Extending Linda to Simplify Application Development

George Wells
Peter Clayton
Department of Computer Science
Rhodes University
Grahamstown, South Africa

Alan Chalmers
Department of Computer Science
University of Bristol
Bristol, U.K.

Abstract ***This paper describes a new implementation of Linda in Java, called eLinda. This system includes extensions designed to simplify the development of distributed applications, and to enhance the efficiency of communication in a distributed memory environment. These features are described, and compared with the extended features found in other Java implementations of Linda (i.e. JavaSpaces and TSpaces), highlighting the power and simplicity of the extensions in eLinda. The application of eLinda to various practical problems is also discussed, and, in particular, the paper focusses on the use of eLinda for parsing visual languages as a specific case study.***

Keywords: Linda, Java, tuple space, visual language parsing

1 Introduction

This paper describes an extended version of Linda, called eLinda, developed by the authors using Java. This system is compared with two other recent Linda implementations in Java (JavaSpaces from Sun Microsystems[2], and TSpaces from IBM[5]) and its applications are discussed.

The Linda model was first proposed in the 1980's by David Gelernter[3]. This approach to distributed and parallel programming offers a number of advantages as it is based on a shared memory paradigm with a small set of simple operations to access shared data.

The extensions introduced in eLinda have been designed with a view to providing increased flexibility for application development and to making some of the underlying communication issues more explicit, thus providing the programmer with a greater level of control of the communication. JavaSpaces and TSpaces also include some extensions and differences to the original Linda model as proposed by Gelernter. These are aimed mainly at support for commercial applications, but TSpaces does include some extra flexibility, along similar lines to the unique features of eLinda.

2 The Linda Programming Model

Linda is a *coordination language* for parallel and distributed processing, providing a communication mechanism based on a logically shared memory space called *tuple space*. 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*). The host language used in this work is Java. Linda effectively provides a programmer with a small set of operations that may be used to place tuples into tuple space (`out`) and to re-

trieve tuples from tuple space (`in`, which removes the tuple, and `rd` which returns a copy of the tuple). The latter two operations also have predicate forms (`inp` and `rdp`) which do not block if the required tuple is not present. 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.

Further details of the Linda model of distributed/parallel programming may be found in [1].

3 The Extensions in eLinda

The extensions to the original Linda model in the eLinda system take three forms. The first, and most important, is a mechanism to allow powerful, customised searching algorithms to be integrated into the eLinda system efficiently. The second is an additional form of output operation, which provides the programmer with a greater degree of control of the underlying network communication. Lastly, support for multimedia data types has been added.

3.1 The Programmable Matching Engine

The first extension is to provide a *Programmable Matching Engine* (PME) for the retrieval of tuples, 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 which has a value that is "close to" some specified value, rather than strictly equal. Such queries can usually be expressed using the standard Linda associative matching methods, but will generally be quite 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. If the tuple space is distributed, searching for a tuple may involve accessing the sections held on all the processors 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 a special form of matcher. For example, in searching for the "largest" tuple, each section of the tuple space would be searched locally for the largest tuple and that returned to the matcher running in the originating process, which would then select the largest of all the replies received. This process is completely transparent to the application, which simply inputs a tuple, using a specialised matcher. Matchers may also perform aggregated operations where a tuple is returned that in some way summarises or aggregates information from a number of tuples. It is also possible to write matchers that return multiple tuples.

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 (a simple example is where a tuple is required that has the *median* value of some field). In such situations the facilities offered by the PME may not be ideal. However, it is important to note that such problems are handled no less efficiently than if the application were to handle them directly, using a conventional Linda dialect. Furthermore, there are often possibilities for minimising the network bandwidth requirements by using the PME. Lastly, the use of the PME will usually simplify application development, particularly where a prewritten matcher is available.

Writing matchers is not a trivial operation. As an indication of the complexity of writing a customised matcher, a matcher to find the total of numeric tuple fields, is written in 175 lines of Java code.

3.2 Explicit Broadcast Communication

Two types of output operation are provided in eLinda to reflect explicitly a choice of optimised internal tuple space communication strategies. The first is a "point-to-point" mechanism (using non-replicated data) and the

second a "broadcast" mechanism (using replicated data). This contrasts with previous Linda systems where data is written to tuple space using a single instruction (`out`), but may then be read using one of two 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.

To allow the programmer to take advantage of this behaviour and the fact that the tuple space may be distributed across many processors, a new output operation, `wr`, has been added in eLinda. This operation broadcasts copies of the tuple throughout the processor network, whereas `out` simply places a single tuple in the local tuple space. These mechanisms provide the programmer with the necessary facilities to express shared, read-only access to data (using `wr` and `rd`), or exclusive, delete/modify access (using `out` and `in`). It should be noted that these forms of usage are not enforced by the system, and, as a result, the semantics of the `wr` operation are identical to those of `out`. The only difference is in efficiency.

3.3 Support for Distributed Multimedia Applications

The third distinctive feature of eLinda is its support for multimedia data. Tuples in eLinda may contain `MultiMediaResource` objects in addition to any of the eight primitive data types supported by Java and `String` objects[1]. The `MultiMediaResource` class acts as a wrapper to the underlying Java Media Framework (JMF) multimedia resource. In particular the implementation of the `MultiMediaResource` class provides support (transparent to the application programmer) for any necessary buffering of data, fetching or streaming of multimedia data across the network, etc. Multimedia applications will not be considered further in this paper, but further details of this aspect of eLinda may be found in [8].

[1] Additionally, any serializable Java object may be added to a tuple, although this limits the type checking that can be performed by the eLinda system

4 Other Linda Implementations in Java

As has already been mentioned, both Sun Microsystems and IBM have released Linda implementations in Java. This section describes their features and compares them with eLinda and the original Yale Linda model. It is worth noting that both JavaSpaces and TSpaces make use of the object-oriented features of Java (i.e. inheritance, polymorphism and interfaces) to circumvent the need for a preprocessor.

4.1 JavaSpaces

JavaSpaces[2] 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 with antituples (called *templates*) 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. A centralised tuple storage approach is used 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. There are also a

few other minor differences from the original Yale Linda model.

4.2 TSpaces

The TSpaces implementation is fairly simple — all that is required is that a single server process be running on the network. Again, this centralised server model may become a performance bottleneck.

TSpaces supports a large number of operations. The basic Linda operations are provided, again with slightly different names. There are also a number of other operations that allow for the deletion of tuples, the input or output of multiple tuples, and operations that specify tuples by means of a "tuple ID" rather than the usual associative matching mechanisms.

Tuples in TSpaces are objects consisting of a number of `Field` objects. The associative matching process then uses `Field` objects with a class type for a wildcard (e.g. `String.class`). Matching can be done using so-called indexed tuples (fields are named; ranges of values may be included; AND and OR operations are supported), and queries using XML[10]. A further interesting feature is the provision of an "event registration" mechanism, whereby an application can be informed when a certain tuple is written to the tuple space. These features are all easily implemented using the PME facilities in eLinda.

Tuples may have an expiration time set, providing similar functionality to the lease mechanism in JavaSpaces, and there is also transaction support. Furthermore, access control mechanisms are provided, based on user names, passwords and groups.

TSpaces also incorporates a feature allowing new commands to be added to the system in a way similar to the PME. However, the addition of new commands is a more complex process than adding a new matcher to an eLinda application. On the other hand, the centralised data storage model in TSpaces means that writing new command handlers for TSpaces is somewhat simpler than writing a customised matcher for eLinda. There is no provision in TSpaces for field values to effectively act as "in/out" parameters, as is required for matching operations such as finding the tuple with a field value closest to some given (input) value.

4.3 Comparison of eLinda with JavaSpaces and TSpaces

In essence, JavaSpaces is a relatively simple implementation of Linda. The only major extensions are the provision of transaction support and leases for commercial applications. TSpaces provides similar commercial features, but extends the basic Linda model considerably with its flexible and extensible matching facilities. Similarly, eLinda provides considerably extended matching functionality, but does not include the support for business applications found in the two commercial implementations.

Some performance comparisons have also been done, and are reported in [9]. Essentially all three systems have similar performance, with TSpaces slightly more efficient than the other two. JavaSpaces has a particularly high overhead for system initialisation.

5 The Application of eLinda

The eLinda system has been designed to support distributed processing applications on networks of workstations. The PME allows eLinda to be used efficiently in application areas where Linda may not otherwise be suited. The discussion in section 3.1 shows this clearly, where an application in a standard Linda system would need to retrieve *all* the potential tuples, compare the field values to locate the one with the maximum value, and then return all the other tuples to the tuple space. In eLinda, this could be handled by a matcher where each local portion of the tuple space is searched for the local maximum, which is then returned to the originating node to select the global maximum from the set of tuples received. In this case the network traffic is decreased, and, more importantly, the application itself is greatly

simplified. In this way eLinda provides powerful, efficient support for complex tuple retrieval and tuple aggregation operations which are useful for many applications.

5.1 Applications of the PME

A few simple, numeric examples of matchers have already been given that may have hinted at some possible applications. However, the PME is far more flexible than these examples would suggest. For example, a matcher could match string fields using some alphabetic measure of "closeness", or even approximate homophonic matching. As another example, matcher could make use of "fuzzy logic" to locate a tuple with some associated degree of certainty of its suitability.

5.2 Visual Language Parsing

As a particular example, which highlights the flexibility and power of the PME, 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, 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 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. For example, tokens may be related by inclusion, by contact, by relative position, 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 [6]. 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[4]. 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 [7].

5.2.1 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, one production from a grammar for State Transition Diagrams might be:

`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.

While picture layout grammars are a powerful formalism for defining visual languages they are difficult to parse efficiently. Golin reports a worst case theoretical complexity result of $O(n^9)$. The main cause of this complexity is that the first stage of the parsing algorithm produces multiple possible results: the *factored multiple derivation* (*FMD*) structure, es-

sentially 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 concurrent implementation.

5.2.2 The Use of eLinda for Parsing Picture Layout Grammars

The heart of the sequential visual parsing process is the algorithm shown in Table 1. A replicated worker pattern was used to parallelise this, with the workers performing the main loop of the program (lines 5–17). The second phase of the parsing process (checking the FMD structure) is also done in parallel. This problem provided considerable scope for the use of the PME facilities. Four new matchers were written to support the application, two of which are specific to the problem domain and two of which are of general applicability:

RHSMatcher 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. It effectively replaces lines 7–10 of the parsing algorithm in Table 1.

ConstraintMatcher This is used when applying rules, to check the constraints of the attributes of the symbols. It effectively implements lines 14 and 15 of the parsing algorithm in Table 1. It returns multiple matching tuples.

SetMatcher This matcher is used by the workers to retrieve a symbol chosen from a set of allowed symbols. 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.

AllMatcher This matcher can be used to retrieve all the tuples matching a given anti-tuple. It is used during the checking phase of the visual parsing application. Again, this matcher could be used by any application that needed to retrieve all the tuples meeting some criterion.

Preliminary testing indicates that the parallel algorithm shows speedup as the number of processing nodes increases. Initial indications are that a serial version may be more efficient, particularly for the first phase of the algorithm (i.e. building the FMD), however further testing and development remains to be done. By the time of the conference the testing process should be completed and detailed results will be presented.

6 Conclusions

The Linda model for parallel and distributed programming has always held much promise due to its inherent simplicity. Until recently this promise has not been realised to any great extent. It now seems that two major companies in the computer industry (Sun Microsystems and IBM) are adding momentum to the adoption of Linda as a viable mechanism for coordinating distributed systems, particularly in Java, to take advantage of its portability.

The work described in this paper builds on the underlying strengths of the Linda approach while adding to its functionality in ways that address the weaknesses of the original model. In addition, the eLinda project has extended the categories of applications to which Linda may be suited through the provision of flexible matching facilities. This is illustrated by the application of eLinda to the complex problem of parsing visual languages using picture layout grammars.

Acknowledgments

This work was supported by the Distributed Multimedia Centre of Excellence in the Department of Computer Science at Rhodes University, with funding from Telkom SA, Lucent Technologies, Dimension Data and THRIP.

Table 1: The Visual Parsing Algorithm

```
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 a new node for p to todo and FMD
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 a new node for p to todo and FMD
17     move next from todo to done
18   return FMD
```

M is the set of input symbols, P is the set of production rules

References

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

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

[3] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, Jan. 1985.

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

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

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

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

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

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

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