New and Improved: Linda in Java

George C. Wells

Department of Computer Science, Rhodes University, Grahamstown, South Africa
G.Wells@ru.ac.za

Abstract

This paper discusses the current resurgence of interest in the Linda coordination language for parallel and distributed programming. Particularly in the Java field, there have been a number of developments over the past few years. These developments are summarised together with the advantages of using Linda for programming concurrent systems. Some problems with the basic Linda approach are also discussed and a novel solution to these is presented. The power and flexibility of the proposed extensions to the Linda programming model are illustrated by considering a number of example applications, including a detailed case study of visual language parsing.

1 Introduction

The Linda[1] coordination language was proposed and developed in the mid-1980's by David Gelernter at Yale[6]. There was a great amount of interest in it as a model for parallel and distributed programming, but this waned through the early 1990's. In recent years there has been a considerable resurgence of interest in Linda, particularly in the Java[2] community.

Linda is a language for distributed and parallel programming that has a very appealing simplicity. It is based on a simple shared-memory paradigm and has only a handful of operations. While this simplicity introduces other problems, particularly with regard to performance and predictability[20], these are not insurmountable and much research was done in the early days of Linda to develop techniques to ameliorate these drawbacks[1,2,4].

The first section of this paper presents a brief overview of Linda. This is followed by a survey of Java implementations of Linda, with an emphasis on the

[1] Linda is a registered trademark of Scientific Computing Associates.
[2] Java is a registered trademark of Sun Microsystems Inc.

recent commercial developments in this area. Some of the problems that are inherent in the Linda model are then discussed, followed by the presentation of our solution to these problems. The power and flexibility of our solution is highlighted by considering two examples of the use of our extensions to the basic Linda programming model. Some limitations of our approach are also considered.

2 Overview of 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*. Thus, the Linda model can be categorised as a form of *virtual shared memory*[12], in which the actual memory system may be physically shared or distributed, but application programmers are provided with a simple, shared-memory model.

The tuple space is accessed using *associative addressing* to specify the required data objects, stored as *tuples*. An example of a tuple with three fields is `("point", 12, 67)`, where `12` and `67` are the x and y coordinates of the point represented by this tuple.

As a coordination language, Linda is designed to be coupled with a sequential programming language (called the *host language*—in our case, Java). Linda provides a programmer with a small set of operations. These operations may be categorised as *output* and *input* operations. There is a single output operation, used to place tuples into tuple space. This is called `out`, and is used as follows: `out("point", 12, 67)`. The input operations are used to retrieve tuples from tuple space. The basic forms are `in`, which removes the tuple from tuple space, and `rd`, which returns a copy of the tuple. The two input operations also have predicate forms (`inp` and `rdp`), which do not block if the required tuple is not present.

For the input operations, the specification of the tuple to be retrieved makes use of an associative matching technique whereby 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, the command `in("point", ?x, ?y)` might be used to retrieve the example tuple above (or any other tuple with a similar structure). The specification of the tuple used in an input operation is called an *antituple*. On successful completion of this input operation the variables `x` and `y` are bound to the values found in the matching tuple. The resultant communication between two processes is illustrated in Figure 1.

This simple model provides for very easy inter-process communication, and

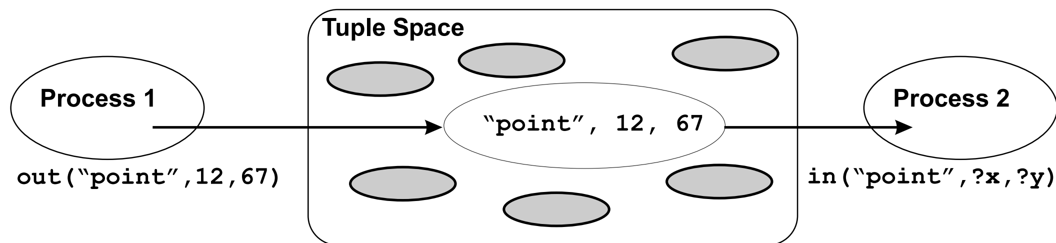

Fig. 1. A Simple Communication Pattern

common synchronisation operations are also easily implemented, using the blocking forms of the input operations. A particular advantage of Linda is that the processes involved in a distributed/parallel computation are completely decoupled, both temporally and spatially. Temporal decoupling arises from the fact that communication is asynchronous: the process that has generated a tuple can continue without waiting, and may even have terminated when the tuple is retrieved by another process. Spatial decoupling arises from the fact that the communication is effectively anonymous: there is no need for processes to be aware of the location of the other processing nodes in the network, and no "addressing information" is required for two processes to communicate. Further details of the Linda programming model may be found in [3].

3 Recent Linda Developments in Java

During the last few years a number of Linda implementations have been developed by research groups and commercial companies using Java as the host language. This paper considers the commercially-developed products, namely JavaSpaces, GigaSpaces, AutevoSpaces and TSpaces.

3.1 JavaSpaces

JavaSpaces[5] is a complex product and relies heavily on a number of other technologies developed by Sun Microsystems. As a result, configuring the JavaSpaces system and its applications is a very complex process.

JavaSpaces supports the basic Linda operations, although with slightly different names. Tuples (called *entries* in JavaSpaces) are created from classes that implement the Jini `Entry` interface, and only public fields that refer to objects are considered. Tuples are transmitted across the network using a non-standard form of serialisation. Matching of tuples is performed using byte-level comparisons of the data, not the conventional `equals()` method, and use is made of object-oriented polymorphism for matching sub-types of a class. Tuple

storage is centralised on a single server, and this may become a performance bottleneck in large systems.

JavaSpaces provides some extended functionality, especially in areas such as support for *transactions* and *leases*, which are important for commercial applications.

3.2 GigaSpaces

GigaSpaces[9] was developed as a commercial implementation of the JavaSpaces specification. As such, it is compliant with the Sun specifications, while adding a number of new features. These include operations on multiple tuples, updating, deleting and counting tuples, and iterating over a set of tuples matching an antituple. There are also distributed implementations of the Java Collections `List`, `Set` and `Map` interfaces, and a message-queuing mechanism.

Considerable attention has been paid to the efficient implementation of GigaSpaces. This includes the provision of facilities such as buffered writes, and indexing of tuples.

There is also support for non-Java clients to access GigaSpaces through the use of the SOAP protocol over HTTP. Lastly, there is support for web servers to make use of GigaSpaces to share session information.

3.3 AutevoSpaces

Like GigaSpaces, AutevoSpaces is a commercial implementation of the JavaSpaces specification. The focus of AutevoSpaces is on enterprise systems requiring high availability, including fail-over, recovery and load-balancing mechanisms. They claim that their "High Availability solution is the only commercially available implementation that provides semantic consistency with the JavaSpaces reference implementation. This consistency is essential to ensuring the correctness and flexibility of large, distributed, mission-critical applications"[10]. AutevoSpaces makes use of a distributed tuple space implementation to provide scalability and the high availability features.

3.4 TSpaces

TSpaces is a Linda system developed by IBM's alphaWorks research division[8,19]. It is considerably extended from the original Linda model, par-

ticularly in terms of support for commercial applications. The TSpaces implementation is extremely simple to setup and configure in comparison to JavaSpaces—all that is required is that a single server process be running on the network.

TSpaces supports a large number of operations, including new operations for the input and output of multiple tuples, and operations that specify tuples by means of a "tuple ID" rather than the usual associative matching mechanisms. There is also the `rhonda` operator, which performs an atomic synchronisation and data exchange operation between two processes. Lastly, there is an event mechanism, providing notification when a specified tuple is written to the tuple space or deleted from it.

In addition to the usual associative matching mechanism, TSpaces allows tuple input using so-called "indexed tuples". In this case, fields may be named, ranges of values may be used, and AND and OR operations may be specified. It is also possible to perform matching on XML data contained in tuples.

Tuples may have an expiration time set (similar to the lease mechanism in JavaSpaces), and there is also transaction support. Furthermore, access control mechanisms are provided, providing functionality similar to UNIX file system permissions.

3.4.1 XMLSpaces

XMLSpaces is a research project, built on TSpaces to extend the limited facilities that it has for matching XML data[14]. The XML support in XMLSpaces is provided by subclassing the `Field` class used by TSpaces. The new `XMLDocField` class overrides the matching method used by TSpaces to provide matching on the basis of the XML content of the field. The matching method may be provided by the application programmer, providing a great deal of flexibility for XML matching operations. A number of matching operations are supported, including the use of XML query languages such as XPath[18].

Further details of these Java implementations of Linda and other related research projects may be found in [17].

4 Problems with Linda

The simple associative matching mechanism used for the retrieval of tuples in Linda works very well in many situations. One-to-one and one-to-many communication patterns are trivial, and implementing semaphores, barrier syn-

chronisation, and other coordination and interprocess communication models is simple. However, situations do arise where the simple associative matching technique is not adequate.

As a simple example, consider a set of tuples, where an application needs to locate the tuple with the minimum value of some field. Using Linda to solve this problem is possible, but is not efficient. The application would need to retrieve *all* of the tuples, using repeated `inp` operations. These tuples would then need to be searched for the one with the minimum value. The tuples would then be returned to the tuple space (including the tuple with the minimum value, if the overall effect is to be that of a `rd` operation). During this procedure the tuples are not accessible by other processes, potentially restricting the degree of parallelism possible. Furthermore, in an implementation with a distributed tuple space, there is a large volume of network traffic generated by this solution.

While this is a simple example, it illustrates a general problem, namely that some applications may need a "global view" of the tuples in tuple space. Other examples include finding tuples with values "close to" some specified value, or lying within a specified range of values. These types of problems cannot be solved efficiently using the standard Linda associative matching technique. While some of the Linda systems described above, notably TSpaces, have provided extensions to the associative matching mechanism, none has addressed these issues.

5 The eLinda System

In an attempt to address the problem described in the preceding section, an alternative, flexible matching mechanism is proposed. We call this the *Programmable Matching Engine* (PME), and our Linda implementation *eLinda*.

The eLinda system is based closely on the standard Linda model. A number of different implementations of eLinda were developed in the course of the author's PhD research[15]. The most complex of these uses a fully-distributed tuple space model where any tuple may reside on any processor/node. The others use a centralised tuple space, with optional local caching of certain tuples. The fully-distributed model poses particular problems for matching, in that many processing nodes may be required to participate in a matching operation.

In addition to the programmable matching facilities, eLinda contains extensions to support the development of distributed multimedia applications. Further details of the eLinda system and its other features can be found in [15].

The Programmable Matching Engine allows the use of more flexible criteria for the associative addressing of tuples. This is useful in situations such as that exemplified above (finding the tuple with the minimum value for some field). As has already been noted, such queries can be expressed using the standard Linda associative matching methods, but will generally be quite inefficient. If the tuple space is distributed, searching for a tuple may involve accessing the sections held on all the processors. Ideally, this should be done in parallel. This problem is handled efficiently in eLinda by distributing the matching engine so that network traffic is minimised, and moving the necessary computation out of the application and into the matcher. For example, in searching for the minimum tuple, each section of the tuple space would be searched locally for the smallest tuple, which would then be returned to the node that originated the operation. The originating matcher would then select the smallest of all the replies received. This process is completely transparent to the application, which simply inputs a tuple, using a specialised matcher. From an application programmer's perspective this could be expressed simply as `in.minimum(?field1, ?=field2)`. The notation that is used is to follow the Linda input operation with the name of the matcher to be used [3]. The field (or fields) to be used by the matcher is denoted by `?=`.

In addition to this simple usage, matchers may also perform *aggregated operations* where a tuple is returned that in some way summarises or aggregates information from a number of tuples. For example, a matcher might calculate the total of numeric fields in some subset of the tuples in tuple space. It is also possible to write matchers that return multiple tuples, similar to the TSpaces "scan" operations.

New matchers are written as Java classes that implement a specific interface. This requires the implementation of two methods. One of these is used when checking all the tuples that are already in tuple space for a possible match. The other is used when the input operation has blocked and individual tuples need to be checked as they are added to the tuple space. These matching methods can make use of a simple library that provides controlled access to tuple space and communication between the distributed matchers.

[3] Note that this is an idealised syntax, such as might be supported by a Linda preprocessor. In practice the usual style of Java method calls is used.

5.2 Possible Applications of the Programmable Matching Engine

The simple examples of matchers given above may have hinted at possible applications, but have focussed on common numeric applications for simplicity. Some other examples of matchers that emphasise the power and flexibility of the Programmable Matching Engine are given below.

- A string matcher could match string fields using some alphabetic measure of "closeness", regular expressions, or even approximate homophonic matching.
- A spatial matcher could compare two fields, taken to be x and y coordinates, in order to locate a tuple corresponding to a point in some two-dimensional space (or, equivalently, in three or more dimensions). This possibility is discussed in more detail in Section 5.4 below.
- 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 select a tuple at random from some subset of the available tuples [4].
- A matcher could be written to extract XML-formatted information from a tuple and perform complex matching operations based on this (providing an equivalent to the XML support in TSpaces).

5.3 A Simple Application: Video-on-Demand

As an initial illustration of the use of the Programmable Matching Engine (and also 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. A number of practical issues such as security, payment verification, etc. are omitted from this application for simplicity.

5.3.1 The Video Server Application

Thisprogram initially places the details of the available videos into a tuple space called "videos". The server then waits for a tuple to be placed into a tuple space called "requests" with a matching supplier name. These request tuples specify a unique access key for the video required, and also contain

[4] By default, eLinda uses a FIFO queuing system.

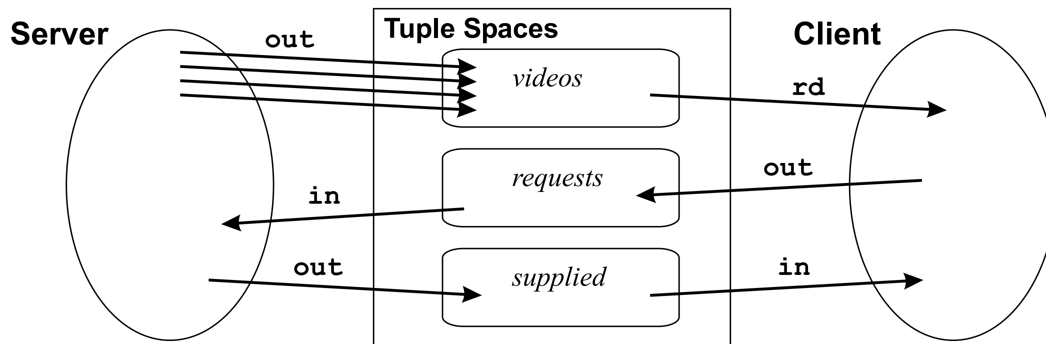

Fig. 2. Control Flow in the Video Server Application

payment details. The payment details are verified, and, if successful, a tuple is placed into a third tuple space, called "supplied". This tuple contains the unique key and a `MultiMediaResource` object that the client can retrieve in order to view the video. This process, and the client's interaction with the tuple spaces, is shown diagrammatically in Figure 2.

5.3.2 The Video Client Application

The outline of the client program is shown in Algorithm 1 (this has been expressed using a simple, procedural pseudocode notation for simplicity). This program is a GUI, event-driven Java application that allows a user to select a video and then view it. The user enters the title of a video, and the "videos" tuple space is then searched for a tuple with a matching title. This makes use of a PME matcher that retrieves the tuple with the minimum value in the cost field. Alternative matchers might also be provided for this purpose, which could take into account other issues, such as the quality of the video and the network bandwidth available. 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 a positive response is received, the payment details are requested from the user and the remainder of the interaction between client and server described above is completed.

Algorithm 1

Get videoName from user
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"

5.3.3 Discussion

While this example is a simple illustration of the principles involved in such an application, and particularly of the use of the Programmable Matching Engine and 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 can simplify the development of such applications. Additionally, it highlights some of the benefits of the basic Linda programming model, such as the spatially-decoupled nature of the communication: neither the client nor the server have to be aware of each other's location in the network. The next section presents a larger, more complex example of the use of eLinda.

5.4 Visual Language Parsing

As a more significant example, which highlights the flexibility and power of the Programmable Matching Engine, we will consider the problem of parsing visual languages in a little more detail. Visual languages are used in many areas to depict situations or activities in a pictorial form which is often easier for human beings to comprehend than a textual format. Examples abound, not least in the field of Computer Science where notations such as flowcharts, state transition diagrams, UML diagrams, etc. are widely used. If such graphical models are to be "understood" by a computer system there is a requirement for parsing them in order to analyse their structure. This 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 (the keywords and other tokens of the language). 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 [11]. The method that we will consider here is the use of *picture layout grammars* (a variation on *attributed multiset grammars*), as developed by Eric Golin[7]. Picture layout grammars provide a particularly flexible and powerful way of expressing the syntax of visual languages. Much of the following discussion is based on a course on visual languages which can be found in [13].

Table 1
Picture Layout Grammar for State Transition Diagrams

```
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
```

5.4.1 Picture Layout Grammars

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, location, 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, for example:

```
State → contains(circle, text)
```

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 is not part of the left hand symbol, but which must be present as part of the *context* in which the rule can be applied. This is usually shown by underlining the context symbols to distinguish them from normal symbols.

A complete grammar for state transition diagrams is shown in Table 1. This describes the common form of state transition diagram as exemplified in Figure 3. In the grammar in Table 1, the production for `Arc` (7) shows the use of a context symbol: the `circle` symbol is not a part of an arc, but must be present as the context in which the `Arc` production can be used.

Formally, an *attributed multiset grammar* can be defined as follows:

Definition 1 *An* attributed multiset grammar *is a six-tuple* (N, Σ, s, I, D, P) *where:*

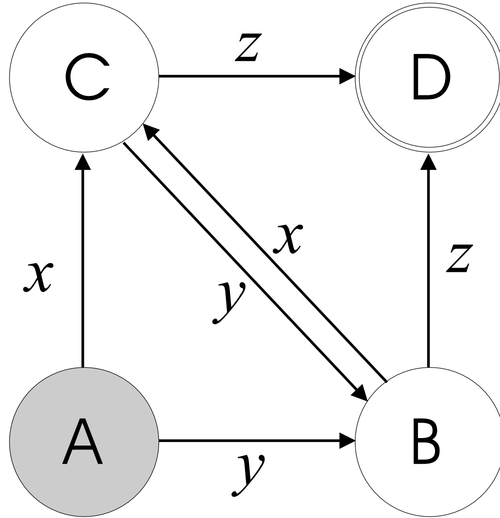

Fig. 3. An Example State Transition Diagram

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

Definition 2 *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*

We then introduce the concept that a picture M is *analyzable*.

Definition 3 *M is analyzable if M has a derivation tree T where:*

The leaf nodes of T spell out M
The root node of T is labelled by 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, 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, with the child nodes fulfilling the constraints of the production used to generate the non-terminal symbol, and the attributes of the non-terminal symbol generated from the attributes of the child nodes by means of the semantic function associated with the production.

The language $S(G)$ recognised by a grammar G can then be formally defined as follows:

Definition 4 $S(G) = \{M \mid M \in \Sigma^* \wedge M \text{ is analyzable over } G\}$

Picture layout grammars are attributed multiset grammars, as defined above, but with some simple restrictions.

While picture layout grammars are a powerful formalism for defining visual languages they are difficult to parse efficiently. Golin reports a theoretical complexity result of $O(n^9)$, although in practice the situation is seldom this bad. 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*), essentially a tree structure with cross-links, giving a directed acyclic graph (DAG). This data structure must then be checked to remove invalid results, and then traversed again to pick a unique valid result. As a result of this complexity, parsers for picture layout grammars can benefit from a distributed or parallel implementation.

5.4.2 The Use of eLinda for Parsing Picture Layout Grammars

The eLinda system is ideally suited to a problem like the parsing of picture layout grammars. The symbols are simply represented by tuples, which contain the class and the attributes as fields. In practice, the terminal symbols of a particular picture need to be kept in two separate data stores: one from which symbols are consumed as they are used (or *reduced*) by the application of productions, and one in which the symbols remain for use in determining the *context* for other productions. Again, this can easily be solved by placing the initial terminal symbols (the output of a graphics editor, or a visual "lexical analyser") in two separate tuple spaces, which we will refer to as the *reduction tuple space* and the *context tuple space*.

The application of the production rules in a picture layout grammar is a task that can easily be done in parallel, as there is no clear sequential association betweensymbols, and exhaustive searching is required to determine the relationships between symbols. This can be handled using a "replicated worker" pattern[5], where each worker obtains a symbol from the reduction tuple space, then searches for a production rule that can be applied, retrieving any other symbols used by the production rule as required, and checking the constraint functions (this involves searching the context tuple space). The

```
("circle", 10, 10, 5)
("text", "A", 7, 7)
```

Fig. 4. Two Example Tuples during Visual Parsing

constraint functions themselves are ideally suited to implementation as specialised matchers in the Programmable Matching Engine (a library of useful spatial functions such as “contains”, “enters”, “touches”, etc. could be written to support the common features of visual languages).

As a concrete example, consider a parser for the language of state transition diagrams, using the grammar in Table 1. One simple step in parsing a state transition diagram is to apply the production rule stating that a State comprises a Circle and a Text object, with the constraint that the Text is contained within the Circle (rule 4 in Table 1). Let us assume that the two tuples shown in Figure 4 describe two such symbols, where the Circle is attributed with the coordinates of the central point (here, $x = 10$ and $y = 10$) and the radius (5). The Text object is attributed with the value of the text (`"A"`) and the coordinates of the top left corner of the bounding box.

Using the approach outlined above, a worker process might retrieve the tuple describing the Circle and begin to search for productions which could be applied. One production which could be applied is the production describing a Double Circle as a circle contained within another circle (rule 8). This might be tried and discarded if no other circle is found with a common centre point. Continuing a search through the productions the process will eventually get to the production describing a State. The tuple corresponding to the Text object could then be retrieved using the Programmable Matching Engine matcher for the “contains” function. At this stage a new tuple can be added to the reduction tuple space, describing the State symbol. This would contain references to the constituent Circle and Text symbols, in order to build up the FMD structure. This process continues until the set of symbols in the reduction tuple space contains only the start symbol.

This parsing stage is then followed by a checking stage. This is required as the parsing process generates many redundant or illegal paths in the FMD. Fortunately, the checking can also be easily parallelised, using the same simple replicated worker pattern.

In implementing the visual parsing application using eLinda, four specialised matchers were developed, as, at a number of points, there is a need to use complex criteria to specify the tuples to be retrieved from tuple space. Of these matchers, two are general-purpose, and may be useful in other applications. While not an accurate metric of code complexity, the number of lines of code does give an approximate indication of the complexity of a matcher, and has been 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 grammar 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).

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 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 tuple space for consideration. Due to some complexities introduced by the parallelisation of the algorithm, this symbol needs to be chosen from a specified set of symbols. This matcher could be also used by any other 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.

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. 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`.

5.4.3 Discussion

Parallelising the 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, separate phase, allows the worker processes to perform the construction phase with little communication or synchronisation.

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

A major advantage of the use of eLinda in this application is that attributed multisets map very naturally to tuple spaces. Additionally the use of the Programmable Matching Engine simplified the parallel version considerably. 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.

More generally, the use of the Linda model also simplified other aspects of the application, such as the barrier synchronisation required between the two phases of the parsing process.

5.5 Limitations of the Programmable Matching Engine

There are some limitations to the kinds of matching operations that are supported by the PME. Notably, some matching operations may require a complete, global view of the tuple space (e.g. where a tuple is required that has the *median* value of some field). In such situations the use of the PME may not be ideal, as all the tuples involved *must* be examined in order to find the result. However, it is important to note that such problems are handled no less efficiently than if the application were forced to handle them directly, using a conventional Linda system.

Furthermore, the use of the PME may help to minimise the network bandwidth requirements in such cases. For example, taking the example of finding the tuple with the median value of some field, the distributed matchers could return lists containing only the specified field values to the requesting node. These lists could be combined to calculate the median value, and then the matching tuple could be requested from the node that held it. In this way, both the number of network transactions and the network bandwidth requirements are minimised in comparison to the standard Linda model. With a centralised tuple space, the improvements are even more dramatic, as only the required result tuple needs to be transmitted across the network to the requesting node. Of course, this comes at the price of higher computational requirements of the server itself.

6 Results and Conclusions

The visual language parser discussed above is a complex application that serves as a very good demonstration of the power and flexibility of eLinda and the Programmable Matching Engine.

Testing has shown that the performance of eLinda is on a par with that of other Java Linda systems such as JavaSpaces, TSpaces and GigaSpaces[17], which is remarkable considering that these other systems are commercially developed products. The testing also revealed the fact that Java is not an ideal platform for fine-grained parallel processing applications on conventional networks[16] (however, the performance of the Java Virtual Machine has been the focus of ongoing development, and this situation is improving). For coarser-grained distributed programming problems, the inherent simplicity of the Linda programming model is highly desirable and has led to the increasing interest in Linda-based approaches, particularly in the Java community.

One of the weaknesses of the simple associative matching mechanism used in the original Linda programming model is that it is limited for some applications. The Programmable Matching Engine developed for eLinda offers a solution that is both simple and elegant, and caters for a range of different implementation strategies. The benefits of the Programmable Matching Engine have been confirmed through its use in a number of different application areas, with both fully-distributed and centralised tuple space implementations.

Future work will focus on the application of the principles introduced in eLinda to the field of web services. The distributed nature of the extensions in eLinda and the increasing use of web services for the transparent provision of information across wide area networks suggest that this may be a fruitful application area for a system based on the Linda model and incorporating the flexible matching approach of the Programmable Matching Engine.

Acknowledgements

This work was supported by the Distributed Multimedia Centre of Excellence in the Department of Computer Science at Rhodes University, South Africa, with funding from Telkom SA, Business Connexion, Comverse, Verso Technologies and THRIP. Financial support was also received from the National Research Foundation (NRF) of South Africa, and from Rhodes University. The author also wishes to acknowledge the valuable advice and support of Alan Chalmers (University of Bristol) and Peter Clayton (Rhodes University) over many years, and the constructive feedback of the anonymous referees, both of the original PPPJ'04 paper and of this extended paper.

References

[1] 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.

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

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

[4] N. Carriero and D. Gelernter. New optimization strategies for the Linda precompiler. In G. Wilson, editor, *Linda-Like Systems and Their Implementation*, Technical Report 91-13, pages 74–83. Edinburgh Parallel Computing Centre, June 1991.

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

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

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

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

[9] GigaSpaces Technologies Ltd. GigaSpaces. URL: `http://www.gigaspaces.com/index.htm`, 2001.

[10] Intamission Ltd. AutevoSpaces: Product overview. URL: `http://www.intamission.com/downloads/datasheets/AutevoSpaces-Overview.pdf`, 2003.

[11] 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.

[12] 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.

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

[14] 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.

[15] G.C. Wells. *A Programmable Matching Engine for Application Development in Linda*. PhD thesis, University of Bristol, U.K., 2001.

[16] 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.

[17] G.C. Wells, A.G. Chalmers, and P.G. Clayton. Linda implementations in Java for concurrent systems. *Concurrency and Computation: Practice and Experience*, 16:1005–1022, August 2004.

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

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

[20] S.E. Zenith. A rationale for programming with Ease. In J.P. Banâtre and D. Le Métayer, editors, *Research Directions in High-Level Parallel Programming Languages*, volume 574 of *Lecture Notes in Computer Science*, pages 147–156. Springer-Verlag, 1992.