset: an unordered collection of attributed visual symbols. The *class* of a symbol corresponds to its type (e.g. label, circle, etc.), while the *attributes* of a symbol specify its features (e.g. text value, position, etc.). Visual languages are then sets of attributed multisets.

The attributed multiset representation of a picture is a flat structure. If we view the picture as an element of a visual language, then it has a complex structure, described by the relationships between the symbols. This structure is defined by the grammar productions of the language. A simple example of such a production is:




```
C 3.5,19 1.5 C0C0C0
T 'A' 3.5,19 000000 Arial
A 5,19 9,19 000000
T '1' 5,19 000000 Arial
C 10.5,19 1.5 FFFFFF
T 'B' 10.5,19 000000 Arial
T '0' 3.5,17.5 000000 Arial
A 3.5,17.5 5.5,15 000000
T '0' 9,19 000000 Arial
A 9,19 5.5,15 000000
A 5.5,12 11,13.5 000000
T 'X' 5.5,12 000000 Arial
T 'C' 5.5,13.5 000000 Arial
C 5.5,13.5 1.5 FFFFFF
C 12.5,13.5 1.5 FFFFFF
C 12.5,13.5 1.4 FFFFFF
T 'D' 12.5,13.5 000000 Arial
A 7,13.5 10.5,17.5 0000000
T '1' 7,13.5 000000 Arial
T 'X' 12,19 000000 Arial
A 12,19 12.5,15 000000
T '0' 4,13.5 000000 Arial
A 4,13.5 5.5,12 000000
```

Figure 5.4: An Example State Transition Diagram and Its Textual Representation

	Production Rule
1:	`STD` → `StateList`
2:	`StateList` → `State`
3:	`StateList` → `(State, StateList)`
4:	`State` → `contains(circle, text)`
5:	`State` → `leaves(State, Transition)`
6:	`Transition` → `labels(Arc, text)`
7:	`Arc` → `enters(arrow, circle)`
8:	`DoubleCircle` → `contains(circle, circle)`
9:	`FinalState` → `contains(DoubleCircle, text)`
10:	`State` → `FinalState`
11:	`GreyCircle` → `isgrey(circle)`
12:	`StartState` → `contains(GreyCircle, text)`
13:	`State` → `StartState`

Table 5.1: A Grammar for State Transition Diagrams

`State` → `contains(circle, text)`

This rule specifies that a state in a state transition diagram is made up of a circle and a textual label. The operator (`contains` in the example above) specifies explicitly the kind of relationship between the constituent elements.

In certain situations it is necessary for a production to include a symbol that does not constitute a part of the left hand symbol, but which must be present as the *context* in which the grammar rule can be applied. This is usually indicated by underlining such a context symbol to distinguish it from a normal symbol. A complete grammar for state transition diagrams is shown in Table 5.1, where the production for `Arc` (7) shows the use of a context symbol. In this case, the `circle` symbol is not a part of an arc, but must be present as the context in which the production for `Arc` can be used.

Formally, an *attributed multiset grammar* can be defined as a six-tuple (N, Σ, s, I, D, P) where:

N is a finite set of *non-terminal* symbols

Σ is a finite set of *terminal* symbols

$s \in N$ is the *start symbol*

I is the attribute names

D is the attribute domains

P is a set of *productions*

A *production* is a triple (R, SF, C) where:

R is a rewrite rule of the form $A \rightarrow M_1/\lambda$, where:

$A \in N$ is the left hand side (LHS)
M_1/λ is the right hand side (RHS)
$M_1 \subset (N \cup \Sigma)$ is a multiset of *ordinary symbols*
$\lambda \subset \Sigma$ is a multiset of *context symbols*
SF is a *semantic function*
C is the *constraints* on the application of R

Note that the simplified notation used for the productions in Table 5.1 includes the rewrite rules and the constraints, but omits the semantic functions.

We then introduce the concept that a picture M is *analysable*:

M is analysable if M has a derivation tree T where:
The leaf nodes of T spell out M
The root node of T is labelled by the start symbol, s
Each interior node n is labelled by a production $p = (R, SF, C)$, where:
labels(RHS(R)) = labels(children n)
C(children n) = true
attributes $n = SF$(children n)

Essentially, this means that a picture can be represented by a tree structure, where the leaf nodes represent the terminal symbols. The interior nodes represent the non-terminal symbols. For each interior node, the child nodes fulfil the constraints of the production used to generate the associated non-terminal symbol, and the attributes of the non-terminal symbol are generated from the attributes of the child nodes by applying the semantic function to the attributes of the children. Part of the parse tree for the state transition diagram of Figure 5.4 is shown in Figure 5.5.

Finally, the language recognised by a grammar G can be formally defined as follows:

$S(G) = \{M \mid M \in \Sigma^* \wedge M \text{ is analysable over } G\}$

Picture layout grammars are attributed multiset grammars, as defined above, but with the following restrictions:

- The attributes may take only a finite number of different values.
- No two terminal symbols may have the same class and attribute values.

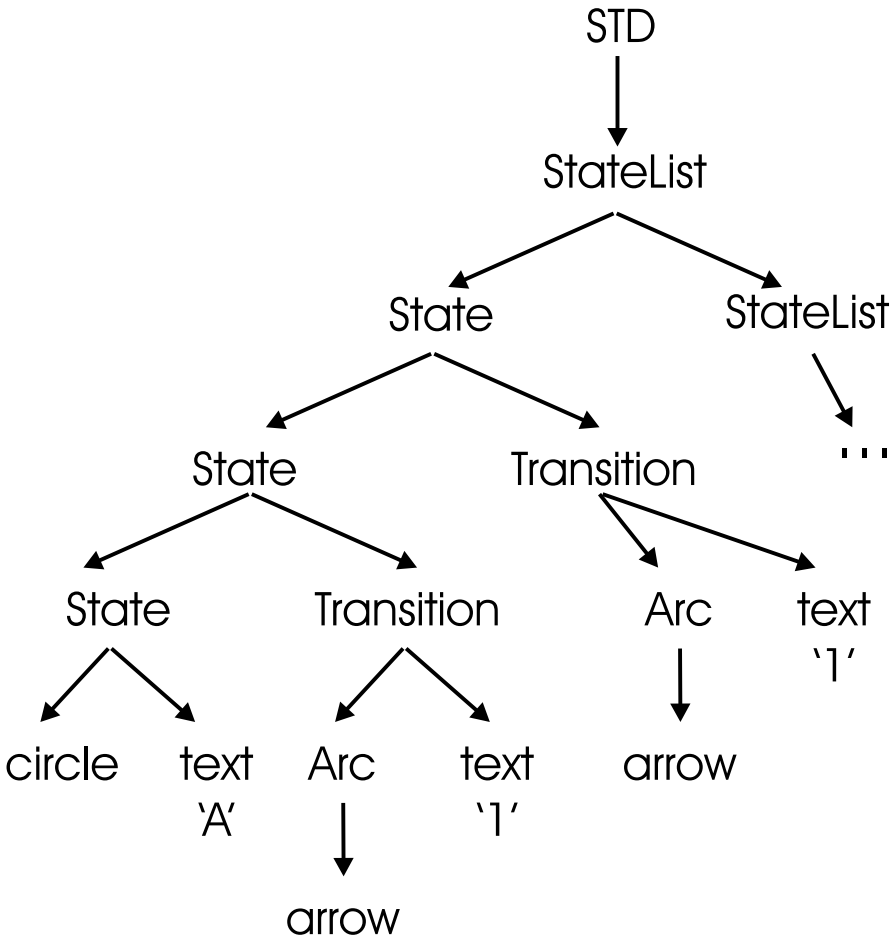


Figure 5.5: Example of a Parse Tree

- The right hand side of a production may consist of a maximum of two symbols (only a single ordinary symbol, or two ordinary symbols, or one ordinary symbol and one context symbol)[1].

While picture layout grammars are a powerful formalism for defining visual languages, they are difficult to parse efficiently. Golin reports a theoretical worst-case complexity result of $O(n^9)$, although in practice the worst-case behaviour seldom arises. The main cause of this complexity is that the first stage of the parsing algorithm produces multiple possible results: the *Factored Multiple Derivation* structure (or *FMD*). This is essentially a derivation tree as defined above, but has cross-links representing the use of context symbols, giving a *directed acyclic graph* (DAG). This data structure must then be checked in the second stage of the parsing process to remove invalid results and inconsistencies that may arise from ambiguities in the grammar. Lastly, the data structure is traversed again to pick a unique valid result. As a result of this complexity, parsers for picture layout grammars can benefit from a parallel implementation.

5.3.2 The Implementation of Golin's Parsing Algorithm

Prior to commencing any practical work on the parallelisation of the parsing algorithm for visual languages, some supporting software was developed. This took the form of a design for a generalised approach to the specification of grammars for visual languages. A number of Java classes were developed to automate the reading in of a grammar specification and its representation in terms of various data structures such as lists of symbols (terminals and nonterminals), a list of production rules, the specification and implementation of the constraint functions and the semantic functions, etc. Classes were also written to represent various data values required during the parsing process, such as production rules and symbol attributes. These supporting classes were utilised in all of the versions of the parsing algorithm that were developed (a sequential version, and two slightly different parallel implementations using eLinda).

Before studying the parallelisation of the parsing algorithm, it is useful to consider the first phase of the original, sequential algorithm (i.e. building the Factored Multiple Derivation structure). This has the form shown in Program Segment 5.3. It will be referred to as the "build" phase in the following discussion of the parallelisation of the parsing algorithm.

[1]This restriction is easily met by simply replacing each n-ary production by a sequence of binary productions.

```
 1 Build(M, P):
 2   for each b ∈ M do
 3     add a terminal node for b to todo and FMD
 4   while todo ≠ ∅ do
 5     next := some element of todo
 6     X := symbol(next)
 7     for each p ∈ P such that X ∈ RHS(p) do
 8       if p = A → {X} then
 9         if constraints satisfied then
10           Add(p, {next})
11       else
12         for each occurrence of X in RHS(p) do
13           let Y be the other symbol in RHS(p)
14           for each old ∈ done such that symbol(old) = Y do
15             if constraints satisfied then
16               Add(p, {next, old})
17     move next from todo to done
18   return FMD

19 Add(p, subnodes):
20   new := create a node for p
21   childlist(new) := subnodes
22   attr(new) := SF(subnodes)
23   for each node n ∈ (todo ∪ done) do
24     if symbol(n) = LHS(p) ∧ attr(n) = attr(new) then
25       childlist(n) := childlist(n) ∪ subnodes
26       discard new
27       return
28   add new to FMD
29   add new to todo
```

Program Segment 5.3: The Sequential Form of the First Phase of the Visual Parsing Algorithm

This is then followed by the second phase: checking the FMD structure, removing any invalid features from the initial parsing phase, and finally locating a valid parse tree within the FMD.

5.3.3 The Parallelisation of the Parsing Algorithm

The eLinda system is ideally suited to a problem like the parsing of picture layout grammars. In this implementation the symbols are simply represented by tuples. The fields of each of these tuples contain the class and the attributes of the corresponding symbol.

The parsing algorithm was parallelised using the replicated-worker pattern[51]. The master process first performs the initialisation of the tuple spaces used by this application. There are five tuple spaces, used for the following purposes:

1. There is a control tuple space, used for communication with the workers, and monitoring the progress of the parsing algorithm.

2. The grammar rules are stored in a tuple space for convenient distribution to the workers.

3. The *todo* set of symbols that still need to be considered by the algorithm is represented by a tuple space. This takes advantage of the fact that an attributed multiset maps very naturally onto the concept of a tuple space (in fact, a tuple space can also be thought of as an attributed multiset).

4. The *done* set of symbols that have been processed by the parsing algorithm is represented by a tuple space in the same way as the *todo* set.

5. The tree relationships for the *FMD* structure are stored in a separate tuple space. This decreases the need for manipulating the tuples in the *todo* and *done* spaces.

The initialisation performed by the master process consists of reading the grammar rules and loading them into the rule tuple space. The terminal symbols are written to the *todo* tuple space, replacing the first `for` loop in the algorithm (lines 2 and 3 of Program Segment 5.3).

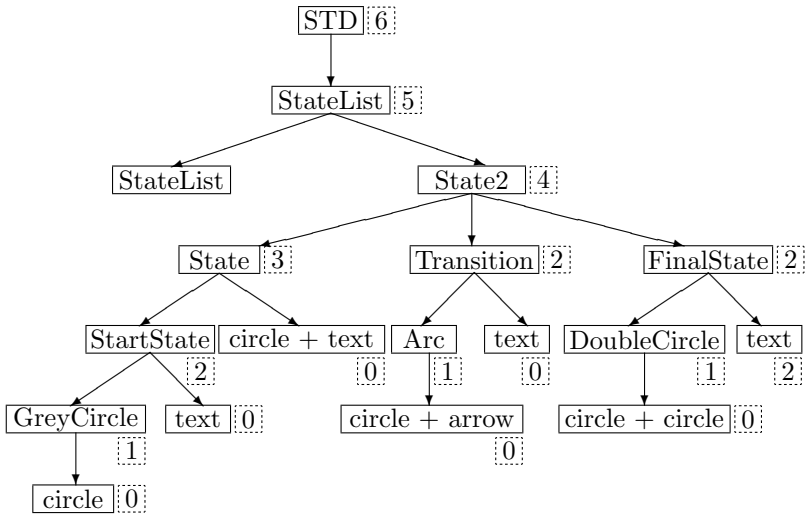
The worker processes then perform the central loop of the build algorithm (lines 4–17 of Program Segment 5.3) in parallel. However, some modifications to the basic algorithm are required. The first, relatively minor, modification is to perform the step

of placing the *next* element in the *done* tuple space as soon as it has been removed from the *todo* space (essentially moving line 17 of the algorithm to immediately after line 5 in Program Segment 5.3). This is done to prevent one process from holding the symbol for a potentially lengthy period of time, during which it might be required by another process.

A more substantial modification was required to keep the workers loosely in step with each other. Without this modification, it was observed that a worker would sometimes find that it was unable to perform any reductions as the symbols that it required were still being produced by other workers. As it is, if one worker does start to "get ahead" of the others then it may still find that it has no further work to do and terminate early, leaving the remaining workers to complete the parsing process. However, the loose synchronisation that was introduced is generally very successful, as can be seen from the results presented in the next chapter. The loose synchronisation used here was handled by means of introducing a number of *parsing levels*. These levels are determined by performing an analysis of the grammar. Firstly, the grammar rules are used to derive a tree corresponding to an abstract string derivation. For example, for the grammar for State Transition Diagrams, the abstract derivation tree is as shown in Figure 5.6. A parsing level is associated with each node of this tree, corresponding to the maximum depth of the subtrees below that node. These levels are shown in the dashed boxes in Figure 5.6 (where they are also summarised below the tree diagram). The parsing process then considers all the symbols at a given level before proceeding to consider the symbols at the next level. In this way the symbols are generated in the order in which they will be required, and workers are prevented from considering higher level symbols until all the lower level symbols are exhausted. At the lowest level this has the effect that all the terminal symbols must be used before the nonterminals generated from them are considered.

Once the first, building phase is completed the master process checks to ensure that the start symbol has been recognised. If this is the case, then the childlists of the start symbol are distributed to the worker processes for checking in parallel. In essence, the checking algorithm followed by the workers is as follows (we will consider the full parallelised version, illustrating the use of eLinda, in the next section):

```
1 Check(FMD, todo):
2   while todo ≠ ∅ do
3     childlist := some element of todo
4     walk the FMD calculating the coverage of each node and its depth
```



Level 0: {arrow, text, circle}
Level 1: {GreyCircle, DoubleCircle, Arc}
Level 2: {FinalState, Transition, StartState}
Level 3: {State}
Level 4: {State2}
Level 5: {StateList}
Level 6: {STD}

Figure 5.6: The Abstract Derivation Tree and Parsing Levels for the State Transition Diagram Grammar

```
5       if the childlist covers M then
6          remove recursive loops
7          prune childlists with identical coverage, favouring childlists
              with greater depth
8          walk the FMD a final time, picking a path that gives M
```

This process makes use of the *done* and *FMD* tuple spaces. For efficiency the workers cache this information in internal data structures (many of the childlists are composed of overlapping sets of symbols), accessing tuple space only once for each tuple during the checking stage.

5.3.4 Use of the Programmable Matching Engine

The parallelisation of Golin's parsing algorithm gives considerable scope for the use of the Programmable Matching Engine's facilities. At a number of points there is a need to use complex criteria to specify the tuples that are to be retrieved from tuple space. Four customised matchers were written for the visual language parser. Of these matchers, two are general-purpose, and may be useful in other applications. As discussed in the previous chapter, the number of lines of code, while not an accurate metric, does give an approximate indication of the complexity of a matcher. Accordingly, the number of lines of Java code are given below for each of the matchers written for the visual parsing algorithm, together with a brief description of the purpose of the matcher.

RHSMatcher (130 lines of code) This matcher is the most complex of those used in the visual language parsing application. It is used to search the tuple space containing the rules, looking for a rule that could be applied to reduce a given symbol X. This requires searching through the rules looking for the symbol X in the right hand side of a rule. If the rule is of the form $A \rightarrow X$ then the matcher additionally checks the constraints (this is straightforward in this case, as there is only the one symbol to consider). This matcher effectively replaces lines 7–10 of the parsing algorithm in Program Segment 5.3 (p. 83).

ConstraintMatcher (114 lines of code) This matcher is used when applying rules of the form $A \rightarrow X\ Y$ or $A \rightarrow Y\ X$ to locate suitable Y symbols to reduce with the current symbol X. In order to do this it has to check the constraints of the attributes of the available Y symbols (usually in conjunction with the attributes of the symbol X). It effectively implements lines 14 and 15 of the

parsing algorithm in Program Segment 5.3. It returns multiple matching tuples, using a Java `Vector`.

SetMatcher (111 lines of code) This matcher is used by the workers to retrieve a symbol (*next*) from the *todo* tuple space for consideration (line 5 in Program Segment 5.3). Due to the introduction of the parsing levels as explained in Section 5.3.3, this symbol needs to be chosen from the set of symbols to be considered at the current parsing level. For example, at level 1 of the parsing process the set of symbols from which *next* can be drawn is {GreyCircle, DoubleCircle, Arc} (see Figure 5.6).

This matcher could be used by any application that had a similar requirement to match tuples where a field has one of a set of defined values. The sets are handled using the `java.util.Set` interface within the matcher, allowing considerable flexibility. In the visual language parser the `java.util.HashSet` class is used, and the contents of the sets are simply `Integer` objects representing the symbol types.

AllMatcher (67 lines of code) This matcher can be used to retrieve all the tuples matching a given anti-tuple (in the same way as the `scan` operation provided by TSpaces). It is employed during the checking phase of the visual parsing application to retrieve all entries in the *FMD* tuple space for a given symbol (i.e. all of the childlists).

Again this matcher could be used by any application that needed to retrieve the set of all tuples meeting some criterion. It returns the multiple tuples in a Java `Vector`.

The parallelised algorithm for the master process is then as follows:

```
1 Master(M, P):
2   for each b ∈ M do
3     todo.out(b)
4   for each p ∈ P do
5     rule.wr(p)
6   signal the workers to start

7   wait for the workers to complete the build algorithm
8   if FMD.rdp(startSym) then
9     childlists := FMD.rdp.all(startSym)
```

```
10      for each childlist c in childlists do
11          todo.out(c)
12    signal the workers to start

13    wait for the workers to complete the checking algorithm
14    if control.rdp(startSym) then
15        print the solution parse tree
```

The waiting and signalling operations referred to above are simply implemented in terms of tuple space operations, using the control tuple space. These are essentially barrier synchronisation points, which are very simple to implement in Linda. Additionally there are a few other, minor steps that have been left out for clarity, such as communicating the start symbol to the workers.

The algorithm followed by each worker is as follows:

```
1  Worker:
2    wait for signal to start
3    Build
4    signal master
5    wait for signal to start
6    Check

7  Build:
8    while level < maxLevels do
9      set := new SetMatcher(symSet[level])
10     next := todo.inp.set(?)
11     while next ≠ null do
12       done.out(next)
13       p := rule.rdp.rhs(next)
14       while p ≠ null do
15         if p = A → {X} then
16           Add(p, {next})
17         else
18           for each occurrence of next in RHS(p) do
19             let Y be the other symbol in RHS(p)
20             oldSet := done.rdp.constraint(Y, next, p)
21             for each Y in oldSet do
22               Add(p, {next, Y})
23         p := rule.rdp.rhs(next)
24       next := todo.inp.set(?)

25 Add(p, subnodes):
26   newAttr := SF(subnodes)
27   n := todo.rdp(LHS(p), newAttr)
```

```
28    if n = null then
29       n := done.rdp(LHS(p), newAttr)
30       if n = null then
31          new := create a node for p
32          add new to todo
33          childlist(new) := subnodes
34          add childlist(new) to FMD
35       else
36          add subnodes to childlist(n) in FMD
37    else
38       add subnodes to childlist(n) in FMD

39 Check:
40    startTuple := todo.inp(startSym, ?childlist)
41    while startTuple ≠ null do
42       starting with childlist, walk the FMD calculating the coverage
            of each node and its depth
43       if childlist covers M then
44          remove recursive loops
45          prune childlists with identical coverage, favouring childlists
               with greater depth
46          walk the FMD a final time, picking a path p that gives M
47          control.out(startSym, p)
48          return
49       startTuple := todo.inp(startSym, ?childlist)
```

5.3.5 Refinements to the Parallel Algorithm

During testing and development the possibility of incorporating some enhancements and optimisations to the algorithm described above became apparent. The primary one among these was to store the grammar rules in the workers, rather than in the rule tuple space. This allows potential rules for application to be located without having to search the tuple space. This affects lines 7–10 of the original algorithm (Program Segment 5.3, p. 83), and removes the need for the RHSMatcher, which implemented this part of the algorithm. The rules are still distributed to the workers using the tuple space, but are simply loaded by the workers into an internal data structure during initialisation (using the AllMatcher). This resulted in the worker processes somewhat more closely resembling the original sequential algorithm.

A further enhancement that was implemented was to simplify the FMD structure during the build phase to decrease the amount of work required during the checking phase. This took the form of checking in the Add function to ensure that childlists do

not refer directly to the parent node (this is one of the aspects covered by the removal of recursive loops in the Check function). In the example grammar for state transition diagrams this arises from the rule `StateList` → `State2 StateList`, which may lead to a `StateList` being included in its own childlist (this also occurs in the serial version of the algorithm). The Check function still needs to check for possible recursion in the tree structure, as this may span a number of levels of the tree in general, but the incidence of recursive loops is decreased (as is the complexity of the *FMD* stucture).

There are other possible ways of organising this parallel application. One would be to provide a matcher for use with retrieving potentially applicable rules (similar to the RHSMatcher described above) that returned the set of *all* matching rules without evaluating any constraints (i.e. a solution intermediate between the use of the RHSMatcher and the first optimisation discussed above).

It should also be noted that the parsing process proceeds in a strict "lock-step" fashion: the master process signals the workers to start, they all perform the first phase, then synchronise again before all commencing the second phase. Further parallelism could be obtained by removing the synchronisation point in the middle of the parsing procedure. In other words, the checking could be started as soon as the goal symbol tokens start to become available. Two strategies are possible in this case.

1. Create two pools of distinct workers: one pool dedicated to the building phase of the algorithm and the other to the checking phase. As soon as results become available from the first pool, the second pool of workers can commence work on the second phase.

2. Retain a single pool of workers capable of performing both tasks (building and checking). However, in this case, as soon as a worker finds that it no longer has work to do for the first phase, it can immediately start work on checking any available results, without synchronising with the master and other worker processes.

Overlapping the two phases in either of these ways would decrease the total processing time, and increase the parallelism of the parsing application. These points are considered further in the discussion of the results in the next chapter.

5.3.6 Comments and Conclusions

Parallelising Golin's visual parsing algorithm presented some interesting opportunities to utilise the unique features of the eLinda system. The nature of the algorithm makes it simple to parallelise. In particular, the fact that the first phase constructs the FMD structure with a degree of redundancy and delays checking to a second distinct step, allows the worker processes to perform this phase without requiring any communication or synchronisation (except for the loose synchronisation described in Section 5.3.3, and even this requires no explicit communication between the workers).

It appears that there may be slightly more redundancy in the FMD structures created by the parallel parser than in the serial version. This would lead to slightly increased workload in the second, checking phase.

A major advantage of the use of eLinda in this area is that attributed multisets map very naturally to tuple spaces. Additionally the use of the Programmable Matching Engine simplifies the parallel version considerably, by encapsulating a number of lines of code from the original algorithms in matchers. More importantly, this problem would be very difficult to parallelise using the standard Linda programming model, due to the need to retrieve tuples subject to arbitrary constraints.

5.4 Summary

The applications presented in this chapter have highlighted the power and flexibility of eLinda, particularly the Programmable Matching Engine and the multimedia facilities. These features are very useful in simplifying the algorithms, allowing the application programmer to focus on the overall structure of the application.

The results obtained from testing these applications are presented in the following chapter.

Chapter 6

Evaluation of eLinda

This chapter presents a qualitative comparison of eLinda with other Linda systems, showing how the features of eLinda, especially the Programmable Matching Engine, are easier to use, and, in general, more powerful than the similar features of the other systems. It also provides quantitative results for the testing that was done using eLinda and the Linda systems developed by Sun Microsystems and IBM.

6.1 Comparison of Features

This section compares the features provided by eLinda with those in other Linda systems. Firstly the two commercial implementations of Linda in Java are considered, and then various other Linda research projects and proposals with similar features.

6.1.1 Comparison with JavaSpaces and TSpaces

Table 6.1 summarises the main functional differences between eLinda, JavaSpaces and TSpaces. Of particular interest are the extensible matching features of eLinda and TSpaces, which will be covered in more detail below.

Both TSpaces and JavaSpaces are intended for use in commercial applications, and so support transaction processing and lease/expiration mechanisms, which may be needed in such environments.

The event notification mechanism in TSpaces is an interesting feature whereby a process can ask to be informed when a tuple (specified by an anti-tuple) is placed into tuple space or deleted. While this feature is not provided in either eLinda or JavaSpaces, it would not be difficult to emulate for output operations using a separate

Feature	**JavaSpaces**	**TSpaces**	**eLinda**
Matching subtypes	Yes	No	No
Fields *must* be objects	Yes	Yes	No
More than one tuple space	Yes	Yes	Yes
Leases/Expiration	Yes	Yes	No
Transactions	Yes	Yes	No
Event notification:			
for writes	No (but `rd`)	Yes	No (but `rd`)
for deletes	No	Yes	No
Extensible matching	No	Yes	Yes
Multimedia support	No	No	Yes
Provides `eval`	No	No	No

Table 6.1: Comparison of JavaSpaces, TSpaces and eLinda

thread that waited for the event using the standard `rd` operation. Event notification for deletions would be more difficult and less efficient to implement, requiring a polling loop that used `rdp` to detect the eventual absence of the deleted tuple.

The Extended Features in TSpaces

As has already been noted, TSpaces contains a number of extensions to the original Linda programming model. These include several new operations, and a mechanism for adding new commands to the system, which will be considered in this section and compared with eLinda.

New Operations One of the notable features of TSpaces is its ability to use XML documents as fields of tuples, and then to retrieve tuples based on simple XML queries. In the version of TSpaces that was studied (i.e. version 2.1.1) the XML support was a new feature, and, as a result, incomplete. Some of the limitations present are as follows:

- The entire XML document must be passed to TSpaces as a single string (it is then parsed internally and stored as an XML document tree).
- Queries can be performed using only a subset of the XQL query language[121].
- Tuples can contain a maximum of one XML field.

At present, eLinda provides no support for XML. However, the Programmable Matching Engine would be an ideal platform for constructing such support if it is required for an eLinda application. This would allow a great deal of flexibility, such as the ability to support pre-formatted XML documents (i.e. already parsed, and represented as a document tree) for increased efficiency. Given the increasing importance and wide-spread use of XML in the computer industry, writing a programmable matcher to support XML queries in eLinda would be a useful exercise.

The second group of new commands in TSpaces are the "scan" commands:

`scan` This operation returns a tuple containing the set of tuples that match a given anti-tuple ("template" in TSpaces terminology).

`consumingScan` This works in the same way as `scan` but removes the tuples from the tuple space (analogous to the difference between `rd` and `in`).

`countN` This returns a count of the number of tuples that match a given anti-tuple.

While eLinda does not provide any of these operations directly, they are once again straightforward to emulate using the features of the Programmable Matching Engine. In fact, the "AllMatcher" developed for the visual parsing application (see p. 88) is exactly what is required for the `scan` and `consumingScan` operations, when used with `rd` and `in`, respectively.

TSpaces also includes a `multiWrite` operation, which takes a tuple containing a number of other tuples as fields, and places them in the tuple space. This may provide some slight performance benefit, as only one network communication is required to send all of the tuples to the processing node on which the tuple space server resides. While eLinda provides no such feature it could again be emulated using the Programmable Matching Engine facilities. This solution would have the aesthetic disadvantage that eLinda would require the use of an "input" operation in this case, while actually performing an "output" (the Programmable Matching Engine facilities are intended for use in matching operations, i.e. for retrieving tuples from tuple space).

TSpaces also supports the deletion of tuples (selected by matching with an anti-tuple) without retrieving them. This will reduce network traffic as the tuple is simply removed from the server without being transmitted back to the originating processing node. Again, this could easily be emulated in eLinda, but there would need to be at least an acknowledgment transmitted back across the network (it is not clear whether there is such an acknowledgement in TSpaces).

Lastly, TSpaces provides an update operation whereby a tuple can be replaced. This makes use of the "tuple ID" (a unique identifier allocated to all tuples by the TSpaces system) to locate the tuple. Once more, this could be easily be done using the Programmable Matching Engine (of course, the lack of a tuple identifier would have to be dealt with in some way such as designating a particular field of the tuple as the "key field"). Note that there would again be the aesthetically unpleasing aspect of using an input operation to perform an operation that is, perhaps, more closely related to the output operations.

Extensible Facilities As has already been noted, TSpaces makes provision for new commands to be added to the system. This feature can be used to extend the matching facilities provided as standard, but is implemented in a very different way to the Programmable Matching Engine in eLinda.

From a design perspective, the two systems approach the problem with a completely different philosophy. The TSpaces approach is to modify the server by embedding a new "factory" and a new command handler into the server to support new commands. The eLinda approach is to allow the programmer to provide an alternative matching algorithm with any anti-tuple. These might be characterised as a "heavy-weight" solution (modifying the server) and a "light-weight" solution. The TSpaces approach does allow the possibility of adding completely new commands, rather than simply modifying the way in which the matching and retrieval operations work. However, it is not clear that there is any great advantage to be gained from this. Even the example presented in the TSpaces programming guide[77] simply shows how an existing feature of TSpaces (i.e. tuple expiration) could be emulated. Essentially a tuple space is a high-level communication abstraction, and as such the operations that one is likely to need are those provided by the original Linda model: input and output of tuples.

The TSpaces implementation also raises concerns that must be addressed in a multi-user (or multi-application) environment. As it is the server that is being reconfigured to support new commands, there is the potential for one application to interfere with another (possibly quite unintentionally). This is addressed to some degree in TSpaces by the requirement that users adding new factories to the system must have the correct authorisation, but this does add to the overall complexity of the process of installing support for new operations, and may still be vulnerable to undesirable interference. The eLinda approach simply associates a new matching operation with a specific anti-tuple, and this is constrained in its actions by the programming interface available to

it.

Moving on to more practical matters, the difficulty of programming a new command handler in TSpaces appears to be roughly equivalent to that involved in writing a new matcher in eLinda. Conceptually, the task might be slightly simpler in TSpaces, due to the centralised tuple space model used (the Programmable Matching Engine approach is oriented towards the distributed tuple space model, and writers of new matchers will generally need to take this into consideration).

From the perspective of an application programmer using a new TSpaces command or a Programmable Matching Engine matcher, the eLinda approach is much simpler. The TSpaces approach requires the application to install a new factory and the associated new command handler, or handlers. The Programmable Matching Engine simply requires that a programmable matcher object be created and associated with an input operation.

6.1.2 Comparison with Research Systems

A number of other research projects based on Linda were discussed in sections 2.1.2 and 2.2.3. Those systems that have features which overlap with the extensions in eLinda will be considered further in this section.

XMLSpaces

The extensions in XMLSpaces are designed primarily to support the matching of XML documents as fields of tuples. This was discussed in Section 6.1.1, when considering the support provided for XML by TSpaces. As noted there, this is a feature that, while it is not currently supported by eLinda, would be easily provided, using the Programmable Matching Engine.

Objective Linda and CO^{3}PS

As discussed in Section 2.1.2, Objective Linda allows the matching method, `match`, for tuples to be overridden. The same mechanism is also provided by CO^{3}PS, which is closely based on Objective Linda. This approach provides a limited subset of the functionality of the Programmable Matching Engine. In particular, the matching is effectively provided by the *tuple*, rather than the anti-tuple. This has the implication that programmers writing classes for tuples need to consider how they may be retrieved. This is somewhat counter-intuitive as it is the anti-tuple that is relevant to input

operations in Linda systems. Associating the matching logic with the anti-tuples (as is done in eLinda) is thus a more natural approach.

Furthermore, Objective Linda constrains the matching to a one-to-one situation: the `match` method is called to determine whether the tuple matches a single anti-tuple. Thus there is no way of providing the functionality inherent in Programmable Matching Engine aggregating matchers. Such aggregating facilities are extremely useful in Linda systems, as can be seen from their inclusion as distinct operations in TSpaces (e.g. `scan`), and their use in applications such as the visual language parser of Section 5.3.

In CO^{3}PS this approach has been used to implement a *reflective architecture* allowing the *non-functional requirements* of an application to be provided for in way that is transparent to the application. The key to this philosophy is that the semantics of the coordination (tuple space) operations may not be altered by the imposition of non-functional requirements. This approach could be modelled using eLinda, as it would be simple to implement matchers that took into account non-functional constraints (such as security, or retrieving tuples to satisfy some performance constraint for the application). However, one of the strengths of the Programmable Matching Engine is that it *does* permit the semantics of tuple retrieval operations to be changed (for example, to return a pseudo-tuple, or to return multiple matching tuples).

The York Coordination Group

The work done on the `collect` and `copy-collect` operations at the University of York has some similarity to the functionality provided by aggregating matchers. These operations allow a collection of tuples to be selected from a tuple space, and moved or copied to another tuple space. This tuple space will generally be a private, local tuple space from which the tuples can be retrieved without interference from other processes. This is very similar to the use of the Programmable Matching Engine "AllMatcher", except that the Programmable Matching Engine solution delivers the tuples directly to the application without using an intermediate tuple space. Which of these two approaches is preferable depends on the requirements of a specific application, but it is likely that most applications will need to retrieve the tuples, in which case the Programmable Matching Engine approach will be more useful.

Liam is the Linda system developed by Campbell, of the York Coordination Group, using the CHAM (Chemical Abstract Machine—see Section 2.1.2, p. 15). As was noted in the earlier discussion, the programming notation used by the CHAM is likely to be

very difficult for most application programmers to use. Additionally, the matching extensions in Liam are also limited to matching a single tuple—there is no possibility of implementing aggregating matchers.

ELLIS

ELLIS was introduced in Section 2.1.2 (p. 15). As an object-oriented implementation of Linda (in EuLisp) it allows the method used for tuple matching to be overridden. However, this method is contained in the class used for tuple spaces, which leads to the rather inelegant solution that new tuple space classes must be developed to support new matchers. This also complicates the support of multiple matchers: the overridden matcher method will need to be parameterised in order to be able to decide which matching algorithm should be used. The Programmable Matching Engine approach of associating a specific matcher object with an anti-tuple is generally easier to use, and results in better application architectures. The programming interface for writing new matchers in ELLIS also appears to be more complex than that for the Programmable Matching Engine.

I-Tuples

As discussed in Chapter 2, the I-Tuples system allows simple updates of the values contained in tuple space to be performed at the server, thus reducing the number of network messages required. As with several of the TSpaces extensions this is not an operation that the Programmable Matching Engine was intended to support directly. However, it can be done very easily. A programmable matcher can use the normal tuple operations and so, after locating the tuple to be updated, it could place the new, updated tuple into tuple space using the usual `out` operation (or `wr`, if appropriate).

From an aesthetic viewpoint this is not ideal, but would not be as inelegant as in the case of emulating some of the TSpace features discussed above. For example, the example considered on page 14 might be handled as follows in eLinda, where the `addUpdateMatcher` retrieved a tuple, incremented the value contained in it and then placed a tuple containing the new value into the tuple space:

```
rd.addUpdateMatcher(?=someValue);
```

6.2 Benchmark and Application Results

This section presents the results of a number of benchmarks and applications developed using eLinda. Some of these are comparative, incorporating results found for other Linda systems, while others examine specific aspects of the eLinda system (such as communication performance, or scalability) in isolation. Additionally, some of the testing compared the different implementations of eLinda. As a reminder, the key features of these three implementations are as follows:

eLinda1 Fully distributed tuple space.

eLinda2 Centralised tuple space.

eLinda3 Centralised tuple space, with broadcast tuples (i.e. those tuples output using the `wr` operation) cached on each processing node.

All the results presented in this section were obtained using the undergraduate teaching laboratory in the Department of Computer at Rhodes University, unless otherwise noted. The computers in this facility are all equipped with 133MHz Pentium I processors and 32MB of main memory, and networked using standard 10Mb/s shared Ethernet. The operating system used was Windows NT 4.0. The version of Java used for all the testing was 1.3.0.

It should be noted that this hardware specification is barely adequate for the execution of Java applications. Sun's minimum recommended memory allowance for Java is 32MB, and the minimum processor recommendation is a 166MHz Pentium processor[143]. For some of the testing the limitations of the hardware became apparent, particularly in the form of excessive virtual memory paging and, in some extreme cases, complete system failures due to insufficient memory. These occurrences will be noted where relevant in the discussion in the following sections.

6.2.1 Communication Benchmark

In order to assess the communication costs in eLinda, a simple communication benchmark program was written. This makes use of two processes, neither of which does any processing but simply waits for a tuple to be deposited into tuple space by the other, and then sends a reply. This is repeated a number of times to get a realistic average communication time, that is, the average time taken to send information from one process to another, communicating through the tuple space. The number of repetitions was generally set to 100, with one exception, discussed below.

Option	Configurations/Values
eLinda Version	eLinda1 eLinda2 eLinda3
Program Type	Multiprocessing (separate computers) Multithreaded (single computer)
Data Size	No data 40 bytes 400 bytes 4000 bytes 40 000 bytes

Table 6.2: Options Tested with the Communication Benchmark

The amount of data included in the tuples was also varied, from none (the tuples contained only a single identifying string of four characters), to an array of 10 000 integer values (i.e. 40 000 bytes of data). Two versions of the program were developed: one using two distinct processes, running on separate computers, and one using two threads on a single machine. The multithreaded version under eLinda1 was repeated 10 000 times, rather than 100 as for all the other tests. The reason for this is that, with the distributed tuple space approach of eLinda1, there is no network communication required for the multithreaded option, and a smaller value for the number of repetitions would have resulted in program running times that would be too small to measure accurately. The various options are summarised in Table 6.2.

This benchmark was run on two different system configurations:

1. 133Mhz Pentium I processor, 32MB memory, 10Mb/s Ethernet, Windows NT 4.0 (the configuration used for all the other testing).

2. 733MHz Pentium III processor, 128MB memory, 100Mb/s Ethernet, Windows 2000 Advanced Server.

Table 6.3 shows the results that were obtained. The results for the multiprocessing version are also graphed in Figure 6.1. There are a number of unexpected findings revealed in these results.

1. There is very little apparent dependency on the data size. In fact, in many cases, the result when using tuples with *no* additional data is the slowest in a group.

2. The faster system configuration (i.e. using faster processors and a faster network) yields results that are no better than the slower system configuration. In one case

System Configuration 1 (Slow)			System Configuration 2 (Fast)		
eLinda1	**Processes**	**Threads†**	**eLinda1**	**Processes**	**Threads†**
No data	41.35	9.64	No data	50.06	1.74
40 bytes	39.88	10.28	40 bytes	49.74	1.44
400 bytes	39.90	9.24	400 bytes	49.44	1.07
4000 bytes	40.29	12.51	4000 bytes	49.43	0.88
40 000 bytes	40.44	9.33	40 000 bytes	49.52	1.10
eLinda2	**Processes**	**Threads**	**eLinda2**	**Processes**	**Threads**
No data	20.56	20.04	No data	20.00	20.00
40 bytes	20.68	19.97	40 bytes	20.00	19.99
400 bytes	20.90	20.08	400 bytes	20.00	20.00
4000 bytes	21.48	19.87	4000 bytes	19.80	20.19
40 000 bytes	20.65	20.02	40 000 bytes	19.99	20.08
eLinda3	**Processes**	**Threads**	**eLinda3**	**Processes**	**Threads**
No data	20.23	20.32	No data	20.00	20.01
40 bytes	19.97	19.97	40 bytes	20.00	19.99
400 bytes	19.99	20.07	400 bytes	20.00	20.00
4000 bytes	19.88	19.86	4000 bytes	19.99	20.20
40 000 bytes	20.06	20.02	40 000 bytes	19.98	19.97

† Repeated 10 000 times.
Note: All times are in seconds.

Table 6.3: Results of the Communication Benchmark

(eLinda1, using processes rather than threads) the "fast" configuration performs noticeably slower than the "slow" configuration, by a factor of around 25%.

3. The one set of results that does agree with intuitive expectations is for eLinda1, when using threads. This is significantly faster on the "fast" system configuration, by a factor that is close to the ratio of the processor speeds (133:733).

It is not clear why these findings do not agree with the obvious expectations. The results for the multithreaded eLinda1 tests indicate that software-related factors (such as garbage collection costs, or object serialisation overheads) can be ruled out as an explanation, as these factors would also have been apparent in that case.

Considering hardware factors, it appears that use of the faster network (100Mb/s) does not have a significant effect on the results (except for eLinda1, where it apparently has a negative effect). Since this communication medium is a factor of ten times faster than the "slow" network, it is to be expected that there would be some noticeable benefit arising from its use. It is possible that the network access negotiation protocol used by Ethernet (collision detection and "back off") is having an adverse effect on the

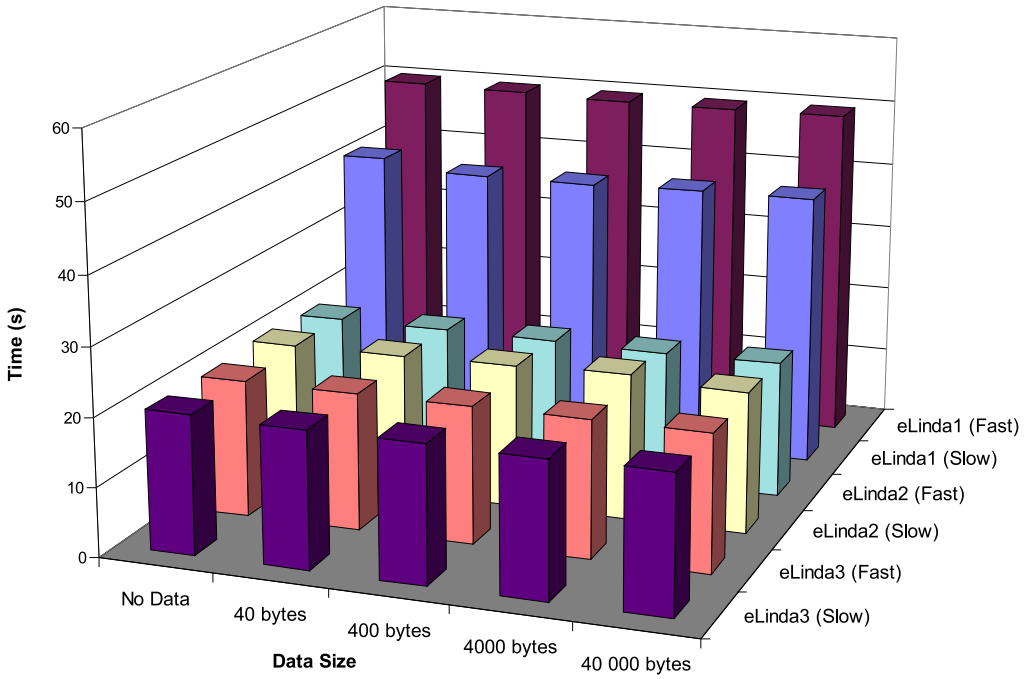


Figure 6.1: Communication Benchmark: Results for Processes

network performance. This would clearly be impacted by faster processors delivering messages to the network interface for transmision more rapidly. However, it seems unlikely that this would result in performance figures that are almost identical to those for the slower network. One other possibility is that the bottleneck in these cases may be the internal system bus that is used to transfer data from the processor to the network interface card. If this bus is clocked at the same speed in both system configurations then this may be the reason for these anomalous results.

There are some interesting results that can be derived from these measurements. Firstly, the cost of network communication in eLinda can be calculated. If we take 20s as the average time for 200 "messages" (i.e. matching pairs of tuple output and input operations), we get an average of approximately 100ms per message. If we consider the results for the multithreaded version of the program in eLinda1, where no network activity is required, we get the following results:

Average time per message (with no data):

Slow configuration: $9.64\text{s}/20\ 000 = 0.48\text{ms}$

Fast configuration: $1.74\text{s}/20\ 000 = 0.087\text{ms}$

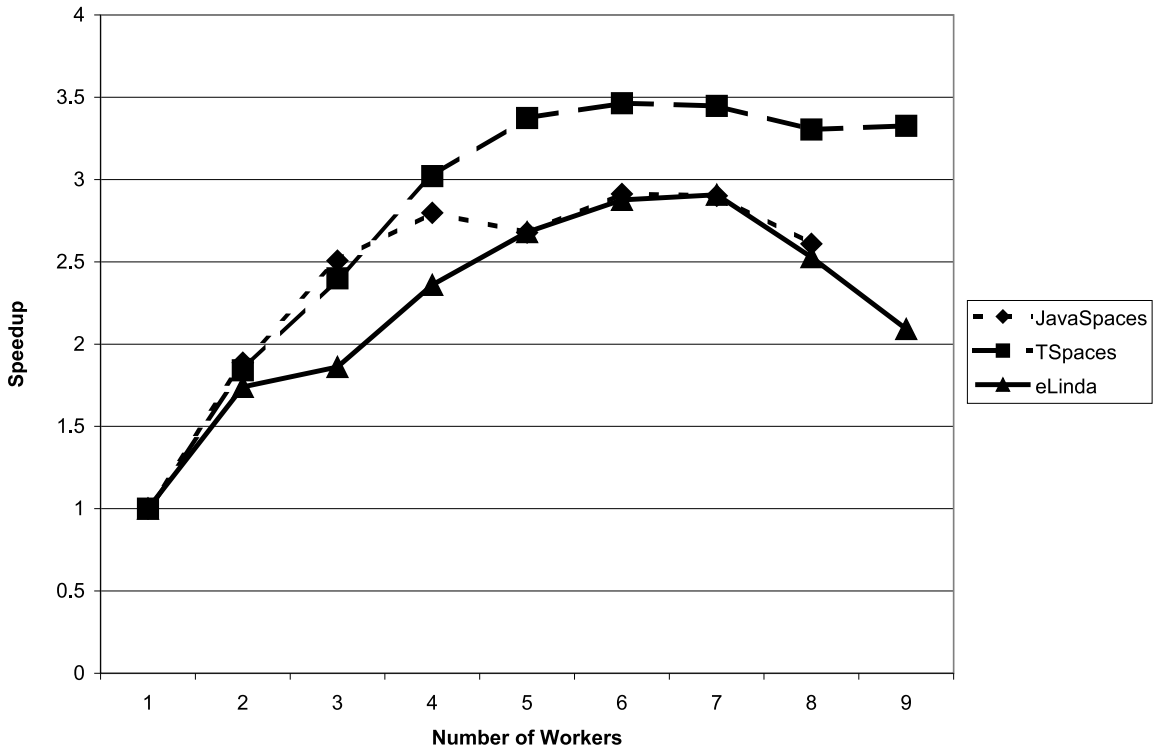


Figure 6.2: Speedup for the Ray-Tracing Application

6.2.2 Ray-Tracing

The ray-tracing application was used mainly as a platform for comparing the three Java Linda systems: eLinda, TSpaces and JavaSpaces. It was adapted from one of the demonstration applications provided with the JavaSpaces system, and thus did not make use of any of the extensions in eLinda. For this comparison only eLinda1 was used.

The results of this study are summarised in Figure 6.2. This graph shows the speedup of the ray-tracing program as the number of worker processes is increased, relative to the time taken by a single worker. What this demonstrates clearly is the similarity between the three implementations: all exhibit very similar behaviour, with the speedup levelling off or, in some cases, decreasing from about seven worker processes. The maximum speedup obtained was approximately 3 to 3.5 times, using six or seven processors, as shown in Figure 6.2. The performance of eLinda and JavaSpaces is very similar, while TSpaces is slightly more efficient, but not markedly so.

Actual timing results for this application are shown in Table 6.4, and are graphed

	Linda System		
Workers	**eLinda**	**TSpaces**	**JavaSpaces**
1	60.66	58.48	63.48
2	34.86	31.78	33.65
3	32.60	24.37	25.32
4	25.70	19.36	22.69
5	22.62	17.34	23.71
6	21.09	16.89	21.80
7	20.87	16.96	21.88
8	23.99	17.70	24.33
9	29.01	17.58	
	Note: Times are all in seconds.		

Table 6.4: Detailed Results for the Ray-Tracing Application

in Figure 6.3. The ray-tracing application allows the size of the segments of the image that are distributed to the workers to be changed. These results were all measured for an image segment size of 200 × 200 pixels (the full image is 640 × 480 pixels). As can be seen from these figures, TSpaces had the best performance overall. The results for eLinda are very close to those for JavaSpaces, and fairly close to those for TSpaces. A maximum of only eight worker processes could be used with JavaSpaces. Over this limit the system became completely unstable.

To assess the impact of different image segment sizes, the measurements were repeated for 50 × 50 pixels, 100 × 100 pixels and 200 × 200 pixels. The best results for all three systems were generally found for larger segment sizes. The results for eLinda showing the impact of the segment size are summarised in Table 6.5. This clearly shows that the application is communication-bound, favouring larger segment sizes.

6.2.3 Visual Language Parsing

The visual language parser was tested in a total of six different configurations. This arose from the use of all three implementations of eLinda, combined with the two different versions of the parser described in Section 5.3.5 (one where the grammar rules were stored in a tuple space, and the other where the rules were cached in the worker processes). The time taken for the build and check phases was measured separately, for each worker process, and also for the master process. The tests were also performed

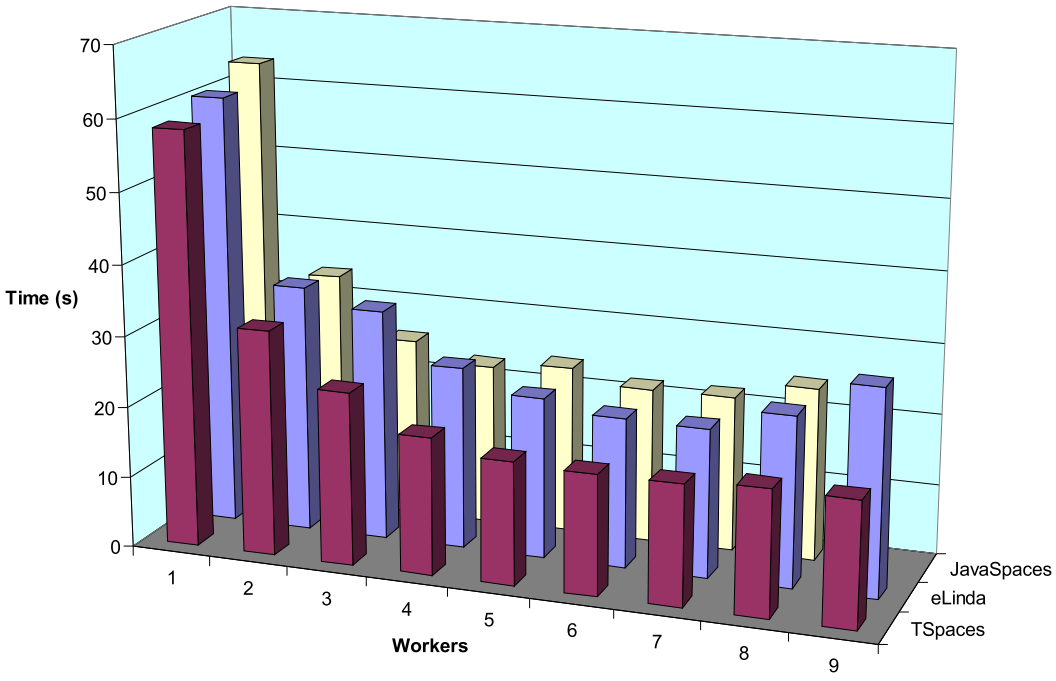
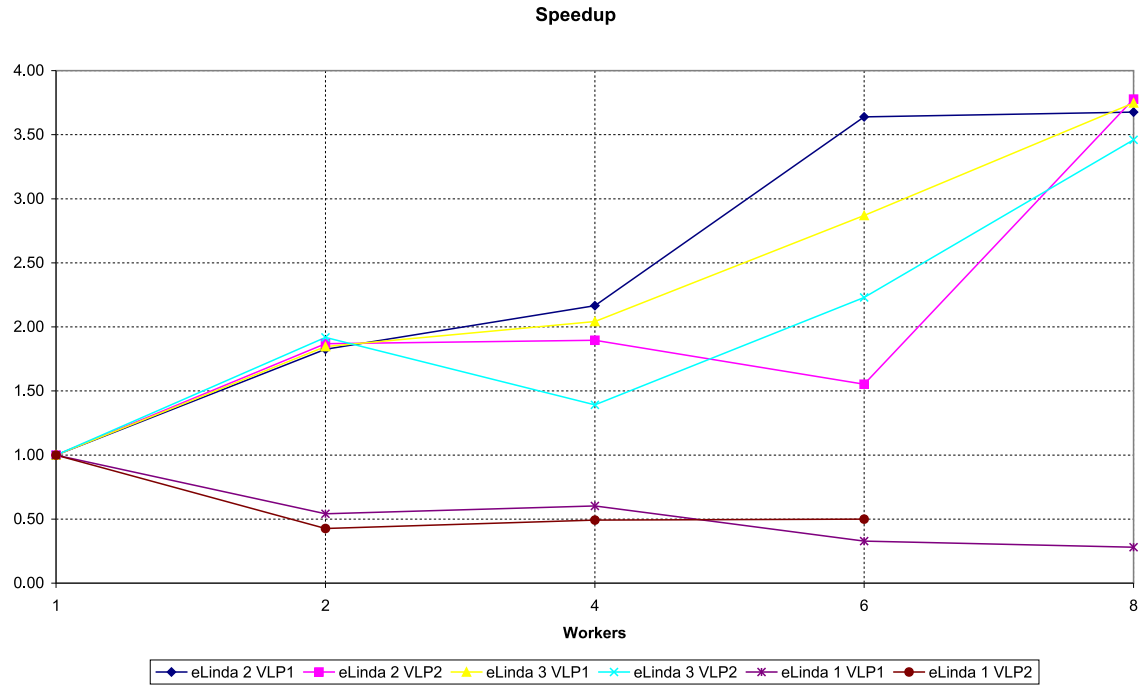


Figure 6.3: Results for the Ray-Tracing Application

	Image Size		
Workers	**50 × 50**	**100 × 100**	**200 × 200**
1	125.82	70.87	60.66
2	66.34	44.55	34.86
3	53.30	59.72	32.60
4	30.65	23.61	25.70
5	28.32	25.62	22.62
6	23.09	15.65	21.09
7	21.15	18.94	20.87
8	22.31	17.31	23.99
9	19.52	24.45	29.01
	Note: Times are all in seconds.		

Table 6.5: Results for Differing Image Segment Sizes for eLinda



Note: VLP 1 refers to the visual parser without caching of the grammar rules, VLP 2 to the version with caching of the rules.

Figure 6.4: Speedup for the Visual Language Parser

for various numbers of workers: 1, 2, 4, 6 and 8. All the measurements in this section were made using a small state transition diagram containing 21 symbols (representing four states with six transitions).

TSpaces and JavaSpaces were not used for this application due to the difficulty of expressing many of the required operations without the support of the Programmable Matching Engine.

With the exception of eLinda1, all the tests showed increasing degrees of speedup, up to the maximum tested configuration of eight processors. This can be seen clearly in Figure 6.4. The average speedup using the eight processor configuration (but excluding eLinda1) was 3.7 times.

This graph also shows the effects of the limited memory installed in the machines used for testing. This caused excessive swapping when using six and eight worker processes in the following configurations:

- eLinda1, with both versions of the visual parser.
- eLinda2, with the first (non-caching) version of the parser.

Furthermore, it was impossible to obtain any results for eight workers with eLinda1 and the second (caching) version of the parser. The "Directory" handlers used in eLinda1 generate approximate load measurements, which are reported by the system. Under normal conditions the measured loads are around 3 or 4. During this testing, for six and eight workers, loads as high as 25 were reported, reflecting the inability of the processing platform to cope with the application.

While the speedup that is obtained is generally good, a comparison with the results obtained from a sequential version of the program highlight the poor performance of communication based on TCP/IP protocols over a 10Mb/s Ethernet network. The best result that was obtained for the parallel versions of this application was 42.52s (for eLinda2 with the second, caching version of the parser, using eight worker processes), where the sequential time is only 0.58s. Breaking this down into the two phases of the parsing process reveals some interesting features:

1. The difference is most marked for the build phase, where the best parallel result was 32.48s, compared to 0.27s for the sequential version, i.e. two orders of magnitude difference.

2. For the checking phase the difference is smaller, with the best parallel result 2.74s (eLinda2, with the first, non-caching version of the parser, using only two workers), and the sequential result 0.31s. While this indicates that the speedup for the checking phase is noticeably less pronounced than for the building phase, it should be noted that the performance using eight worker processes was still only 2.95s—there is not a great deal of difference between two workers and eight workers. It should also be noted that the difference between the sequential and parallel results here is only one order of magnitude.

The Effect of Synchronisation

In Section 5.3.3 the need for the introduction of "parsing levels" to loosely synchronise the worker processes was discussed. The effects of this can be seen clearly in the results that were obtained. A typical run with eight workers gave the set of timings shown in Table 6.6 (these times are for the build phase only). These figures show how one of the workers (number 4) did get ahead of the others, as explained in Section 5.3.3, finishing in only 5.73s. This behaviour was observed about 12% of the time. The workload of the other worker processes is well balanced, with the remaining results ranging from 29.01s to 31.31s.

Worker Number:	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
	30.31	30.09	29.04	5.73	29.76	29.01	29.53	29.02

Note: Times are all in seconds.

Table 6.6: Sample Results for the Build Phase with Eight Worker Processes

6.2.4 Multimedia Performance

The Java Media Framework (JMF) supports a wide variety of different protocols for the transmission and presentation of audio and video data[140]. Of the many formats available, JPEG and H.263 were selected for testing, as these are widely used in practice. The JPEG (Joint Photographic Experts Group) image format is extensively used on the Internet[83]. On the other hand, H.263 (a standard produced by the International Telecommunications Union, ITU) is commonly used in video-conferencing systems[1, 81]. All the testing described here was done using RTP (the Real-time Transport Protocol) for the transmission of the data across the network.

The performance of the Java Media Framework for video transmission was measured for a number of different image sizes. This was done for video-conferencing conditions, using a low-cost commodity video camera (a Logitech QuickCam Express). This has a maximum video capture rate of 15 frames per second at low resolution (160×120 pixels), using the custom software provided by the manufacturers of the camera. The frame capture rate of the camera was set to nine frames per second for the testing reported on here. The computers used for this testing were two 500MHz Pentium III processors, using the Windows 2000 operating system. The machine doing the video capture was equipped with 192MB of memory; that doing the display had 64MB of memory. They were connected by a standard 10Mb/s Ethernet network. It should also be noted that the encoding and decoding of the multimedia data was all performed by the Java Media Framework software—there was no hardware *codec* support. The results of this test are shown in Table 6.7.

These figures show adequate performance for desktop video-conferencing purposes, in most cases. The performance using the H.263 format is slightly better than that for the JPEG format for low resolutions, but deteriorates more rapidly as the image size is increased.

Resolution	**Video Format**	
(in pixels)	**JPEG/RTP**	**H.263/RTP**
128×96	7.9–9.0	7.9–10.0
176×144	6.4–7.9	7.9–8.5
320×240	5.0–7.4	†
352×288	4.5–5.5	0.5

Note: Results are all in Frames per Second (fps).
† Resolution not supported for RTP with H.263 in the JMF.

Table 6.7: Multimedia Transmission Results

6.3 Summary

This chapter has demonstrated the power and expressiveness of the Programmable Matching Engine. In particular it has shown how almost all other proposed extensions to Linda, of a similar nature, may be emulated using the Programmable Matching Engine. The benchmark and timing tests for the applications presented in this chapter have demonstrated that, while the performance of the current Java implementations of eLinda is not good, it is very close to that of JavaSpaces and TSpaces (both of which were produced by large commercial enterprises). Additionally, these results have highlighted the poor performance of the TCP/IP protocols in Java, and their unsuitability for use in communication-intensive parallel applications.

The following chapter draws some overall conclusions from the work done on eLinda, and the results presented in this chapter. It also proposes a number of areas for further research that would address some of the negative findings, and build on the existing strengths of eLinda.

Chapter 7

Conclusions and Future Research Directions

This chapter presents an analysis of the results found and presented in the previous chapter. This is followed by a discussion of possible future research directions arising from this work and the results found. Lastly, some final conclusions are presented.

7.1 Analysis of Results

The results presented in the previous chapter fell into two categories: *qualitative* and *quantitative*. They will be discussed here under these two headings.

7.1.1 Qualitative Results

The significant number of projects that have, in one way or another, extended the facilities of the matching operations in Linda points to the weakness that is embodied in the original programming model proposed by Gelernter. The nature of the extensions ranges from those exemplified by TSpaces, where the tuple space server itself is reconfigured to support new operations, through proposals such as I-Tuples and the "mobile coordination" proposals that aim to increase the efficiency of simple updates, to systems like XMLSpaces, Objective Linda and Liam where the matching process is specified by overriding the matching method used, in some cases for very specific purposes (e.g. XMLSpaces).

What is found in comparing these proposals with the eLinda Programmable Matching Engine is that the Programmable Matching Engine proposal can emulate all

of these alternative approaches. Furthermore, the Programmable Matching Engine approach allows for a range of tuple space implementation techniques, ranging from fully distributed to centralised, whereas most of the other systems are implemented only for centralised configurations.

In many cases the Programmable Matching Engine approach is more intuitive and elegant than the alternatives. The approach adopted in Objective Linda and CO^{3}PS, where the matching method in an object/tuple can be overridden, fits well with the object-oriented philosophy of Java. However, it limits matching operations to one-to-one situations, and fixes the matching possibilities at the time that the tuple class is written. By providing the matcher as an independent object, the Programmable Matching Engine approach opens up the possibilities of aggregating operations, and provides the flexibility of being able to apply many matchers to a single type of tuple. The approach used in TSpaces, while providing a high degree of flexibility, effectively requires the (complex) reconfiguration of the tuple space server to support new operations.

Additionally, the support for distributed multimedia applications provided in e-Linda greatly simplifies the development of this important class of applications, by encapsulating all of the details of the transmission and presentation of multimedia resources.

7.1.2 Quantitative Results

The performance results obtained for eLinda should be viewed in the light that the Java implementation is intended as a demonstration of the new concepts proposed in eLinda, and not necessarily as a production system for the development of parallel applications. What is notable is that the performance of eLinda is reasonably close to that of TSpaces, and generally as good or better than that of JavaSpaces—two systems developed by very large organisations with considerable resources. This was shown clearly by the results of the ray-tracing application, where eLinda was 24% slower than TSpaces, and JavaSpaces was 29% slower than TSpaces (Section 6.2.2, p. 104).

That the bottleneck in the performance arises from the use of the TCP/IP protocols in Java is also clear: 100ms per network "message" is a very high cost (Section 6.2.1, p. 100). In this regard, the anomalies detected when testing the communication performance of a fast system configuration need to be investigated further.

Given the poor overall performance, the scalability demonstrated by the example applications running on eLinda demonstrates the viability of the fundamental approach. This can be seen in the ray-tracing application where a speedup of 2.9 times was achieved with seven processors (Section 6.2.2, p. 104). Even better was the speedup obtained for the visual parser (3.7 times), which had not levelled off at the maximum number of processors used (i.e. eight processors—Section 6.2.3, p. 105). This is an encouraging basis for the possible development of an eLinda implementation using a more suitable network and processor architecture.

With regard to the multimedia extensions in eLinda, the performance demonstrated in this area is good for applications such as desktop videoconferencing. Adequate performance for more demanding applications (such as high-resolution entertainment broadcasts) would require the use of more sophisticated codec support, possibly with hardware assistance.

7.2 Future Directions for Research

This section discusses a number of possible areas arising from this research work that would be interesting, or useful, to explore in order to further develop the eLinda approach to the development of concurrent systems. These have been classified into three categories: new features, implementation issues and applications.

7.2.1 New Features

Arising from the comparison of the Programmable Matching Engine facilities with the similar features of other extended Linda systems (notably, TSpaces and I-Tuples) there were some facilities of those systems that could be emulated in eLinda, but it was noted that these solutions were aesthetically unpleasing. This arose from the use of the Programmable Matching Engine, which is intended primarily for increasing the flexibility of the matching for *input* operations, to emulate *output* or *update* operations. This suggests that it might be useful to develop new facilities for eLinda, similar to the Programmable Matching Engine, but intended for output and/or update operations.

This would require extensions to the `out` and `wr` operations, and possibly adding a further new operation along the lines of the `tmexec` operation in I-Tuples, or the `update` and `multiUpdate` operations in TSpaces.

7.2.2 Implementation Issues

The implementation of eLinda in Java, using the Internet protocols for communication has proved successful as a demonstration of the validity of the ideas embodied in the extensions to the conventional Linda programming model. However, the lack of performance for concurrent programming in this environment is very apparent. Accordingly, it would be of great interest to develop other implementations of eLinda for alternative hardware platforms, such as a tightly-coupled multicomputer architecture, or a multiprocessor system with physically shared memory.

The approach taken by the Hyperion system (utilising a native code compiler for Java, and emulating the shared Java memory model for distributed Java threads—see Section 2.2.2) appears to hold potential for the efficient execution of multithreaded Java programs on distributed "network of workstation" platforms. Given the good communication performance of the multithreaded version of the eLinda1 system (Table 6.3), Hyperion may be a useful platform for the implementation of eLinda.

On the subject of implementation platforms, it would also be useful to explore further the benefits that may be found from using high speed networks with widely-available, commodity workstations.

Arising from the initial performance results for the multimedia support in eLinda, it would be interesting to explore other data formats supported by the Java Media Framework. Similarly, the performance benefits to be obtained from utilising specialised hardware for encoding and decoding multimedia data streams should be investigated.

7.2.3 Applications

Visual Language Parsing

As was discussed in Chapter 5, there is some scope for increasing the overall parallelism of the visual language parser by overlapping the execution of the two phases of the algorithm (i.e. building the factored multiple derivation data structure, and the subsequent checking of it). Some ideas for implementing this approach were presented in Section 5.3.5.

In addition to this enhancement, there are a number of other aspects of this application that could be explored further:

- Performing the building step as a sequential program stage, with the checking stage done in parallel. This may yield better overall results as the parallel exe-

cution of the second stage showed more promising results than that of the first stage.

- Sharing information from the checking stage among the workers. Currently the worker processes do not share any information during this stage. However, due to the redundancy present in the factored multiple derivation data structure, this means that there is a considerable amount of duplication of effort between the workers.

  Naturally, sharing this information will increase the communication load on the system, and so would have to be approached carefully with the current implementations of eLinda. In this regard, it may be possible to use the concept of "parsing levels" that was introduced in the parallelisation of this algorithm. In this case, only information for the higher levels would be shared, in order to reduce the communication requirements (at the lower levels the information required is more easily derived).

Various combinations of these different approaches could also be studied in order to obtain optimal performance from the parser.

Multimedia Applications

It would be useful to develop some full-scale multimedia applications, such as a video-conferencing system or computer-based education lessons. These could be used to explore the multimedia features and the performance of eLinda and the Java Media Framework further.

New Application Areas

Given the unsuitability of the current Java implementation for high performance, parallel processing it would interesting to explore the possibilities of applying the current implementations of eLinda to coarse-grained, distributed applications with more favourable communication:computation ratios. An application such as searching the World-Wide Web might benefit from the simplicity of the Linda programming model, and could make good use of the Programmable Matching Engine facilities to support flexible matching for such searches.

7.3 Conclusion

eLinda and the related research and production systems described here have clearly shown the need for extensions to the original associative matching model in Linda. The Programmable Matching Engine provides an elegant, simple, but highly expressive solution to the problem of implementing complex matching operations in Linda. Its ability to emulate the other proposals for extended matching facilities, and to exceed their capabilities is clear. Particularly in the area of distributed multimedia applications the use of eLinda can greatly simplify the work of application developers, building on the inherent simplicity of the original Linda approach.

While it is apparent that the current Java implementations of eLinda do not provide the necessary performance for serious use, this work has served as a significant step towards validating the novel concepts in eLinda as powerful tools for simplifying application development.

Bibliography

[1] 4i2i Communications Ltd. H.263 video coding. URL: `http://www.4i2i.com/-h263_video_codec.htm`, 1998.

[2] S. Ahuja, N. Carriero, and D. Gelernter. Linda and friends. *IEEE Computer*, 19(8):26–34, August 1986.

[3] S. Ahuja, N. Carriero, D. Gelernter, and V. Krishnaswamy. Matching language and hardware for parallel computation in the Linda machine. *IEEE Trans. Computers*, 37(8):921–929, August 1988.

[4] B.G. Anderson and D. Shasha. Persistent Linda: Linda + transactions + query processing. In Banâtre and Métayer [9], pages 93–109.

[5] G. Antoniu, L. Bougé, P. Hatcher, M. MacBeth, K. McGuigan, and R. Namyst. The Hyperion system: Compiling multithreaded Java bytecode for distributed execution. *Parallel Computing*, 27(10):1279–1297, September 2001.

[6] C. Austin and M. Pawlan. *Advanced Programming for the Java 2 Platform*. Addison-Wesley, September 2000.

[7] A. Baba and J. Tanaka. Eviss: a visual system having a spatial parser generator. In *Proc. Asia Pacific Computer Human Interaction*, pages 158–164, 1998.

[8] H.E. Bal, M.F. Kaashoek, and A.S. Tanenbaum. Orca: a language based on shared data-objects. In Wilson [164], pages 5–13.

[9] J.P. Banâtre and D. Le Métayer, editors. *Research Directions in High-Level Parallel Programming Languages*, volume 574 of *Lecture Notes in Computer Science*. Springer-Verlag, 1992.

[10] R. Baraglia, D. Laforenza, and R. Perego. Programming a workstation cluster with PVM and Linda: a qualitative and quantitative comparison. In *Proc.*

AICA'93 International Section: Parallel and Distributed Architectures and Algorithms, pages 101–114, 1993.

[11] A. Baratloo, M. Karaul, Z. Kedem, and P. Wyckoff. Charlotte: Metacomputing on the Web. In *Proceedings of the 9th Conference on Parallel and Distributed Computing Systems*, 1996.

[12] R. Bjornson. *Linda on Distributed Memory Multiprocessors*. PhD thesis, Yale University, 1992. Technical Report 931.

[13] T. Brecht, H. Sandhu, M. Shan, and J. Talbot. ParaWeb: Towards world-wide supercomputing. In *Proceedings of the Seventh ACM SIGOPS European Workshop on System Support for Worldwide Applications*, 1996.

[14] P. Broadbery and K. Playford. Using object-oriented mechanisms to describe Linda. In Wilson [164], pages 14–26.

[15] A. Burns and A.J. Wellings. *Real-Time Systems and Their Programming Languages*. Addison-Wesley, 1990.

[16] P. Butcher. Lucinda. In Wilson [164], pages 27–38.

[17] P. Butcher, A. Wood, and M. Atkins. Global synchronisation in Linda. *Concurrency: Practice and Experience*, 6(6):505–516, September 1994.

[18] P. Butcher and H. Zedan. Lucinda—a polymorphic Linda. In Banâtre and Métayer [9], pages 126–146.

[19] C.J. Calsen, I. Cheng, and P.L. Hagen. The AUC C++ Linda system. In Wilson [164], pages 39–73.

[20] D. Campbell, H. Osborne, A. Wood, and D. Bridge. Parallel case base retrieval: an implementation on distributed Linda. In *Proc. Ninth IASTED International Conference: PDCS '97*, pages 525–530. IASTED/Acta Press, 1997.

[21] D. Campbell, A. Wood, H. Osborne, and D. Bridge. Linda for case base retrieval: A case for extending the functionality of Linda and its abstract machine. In *Proc. Thirty-First Annual Hawaii International Conference on System Sciences*, pages 226–235. IEEE Computer Society, 1998.

[22] D.K.G. Campbell. Constraint matching retrieval in Linda: extending retrieval functionality and distributing query processing. Technical Report YCS 285, University of York, 1997.

[23] S. Cannon and L. Denys. A self-configuring distributed kernel for satellite networks. In P.H. Welch and A.W.P. Bakkers, editors, *Communicating Process Architectures 2000*, volume 58 of *Concurrent Systems Engineering Series*, pages 109–119. IOS Press, September 2000.

[24] S.R. Cannon and D. Dunn. Adding fault-tolerant transaction processing to LINDA. *Software—Practice and Experience*, 24(5):449–466, May 1994.

[25] B. Carpenter, G. Fox, S. H. Ko, and S. Lim. Object serialization for marshaling data in a Java interface to MPI. *Concurrency: Practice and Experience*, 12(7):539–553, June 2000.

[26] N. Carriero and D. Gelernter. The S/Net's Linda kernel. *Operating Systems Review*, 19(5):54–71, March 1985.

[27] N. Carriero and D. Gelernter. Applications experience with Linda. *ACM SIGPLAN Notices*, 23(9):173–187, September 1988.

[28] N. Carriero and D. Gelernter. How to write parallel programs: A guide to the perplexed. *ACM Computing Surveys*, 21(3):323–357, September 1989.

[29] N. Carriero and D. Gelernter. Linda in context. *Comm. ACM*, 32(4):444–458, April 1989.

[30] N. Carriero and D. Gelernter. *How to Write Parallel Programs: A First Course*. The MIT Press, 1990.

[31] N. Carriero and D. Gelernter. New optimization strategies for the Linda precompiler. In Wilson [164], pages 74–83.

[32] N. Carriero, D. Gelernter, D. Kaminsky, and J. Westbrook. Adaptive parallelism with Piranha. Technical Report 954, Yale University, 1993.

[33] N. Carriero, D. Gelernter, and J. Leichter. Distributed data structures in Linda. In *Proc. 13th Symposium on Principles of Programming Languages*, pages 236–242, January 1986.

[34] A.G. Chalmers. *A Minimum Path Parallel Processing Environment*. PhD thesis, University of Bristol, 1991.

[35] M. Chen, P. Townsend, and J.A. Vince, editors. *High Performance Computing for Computer Graphics and Visualisation*. Springer-Verlag, 1996.

[36] S. Chok and K. Marriott. Parsing visual languages. In *Proc. 18th Australasian Computer Science Conference*, pages 90–98, February 1995.

[37] P. Ciancarini. Parallel logic programming using the Linda model of computation. In Banâtre and Métayer [9], pages 110–125.

[38] P. Ciancarini, F. Franze, and C. Mascolo. Using a coordination language to specify and analyze systems containing mobile components. *ACM Trans. on Software Engineering and Methodology*, 9(2):167–198, 2000.

[39] P.G. Clayton, F.K. de Heer Menlah, G.C. Wells, and E.P. Wentworth. An implementation of Linda tuple space under the Helios operating system. *South African Computer Journal*, 6:3–10, March 1992.

[40] P.G. Clayton and G.M. Rehmet. Implementing adaptive parallelism on a heterogeneous cluster of networked workstations. In *Proc. Research Manpower in Computer Science: Report Back Conference*, December 1994.

[41] P.G. Clayton and G.M. Rehmet. Implementing adaptive parallelism on a heterogeneous cluster of networked workstations. In *Proc. 1995 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'1995)*, pages 571–580, November 1995.

[42] R. Cleaveland, S.A. Smolka, et al. Strategic directions in concurrency research. *ACM Computing Surveys*, 28(4), December 1996.

[43] J. Conde. Mobile agents in Java. URL: `http://wwwinfo.cern.ch/asd/rd45/-white-papers/9812/agents2.html`, December 1998.

[44] H.M. Deitel, P.J. Deitel, T.R. Nieto, T.M. Lin, and P. Sadhu. *XML: How to Program*. Prentice-Hall, 2001.

[45] A. Douglas, A. Wood, and A. Rowstron. Linda implementation revisited. In P. Nixon, editor, *Transputer and occam Developments (Proc. 18th World Occam*

and Transputer User Group Technical Meeting), pages 125–138. IOS Press, April 1995.

[46] C. Faasen. Intermediate uniformly distributed tuple space on Transputer meshes. In Banâtre and Métayer [9], pages 157–173.

[47] A. Ferrari. JPVM: The Java Parallel Virtual Machine. URL: `http://-www.cs.virginia.edu/~ajf2j/jpvm.html`, February 1999.

[48] M.J. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. Computers*, 21(9):948–960, 1972.

[49] M. Foster, N. Matloff, R. Pandey, D. Standring, and R. Sweeney. I-Tuples: A programmer-controllable performance enhancement for the Linda environment. In H.R. Arabnia, editor, *Proc. International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2001)*, pages 357–361. CSREA Press, June 2001.

[50] E. Freeman and S. Hupfer. A model for 3D interaction with hierarchical information spaces. Technical Report Yale-CS TR 1071, Yale University, May 1995. Position Paper, CHI '95 Research Symposium.

[51] E. Freeman, S. Hupfer, and K. Arnold. *JavaSpaces Principles, Patterns, and Practice*. Addison-Wesley, 1999.

[52] R.P. Futrelle and N. Nikolakis. Efficient analysis of complex diagrams using constraint-based parsing. In *International Conference on Document Analysis and Recognition (ICDAR-95)*, pages 782–790, August 1995.

[53] R.P. Futrelle and N. Nikolakis. Diagram analysis using context-based constraint grammars. Technical Report NU-CCS-96-01, College of Computer Science, Northeastern University, 1996.

[54] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing*. Scientific and Engineering Computation. MIT Press, 1994.

[55] G.A. Geist, J.A. Kohl, and P.M. Papadopoulos. PVM and MPI: A comparison of features. *Calculateurs Parallèles*, 8(2), 1996.

[56] D. Gelernter. Generative communication in Linda. *ACM Trans. Programming Languages and Systems*, 7(1):80–112, January 1985.

[57] D. Gelernter. Current research on Linda. In Banâtre and Métayer [9], pages 74–76.

[58] D. Gelernter and N. Carriero. Coordination languages and their significance. *Comm. ACM*, 35(2):97–107, February 1992.

[59] A.S. Glassner, editor. *An Introduction to Ray Tracing*. Academic Press, 1989.

[60] E.J. Golin. *A Method for the Specification and Parsing of Visual Languages*. PhD thesis, Brown University, 1991.

[61] P. Gray and V. Sunderam. IceT: Distributed computing and Java. In *Proceedings of ACM 1997 Workshop on Java for Science and Engineering*, June 1997.

[62] D. Grune, H.E. Bal, C.J.H. Jacobs, and K.G. Langendoen. *Modern Compiler Design*. Worldwide Series in Computer Science. John Wiley and Sons, 2000.

[63] P.B. Hansen. The programming language Concurrent Pascal. *IEEE Trans. Software Engineering*, 1(2):199–207, June 1975.

[64] P.B. Hansen. An evaluation of the Message-Passing Interface. *ACM Sigplan Notices*, 33(3):65–72, March 1998.

[65] C.G. Harrison, D.M. Chess, and A. Kershenbaum. Mobile agents: Are they a good idea? Technical report, IBM, March 1995.

[66] W. Hasselbring. Combining SETL/E with Linda. In Wilson [164], pages 84–99.

[67] R. Hempel. The MPI standard for message passing. In W. Gentzsch and U. Harms, editors, *High-Performance Computing and Networking, International Conference and Exhibition, Proceedings, Volume II: Networking and Tools*, volume 797 of *Lecture Notes in Computer Science*, pages 247–252. Springer-Verlag, 1994.

[68] G. Hilderink. CSP for Java. URL: `http://www.rt.el.utwente.nl/javapp`, October 2000.

[69] G. Hilderink, A. Bakkers, and J. Broenink. A distributed real-time Java system based on CSP. In *The Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, ISORC 2000*, pages 400–407, March 2000.

[70] C.A.R. Hoare. Communicating sequential processes. *Comm. ACM*, 21(8):666–677, August 1978.

[71] C.A.R. Hoare. *Communicating Sequential Processes*. International Series in Computer Science. Prentice Hall, 1985.

[72] T. Holvoet. *An Approach for Open Concurrent Software Development*. PhD thesis, Department of Computer Science, K.U.Leuven, December 1997.

[73] T. Holvoet and Y. Berbers. Reflective programmable coordination media. In H.R. Arabnia, editor, *Proc. International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2001)*, pages 1236–1242. CSREA Press, June 2001.

[74] S. Hupfer, D. Kaminsky, N. Carriero, and D. Gelernter. Coordination applications of Linda. In Banâtre and Métayer [9], pages 187–194.

[75] D.C. Hyde. Introduction to the programming language occam. URL: `http://www.eg.bucknell.edu/~cs366/occam.html`, March 1995.

[76] IBM. TSpaces. URL: `http://www.almaden.ibm.com/cs/TSpaces/index.html`.

[77] IBM. The TSpaces programmer's guide. URL: `http://www.almaden.ibm.com/cs/TSpaces/html/ProgrGuide.html`.

[78] IBM. TSpaces user's guide. URL: `http://www.almaden.ibm.com/cs/TSpaces/html/UserGuide.html`.

[79] IBM. The TSpaces vision. URL: `http://www.almaden.ibm.com/cs/TSpaces/html/Vision.html`.

[80] IBM. IBM Aglets software development kit. URL: `http://www.trl.ibm.co.jp/aglets/`, June 2000.

[81] ITU-T. Video coding for low bit rate communication. ITU Recommendation H.263 (02/98), 1998.

[82] S. Jagannathan. Expressing fine-grained parallelism using concurrent data structures. In Banâtre and Métayer [9], pages 77–92.

[83] Joint Photographic Experts Group. JPEG home page. ISO/IEC JTC1 SC29 Working Group 1. URL: `http://www.jpeg.org/`.

[84] G. Jones. *Programming in occam.* International Series in Computer Science. Prentice-Hall, 1987.

[85] D.L. Kaminsky. *Adaptive Parallelism with Piranha.* PhD thesis, Yale University, May 1994.

[86] T. Kielmann. Object-Oriented Distributed Programming with Objective Linda. In *First International Workshop on High Speed Networks and Open Distributed Platforms*, St. Petersburg, Russia, June 1995.

[87] T. Kielmann. *Objective Linda: A Coordination Model for Object-Oriented Parallel Programming.* PhD thesis, University of Siegen, Germany, 1997.

[88] T. Kielmann, P. Hatcher, L. Bougé, and H.E. Bal. Enabling Java for high-performance computing: Exploiting distributed shared memory and remote method invocation. *Comm. ACM*, October 2001. Accepted for publication in the special issue on High Performance Java.

[89] A.J.F. Kok. *Ray Tracing and Radiosity Algorithms for Photorealistic Image Synthesis.* PhD thesis, Delft University of Technology, 1994.

[90] E. Kühn. Introduction: How to approach the virtual shared memory paradigm. *Parallel and Distributed Computing Practices*, 1(3), June 1998. URL: `http://-orca.st.usm.edu/pdcp/vols/vol01no3introduction.html`.

[91] D. Lea. *Concurrent Programming in Java: Design Principles and Patterns.* Addison-Wesley, 1997.

[92] M.L. Liu. On the power of abstraction—a look at the paradigms and technologies in distributed applications. In *2001 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2001)*, June 2001. Poster presentation.

[93] N. MacDonald. Linda work in Edinburgh. In Wilson [164], pages 100–104.

[94] K. Marriott and B. Meyer. The classification of visual languages by grammar hierarchies. *Journal of Visual Languages and Computing*, 8(4):375–402, August 1997.

[95] S. Matsuoka and S. Kawai. Using tuple space communication in distributed object-oriented languages. In *Proc. OOPSLA '88*, pages 276–284, September 1988.

[96] D. May. Occam. *SIGPLAN Notices*, 18(4):69–79, April 1983.

[97] D. May and R. Taylor. Occam: An overview. *Microprocessors and Microsystems*, 8(2):73–79, March 1984.

[98] R. Menezes. Ligia: Incorporating garbage collection in a Java based Linda-like run-time system. In *Proc. Second Workshop on Distributed Systems (WOSID'98)*, pages 81–88, 1998.

[99] R. Menezes and A. Wood. Garbage collection in open distributed tuple space systems. In *Proc. Fifteenth Brazilian Computer Networks Symposium—SBRC '97*, pages 525–543, May 1997.

[100] R. Menezes and A. Wood. Coordination of distributed I/O in tuple space systems. In *Proc. Thirty-First Annual Hawaii International Conference on System Sciences*, pages 216–225. IEEE Computer Society, 1998.

[101] R. Menezes and A. Wood. Using tuple monitoring and process registration in the implementation of garbage collection in open Linda-like systems. In *Proc. Tenth IASTED International Conference: PDCS'98*, pages 490–495. IASTED/Acta Press, October 1998.

[102] B. Meyer. Pictures depicting pictures: On the specification of visual languages by visual grammars. Technical Report TRHA139, Fern Universität, 1992.

[103] B. Meyer. Pictures depicting pictures: On the specification of visual languages by visual grammars. In *IEEE Workshop on Visual Languages*, pages 41–47, September 1992.

[104] N. Minar. Computational media for mobile agents. URL: `http://-nelson.www.media.mit.edu/people/nelson/research/dc/dc.html`, December 1996.

[105] MPI Forum. Message Passing Interface (MPI) Forum home page. URL: `http://-www.mpi-forum.org/`.

[106] Object Management Group (OMG). CORBA basics. URL: `http://-www.omg.org/gettingstarted/corbafaq.htm`.

[107] S.S. Owicki and L. Lamport. Proving liveness properties of concurrent programs. *ACM Trans. Programming Languages and Systems*, 4(4):455–495, 1982.

[108] A.E. Paalder. A C++ implementation of a parser for visual languages based on relational grammars. Master's thesis, Universiteit Leiden, August 1994. `http://www.liacs.nl/MScThesis/paalder.94.ps.gz`.

[109] P.S. Pacheco. *A User's Guide to MPI.* University of San Francisco, March 1998. URL: `ftp://math.usfca.edu/pub/MPI/mpi.guide.ps`.

[110] J. Padget, P. Broadbery, and D. Hutchinson. Mixing concurrency abstractions and classes. In Banâtre and Métayer [9], pages 174–186.

[111] G.A. Papadopoulos and F. Arbab. Coordination models and languages. In Marvin V. Zelkowitz, editor, *The Engineering of Large Systems*, volume 46 of *Advances in Computers*, pages 329–400. Academic Press, August 1998.

[112] E. W. Parsons, M. Brorsson, and K. C. Sevcik. Predicting the performance of distributed virtual shared-memory applications. *IBM Systems Journal*, 36(4), 1997. URL: `http://www.research.ibm.com/journal/sj/364/parsons.html`.

[113] B.T. Phong. Illumination for computer generated pictures. *Comm. ACM*, 18(6):311–317, June 1975.

[114] S. Raina. Virtual shared memory: A survey of techniques and systems. Technical Report CSTR-92-36, Department of Computer Science, University of Bristol, December 1992.

[115] S. Raina. *Emulation of a Virtual Shared Memory Architecture.* PhD thesis, University of Bristol, 1993.

[116] G.M. Rehmet. Adaptive parallelism under Linda. In *Proc. Eighth National Computer Science research Students' Conference*, June 1993.

[117] G.M. Rehmet. Remora: Implementing adaptive parallelism on a heterogeneous cluster of networked workstations. Master's thesis, Rhodes University, 1995.

[118] G.M. Rehmet and P.G. Clayton. Peforming parallel computations over existing networks. In *Proc. Conference of the Computer Society of South Africa (CSSA'93)*, August 1993.

[119] J. Rekers. The use of graph grammars for defining the syntax of graphical languages. Technical report, Department of Computer Science, Leiden University, November 1994.

[120] J. Rekers. A course on visual languages. URL: `http://-www.wi.leidenuniv.nl/CS/SEIS/vislang/VLcourse.html`, 1995.

[121] J. Robie, J. Lapp, and D. Schach. XML Query Language (XQL). URL: `http://www.w3.org/TandS/QL/QL98/pp/xql.html`, September 1998.

[122] D. Rossi. Jada: Multiple object spaces for Java. URL: `http://-www.cs.unibo.it/~rossi/jada`.

[123] D. Rossi, G. Cabri, and E. Denti. Tuple-based technologies for coordination. In A. Omicini, R. Tolksdorf, G. Weiss, and F. Zambonelli, editors, *Coordination Models and Applications for Agents*. Springer, 2001.

[124] A. Rowstron. Mobile co-ordination: Providing fault tolerance in tuple space based co-ordination languages. URL: `http://www.research.microsoft.com/-~antr/papers/mobile.ps.gz`, 1999.

[125] A. Rowstron, A. Douglas, and A. Wood. A distributed Linda-like kernel for PVM. In J. Dongarra, M. Gengler, B. Tourancheau, and X. Vigouroux, editors, *Proc. EuroPVM '95*, pages 107–112, 1995.

[126] A. Rowstron and A. Wood. An efficient distributed tuple space implementation for networks of workstations. In L. Bougé, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *Proc. Euro-Par '96*, volume 1124 of *Lecture Notes in Computer Science*, August 1996.

[127] A. Rowstron and A. Wood. Solving the Linda multiple rd problem. In P. Ciancarini and C. Hankin, editors, *Coordination Languages and Models, Proc. Coor-*

dination '96, volume 1061 of *Lecture Notes in Computer Science*, pages 357–367. Springer-Verlag, 1996.

[128] A. Rowstron and A. Wood. BONITA: a set of tuple space primitives for distributed coordination. In *Proc. Thirtieth Annual Hawaii International Conference on System Sciences*, pages 379–388, Hawaii, 1997. IEEE Computer Society Press.

[129] U. Rüde. Vorlesungsskript technik des wissenschaftlichen rechnens. URL: `http://wwwzenger.informatik.tu-muenchen.de/lehre/skripten/wissrech/skript.html`, June 1995.

[130] S. Schneider. *Concurrent and Real-time Systems: The CSP Approach.* Wiley, 1999.

[131] G. Schoinas. POSBYL: Implementing the blackboard model in a distributed memory environment using Linda. In Wilson [164], pages 105–116.

[132] T.E. Schouten and H. van Nieuwkerk. RTLINDA: A Linda-like system for a real-time VMEbus environment. In Wilson [164], pages 117–123.

[133] H. Schulzrinne. RTP: About RTP and the Audio-Video Transport Working Group. URL: `http://www.cs.columbia.edu/~hgs/rtp/`, 1999.

[134] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A transport protocol for real-time applications. *Internet Draft RFC 1889*, January 1996.

[135] H. Schulzrinne, A. Rao, and R. Lanphier. Real Time Streaming Protocol (RTSP). *Internet Draft RFC 2326*, April 1998.

[136] Scientific Computing Associates. Virtual shared memory and the Paradise system for distributed computing. Technical report, Scientific Computing Associates, April 1999.

[137] D. Shires, R. Mohan, and A. Mark. An evaluation of HPF and MPI approaches and performance in unstructured finite element simulations. In H.R. Arabnia, editor, *Proc. International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2001)*, pages 13–19. CSREA Press, June 2001.

[138] A. Smith. Towards wide-area network Piranha: Implementing Java-Linda. URL: `http://www.cs.yale.edu/homes/asmith/cs690/cs690.html`.

[139] A.B. Sudell. Design and implementation of a tuple-space server for Java. URL: `http://www.op.net/~asudell/is/linda/linda.html`, December 1998.

[140] Sun Microsystems. Java Media Framework API. URL: `http://java.sun.com/-products/java-media/jmf/index.html`.

[141] Sun Microsystems. Jini connection technology. URL: `http://www.sun.com/-jini`.

[142] Sun Microsystems. Object serialization. Java 2 SDK, Standard Edition Version 1.3.0 Documentation, 2000.

[143] Sun Microsystems. System requirements. Java 2 SDK, Standard Edition Version 1.3.0, README File, 2000.

[144] V.S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–339, December 1990.

[145] O. Thomas. A Linda kernel for Unix networks. In Wilson [164], pages 124–128.

[146] R. Thomas, L.R. Rogers, and J.L. Yates. *Advanced Programmer's Guide to UNIX System V*. Osborne McGraw-Hill, 1986.

[147] R. Tolksdorf. Laura: A coordination language for open distributed systems. Technical Report 1992/35, Technische Universität Berlin, 1992.

[148] R. Tolksdorf. Laura: A coordination language for open distributed systems. In *Proc. 13th IEEE International Conference on Distributed Computing Systems ICDCS 93*, pages 39–46, 1993.

[149] R. Tolksdorf and D. Glaubitz. Coordinating web-based systems with documents in XMLSpaces. URL: `http://flp.cs.tu-berlin.de/~tolk/xmlspaces/-webxmlspaces.pdf`, 2001.

[150] L.G. Valiant. A bridging model for parallel computation. *Comm. ACM*, 33(8):103–111, August 1990.

[151] R. van der Goot, J. Schaeffer, and G. Wilson. Safer tuple spaces. In D. Garlan and D. Le Métayer, editors, *Proc. 2nd Int. Conf. on Coordination Models and Languages*, volume 1282, pages 289–301, Berlin, Germany, 1997. Springer-Verlag, Berlin.

[152] P. Walker. The Transputer. *Byte*, May 1985.

[153] P.H. Welch. "wot, no chickens?". URL: `http://www.cs.ukc.ac.uk/projects/ofa/java-threads/0.html`, September 1996.

[154] P.H. Welch. Communicating Sequential Processes for Java (JCSP). URL: `http://www.cs.ukc.ac.uk/projects/ofa/jcsp`, August 2000.

[155] P.H. Welch and J.M.R. Martin. Formal analysis of concurrent Java systems. In P.H. Welch and A.W.P. Bakkers, editors, *Communicating Process Architectures 2000*, volume 58 of *Concurrent Systems Engineering Series*, pages 275–301. IOS Press, September 2000.

[156] G.C. Wells and A.G. Chalmers. An extended Linda system using PVM. In *Proc. 1995 PVM Users' Group Meeting*, May 1995. URL: `http://www.cs.cmu.edu/Web/Groups/pvmug95.html`.

[157] G.C. Wells and A.G. Chalmers. Extensions to Linda for graphical applications. In *Proc. International Workshop on High Performance Computing for Computer Graphics and Visualisation*, pages 174–181, July 1995. Reprinted in [35].

[158] G.C. Wells, A.G. Chalmers, and P.G. Clayton. An extended version of Linda for Transputer systems. In B.C. O'Neill, editor, *Parallel Processing Developments (Proc. 19th World Occam and Transputer User Group Technical Meeting)*, pages 233–240. IOS Press, April 1996.

[159] G.C. Wells, A.G. Chalmers, and P.G. Clayton. An extended version of Linda for distributed multimedia applications. *SAICSIT '99*, November 1999. URL: `http://www.cs.wits.ac.za/~philip/SAICSIT/SAICSIT-99/papers_ideas.html`.

[160] G.C. Wells, A.G. Chalmers, and P.G. Clayton. A comparison of Linda implementations in Java. In P.H. Welch and A.W.P. Bakkers, editors, *Communicating Process Architectures 2000*, volume 58 of *Concurrent Systems Engineering Series*, pages 63–75. IOS Press, September 2000.

[161] G.C. Wells, A.G. Chalmers, and P.G. Clayton. Extending Linda to simplify application development. In H.R. Arabnia, editor, *Proc. International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2001)*, pages 108–114. CSREA Press, June 2001.

[162] T. Whitted. An improved illumination model for shaded display. *Comm. ACM*, 23(6):343–349, June 1980.

[163] G. Wilson. Improving the performance of generative communication systems by using application-specific mapping functions. In Wilson [164], pages 129–142.

[164] G. Wilson, editor. *Linda-Like Systems and Their Implementation.* Technical Report 91-13. Edinburgh Parallel Computing Centre, June 1991.

[165] A. Wood. Coordination with attributes. In P. Ciancarini and A.L. Wolf, editors, *Coordination Languages and Models: Proc. Third International Conference COORDINATION '99*, volume 1594 of *Lecture Notes in Computer Science*, pages 21–36, 1999.

[166] A.M. Wood and D.K.G. Campbell. Generic operations for concurrent knowledge manipulation architectures. In *Proc. Applied Informatics '98*, 1998.

[167] World Wide Web Consortium. Extensible markup language (XML). URL: `http://www.w3.org/XML`.

[168] World Wide Web Consortium. XML Path language (XPath) version 1.0. W3C Recommendation, URL: `http://www.w3.org/TR/xpath.html`, November 1999.

[169] World Wide Web Consortium. XQuery 1.0: An XML Query Language. W3C Working Draft, URL: `http://www.w3.org/TR/xquery/`, June 2001.

[170] P. Wyckoff, S. W. McLaughry, T. J. Lehman, and D. A. Ford. T Spaces. *IBM Systems Journal*, 37(3):454–474, 1998.

[171] M.R. Zargham. *Computer Architecture: Single and Parallel Systems.* Prentice-Hall, 1996.

[172] S.E. Zenith. The axiomatic characterization of Ease. In Wilson [164], pages 143–152.

[173] S.E. Zenith. A rationale for programming with Ease. In Banâtre and Métayer [9], pages 147–156.

Appendix A

Writing New Matchers

This appendix gives a more detailed overview of the support methods provided by the eLinda system for writing new matchers, explaining in greater detail the principles discussed in Chapter 4. It also presents a complete example of a matcher written for the Programmable Matching Engine.

A.1 Support Functions

The support functions provided by the eLinda system fall into one of two categories: those provided by the tuple and anti-tuple objects and those provided by the tuple space manager (for communication and accessing the tuple space).

A.1.1 Tuple Support Functions

There are two methods of the `Tuple` class that may be used when writing new matchers. These are summarised in Table A.1.

In the eLinda system the `AntiTuple` class is a subclass of the `Tuple` class, and so inherits the methods of Table A.1, while adding a number of others. These are summarised in Table A.3. These methods allow a matcher to determine the exact nature of the matching operation (i.e. to distinguish between the four different input operations), which fields it should make use of (i.e. those marked with `?=` in the ideal syntax), and also to apply the standard matching algorithm in various ways.

`boolean localTuple()` Returns an indication of whether the tuple is considered to "belong" to the local processing node. This allows a matcher to exclude non-local tuples if necessary (for example, to prevent double counting of tuples that have been broadcast).
`String getSignature ()` Returns a string representing the type signature of the tuple. This can be used to extract the type of each field, which may be necessary in a matcher (for example, to detect attempts to perform arithmetic matching operations on non-numeric fields). The field types are encoded as shown in Table A.2.

Table A.1: Methods of the `Tuple` Class Used in Matching

Code	**Type**
s	String
b	byte
h	short
i	integer
l	long
f	float
d	double
t	boolean
c	char
m	MultiMediaResource object
o	Object (used for any serializable Java object)

Table A.2: Encoding of Tuple Field Types

`long getMatcherMask ()` Get the bit mask specifying which fields should be used by the programmable matcher.
`boolean isMatcherField (int index)` Tell whether a field is a matcher field (i.e. one specified for use with a programmable matcher). This simply checks the matcher bit mask set by `setMatcherMask`.
`boolean isWild (int index)` Tell whether a field is a wildcard or not.
`long getFieldMask ()` Return a bit mask where wildcard fields are 0 and other fields are 1. The least significant bit of the bit mask corresponds to the first tuple field.
`boolean maskedMatch (Tuple t, long mask)` Matching algorithm to see if a tuple matches this anti-tuple, comparing only the fields specified by a bit mask. If there is a 1 in the bit mask then the comparison is made.
`boolean standardMatch (Tuple t)` Matching algorithm to see if a tuple matches this anti-tuple. This takes wildcards into account.
`boolean isPredicate ()` Tell whether the tuple space operation for this anti-tuple is a predicate form (i.e. `inp` or `rdp`).
`boolean isIn ()` Tell whether the tuple space operation for this anti-tuple is a deleting form (i.e. `in` or `inp`).

Table A.3: Matching Methods of the `AntiTuple` Class

`int scatter (AntiTuple at)` Distributes the given anti-tuple across the network to remote matchers. It returns the index number to be used for gathering the results.
`Tuple[] gather (int gatherIndex)` Returns the tuples resulting from a scatter operation. The parameter is the value returned by the corresponding `scatter` method.
`void broadcastDelete (Tuple t)` Broadcast a message to all other tuple spaces to delete the specified tuple (or anti-tuple).
`void replaceTuple (Tuple t)` Replaces the specified tuple into the tuple space. This can be used by matchers to return unneeded tuples.
`void replaceTuples (Tuple[] t)` Replaces the given array of tuples into the tuple space. This can be used by matchers to return unneeded tuples from scatter/gather operations. Null entries in the array are ignored.

Table A.4: Communication and Tuple Space Access Methods

A.1.2 Communication and Tuple Space Access

New matchers may interact directly with the eLinda system to access and replace tuples in the tuple space, communicate requests to other processors and subsequently retrieve the results of such requests, etc.

The methods available to a matcher for communication and tuple space access are all static methods of the `TSManager` class. They are summarised in Table A.4.

A.2 An Example Matcher

The example is a simple numeric matcher that returns the tuple with a specified field (or fields) closest to some given value(s). Matchers written for similar purposes would follow a very similar pattern and be of comparable complexity.

```
// ClosestMatcher.java

import gcw.eLinda.*;

/** This class implements a programmable matcher for the eLinda
  * system.
  * The result of this matcher is a tuple which contains
  * the closest match for the specified numeric fields of the
```

```
  * tuples.
  */
public class ClosestMatcher
    implements ProgrammableMatcher, java.io.Serializable
  { /** Type signature of the tuples.
      */
    private String signature;

    public ClosestMatcher ()
      {
      } // constructor

    private double findDiff (Tuple t, AntiTuple at)
        throws MatcherException
      { double diff = 0.0;
        int k;
        try
          { for (k = 0; k < signature.length(); k++)
              if (at.isMatcherField(k))
                switch (signature.charAt(k))
                  { case 'b':
                      diff += Math.abs(at.unpackByte(k) -
                        t.unpackByte(k));
                      break;
                    case 'h':
                      diff += Math.abs(at.unpackShort(k) -
                        t.unpackShort(k));
                      break;
                    case 'i':
                      diff += Math.abs(at.unpackInt(k) -
                        t.unpackInt(k));
                      break;
                    case 'l':
                      diff += Math.abs(at.unpackLong(k) -
                        t.unpackLong(k));
                      break;
                    case 'f':
                      diff += Math.abs(at.unpackFloat(k) -
                        t.unpackFloat(k));
                      break;
                    case 'd':
                      diff += Math.abs(at.unpackDouble(k) -
                        t.unpackDouble(k));
                      break;
                    default:
                      System.out.println("INTERNAL ERROR:" +
                        " ClosestMatcher: Non-numeric field!");
                      break;
```

```
            }
        }
      catch (InvalidTupleException e)
        { throw new MatcherException("ClosestMatcher: " +
              "Exception unpacking tuple: " + e.toString());
        }
      return diff;
    } // findDiff

  public Tuple matchList (AntiTuple a, TupleIterator it)
      throws MatcherException
    { long mask = a.getMatcherMask();
      if (mask == 0)
        throw new MatcherException("ClosestMatcher: " +
            "No matcher mask specified.");

      Tuple n = null, closest = null;
      boolean hadResults = false;
      int k, bitPos;
      double diff = 0.0;
      signature = a.getSignature();

      for (k = 0, bitPos = 1;
           k < signature.length();
           k++, bitPos <<= 1)
        if ((mask & bitPos) != 0)
          { if ("bhilfd".indexOf(signature.charAt(k)) == -1)
              throw new MatcherException("ClosestMatcher: " +
                "Non-numeric field specified (" + k + ").");
          }

      // If we survived the checking and setup distribute the
      // request to the other TSM's
      int gatherIndex = TSManager.scatter(a);

      // Now start iteration through the tuples
      try
        { n = it.firstTuple();
          while (n != null) // Calculate diffs
            { if (n.localTuple() && a.maskedMatch(n, ~mask))
                // Don't duplicate work of remote matchers
                if (! hadResults) // This is the first match
                  { diff = findDiff(n, a);
                    closest = n;
                    hadResults = true;
                  }
                else // Second or subsequent match
                  { double nDiff = findDiff(n, a);
```

```
                if (nDiff < diff)
                  { diff = nDiff;
                    closest = n;
                  }
              }
          n = it.nextTuple();
        }
      it.end();
    }
  catch (IterationException e)
    { throw new MatcherException("ClosestMatcher: " +
          "failed during iteration: " + e.toString());
    }

  // Now gather the global results
  Tuple[] results = TSManager.gather(gatherIndex);
  int closestIndex = -1;
  if (results != null)
    { for (k = 0; k < results.length; k++)
        if (results[k] != null)
          { if (! hadResults) // This is the first
              { diff = findDiff(results[k], a);
                closest = results[k];
                closestIndex = k;
                hadResults = true;
              }
            else // Second or subsequent match
              { double nDiff = findDiff(results[k], a);
                if (nDiff < diff)
                  { diff = nDiff;
                    closest = results[k];
                    closestIndex = k;
                  }
              }
          }
    }

  // Now sort out the results
  if (hadResults) // We have an answer!
    { if (a.localTuple())
        TSManager.broadcastDelete(a);
      if (a.isIn())
        // Must sort out tuples in tuple space
        { if (closestIndex != -1)
            // It was a scatter/gather tuple
            { results[closestIndex] = null;
                // Don't return this one
            }
```

```
              else // It was a local tuple
                try
                  { for (n = it.firstTuple();
                         n != null;
                         n = it.nextTuple())
                      // Restart local iterator
                      if (n.equals(closest))
                        { it.deleteCurrentTuple();
                          break;
                        }
                  }
                catch (IterationException e)
                  { throw new MatcherException(
                        "ClosestMatcher: failed during " +
                        "iteration: " + e.toString());
                  }
                TSManager.replaceTuples(results);
                  // Throw the unwanted ones back
            }
          return closest;
        }

      return null; // No result
    } // matchList

  public boolean match (AntiTuple a, Tuple t)
      throws MatcherException
    { if (a.maskedMatch(t, ~a.getMatcherMask()))
        return true; // Must be the closest so far!
      else
        return false;
    } // match

  /** Override toString to give matcher name.
    */
  public String toString()
    { return "ClosestMatcher";
    } // toString

} // class ClosestMatcher
```

Appendix B

Preprocessor Support

Traditionally Linda systems have made use of a preprocessor to integrate Linda (the coordination language) with the host language[19, 31]. This style of approach has been adopted in eLinda, but a preprocessor has not been written. This is partly due to the fact that the provision of a preprocessor was seen as incidental to the main area of interest, namely focussing on the programmable matching mechanism, distributed multimedia and efficiency concerns.

The author has previously had experience in implementing such a preprocessor, using C as the host language[39]. This appendix outlines some of the issues that a Java preprocessor for eLinda would need to take into account.

B.1 Requirements for an eLinda Preprocessor

All of the eLinda examples presented in this thesis have made use of the so-called "ideal syntax" that a typical preprocessor would support. As an example, consider the following simple code extract: `in("point", ?x, ?y)`. In eLinda without a preprocessor (or after preprocessing was done) this would actually appear as follows:

```
AntiTuple template = new AntiTuple("sii");
Tuple retrieved = null;
template.addString("point");
template.addWild('i');
template.addWild('i');
retrieved = space.in(template);
x = retrieved.unpackInt(1);
y = retrieved.unpackInt(2);
```

Note that this example has been simplified slightly—all exception-handling code has been omitted, as has the code required to declare and open the tuple space (called `space` in the example code above).

In order to perform this kind of transformation a preprocessor must parse the original source program, building up a symbol table with type information. As the syntax of Java for type declarations is considerably simpler than that of C this would not be particularly difficult. When the preprocessor encountered an eLinda operation, such as the `in` shown above, the type information from the symbol table could be used to construct the correct *type signature* required by the `AntiTuple` constructor (i.e. `"sii"` here), and also to specify the type of the wildcard fields (there is some redundancy here, which is useful for error-checking). The preprocessor would also need to generate the necessary assignments, required for the binding of the retrieved tuple values to the wildcard variables.

This is complicated slightly when the Programmable Matching Engine mechanisms must be supported. As an example of this, the following program code extract shows the Programmable Matching Engine being used to retrieve a tuple corresponding to some multimedia resource with a minimum cost value:

```
rdp.minimum(?videoStore, "Star Wars", ?=cost)
```

After preprocessing the modified source code would appear as follows (again, the code required for exception-handling and declaration of the tuple space has been omitted, as has the "unpacking" of the retrieved tuple):

```
AntiTuple template = new AntiTuple("ssf");
template.addWild('s'); // Video store name
template.addString("Star Wars"); // Video name
template.addWild('f'); // Cost
template.setMatcher(new MinimumMatcher());
template.setMatcherMask(0x4); // Bit mask
Tuple videoTuple = videoSpace.rdp(template);
. . .
```

Notice here how the necessary matcher has been created, and it and the bitmask specifying the field that must be used by the matcher have been added to the anti-tuple.

It should also be noted that the multimedia features in eLinda require no special support from the preprocessor. The only interaction is that the `MultiMediaResource` class is recognised as a known type by the preprocessor, and encoded as `'m'` in the type signatures for tuples and anti-tuples, as shown in the example above.

B.1.1 Tuple Space Analysis

One of the tasks normally performed by a Linda preprocessor is an analysis of the usage of the tuples. This allows the preprocessor to perform what amounts to a refactorisation of the tuples into disjoint sets that may be stored in separate tuple spaces[19, 31]. This kind of analysis requires a global view of an application so that all processes are considered. For example, it may be found that the master process in a particular application outputs tuples of the form `("work", someInt)`, and then retrieves tuples of the form `("result", ?someInt)`. In turn, the analysis of the worker processes will probably discover that they input tuples of the form `("work", ?someInt)` and create tuples of the form `("result", intResult)`. In a simple case like this it is easy to see that there are two distinct groups of tuples being used in this example: the work specifications and the results. Both sets of tuples have the same type signature, namely (*string, int*).

The first optimisation that a preprocessor can perform is to create two separate tuple spaces for the two groups of tuples. This has the advantage of preventing the master process from having to search through outstanding work specifications when retrieving results, and vice versa for the workers. Having done this, the second optimisation that can be performed by the preprocessor is to remove the strings from the tuples. These were initially necessary to distinguish work requests from results. Now that these groups of tuples are stored in separate tuple spaces this distinction is implicit in the specification of the tuple space to be used for a particular operation. Decreasing the redundant information in the tuples has the obvious advantage of decreasing the work that must be done by the matching algorithm.

The example considered above was extremely simple. In general, the task of performing the analysis and then the optimisation of the tuple space access operations may be considerably more complex. However, it is a valuable feature of a preprocessor, as it allows programmers to focus their attention on the development of the application at hand by automating the optimisation of the tuple space operations.

The eLinda system includes the features that would be required to support these optimisations. These are the ability to create distinct tuple spaces, based on a combination of a programmer-specified name and type signature information.

Further details of the analysis and optimisation performed by a typical Linda preprocessor can be found in [19, 31, 39]. An interesting perspective on the limitations of automatic optimisation for Linda systems can be found in [163].

B.2 Preprocessing: JavaSpaces and TSpaces

The initial design of eLinda was completed before JavaSpaces and TSpaces were released. Neither of these systems uses a preprocessor, but rather they both make use of the object-oriented features of Java to partially circumvent the need for preprocessing. Notably, they both use the standard syntax of Java for their tuple space operations.

On the negative side, neither system provides any kind of optimisation of the tuple space access operations. The sort of analysis and optimisation that was described earlier would need to be performed explicitly by programmers. Both systems do provide the ability to create multiple, distinct tuple spaces that can be used for this purpose.

The two systems are essentially very similar, but differ in the details of how tuples and anti-tuples are created. These mechanisms will be considered briefly in the next two sections.

B.2.1 TSpaces

TSpaces provides two ways of creating tuples:

1. Objects can be passed to the constructor of the `Tuple` class to create a tuple with these objects as its fields. An anti-tuple is created in exactly the same way, but using a `Class` object to specify the type of a wildcard field, rather than a normal data-containing object.

2. Programmers can create their own classes as subclasses of the provided `SubclassableTuple` class. It appears from the documentation that the matching method (i.e. `SuperTuple.matches()`) could be overridden in this case, providing a similar facility to that found in Objective Linda.

The objects created in either of these two ways may then be passed as parameters to the methods of the `TupleSpace` objects that implement the client side of the tuple spaces.

B.2.2 JavaSpaces

JavaSpaces implements support for tuples slightly differently. It provides a Java interface, called `Entry`, as part of the Jini system. This interface must be implemented by any class in an application that is to be used as a tuple. Any public object reference[1]

[1]The use of private fields or of primitive data fields is *not* supported.

contained in such a class is then effectively a field of the tuple. The wildcard fields in anti-tuples are specified by setting the corresponding reference fields to `null`.

As is the case for TSpaces, objects created in this way may be used with the methods of the `JavaSpace` objects that implement the tuple spaces.

B.2.3 Application to eLinda

The approaches adopted by TSpaces and JavaSpaces could be adopted for use in a redesigned eLinda system. This would have the advantage that the syntax is that with which Java programmers are already familiar. Additionally, the need to perform a separate preprocessing stage during compilation would be removed, with obvious performance benefits for the program development process. However the ability to perform automatic optimisations is not possible with this model, moving the responsibility of handling potentially complex optimisation steps onto the programmers using the system.