The implementation of Linda In C and associated problems

Stephan Bettermann
Edith Cowan University

Recommended Citation

This Thesis is posted at Research Online.
https://ro.ecu.edu.au/theses_hons/394
You may print or download ONE copy of this document for the purpose of your own research or study.

The University does not authorize you to copy, communicate or otherwise make available electronically to any other person any copyright material contained on this site.

You are reminded of the following:

- Copyright owners are entitled to take legal action against persons who infringe their copyright.

- A reproduction of material that is protected by copyright may be a copyright infringement. Where the reproduction of such material is done without attribution of authorship, with false attribution of authorship or the authorship is treated in a derogatory manner, this may be a breach of the author’s moral rights contained in Part IX of the Copyright Act 1968 (Cth).

- Courts have the power to impose a wide range of civil and criminal sanctions for infringement of copyright, infringement of moral rights and other offences under the Copyright Act 1968 (Cth). Higher penalties may apply, and higher damages may be awarded, for offences and infringements involving the conversion of material into digital or electronic form.
The Implementation of Linda in C
and associated Problems

By

Stephan Bettermann

A Thesis
Submitted in Partial Fulfilment of the Requirements for the Award of
Bachelor of Applied Science (Information Science) Honours
at the Faculty of Science and Technology
Edith Cowan University
Perth, Western Australia, Australia

Supervisor: Geoff Sutcliffe
Date of Submission: 13 January 1992
USE OF THESIS

The Use of Thesis statement is not included in this version of the thesis.
ABSTRACT

The parallel paradigm is accepted as playing a role of ever increasing importance. As expressed by Andrews (1975) quoted in Gelernter (1985, p. 80), there are "three basic kinds of mechanism, and three corresponding models of concurrent programming: monitors (shared variables), messages passing, and remote operations."

The Linda model, first defined in 1983 by David Gelernter, represents a new fourth model.

Linda provides "mechanisms for inter-process communication, process creation, and inter-process synchronisation" (Berndt, 1989, p. 1), in the form of six Linda operators, that are injected into a host language producing a parallel dialect of the host language. Processes communicate and synchronise by adding to, reading, and possibly removing "information structured as tuples" (Yuen and Wong, 1990, p. 2) from a common data area called the tuple space.

The programming language C was originally designed by Dennis Ritchie. Like Linda it is very simple in its design. C compilers for different machine architectures are widely available. Furthermore, as Kernighan and Ritchie (1988, p. xi) point out, "C is not tied to any particular hardware or system". These two factors make systems written in C portable.

This thesis describes the embedding of Linda into C resulting in DOS-C-Linda. DOS-C-Linda is targeted for an environment of PC's (Personal Computers) running DOS (Disk Operating System) linked by a LAN (Local Area Network) accessed through NetBIOS.

The Linda operators are injected via a Linda library. The tuple space and PC's
where processes created by the process operators may be run, are provided by the
**DOS-C-Linda Machine.** The DOS-C-Linda Machine consists of a **Tuple Server** and
multiple **Client Shells**, each on a separate PC connected to a LAN. The Tuple Server
provides and manages the tuple space and the DOS-C-Linda Machine.

A performance analysis of DOS-C-Linda reveals that applications using
DOS-C-Linda can show the same performance gains that applications using Linda
running on specialised hardware and software have shown.

Two applications have been implemented using DOS-C-Linda. The first application
is the dining philosophers problem posed by Dijkstra, the second a parallel genetic
algorithm.
DECLARATION

I certify that this thesis does not incorporate, without acknowledgment, any material previously submitted for a degree or diploma in any institution of higher education and that, to the best of my knowledge and belief, it does not contain any material previously published or written by another person except where due reference is made in the text.

Signature
Date  13.2.1992

Edith Cowan University  Stephan Bettermann
ACKNOWLEDGMENTS

My supervisor Geoff Sutcliffe supplied the source code for the genetic algorithm. I would like to thank him for sitting down with me to turn it into a parallel version. I also must thank him for scrupulously proof-reading all draft copies and giving helpful and constructive guidance in the writing of the proposal and thesis as well as in the development of C-Linda.

I would like to thank my fellow student Vincent Clive Phillips for proof-reading all draft copies, giving helpful hints and guidance, and for being a worthy opponent.

But most of all I must thank my parents Karl-Heinz and Petra Bettermann. I owe them much gratitude for without their help, support and understanding I would not have had the chance to study.
TRADEMARK ACKNOWLEDGMENTS

The following terms are known to be trademarks:

IBM and NetBIOS are registered trademarks of International Business Machines Corporation.

Microsoft and Microsoft C are registered trademarks of Microsoft Corporation.

Ethernet is a registered trademark of Xerox Corporation.

Turbo C is a registered trademark of Borland International, Inc.

Intel is a trademark of Intel Corporation.

Linda is a trademark of Scientific Computing Associates, New Haven.
# Table of Contents

1 Introduction ............................................. 1
   1.1 Background ....................................... 1
   1.2 Significance of the Study ......................... 2
   1.3 Research Objectives ................................ 3
   1.4 Limitations ....................................... 3
   1.5 Structure of Thesis ................................ 4

2 Theoretical Framework ..................................... 6
   2.1 Concurrency Models ................................ 6
      2.1.1 Remote Operations ............................ 6
      2.1.2 Shared Memory Models ......................... 7
         2.1.2.1 Semaphores ................................ 7
         2.1.2.2 Monitors .................................. 8
      2.1.3 Message Passing ............................... 8
   2.2 Linda ............................................. 8
      2.2.1 The Tuple Space ................................ 9
      2.2.2 Tuples ....................................... 10
      2.2.3 Tuple Matching ................................ 11
      2.2.4 Process Creation ................................ 12
      2.2.5 The Operators ................................ 13
         2.2.5.1 out ...................................... 13
         2.2.5.2 in ....................................... 13
         2.2.5.3 inp ...................................... 14
         2.2.5.4 rd ....................................... 14
         2.2.5.5 rdp ..................................... 15
         2.2.5.6 eval ..................................... 15

Edith Cowan University

Stephan Bettermann
# Table of Contents

2.2.6 Discussion .. 15
2.3 The Physical Environment .. 16
2.3.1 NetBIOS .. 16

3 Literature Review .. 19
3.1 Literature on Linda .. 19
3.2 Previous Implementations .. 21
3.2.1 Languages .. 21
3.2.2 Environments .. 21
3.3 Applications .. 22
3.4 Significant Issues .. 23
3.4.1 Addresses or References .. 23
3.4.2 The eval operator .. 24
3.5 The Need for the Research .. 25

4 Design .. 26
4.1 The DOS-C-Linda Machine .. 26
4.2 The DOS-C-Linda Operators .. 27
4.2.1 Data Tuple Operators .. 28
4.2.2 Tuple Specification .. 30
4.2.2.1 Basic Types .. 31
Formal Fields .. 34
4.2.2.2 Complex Types .. 35
Arrays .. 35
Structures .. 36
Arrays and Structures .. 39
Pointers to Values .. 40
4.2.3 Live Tuple Operators .. 43
4.3 Linda Library or Linda Compiler .. 45
4.3.1 Linda Compiler .. 45
4.3.2 Linda Library .. 46

Edith Cowan University
Stephan Bettermann
5 Implementation ........................................... 48

5.1 The DOS-C-Linda Machine ............................. 48

5.1.1 The Tuple Server ..................................... 50

5.1.2 The Client Shells ..................................... 51

5.1.2.1 Information Transferal between Client Shells and Clients 52

5.1.3 Tuple Server - Client Shell Communication .......... 54

5.1.3.1 Logging Client Shells into the Tuple Server .......... 54

5.1.3.2 Executing commands in Client Shells ................. 54

5.1.3.3 Logging Client Shells out .......................... 54

5.1.3.4 Shutting Down the Tuple Server ...................... 55

5.1.3.5 Communication Problems ........................... 55

5.2 Implementing the DOS-C-Linda Operators................. 55

5.2.1 The linda_out operator .............................. 56

5.2.1.1 Data Tuple Representation ......................... 56

5.2.1.2 Building of Data Tuples ............................ 57

5.2.1.3 Sending Data Tuples ............................... 63

5.2.2 The linda_in Operator ............................... 67

5.2.2.1 Matching Tuples .................................... 68

5.2.2.2 Compressing Matching Tuples ....................... 74

5.2.2.3 Mapping Tuples ..................................... 74

5.2.2.4 Mapping References ............................... 75

5.2.2.5 Mapping Structures ............................... 76
# Table of Contents

Mapping Arrays ........................................... 76

5.2.3 The linda_inp operator .......................... 77
5.2.4 The linda_rd Operator ............................ 77
5.2.5 The linda_rdp Operator ........................... 77
5.2.6 The linda_eval Operator .......................... 78
  5.2.6.1 Sending Processes ................................ 78
  5.2.6.2 Storing Processes ............................. 79
5.2.7 The linda_evalp operator ......................... 81

6 Validation and Evaluation ............................ 83
  6.1 Validation ......................................... 83
    6.1.1 Unit Testing .................................... 85
    6.1.2 System Testing .................................. 85
  6.2 Evaluation ......................................... 86
    6.2.1 Hardware and Software Environments ............ 87
    6.2.2 DOS-C-Linda Operator Execution Delays ....... 88
    6.2.3 Tuple Transfer Time ............................. 92
    6.2.4 Communication Costs ............................. 93
    6.2.5 Matrix Multiplication Problem .................. 95
    6.2.6 Conclusions ..................................... 99
  6.3 Implemented Applications ......................... 100
    6.3.1 The Dining Philosophers ......................... 100
      6.3.1.1 First Method ................................ 101
      6.3.1.2 Second Method ................................ 102
    6.3.2 Parallel Genetic Algorithm ....................... 102

7 Summary ................................................. 104
  7.1 The Linda Paradigm .................................. 104
  7.2 The Implementation .................................. 104
  7.3 Performance Analysis ................................ 105
  7.4 Future Research Directions ......................... 105
Table of Contents

7.5 Conclusions .................................................. 106

8 Bibliography .................................................. 107

Appendix A - Floppy Disk ..................................... 112

Appendix B - User Manual ..................................... 113
LIST OF FIGURES

Figure 1: The Client Table ..................................... 50
Figure 2: Network Information Record ............................ 52
Figure 3: Data Structure representing Tuples ..................... 56
Figure 4: Internal representation of structure_array ............. 60
Figure 5: Non-contiguous memory areas needed for values string .... 60
Figure 6: Internal representation of pointer structure ............ 62
Figure 7: Format of Buffer used to transfer Tuples over network .... 63
Figure 8: List of Tuples with the same Tuple Signature .......... 65
Figure 9: The Tuple Space ...................................... 66
Figure 10: Set of Matching Tuples ................................ 69
Figure 11: Set of Lodged Requests for Tuple Signature .......... 71
Figure 12: List of Outstanding Processes .......................... 81
Figure 13: Execution time of multiplying 64x64 matrices .......... 96
Figure 14: Execution time of multiplying 32x32 matrices .......... 97
Figure 15: Effect of Grain Size on Execution Times .............. 99
LIST OF TABLES

Table 1: Field Type Specifiers for Basic Types .................................. 31
Table 2: Type Modifiers for Basic Types ........................................... 32
Table 3: Field Type Specifier Normalisation ....................................... 67
Table 4: Execution Delays of the C-Linda Operators ............................ 90
Table 5: Transfer times of Linda operators compared to message passing ... 94
Table 6: Execution Times of multiplying 32x32 matrices ....................... 98
1 INTRODUCTION

1.1 Background

David Gelernter of Yale University first proposed the Linda paradigm in 1983 (Whiteside and Leichter, 1988, p. 192). Since then it has received much attention as a mechanism for extending existing languages into the parallel paradigm. Linda is a parallel programming mechanism that entails the use of a small set of operators to manipulate a common data area called the tuple space. Injecting Linda into a host language X creates a parallel dialect of X called X-Linda.

Linda represents a new approach of providing parallelism. As expressed by Andrews (1975) quoted in Gelernter (1985, p. 80), there are:

Three basic kinds of mechanism, and three corresponding models of concurrent programming: monitors (shared variables), messages passing, and remote operations.

Gelernter (1985, p. 80) argues that Linda "is sufficiently different from all three to constitute a fourth model." Berndt (1989, p. 1) expresses the advantage of this fourth model by revealing that:

One of the difficulties associated with parallel programming is communication among cooperating processes. Linda simplifies the problem of parallel process communication by supporting an uncoupled programming style. All process communication is via Linda operations on tuple space [sic]. This uncoupled process model frees the programmer from the low-level concerns about explicit synchronisation common in many parallel programming systems.
Introduction

Linda is a new parallel model that promotes a highly uncoupled programming style. This frees the programmer from explicitly being concerned with other processes.

1.2 Significance of the Study

Authorities such as Weston (1990, p. 81), Jellinghaus (1990, p. 70), Bortmann, Herdieckerhoff and Klein (1988, p. 1) see the parallel paradigm as playing a role of ever increasing importance. The well matured area of single thread execution is coming to the end of its technology curve. Linda is an easy way into the newer parallel paradigm by extending existing and well understood languages.

Even though C is the most common host language for Linda injections, no evidence is presented in the literature of an embedding of Linda into C for a Local Area Network (LAN) of Personal Computers (PC's) running DOS (Disk Operating System).
1.3 Research Objectives

The primary objective of this research has been to develop a C-Linda called DOS-C-Linda. The target environment is a LAN of PC's running DOS. The LAN interface to be used is NetBIOS, an extension to BIOS (Basic Input Output System).

The development of DOS-C-Linda encompasses the resolution of the following problems:

- How can the Linda operators be injected into C?
- How can the Linda environment be provided on a LAN of PC's running DOS?

The secondary objective is to compare the DOS-C-Linda with other implementations, such as the S/Net Linda Kernel by Carriero and Gelernter (1986), and Modula-Linda by Bormann and Herdieckerhoff (1989). This will place DOS-C-Linda into perspective with other research.

1.4 Limitations

A complete performance analysis of DOS-C-Linda is out of the scope of this thesis. A brief analysis is carried out by comparing DOS-C-Linda with the S/Net Linda Kernel and Modula-Linda.

Making the implementation fault tolerant is also out of scope. Methods of introducing fault tolerance into Linda are briefly investigated in the literature review.
1.5 Structure of Thesis

As the thesis involves the development of software it is appropriate that the structure of the Software Development Life Cycle (SDLC) is adopted. The SDLC consists of the following steps:

1. Analysis
2. Design
3. Implementation
4. Testing
5. Quality Evaluation
6. Maintenance

This defines the structure of the thesis. The second chapter, the theoretical framework, defines Linda and its place in the world of parallelism. The physical environment, in which DOS-C-Linda was implemented, is examined.

Complementary to the analysis phase, other Linda implementations are studied in the literature review in the third chapter.

The fourth chapter describes the design of DOS-C-Linda, involving the combination of the Linda paradigm and the physical environment. The issues of how the Linda operators are injected into C, and how the Linda environment is provided in the physical environment, are resolved.

The fifth chapter describes the implementation of DOS-C-Linda. The data structures used by DOS-C-Linda are presented.

The sixth chapter is concerned with the testing and evaluation of DOS-C-Linda. It consists of three sections. The first section describes the methods used to ensure that
Introduction

DOS-C-Linda works correctly.

The second section compares DOS-C-Linda with other Linda implementations. As explained in section 1.4, this evaluation is rather small. It consists of two programs that provide performance data. The first program is a simple "ping-pong" program that compares the communication costs of DOS-C-Linda against those of the S/Net Linda Kernel and Modula-Linda. The second program is a matrix multiplication program. It is modelled after a matrix multiplication program Carriero and Gelernter (1986) use to provide performance data for the S/Net Linda Kernel. This allows comparison and evaluation of DOS-C-Linda.

The third section describes the implementation of two applications using DOS-C-Linda. The first application solves the Dining Philosophers problem posed by Dijkstra in 1971. The second application is a parallel genetic algorithm.

Chapter seven is a summary of the thesis. Improvements and additions that might be included in future implementations are discussed.

Appendix A is a floppy disk containing all the source code and executable files of DOS-C-Linda, example programs, and programs used for the evaluation phase.

Appendix B is the User Manual for DOS-C-Linda. The user is assumed to be familiar with Linda and the C programming language.
2 Theoretical Framework

This chapter is divided into three sections. The first section examines existing concurrent programming models. The second section introduces, and defines, the Linda model. The third section describes the physical environment that DOS-C-Linda is to run in.

2.1 Concurrency Models

Concurrent processes working on the same problem must, at some point in time, communicate, requiring process communication and synchronisation. Andrews (1975) quoted in Gelernter (1985, p. 80) states that there are three models that provide these capabilities:

1. remote operations,
2. shared memory,
3. and message passing.

2.1.1 Remote Operations

One type of remote operation is remote procedure calling. Remote procedure calling is based on the concept of the conventional procedure call. However unlike conventional procedure calls, remote procedure calls invoke procedures residing on processors separate from the processor on which the calling process resides. As in conventional procedure calling data communication between the remote procedure and the caller occurs through a parameter list. The calling process suspends until the remote procedure has completed execution, thus making remote procedure calling impractical for concurrently executing modules.
For a more detailed discussion on remote procedure calling the reader is referred to Coulouris and Dollimore (1988).

2.1.2 Shared Memory Models

The shared memory model enables processes to communicate and synchronise by providing some shared memory. "However, when processes operate on shared objects at the same time, a collision is bound to result, unless some kind of control is imposed on the manner in which the objects are to be shared between the processes" (Krishnamurthy, 1989, p. 42). Semaphore and monitor mechanisms provide this required control.

2.1.2.1 Semaphores

In 1965 Edsger Dijkstra devised semaphores that control access to a shared data area (Sebesta, 1989, p. 357). Access to the data area is controlled by two operators, $P$ and $V$. Processes use these operators to seize and relinquish exclusive control over the shared data area.

Calling the $P$ operator before accessing the data area signals the desire for exclusive ownership. If another process is currently accessing the data area, $P$ queues the request and blocks. After accessing the data area the operator $V$ is called to relinquish control of the data area.

---

1 P stands for the Dutch word *proberen*, which means *to probe* or *to try* (Sebesta, 1989, p. 346). Krishnamurthy (1989, p. 49) states that P stands for the Dutch word *passeren* which means *to pass*.

2 Sebesta (1989, p. 357) states that V stands for the Dutch word *verhogen* which means *to increase*. Krishnamurthy (1989, p. 49) states that V stands for *vrijgeven* meaning *to release* or *to give away*. 

---

Edith Cowan University Stephan Bettermann
Theoretical Framework

The problem with this mechanism is that the programmer controls exclusive ownership over the data area (Sebesta, 1989, p. 360). Failing to call \( v \) means other processes may remain blocked; not calling \( p \) means the data area is not guaranteed to be mutually exclusive. The operators \( p \) and \( v \) do not have to be called to access the data area. The use of semaphores is voluntary.

2.1.2.2 Monitors

Monitors were devised to overcome the problem inherent in semaphores. Instead of providing a common data area and operators to control access to it, the operators needed to manipulate the common data are supplied. The common data area and the operators are encapsulated into an abstract data type. This overcomes the weaknesses of semaphores, as access to the data structure can only be obtained by using the operators.

2.1.3 Message Passing

Unlike the two models above, message passing involves the direct passing of data between processes.

To pass a message from one independent process to another, the processes must be synchronised with each other. When a process wants to receive a message it indicates its interest and waits for another process to indicate it wants to send a message. When a process is ready to send a message it signals its desire to do so and waits for another process to signal that it wants to receive a message. The message exchange occurs only when the two processes rendezvous.

For a more detailed discussion on the shared memory and message passing models the reader is referred to Sebesta (1989).
2.2 Linda

The Linda paradigm offers a fourth alternative to these models. In the same manner that monitors incorporate and extend the concept of semaphores, Linda builds on the concepts of semaphores, monitors, and message passing.

Linda makes a common data area, the tuple space, accessible to all Linda processes via the Linda operators. The tuple space can only be manipulated with the Linda operators, giving it an object oriented appearance.

2.2.1 The Tuple Space

The tuple space is a collection of tuples. Processes communicate and synchronise by placing into, and reading and removing tuples from, the tuple space. Any number of identical tuples may exist in the tuple space. Tuples placed into the tuple space remain in the tuple space until removed by a process. The tuple space is an associative memory in that tuple access is not by use of an address but by type and content. Process creation takes place by placing a live tuple into the tuple space.

The data integrity problems inherent with common data areas, as discussed in section 2.1.2, are handled by Linda in an interesting manner. A tuple cannot be updated simultaneously by two processes as "all tuples must be removed [from the tuple space], updated, and replaced. This process makes explicit the access right to a tuple" (Berndt, 1989, p. 14), because the removal of the tuple from the tuple space makes it unavailable to other processes. Therefore, "access conflicts are dealt with by the nature of the tuple space" (Berndt, 1989, p. 14).
2.2.2 Tuples

A tuple is an ordered collection of fields. The number of fields determines a tuple’s arity. Fields are either actual or formal. Actual fields have both a type and a value. Formal fields have a type but no value. The type of a field is drawn from the host language into which Linda is injected. The type of a tuple is the cross-product of its field types.

Syntactically a field is a variable or a value. Assume the following C variable definition exists:

```c
int the_answer = 42;
```

The following is a tuple containing one actual field of type `int` with value 42. Therefore the type of the tuple is `int`:

```
( the_answer )
```

Preceding a field with a '?' identifies it as a formal. Thus the following is a tuple containing one formal field of type `int`:

```
( ?the_answer )
```

Similarly, the following is a tuple containing two fields. The first field is actual of type `int` with value 42, the second is formal of type `int`. Therefore the type of the tuple is `int x int`:

```
( the_answer, ?the_answer )
```
Theoretical Framework

2.2.3 Tuple Matching

The semantics of tuple matching are as follows. Tuples match if they are of the same arity and the corresponding fields of the two tuples match. The following rules apply to tuple field matching:

- Actual fields match formal fields if the types correspond.

- Actual fields match if the types correspond and the values match (the host language supplies the equality function).

Note that formal fields do not match formal fields, as stated by Whiteside and Leichter (1988, p. 193), Xu and Liskov (1989, p. 199), Jellinghaus (1990, p. 75), and Sutcliffe and Pinakis (1990, p. 2). The reason for this is historical. The first Linda implementation was an injection of Linda into C (Whiteside and Leichter, 1988, p. 193); C does not have mechanisms to represent formal tuple fields. Formal fields not matching formal fields is acceptable because, as formal fields do not have values, there is nothing to match.

Assume the following C variable definitions exist:

```c
char character = 'C';
int integer = 42;
```

The following two tuples:

- `(character, integer)` of type `char x int`
- `(character, integer)` of type `char x int`

match because the two tuples are of the same arity of 2, and because the corresponding fields match. They match because an actual field matches a formal
field when the types correspond. As the first tuple’s first field’s type of char
correspond with the second tuple’s first field’s type of char the first fields of the
two tuples match. The second fields also match because their types of int
correspond.

The following two tuples:

( character, integer ) of type char × int
( 'C', ?integer ) of type char × int

also match because the first fields’ types of char correspond and the value of 'C'
matches. The second fields also match because their types of int agree.

The tuples:

( ?integer, ?integer ) of type int × int
( character, ?integer ) of type char × int

do not match because the types of the first fields int and char) do not correspond
and because the second fields are both formal.

2.2.4 Process Creation

Process creation occurs by placing a live tuple into the tuple space. For each field in
a live tuple a separate process is created to evaluate it. These processes run in
parallel with the process that placed the live tuple into the tuple space. Upon
completion of all the processes, the live tuple turns into an ordinary data tuple
indistinguishable from any other data tuple. The values of the fields in the data tuple
are evaluated by the created processes.
2.2.5 The Operators

Linda provides the following six operators:

```
out
in
inp
rd
rdp
eval
```

### 2.2.5.1 out

The out operator places a tuple into the tuple space. The tuple:

```
( "The Answer is", 42 ) of type [] x int
```

is placed into the tuple space by the following statement:

```
out( "The Answer is", 42 )
```

The tuple contains two fields, the first field of type [] with value "The Answer is", the second of type int with value 42. Therefore the type of the tuple is [] x int.

### 2.2.5.2 in

The argument of the in operator defines a tuple template. The in operator finds a tuple in the tuple space that matches the tuple template and extracts it from the tuple space. If more than one matching tuple exists in the tuple space, one tuple is chosen.
nondeterministically\(^3\). If no matching tuple exists, the process that initiated the in operation blocks until a matching tuple is placed into the tuple space. If at a later point in time a matching tuple is placed into the tuple space, outstanding requests for that tuple are serviced in a nondeterministic order. Thus the process that first requested the tuple and subsequently blocked is not necessarily serviced first.

Actual fields of the extracted tuple are assigned to corresponding formal fields of the tuple template. Assume the following tuple exists in the tuple space:

\[ ( \text{"The Answer is"}, 42 ) \] of type \([ \] \times \text{int}\)

The following in operation removes this tuple from the tuple space and assigns 42 to the formal field \(?The\_Answer\):

\[
\text{in( "The Answer is", ?The\_Answer )}
\]

### 2.2.5.3 inp

The inp operator is the predicate form of the in operator. It is similar to the in operator except that it does not block if the tuple space does not contain a matching tuple. inp indicates failure by returning \text{False}.

### 2.2.5.4 rd

The rd operator is very similar to in. The difference is that matching tuples are not removed from the tuple space.

---

\(^3\) The use of nondeterminism when choosing among alternative processes in the parallel paradigm was introduced by Dijkstra in 1975. It is believed that this provides a form of fairness (Sebesta, 1989, p. 365).

---

Edith Cowan University

Stephan Bettermann
2.2.5.5 rdp

The rdp operator is the predicate form of rd. If no matching tuple exists in the tuple space rdp indicates failure by returning False.

2.2.5.6 eval

The eval operator places live tuples into the tuple space. Assuming a function Square_Root exists that returns the square root of it's integer parameter, the following statement places a live tuple containing two fields into the tuple space:

\[
\text{eval}( \text{Square_Root}(16), "Square Root of 16")
\]

The second field trivially evaluates to itself. For the first field the process Square_Root is created and run in parallel with the process that created the live tuple. When Square_Root has evaluated the first field to be of value 4, the live tuple turns into the following data tuple:

\[
(4, "Square Root of 16")
\]

2.2.6 Discussion

The Linda model offers the two properties of Space Uncoupling and Time Uncoupling simultaneously (Gelernter, 1985). Space Uncoupling refers to the address-space-disjoint property of Linda processes. All Linda processes live in their own address space. The semaphore and monitor models require processes to share some or all of the address space.

Time Uncoupling refers to the persistence property of tuple spaces. Tuples placed into a tuple space remain there until removed. A process wanting to communicate with another places a tuple into the tuple space. The other process may remove this
tuples at any point in time. The message passing model requires the two processes to rendezvous, the message is only exchanged if both processes converge at the same point in time. Despite offering both Space and Time Uncoupling the Linda model remains general enough to implement semaphores, monitors, and message passing, as demonstrated by Gelernter (1985).

2.3 The Physical Environment

Physical environments that can support concurrently executing processes are parallel computers, such as the S/Net, Hypercube, and the Parawell-1, and networks of autonomous processors. This thesis focuses on the implementation of a C-Linda for a LAN of PC’s using the IBM NetBIOS interface.

A PC consists of a single CPU. Each PC has its own address space. The PC’s used run the DOS operating system. DOS is a single thread operating system, i.e. only one process is executing at any time. The PC’s communicate with each other through a LAN. The PC’s are connected to the LAN by a LAN Adaptor placed into one of the PC’s expansion slots. Facilities to access the LAN are provided by NetBIOS.

For a more detailed discussion on local area networks the reader is referred to the literature, such as Schwaderer (1988), and Coulouris and Dollimore (1988).

2.3.1 NetBIOS

NetBIOS is an extension to BIOS, which is part of DOS. "The IBM NetBIOS API [Application Program Interface] provides a programming interface to the LAN so that an application program can have LAN communication" (International Business Machines Corporation [IBM], 1988, p. 1-9). The NetBIOS interface provides two types of communication. Session Support guarantees that data sent is delivered.
Datagram Support does not guarantee delivery. To provide some degree of fault-tolerance DOS-C-Linda uses Session Support.

NetBIOS is accessed via a Network Control Block (NCB). A NetBIOS command to be executed is placed into the NCB along with any information required by the command. The NetBIOS command is then issued by placing the memory address of the NCB into the ES/BX (Extra Segment Register/Base Register) pair, and calling the software interrupt 5Ch.

NetBIOS allows two modes of operation. A program that calls a NetBIOS command can block until the NetBIOS command has completed, or it can continue execution immediately by issuing an outstanding NetBIOS command. An outstanding NetBIOS command is associated with an NCB, and only one such an association is permitted per NCB.

All NetBIOS commands return a code into the NCB indicating the status of the operation. A NetBIOS command that blocks the caller returns a code only upon completion. An outstanding NetBIOS command returns an immediate return code, and a final return code when the outstanding command finally completes.

For each outstanding NetBIOS command NetBIOS allows the specification of a function, called a post-routine, that is executed when the outstanding NetBIOS command completes. A function is specified to be an outstanding NetBIOS command’s post-routine by placing the address of the function into the NCB of the NetBIOS command. Placing a NULL address into the NCB specifies that the outstanding NetBIOS command has no post-routine. Post-routines are interrupt functions. When an outstanding NetBIOS command completes, the currently executing function is interrupted and the post-routine is executed.

NetBIOS communication is based on the message passing model discussed in section 2.1.3. PC’s using the NetBIOS interface are identified on the network by one or
more NetBIOS names. NetBIOS names are assigned to PC’s by issuing an 
*NCB.Add.Name* command. Communication between NetBIOS names occurs through 
*NetBIOS sessions*. NetBIOS sessions between two PC’s are established by one PC 
issuing an *NCB.Listen* command, and the other issuing an *NCB.Call* command. 
NetBIOS identifies sessions with *Local Session Numbers*. Once NetBIOS sessions 
are established, processes on the two PC’s may communicate. Data between two 
PC’s is exchanged by one process issuing an *NCB.Receive* command, the other 
issuing an *NCB.Send* command. To close a session one PC issues the 
*NCB.Close.Session* command. NetBIOS names are unassigned by issuing the 
*NCB.Delete.Name* command.

Data communication occurs by placing the memory address and size of the data to be 
transferred into the NCB and issuing the *NCB.Send* command. NetBIOS limits the 
size of buffers that may be sent across the network to one DOS segment. Therefore 
buffers may be of any size between 0 and 64K’s. The session partner receives the 
data into a *communication buffer*. The address and size of the buffer is placed into 
the NCB before an *NCB.Receive* command is called. The *NCB.Receive* command 
should be issued before the *NCB.Send* command. Upon receiving the buffer, 
NetBIOS places the size of the received buffer into the NCB.

NetBIOS allows the specification of a *timeout* when establishing a session. When 
NetBIOS commands do not complete within a specified time they timeout. This is 
useful for detecting lost communications.

For a more detailed discussion on NetBIOS the reader is referred to the IBM Local 
3 Literature Review

This chapter establishes what has been done in the area of Linda research. It consists of five sections. The first section identifies where a complete overview of Linda and the concepts involved may be found. Two areas of current Linda research are identified, and the significance of Linda is presented. The second section confirms the wide use and acceptance of Linda. The third section lists applications that have been implemented using Linda. The fourth section identifies issues described in the literature significant when implementing a C-Linda. The fifth section outlines reasons for embedding Linda into C for a LAN of PC's.

3.1 Literature on Linda

A good introduction to the Linda paradigm is provided by Gelernter (1985). In this article Gelernter gives a complete overview of Linda and the concepts involved. Linda is compared to other parallel programming models. Examples of the Linda operators are given, thus explaining the semantics of the operators very clearly. Gelernter furthermore uses example applications to illustrate the usefulness of Linda.

Weston (1990) reasons that only parallel software will be able to supply the processing power that future applications require. The parallel architecture already exists, "but without the right software tools, programs can't easily take advantage of the parallel hardware." (Weston, 1990, p. 81). According to Weston (1990), Linda is a very convenient way of extending existing systems into the parallel paradigm. He suggests methods of translating sequential programs into parallel ones.

An interesting observation was made by Carriero and Gelernter (1989b), who report that certain problems are naturally parallel, and can therefore be expressed a lot

Edith Cowan University
Stephan Buttermann
easier in a parallel language than in a sequential one. They give an example of finding primes, a problem they believe to be naturally parallel. Sebesta (1989, p. 345) asserts that "Many problem domains lend themselves naturally to concurrency." Sommerville (1989, p. 184) acknowledges that "there are some applications ... where a parallel approach is a completely natural one."

The Linda model of process communication and synchronisation has also been extended into the hardware domain. Ahuja, Carriero, Gelernter, and Krishnaswamy (1988) have proposed a parallel computer called the Linda Machine, specifically "designed to support the Linda parallel programming environment in hardware." It is a computer consisting of processors that are surrounded by Tuple Space. A similar architecture has also been proposed by Bormann and Herdieckerhoff (1989).

Xu and Liskov (1989) investigate the building of fault tolerant Linda systems. "Many parallel computations are long lived; fault tolerance is particularly important to them" (Xu and Liskov, 1989, p. 205). The fault tolerant design by Xu and Liskov (1989) is based on a distributed and replicated tuple space. Each Linda process maintains a portion of the tuple space. Tuples placed into the tuple space portion are replicated in the tuple space portions of the node's neighbours. Thus if one Linda process crashes, the tuple will still be available in the neighbour's tuple space portion. Besides supplying some degree of fault tolerance, the distributed tuple space and the replication of tuples "can also prevent the tuple space from becoming a bottleneck" (Xu and Liskov, 1989, p. 205).

Kambhatla (1990) also investigates the introduction of fault tolerance into Linda systems. Kambhatla (1990, p. 2) reveals that Linda's characteristics of time decoupling, space decoupling, and nondeterminism make it "a particularly suitable model for fault-tolerant [sic] applications." Kambhatla's approach to achieving fault tolerance involves the "periodic checkpointing of process states" (Kambhatla, 1990, p. 4), and the logging of messages sent between processes. Upon detection of a failure each process individually recovers by restarting from the last valid checkpoint.

Edith Cowan University

Stephan Bettermann
Linda’s properties of time decoupling, space decoupling, and nondeterminism make it possible for each process to recover individually.

3.2 Previous Implementations

3.2.1 Languages


3.2.2 Environments

The environments Linda has been implemented in are as diverse as the host languages. Linda was originally designed for the SBN network computer (Gelernter, 1985, p. 81) and implemented by Nicholas Carriero on the S/Net (Whiteside and Leichter, 1988, p. 193). Carriero and Gelernter (1989b, p. 445) report Linda implementations for "a wide variety of machines: shared-memory multicomputers like the Encore Multimax, Sequent Balance and Symetry and Alliant FX/8;
distributed-memory multicomputers like the Intel iPSC/2 and the S/Net; and Vax/VMS-based local area nets.

Pinakis (1991) reports of a C-Linda for a local area network of Sun Workstations running UNIX, and of a Joyce-Linda for "the VAX 68030 and SPARC architectures" (Pinakis, 1991, p. 5) under BSD Unix. Prolog-Linda by Sutcliffe and Pinakis (1990) runs on "a network of Sun SPARC stations under SunOS 4.0.3., ... connected via an Ethernet" (Sutcliffe and Pinakis, 1990, p. 3) using TCP/IP. An Arity Prolog version has also been implemented for a "network of IBM PS/2 55SXs using MS-DOS 3.3" (Sutcliffe, 1991, p. 212) connected by an Token Ring using NetBIOS.


### 3.3 Applications

Linda has been used for a range of applications as reported by Carriero and Gelernter (1989b, p. 445):

[Linda] has been used for a wide variety of parallel programming experiments, including matrix multiplication and LU decomposition ..., DNA sequence comparison and parallel database search ..., travelling salesman, expert systems ..., charged particle transport, finite element equation solvers ..., linear programming and others.

Carriero and Gelernter (1989b, p. 445) further reveal that Linda is being used for a sparse system solver by Ashcraft, Carriero, and Gelernter (1986). Whiteside and Leichter (1988) report of using Linda "in the generation of ray-tracing displays of
fractal images ..., and in executing a parameter sensitive analysis for rocket plume simulations" (Carriero and Gelernter, p. 445).

Sutcliffe and Pinakis (1990, p. 7) report that Prolog-Linda "may be used to introduce parallelism into automated deduction systems", and subsequently Sutcliffe (1991) implements a Parallel Linear & UR-Derivation System called GLDUR, using Prolog-Linda.

3.4 Significant Issues

Jellinghaus (1990) reveals some important issues that need to be considered when implementing a C-Linda. The first issue refers to pointers. Pointers are two things: an address, and a reference to an object. The second issue refers to the "tricky linguistic issues which involve the eval operator" (Jellinghaus, 1990, p. 76).

3.4.1 Addresses or References

The issue of pointers being both addresses and references to objects is especially relevant in C where pointers occupy a significant role. Assume the following C declaration exists:

\[
\begin{align*}
\text{int integer = 42;} \\
\text{int *pointer = &integer;}
\end{align*}
\]

The following C statements are legal:

\[
\begin{align*}
\text{if ((*pointer == 42) && (pointer == &integer))} \\
\text{++pointer;}
\end{align*}
\]

The variable pointer is both a reference to the integer value 42 of the variable integer, and an address of where the variable integer is stored. Both are
expressible. A C-Linda should therefore allow pointers to be used both as references and as addresses. As a C-Linda cannot determine which of the two is meant, the user must disambiguate this by specifying whether the field is an address or a reference to an object. A method of doing this is discussed in section 4.2.2.2.

3.4.2 The eval operator

The eval operator presents a range of problems. The difficulty of implementing the eval operator is best demonstrated by its limited implementation. Evidence of the eval's implementation is found in the following: C-Linda by Berndt (1989), Prolog-Linda by Sutcliffe and Pinakis (1990), and the implementation described by Gelernter (1989). Pinakis (1991) does not provide the eval operator directly, but provides the capability to create processes using the in, out, and rd operators. Jellinghaus (1990, p. 81) has not implemented the eval operator for his Eiffel Linda. Joyce-Linda by Pinakis and McDonald (1991) also does not provide the eval as the capability of process creation already exists in Joyce (Pinakis and McDonald, 1991, p. 5). The C-Linda implementation by Carriero and Gelernter (1986) does not have the eval operator implemented (Carriero and Gelernter, 1986, p. 112). No evidence exists in the available literature to suggest that the eval operator has been implemented in the remaining implementations listed in section 3.2.1.

The issue of where the processes to be created come from is not satisfactorily resolved in the literature. They can originate from the Tuple Space or from the process that called the eval operator. If the calling process supplies the processes to be created, a second decision arises because the processes can be subroutines of the calling process, or executable files of their own.

Processes created to evaluate fields of live tuples need to return the resultant value. A further problem exists when processes created cannot return evaluated fields. This problem was discussed by Pinakis and McDonald (1991), as Joyce, the host language
for their Joyce-Linda, does not allow "agents to return values" (Pinakis and McDonald, 1991, p. 5). Similarly, programs under most operating systems (including DOS) can only return a single integer. A Linda implementation under such operating systems cannot expect processes to return values. In these situations live tuples cannot turn into data tuples.

3.5 The Need for the Research

As can be seen from the literature review, an embedding of Linda into C on a local area network of PC's has not been implemented. Most Linda implementations are for specialised and expensive hardware and software. PC's are the most inexpensive and widely available computing resource. The research conducted shows how Linda can be embedded into C on a LAN of PC's.

The performance of such an implementation must be determined to prove the hypothesis that a LAN of PC's can show the performance gains that other implementations have shown. The performance data available limits this comparison to the S/Net Linda Kernel and Modula-Linda.
4 Design

A Linda environment consists of a tuple space allowing multiple Linda processes, called Clients, to manipulate the tuple space using the Linda operators. Clients can create new processes, using the operator eval, that execute in parallel. This chapter describes the design of a Linda environment and the operators for a physical environment of PC's running DOS linked via a LAN.

Each Client runs on one PC. The tuple space also occupies one PC. Communication between the PC's running the Clients and the PC running the tuple space occurs through the LAN. Therefore communication has to be established between the PC running the tuple space, and the PC's running the Clients. For communication between two PC's to be possible, a network session is established between the two.

Since multiple tuple spaces may exist on a given network, for a Client to establish communication with a tuple space, the Client would have to specify the desired tuple space. This is undesirable as Linda would have to supply an operator that allows the user to specify which tuple space to access.

When Clients create processes using the eval operator, the new processes must be executed in parallel to the Clients that create them. As DOS is a single process operating system (see 2.3), the process must be executed on another PC. The DOS operating system does not provide mechanisms to execute a process on another PC, therefore that capability must be provided by DOS-C-Linda.

4.1 The DOS-C-Linda Machine

The DOS-C-Linda Machine implements a Linda environment. It provides a tuple space and processors on which Linda processes may be executed. Internally the
DOS-C-Linda Machine consists of a Tuple Server, and multiple Client Shells, each on a separate PC connected to a LAN. The Tuple Server provides and manages the tuple space and the DOS-C-Linda Machine.

Client Shells are command shells logged into the Tuple Server. When a Client Shell logs into the Tuple Server, a network session is established between the Tuple Server and the Client Shell. Client Shells are used to run processes created by the Linda operator eval, and also allow the user to execute operating system commands, here DOS commands. Clients that run in Client Shells have access to the tuple space provided by the Tuple Server.

The existence of a Client Shell allows the establishment of communication with the tuple space before Clients are started. Clients do not need to establish communication themselves.

4.2 The DOS-C-Linda Operators

This section describes the design of the DOS-C-Linda operators' interfaces. There are two types of operators: operators dealing with data tuples, and an operator dealing with live tuples. The data tuple operators are:

- `out`
- `in`
- `inp`
- `rd`
- `rdp`

The operator dealing with live tuples is `eval`. A predicate form of the `eval` operator has also been included in DOS-C-Linda. The `evalp` operator returns `False` if there is no free processor that the process may run on.
To eliminate conflicts with predefined functions, `linda_` precedes all operators. Thus the DOS-C-Linda operators are:

```c
linda_out
linda_in
linda_inp
linda_rd
linda_rdp
linda_eval
linda_evalp
```

### 4.2.1 Data Tuple Operators

The Linda operators dealing with data tuples have tuples as their arguments. The `out` operator places the tuple specified in the argument list into the tuple space. The `in`, `inp`, `rd`, and `rdp` operators find a tuple that matches the tuple template specified in the argument list, and assign all the matching tuple's actuals fields' value to the corresponding formals fields of the tuple template. Since a tuple may consist of any number of fields the argument lists of the DOS-C-Linda operators must be of variable length.

The variable length parameter list mechanism available in C, the host language, requires the programmer to control the retrieval of the parameters. The retrieval of the parameters is only possible with knowledge of the types and number of the parameters. This knowledge is usually supplied by a *format string* as the first parameter, as exemplified by the predefined `printf` function. The format string contains the required information about the number and types of the following parameters. The format string is therefore used to specify what the types of the tuple fields are. Using the `printf` function's format string makes the DOS-C-Linda operators dealing with data tuples look like the `printf` function, an appropriate

\*\*\* `printf` is declared in the ANSI C standard `stdio.h` header file.\*\*\*

---

Edith Cowan University  
Stephan Bettermann
similarity.

The format string is also a convenient mechanism to differentiate between addresses and references, an issue discussed in section 3.4.1. As the user is specifying the type and number of the variable arguments, tuple fields can be specified to be either addresses, or pointers to data objects by the user. The use of the format string solves the problem that "in C ... one cannot dereference a pointer and determine what structure [data object] is being referenced" (Jellinghaus, 1990, p. 77), as the user specifies the types of the data objects being referenced in the format string.

The predicates inp, rdp, and evalp return False upon failure. In C, functions usually indicate failure by returning the integer 0, and success by returning any non-zero integer such as 1. The DOS-C-Linda operators may however encounter other situations that cause failure, such as running out of memory. The DOS-C-Linda operators should therefore return a code indicating the situation that caused the failure. Therefore the DOS-C-Linda operators indicate success by returning 0, and failure by returning a non-zero integer error code detailing the situation that caused the failure.

Considering the arguments expected by the data tuple operators, and the return codes, the C function specifications for the DOS-C-Linda data tuple operators are as follows:

```c
int linda_out( char *format_string, ... );
int linda_in( char *format_string, ... );
int linda_inp( char *format_string, ... );
int linda_rd( char *format_string, ... );
int linda_rdp( char *format_string, ... );
```
4.2.2 Tuple Specification

This section describes how tuples are specified using the format string. All C types have to be expressible, including the complex types arrays and structures. The user has to be able to express both pointers and references to object.

The values of the tuple fields, or pointers to the values, are passed to the operators as arguments after the format string. The format string consists of a sequence of field type specifications, specifying the types of the following arguments. Field type specifications begin with a polarity character and end with a field type specifier. The polarity character is either % to indicate that the field is actual, or ? for formal fields. The field type specifier is a character sequence defining the type of the next argument. A field type specifier can be either complex or basic. Between the polarity character and the field type specifier, optional pointer fields are possible. This indicates that the argument needs to be dereferenced one or more times to obtain the field’s value.
4.2.2.1 Basic Types

The field type specifiers for the basic types are adopted from the field type specifiers used in the format string of the printf. Another type, byte, is also included for convenience. Below is a table of the basic types and the corresponding field type specifier:

**Table 1: Field Type Specifiers for Basic Types**

<table>
<thead>
<tr>
<th>Field Type Specifier</th>
<th>Argument Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>d, i</td>
<td>int; decimal integer</td>
</tr>
<tr>
<td>u, o, x, X</td>
<td>unsigned int; unsigned integer</td>
</tr>
<tr>
<td>f, e, E, g, G</td>
<td>double;</td>
</tr>
<tr>
<td>c</td>
<td>char; character</td>
</tr>
<tr>
<td>b</td>
<td>unsigned char; character</td>
</tr>
<tr>
<td>p</td>
<td>void *; address</td>
</tr>
<tr>
<td>s</td>
<td>char *; character string. Null terminated.</td>
</tr>
</tbody>
</table>

Some argument types have multiple field type specifiers to keep the DOS-C-Linda operators orthogonal to the printf format string. The printf uses these equivalent field type specifiers to display the same type in a different format. Using \( x \), for example, causes an unsigned integer to be displayed in hexadecimal format with the alphabetical digits (a - f) displayed in lowercase. Using \( X \) on the other hand causes the alphabetic digits to be displayed in uppercase.

Following is an example of specifying a tuple with seven actual fields. The fields are a signed integer of value 42, unsigned integer of value 42, a double of value
3.14, the character 'C', a byte of value 0, a pointer of value NULL, and the 'N' terminated string "This is a string":

( "%d %u %f %c %b %p %s",
  42, 42u, 3.14, 'C', 0x00, NULL, "This is a string" )

As is the case with the printf, type modifiers are used to extend these basic types. Below is a table of the type modifiers, the basic types they can be applied to, and the corresponding argument type:

**Table 2: Type Modifiers for Basic Types**

<table>
<thead>
<tr>
<th>Type Modifier</th>
<th>Applied to</th>
<th>Argument Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>p</td>
<td>void _far *; far pointer</td>
</tr>
<tr>
<td>N</td>
<td>p</td>
<td>void _near *; near pointer</td>
</tr>
<tr>
<td>h</td>
<td>d, i</td>
<td>short int; short signed integer</td>
</tr>
<tr>
<td></td>
<td>o, u, x, X</td>
<td>short unsigned int; short unsigned integer</td>
</tr>
<tr>
<td></td>
<td>f, c, E, g, G</td>
<td>float</td>
</tr>
<tr>
<td>l</td>
<td>d, i</td>
<td>long int; long signed integer</td>
</tr>
<tr>
<td></td>
<td>o, u, x, X</td>
<td>long unsigned; long unsigned integer</td>
</tr>
<tr>
<td>L</td>
<td>f, e, E, g, G</td>
<td>long double</td>
</tr>
</tbody>
</table>

The type modifier h is applicable to the floating point field type specifiers as DOS-C-Linda must differentiate between floating point numbers and doubles. Applying the type modifier h to floating point field type specifiers indicates that the tuple field is of type float. Microsoft C Version 6 and Turbo C++ Version 1.0 store floating point numbers in four bytes, doubles in eight bytes. The need to distinguish

---

5 The symbolic constant NULL is declared in the ANSI C standard stdio.h header file.

Edith Cowan University

Stephan Bettermann
between floats and doubles arises out of the following situation. A tuple in the tuple space may contain actual fields of type float. The tuple is extracted and the values of the actual fields have to be assigned to the corresponding formal fields of the specified tuple used to find the matching tuple. As tuple fields are syntactically C variables, the size of the variable must be known.

The printf function does not need to differentiate between doubles and floats; ANSI C promotes floats passed into functions with variable length argument list into doubles as specified in §A.7.3.2 of the Draft ANSI C standard listed in Kernighan and Ritchie (1988, p. 202). The format string of the printf function therefore does not have to differentiate between floats and doubles, explaining the absence of a mechanism to specify floats.

Following is an example using the types listed in the table above. Assume the following C definitions exist:

```c
void _far * far_pointer;
void _near * near_pointer;
short int short_integer;
short unsigned short_unsigned;
float floating_point_number;
long int long_integer;
long unsigned long_unsigned;
long double long_double;
```
The tuple specification for a tuple containing the variables defined above is as follows:

```
( "%Fp %Np %hd %hu %hf %ld %lu %Lf",
  far_pointer,
  near_pointer,
  short_integer,
  short_unsigned,
  floating_point_number,
  long_integer,
  long_unsigned,
  long_double )
```

**Formal Fields**

Tuple templates are used in the linda_in, linda_rd, linda_inp, and linda_rdp functions to find a matching tuple in the tuple space. Formal fields in the tuple template are assigned the values of the corresponding actual fields of the matching tuple from the tuple space. Syntactically tuple fields are C variables. When a tuple is extracted from the tuple space that contains actual fields whose values have to be assigned to formals fields of the tuple template, these values have to be assigned to C variables. For DOS-C-Linda to have the capability of assigning actual values to variables, it must have the addresses of the variables. Therefore formal tuple field specifications must always be accompanied by the addresses of the C variables the values are to be assigned to, in the variable length argument list.

The linda_out operator may also specify tuples containing formal fields. To keep this operator orthogonal to the other data operators, formal tuple field specifications must also be accompanied by an address in the variable length argument list. As formal fields do not have a value, and the linda_out operator does not assign any values to formal fields, this address can be specified as NULL. This is appropriate as NULL is assigned to pointers that point to nothing.
4.2.2.2 Complex Types

The complex types arrays and structures should also be expressible. These types are not expressible in the printf function's format string, therefore the mechanisms to allow the expression of these types have to be defined.

Arrays

When dealing with arrays, the size of the array must be specified. This may be done by the presence of an array size field after the optional pointer field in the format string. To allow for specification of dynamic arrays, a mechanism to place the array size field in the argument list and not in the format string should be provided. The # symbol is an appropriate choice because it is a common abbreviation for a number. Therefore an array size field is either an integer or a #. When using a # the array size is taken from the next argument in the variable argument list, and used to replace the # in the format string. Assuming the following C variable definitions exists:

    int array[100];

The following is a tuple specification for the array:

    ( "%100d", array )

Alternatively the specification could be:

    ( "%#d", 100, array )

DOS-C-Linda then replaces the # with the array size, here 100. Thus the following two tuples are equivalent:
Structures

Structures pose a special problem. As structures are collections of variables, structures may be viewed as tuple fields themselves having tuple fields. Thus the individual elements of structures must be accessed. C, however, does not define the physical layout of structures or unions (§A.8.3 of the Draft ANSI C standard listed in Kernighan and Ritchie, 1988, p. 213). The layout is defined by the implementation and is often affected by any enabled optimisations such as Word Alignment. Therefore it is up to the programmer to define the internal layout of structures. The C pack pragma is used in Microsoft C to ensure the internal layout matches that specified by the programmer. Turbo C++ ignores this pragma, and thus the Word Alignment option must be turned off. This is done by turning off the Word Alignment option in the Options Compiler Code Generation menu.

Structures are defined by enclosing the structure's field definitions in open and closing braces ({}) as done in C. A { as the first character of the field type specifier indicates that the following is a structure.

The structure itself can be indicated to be either formal or actual. As all fields within the structure have polarity characters, the polarity character of the entire structure is only used to indicate the beginning of the structure's field type specification. Either polarity character is acceptable. The presence of a polarity character keeps structure specifications orthogonal to other type specifications.

The next issue that must be considered is whether structures should be passed by reference or by value. In ANSI C structures are passed to functions by value, unless

---

6 Word Alignment refers to the alignment of data items to even-byte addresses.
specified otherwise. When specifying a structure as a tuple field, and the structure itself has both actual and formal fields, the address of the formal fields in the structure must be known so values may be assigned to them. If structures are passed by value the addresses of any formal fields within the structure cannot be determined; they must be obtained through another mechanism. If structures are passed by reference the addresses of any formal fields within the structure may be determined; the starting address of the structure is known and the internal layout is specified. This issue is further discussed in the next chapter. Until this issue is resolved in section 5.2.1.2, structures containing only actual fields are passed by value, structures containing formals are passed by reference.

The tuple specification for the following definition:

```c
struct {
    int  integer;
    float pi;
    char * string;
} structure = {42, 3.14f, "This is a string");
```

would be:

```c
( "%d %f %s", structure )
```
Design

Structures may themselves contain structures as fields. For example, assume the following declaration and definition exist:

```c
typedef struct
{
    int         integer;
    long double long_double;
    void *      far_pointer;
} STRUCTURE_TYPE;

struct
{
    int         integer;
    STRUCTURE_TYPE structure;
    unsigned char byte;
} structure = {42, {43, 3.14L, NULL}, 0x00};
```

Following is a tuple specification for a tuple containing one single field:

```c
( "%d %i %Lf %Fp %b", structure )
```
Arrays and Structures

C allows arrays and structures to be combined. The mechanisms to specify arrays and structures introduced above are easily combined. For the following declaration and definition:

```c
typedef unsigned char BYTE_TYPE;

struct {
    unsigned short unsigned_short;
    BYTE_TYPE byte_stream[10];
} structure;
```

the tuple specification is:

```c
( "%hu %10b", structure )
```

An array of structures containing one array as defined below:

```c
typedef unsigned char BYTE_TYPE;

struct {
    unsigned short unsigned_short;
    BYTE_TYPE byte_stream[10];
} structure_array[20];
```

is specified as follows:

```c
( "%20%hu %10b", structure_array )
```

The use of #'s to define sizes of arrays that are fields of structures requires care.
Any #'s within structures are replaced before the structure is parsed. Hence all array sizes need to be in the argument list before the structure's address. For the following declaration and definition:

```c
typedef unsigned char BYTE_TYPE;

struct {
    BYTE_TYPE first_array[10];
    float second_array[20];
    long double third_array[30];
} structure_array[40];
```

the specification is as follows:

```c
( "%#{%#b %#hf %#Lf}'',
  40, 10, 20, 30, structure_array )
```

### Pointers to Values

Pointers should be expressible as both references to data objects, and as addresses. Addresses are expressed using the field type specifier p or the extended forms Fp and Np (see 4.2.2.1). References to data objects are expressed using the optional pointer fields. Pointer fields adopt their format from C, that is * for a pointer, ** for a pointer to a pointer and so on. As the type of the object being referenced must also be specified, the optional pointer field is followed by a field type specification defining the type of the data object being referenced. This allows pointers to be expressed as references to data objects, and as addresses.

For the following definition:

```c
int integer = 42;
```
a tuple specification could look like this:

( "%*d", &integer )

This notation can be used with any of the basic and complex types. For the following declaration and definitions:

```c
int the_answer = 42;

typedef struct
{
    int * the_answer;
    char * the_question;
} STRUCTURE_TYPE;

STRUCTURE_TYPE structure =
{
    &the_answer,
    "What is the answer?"
};
```

the specification is:

( "%{%*d \%s}", &structure )

Note that the value of the first tuple field is 42, the value of the integer being pointed to. The tuple is not isomorphic to structure's type of STRUCTURE_TYPE.
Similarly pointers to pointers may be used as shown in the following example:

```c
int integer = 42;
int *pointer = &integer;
int **pppointer = &pointer;
int ***pppppointer = &pppointer;
```

The tuple specification for a pointer to a pointer to a pointer to a pointer to the integer value 42 is:

```c
( "%****d", &pppppointer )
```

Pointers to arrays as defined below:

```c
int array[20];
int *pointer = array;
```

are defined as follows:

```c
( "%*20d", pointer )
```

Note that %s is not the same as %c. The %s refers to a '\0' terminated sequence of characters, whereas the latter refers to a pointer to a single character. For the following definitions:

```c
char character = 'C';
char *character_pointer = &character;
char *string = "This is string";
```

the correct tuple specification containing all these fields is:

```c
( "%c %c %s", character, character_pointer, string )
```
4.2.3 Live Tuple Operators

The `eval` operator is used to place live tuples into the tuple space. Live tuples contain any number of fields. For each field a process is created to evaluate the field. Upon completion of all processes the live tuple turns into a data tuple indistinguishable from any other data tuple. The processes created are run in parallel to the process that placed the live tuple into the tuple space. This section is concerned with the design of the `eval` operator, and its predicate form `evalp`.

As discussed in section 3.4.2, programs under DOS can only return an integer. This is inadequate as created processes need to return the type and value that live tuple fields evaluate to. Two options exist:

1. DOS-C-Linda supplies the mechanism enabling processes to return the evaluated field's value and type.

2. The process do not create live tuples whose fields are evaluated by created processes running in parallel. They simply create processes that run in parallel. This option does not destroy the expressiveness of Linda as the evaluation of live tuple fields can be simulated using the data tuple operators. Created processes simply remove an agreed upon tuple from the tuple space upon completion, and replace the appropriate field with the evaluated field.

A C-Linda compiler could easily add the required code to accept any parameters passed into the process, and the code required to return the value and type that the field evaluates to. A C-Linda library does not have the capability of adding the required code to the process. Instead two additional functions must be supplied that

---

7 The literature does not specify whether evaluated fields can be formal; the type of the field is always evaluated.
perform these functions. The programmer would then have to explicitly obtain the parameters passed into the process, and explicitly return the type and value the live tuple field evaluates to, by calling the appropriate functions.

This option however is undesirable as two additional operators would have to be provided. These operators would essentially perform the same tasks as the in and out operators. The capability of passing parameters into processes created with the process operators, and having live tuple evaluate to data tuples can be simulated by using native Linda operators, and thus does not add expressiveness to Linda. The second option is preferable because no new operators have to be introduced, yet the full semantics of the eval operator are still available.

Therefore the eval and the evalp operators should not create live tuples as such. They should simply create processes that run in parallel to the Clients that called the eval or evalp operators. Created processes need to be in executable files, and therefore the process operators only have one parameter: the name of the executable file.

The evalp operator is required to return True or False. Like the data tuple operators the process operators can also encounter situations that result in their failure. Thus the process operators also return codes. The function specifications for the process operators must therefore be:

\[
\text{linda_eval( char *process_name );}
\]
\[
\text{linda_evalp( char *process_name );}
\]

Where process_name is the operating system filename of the executable file that represents the process to be created.
4.3 Linda Library or Linda Compiler

The Linda operators may be injected into the C language as a Linda compiler or pre-compiler, or as a library of Linda operators. This section considers the arguments for both options and decides how DOS-C-Linda should be injected.

4.3.1 Linda Compiler

When specifying structures as tuple fields the physical layout must be known to DOS-C-Linda as structures themselves contain tuple fields. C does not define the physical layout of structures, this is left to the compiler implementor. A Linda library cannot determine where the fields within structures are, and must shift the responsibility of supplying this knowledge to the user. Most C compilers however allow the user to control the layout of structures, thus the user can specify the correct layout. A C-Linda compiler does not depend on the user to specify the layout of structures.

Jellinghaus (1990, p. 76) points out that a Linda compiler could optimise the matching process at compile-time. Tuples only match if they are of the same arity. A compiler could determine a tuples arity at compile-time, thus reducing the number of tuples that are searched for a match at run-time. Proficient search algorithms, such as binary searching, can however make the benefits gained from such optimisations negligible.

A compiler would allow the detection of erroneous use of the Linda operators at compile time. If for example the format string approach is taken as discussed in section 4.2.1, the compiler could validate the format string. It must be noted however that the printf's format string is not checked for validity either, as the format string is not a C construct, but simply a character string being passed into a function. Even though the validation of format strings by the compiler would be
commendable, validating format strings would extend the C language. Extending the language is undesirable as Linda should only supply the operators.

As discussed in section 2.2.5.6 the eval and the evalp operators create processes. These processes may either be executable files of their own, or subroutines of the Clients calling the process operators (see 3.4.2). A Linda compiler can easily produce executable images of subroutines for the process operators. A Linda library however does not have this capability, and thus only executable files may be used for the eval and evalp operators.

4.3.2 Linda Library

Providing the Linda operators as a library means that multiple compilers may be used by the user. Thus the user may use the compiler that best suits their needs. Providing the Linda operators as a library also becomes easier as no effort has to be spend writing a C compiler.

All C functions are supplied by libraries. Supplying the Linda operators as keywords results in extending the C language. Providing Linda as a library means that C itself is not extended. The library merely gives the user the capabilities of process communication, synchronisation, and creation, in the same manner the standard C libraries do.

4.3.3 Summary

A Linda library depends on correct specifications by the user of the internal layout of structures. A compiler knows the internal layout of structures. Most C compilers allow the programmer to control the layout of structures, thus only leaving a C-Linda compiler the advantage of being able to validate that the user has indeed specified the correct layout.

Edith Cowan University

Stephan Bettermann
A Linda compiler could use subroutines of Clients calling the eval and evalp operators to evaluate fields of live tuples. A Linda library does not have the capability to access portions of Clients’ executable files, and therefore executable files must be used for the eval and evalp. The literature does not however specify where the processes evaluating fields of live tuples originate. Using executable files to evaluate live tuple fields is therefore not illegal.

A Linda library is orthogonal to C. All C functions are supplied by libraries. The use of a library allows the user to use the C compiler that best suits their needs. Writing a Linda library is also an easier task than writing a C-Linda compiler. A DOS-C-Linda library offers more advantages than a C-Linda compiler. Therefore a C-Linda operating in a Local Area Network of PC’s running DOS should be injected as a library.
5 IMPLEMENTATION

This chapter describes the implementation of the DOS-C-Linda operators, and the supporting environment, the DOS-C-Linda Machine.

5.1 The DOS-C-Linda Machine

The DOS-C-Linda Machine consists of a Tuple Server and multiple Client Shells (see 4.1). Therefore two programs are required: (a) a Tuple Server program, and (b) a Client Shell program. The Tuple Server and the Client Shells are operating system shells that allow the execution of operating system commands and programs (DOS commands). Client Shells log into the Tuple Server and NetBIOS sessions are established, allowing communication between the Tuple Server and Client Shells.

When the Tuple Server is started it issues the following NetBIOS commands:

- it resets the LAN adaptor by calling the NCB.Reset command,

- an NCB.Add.Name to add its name to the Local Name Table,

- an outstanding NCB.Listen command allowing any NetBIOS station to establish communication thereby logging in,

- and an outstanding NCB.Receive.Any command, to allow any Client to send data to the Tuple Server.

Because multiple Tuple Servers may exist on a local area network, the Tuple Server's NetBIOS name must be specified by the user of the DOS-C-Linda Machine.
For outstanding NetBIOS commands no post-routines are specified (see 2.3.1). Instead the return code in an outstanding NetBIOS command’s NCB is used to determine when the outstanding NetBIOS command has completed.

The Tuple Server has two input channels: (a) The keyboard where users enter DOS commands, and (b) the network through which Client Shells and Clients send requests. The Tuple Server polls the keyboard and the NCB’s of outstanding NetBIOS commands continuously.

When a Client Shell is started it issues the following commands:

- an NCB.Reset command to reset the LAN adaptor,

- an NCB.Add.Name to add its name to the Local Name Table,

- an NCB.Call command to establish communication with the Tuple Server, thereby logging in,

- and an outstanding NCB.Receive command to allow the Tuple Server to send data.

Because multiple Tuple Servers and multiple Client Shells may exist on a local area network, the name of the Tuple Server to log into, and the name of the Client Shell must be specified by the user of the DOS-C-Linda Machine.

Client Shells also have two input channels: (a) The keyboard where users may enter DOS commands, and (b) the network through which communication with the Tuple Server occurs. Client Shells poll the keyboard and the NCB’s of outstanding NetBIOS commands continuously for requests from users and the Tuple Server.
5.1.1 The Tuple Server

When Client Shells log into the Tuple Server, the Tuple Server must store information about the sessions with the Client Shells. This information is stored by the Tuple Server in the following data structure:

![Diagram of the Client Table]

**Figure 1: The Client Table**

*Clients* is a record containing three fields. The *Tuple Server Name* is the NetBIOS name of the Tuple Server. The *Number of Sessions* is the number of sessions that the Tuple Server currently has. The *Client Table Size* is the current size of the Client Table.

The *Client Table* is a dynamic table of *Client Table Records*. Each Client Table Record has a *Client Status* marking the Client Shell as *Busy* or *Ready*, and a pointer to a *Session Record* containing all the information needed for NetBIOS communication, and therefore represents a session with a Client Shell. The Tuple
Implementation

Server marks Client Shells Busy when they are running a Client, and Ready when Client Shells are idle.

The Local Name in the Session Record is the Tuple Server Name. This duplication is desirable as Session Records contain all the information about sessions. The Tuple Server Name is required in Clients, since the Client Table may be empty and therefore no Session Record exists that stores the Tuple Server Name. Client Shells and the Tuple Server use the same networking routines and therefore this duplication is accepted.

When Client Shells log into the Tuple Server, the Tuple Server adds a Client Table Record to the Client Table for each Client Shell. When Clients or Client Shells send messages, the Tuple Server's outstanding NCB.Receive.Any command completes. NetBIOS provides the Local Session Number of the session that caused the outstanding command to complete. The Client Table can be searched for the Session Record using the Local Session Number, thereby determining which session to use for a response.

5.1.2 The Client Shells

Client Shells have two tasks: (a) to let users execute DOS commands, including Clients, and (b) to provide a processor where processes created by the linda_eval and linda_evalp operators may be executed. Clients communicate with the Tuple Server through the NetBIOS session opened by the Client Shell in which they are running. Therefore the Client Shell has to transfer:

- the Adaptor Number,
- the Local Session Number,
- the Local Name Number,
- the addresses of outstanding NCB's,
- the addresses and sizes of the buffers,
Implementation

- and the Local and Remote Names

to the Clients so the Clients may communicate with the Tuple Server through the session opened by the Client Shell.

5.1.2.1 Information Transferal between Client Shells and Clients

Information is usually transferred to programs through the command-line parameter list. Transferring information to Clients through command-line parameter lists is undesirable however. It would introduce the programming convention that DOS-C-Linda reserves some command-line parameters. A better option is to transfer this information in a hidden fashion.

The DOS operating system makes the software interrupts 60_h through to 66_h inclusive available for user programs. The diagram below illustrates how Client Shells transfer information to Clients using a software interrupt:

---

Figure 2: Network Information Record
The Client Shell allocates space for a *Network Information Record*, the communication buffers and a Session Record. The Client Shell then places the address of the Network Information Record into interrupt 60h. Clients running in Client Shells accesses the Network Information Record by reading interrupt 60h the first time a DOS-C-Linda operator is called. Clients detect that they are not running in Client Shells when interrupt 60h has the value NULL.

The advantage of using a software interrupt instead of command-line parameters to transfer information between Client Shells and Clients is threefold:

1. **The information transferal remains hidden to the programmer.**

2. **The programming convention that DOS-C-Linda reserves a software interrupt is more readily acceptable than reserving command-line parameters because command-line parameters are frequently used.** Software interrupts are provided by the DOS operating system, not by C, and are therefore rarely used.

3. **The third advantage lies in the permanent nature of the interrupt vector.** Since Client Shells are DOS shells, the user could execute another Client Shell within a Client Shell by entering the name of the Client Shell program as a DOS command. This would be disastrous as Client Shells reset the LAN adaptor (see 5.1). Nested Client Shells would sever the sessions established with the Tuple Server. When using a software interrupt, Client Shells can detect they are running within Client Shells because the software interrupt will be set. Therefore the problem of starting a Client Shell within a Client Shell can be detected and prevented.
5.1.3 Tuple Server - Client Shell Communication

This section describes how the Client Shells and the Tuple Server communicate. Communication occurs when Client Shells log into the Tuple Server, and when operating system commands are entered in Client Shells.

5.1.3.1 Logging Client Shells into the Tuple Server

The Tuple Server has an NCB.Listen command outstanding at all times. Client Shells log into Tuple Servers using the NCB.Call command, thereby establishing a NetBIOS session. When an outstanding NCB.Listen command completes NetBIOS provides the NetBIOS name of the station that issued the NCB.Call command. The Tuple Server adds a Client Record to the Client Table (see 5.1.1) and issues another outstanding NCB.Listen to allow further Client Shells to log in.

5.1.3.2 Executing commands in Client Shells

When users want to execute DOS commands in Client Shells, the Client Shells must ask the Tuple Server for permission to do so as the Tuple Server might already have sent a process to the Client Shells and subsequently marked them Busy (see 5.2.6). When Client Shells are marked Busy the Tuple Server denies permission to execute commands, otherwise permission is granted and the Client Shells are marked Busy. Client Shells notify the Tuple Server when the DOS commands have completed and the Tuple Server marks the Client Shells Ready.

5.1.3.3 Logging Client Shells out

When a Client Shell wants to log out of the Tuple Server, it requests permission to do so from the Tuple Server. The Tuple Server grants the Client Shell permission to log out when it is marked Ready in the Client Table, and denies permission when it is marked Busy. The Tuple Server notifies the Client Shell that permission is
Implementation

granted or denied. If permission is granted, after notifying the Client Shell the Tuple Server closes the NetBIOS sessions and deletes the Client Shell's Client Records from the Client Table. The Client Shell may then delete its NetBIOS name from its Local Name Tables and terminate.

5.1.3.4 Shutting Down the Tuple Server

When the Tuple Server is shut down (a) the tuple space must be discarded, (b) all Client Shells must be logged out, and (c) all outstanding processes (see 5.2.6) must be deleted. Client Shells that cannot be logged out are simply cut off, i.e. the session is simply closed.

5.1.3.5 Communication Problems

When a Client Shell or the Tuple Server detect that the DOS-C-Linda Machine is in an unstable state, the DOS-C-Linda Machine is shut down. The DOS-C-Linda Machine enters an unstable state when a Client Shell is lost, a NetBIOS command returns a code indicating a breakdown in communication, or a Client Shell and the Tuple Server are out of synchronisation. Synchronisation is lost when the Tuple Server receives a command from a Client Shell that is not expected, or a Client Shell receives a message from the Tuple Server that is not expected.

5.2 Implementing the DOS-C-Linda Operators

The implementation of the DOS-C-Linda operators is now described. The operators are described in the order that they were implemented.
5.2.1 The linda_out operator

When the linda_out operator is called, the tuple to be placed into the tuple space must be built from the variable length argument list. The Client must then send the tuple to the Tuple Server so it may be placed into the tuple space.

5.2.1.1 Data Tuple Representation

Data tuples are represented with the following data structure:

![Data Structure representing Tuples](image)

Tuples are structures with three fields. The first field is a pointer to the tuple's format string. The second field is a pointer to a values string representing the values of all the actual fields within the tuple, in sequential order. Unlike the format string which is terminated with the '\0' character, the values string cannot be terminated with a '\0' as values of tuple fields may include this character. This is also the case.
for any other special character that may be used to indicate the end of the values string. Therefore a third field is used to store the length of the values string.

5.2.1.2 Building of Data Tuples

Data tuples are assembled from the variable length argument lists of the DOS-C-Linda operators. The values of actual fields that are of basic data types are retrieved off the run-time stack and copied into the values string using the va_arg function. Care must be taken with tuple fields of types char and float. These types are promoted by C when passed into functions. Characters are promoted to integers, and floats are promoted to doubles.

The retrieval of the complex data types arrays, structures, and pointers requires additional effort. The next sections explain the retrieval of these types.

Structures

A structure in C is a collection of fields that may be individually accessed. A tuple field that is a structure may therefore be viewed as a collection of tuple fields. As tuple fields are either actual or formal, and as formal fields have no value, structures that contain actual and formal fields require that the individual fields be accessed. Assuming the following declarations exist:

```c
typedef struct
{
    int integer;
    float floating_point_number;
} STRUCTURE_TYPE;

STRUCTURE_TYPE out_structure = {42, 3.14f};
```

va_arg is declared in the ANSI C standard stdarg.h header file.

Edith Cowan University
Stephan Bettermann
The following tuple:

\[( "\%\%d \%h\f\" , \text{out\_structure} ) \]

is placed into the tuple space with the following DOS-C-Linda function call:

\[
\text{linda\_out( } "\%\%d \%h\f\" , \text{out\_structure} );
\]

The \text{linda\_out} call above should only copy the integer value 42 of the first field into the values string of the tuple. ANSI C by default passes structures to functions by value (see 4.2.2.2). The \text{stdarg} library does not supply a function that allows the individual retrieval of structures' fields; the supplied \text{va\_arg} function removes entire structures\(^9\). This is insufficient as only the first field of the structure in the argument list must be placed into the values string. DOS-C-Linda therefore requires that structures be passed by reference. As the user specifies the internal layout of structures in the format string, this method allows access to the individual fields of structures.

**Arrays**

Arrays in C are passed by reference. Therefore when a tuple field is specified as an array, the variable length argument list contains the address of the first array elements. Because the array itself is not placed onto the run-time stack, character and float array elements are not promoted. For actual array tuple fields the entire contiguous area of memory constituting the array is simply copied into the values string.

\(^9\) Separate \text{va\_arg} calls to retrieve fields of structures individually does not work correctly.
Note that arrays of pointers are not directly expressible in DOS-C-Linda. They can however be expressed by defining an array of structures containing a pointer. An array of pointers as defined below:

```c
int one, two, three;
int *array[3];

array[1] = &one;
array[2] = &two;
array[3] = &three;
```

is specified as:

```c
("%3(%*d)", array)
```

### Arrays and Structures

Arrays and structures may be combined. As structures may be viewed as containing tuple fields themselves (see 5.2.1.2), the values in an array of structures, that are copied into the values string, may not be in contiguous memory locations. This is demonstrated in the example below. The following definition:

```c
STRUCTURE_TYPE structure_array[3] =
{
    {1, 1.0f},
    {2, 2.0f},
    {3, 3.0f}
};
```
defines an array of three structures. Assuming `structure_array`'s internal layout has not been modified by the compiler, the array is stored in memory as shown below:

```
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.0f</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2.0f</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3.0f</td>
<td>3</td>
</tr>
</tbody>
</table>
```

**Figure 4: Internal representation of `structure_array`**

The values string for the following tuple:

```
( "%3{?d %hf}", &structure_array )
```

would contain the unshaded areas of `structure_array` as shown below:

```
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.0f</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2.0f</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3.0f</td>
<td></td>
</tr>
</tbody>
</table>
```

**Figure 5: Non-contiguous memory areas needed for values string**

As the unshaded areas are not contiguous in memory, the entire array must be traversed and each element copied individually. The values of an array of references also may not occupy a contiguous area of memory, as each array element must be dereferenced to locate the value.

The treatment of structures with arrays as fields introduces no problems. As structures are viewed as tuple fields that contain tuple fields themselves, arrays that are fields of structures are retrieved in the same manner as tuple fields that are
arrays. However instead of obtaining the address of an array’s first element from the run-time stack, the address of an array’s first element is simply the address of the structure’s field.

**Pointers**

References to data objects are handled in the following manner. The pointer is dereferenced one or more times and the value is copied into the values string. The following definitions:

```c
int integer = 42;
int *pointer = &integer;
int **ppointer = &pointer;
int ***ppppointer = &ppointer;
```
are represented internally as shown below:

![Diagram of pointer structure]

**Figure 6: Internal representation of pointer structure**

The value of the only field in the following tuple:

```c
("%*****d", &pppointer)
```

is 42 as `pppointer` is indicated to be a reference to a data object. Therefore `pppointer` is dereferenced to locate the value 42 which is then placed into the `values` string.
5.2.1.3 Sending Data Tuples

NetBIOS data communication occurs by placing the address and size of the memory area to be transmitted into the NCB. Calling the NetBIOS.Send command causes the data contained within the memory area to be transmitted to the session partner. The memory area transmitted must be a contiguous area of memory. Therefore tuples are placed into buffers of the following format:

![Diagram of buffer format](image)

*Figure 7: Format of Buffer used to transfer Tuples over network*

*Command Blocks* precede all data tuples contained in buffers. A Command Block contains a code identifying the DOS-C-Linda operator that sent the buffer, the length of the tuple’s format string, and the length of the tuple’s values string. Because NetBIOS limits the amount of data that may be sent and received to 64K, DOS-C-Linda limits the size of buffers to 64K.

When a Tuple Server’s outstanding NCB.Receive.Any command completes, the Tuple Server checks the Command Block to determine which Linda operator sent the...
buffer. Upon receipt of buffers that were sent by linda_out operators, the Tuple Server reassembles the tuples and places them into the tuple space.

5.2.1.4 The Tuple Space

The tuple space is a collection of tuples. The reading and possible extraction of tuples requires the location of tuples that match tuple templates. To optimise the matching process the tuple space is maintained in an order that facilitates the efficient location of tuples.

The property that tuples match only if the types of the tuples match is used to optimise the matching process. By extracting the type information from a tuple's format string a *Tuple Signature* is obtained. The Tuple Signature obtained from the following format string:

```
"%d %hf ?100d %10{%8Ld ?*x}"
```

is:

```
"dhf100d10{8Ld*x}"
```

Tuple Signatures contain only information about the types of tuples' fields. Multiple
tuples with the same Tuple Signature may exist in the tuple space. The following diagram shows how tuples with the same Tuple Signature are stored:

Figure 8: List of Tuples with the same Tuple Signature
The following data structure is used to store the Signature Records:

Signature Records are kept in a Signature Table. The Signature Table grows and shrinks depending on the number of Tuple Signatures in the tuple space. The Signature Table is kept in the alphabetical order of the Tuple Signatures in the Signature Records. This allows an efficient search algorithm, such as the binary search\(^{10}\), to locate a collection of tuples with the same Tuple Signature.

When a new tuple is inserted into the tuple space, a binary search is used to locate the position where the Signature Record for the new tuple should be. If no Signature Record already exists, a new Signature Record is added into the Signature Table at the located position, thereby keeping the Signature Table in order. After locating or creating the Signature Record, the new tuple is added to the list hanging off the

---

\(^{10}\) The binary search, of the order \(\log_2 n\), is regarded as the most efficient search algorithm.
Signature Record as shown in Figure 8.

Because Tuple Signatures are used to locate a set of tuples that may match, equivalent field type specifiers (see 4.2.2.1) must be converted to a common field type specifier. For example the two tuples:

\[
(\text{"\%d \%x \%g"}, 42, 42u, 3.14f) \\
(\text{"\%i \%u \%f"}, 42, 42u, 3.14f)
\]

match even though the Tuple Signatures of "dxfg" and "iufl" respectively do not match. The following table shows the field type specifier normalisation:

**Table 3: Field Type Specifier Normalisation**

<table>
<thead>
<tr>
<th>Field Type Specifiers</th>
<th>Converted to</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>d</td>
</tr>
<tr>
<td>o, x, X</td>
<td>u</td>
</tr>
<tr>
<td>e, E, g, G</td>
<td>f</td>
</tr>
</tbody>
</table>

After normalisation the two tuples above have the Tuple Signature "dul".

### 5.2.2 The linda_in Operator

The linda_in operator finds a tuple in the tuple space that matches the tuple template specified as linda_in's argument. Therefore the tuple template must also be assembled from the variable length argument list. For all intents and purposes the tuple template is an ordinary data tuple, and is built in the manner described in section 5.2.1.2. The tuple template is sent to the Tuple Server in the manner as described in section 5.2.1.3.
When the Tuple Server receives a buffer that was sent by a \texttt{linda\_in} operator, the Tuple Server locates tuples in the tuple space that \emph{match} the reassembled tuple template. A matching tuple is then removed from the tuple space and sent to Clients in the manner described in section 5.2.1.3. If no matching tuples exist in the tuple space, the Client is left waiting until a matching tuple is placed into the tuple space.

Upon receiving a matching tuple, Clients \emph{map} the actual fields onto any corresponding formal fields of the tuple template. To reduce network traffic, the Tuple Server \emph{compresses} matching tuples before sending them to Clients.

\subsection*{5.2.2.1 Matching Tuples}

Tuples are read or removed using the \texttt{linda\_in}, \texttt{linda\_inp}, \texttt{linda\_rd}, and \texttt{linda\_rdp} operators. These operators locate a matching tuple in the tuple space. The semantics of tuple matching are described in section 2.2.3. When these operators are called the Tuple Server derives the tuple template's tuple signature and uses a binary search to locate a set of tuples with the same signature in the tuple space. The Tuple Server then searches the located set of tuples for matching tuples.
The entire list of tuples is simply traversed and all the matching tuples are stored in the following data structure\textsuperscript{11}:

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{matching_tuple_set.png}
\caption{Set of Matching Tuples}
\end{figure}

The \textit{Matching Tuple Table} is a growing table of pointers to the Tuple List Elements of Tuples (see Figure 8) that match the Tuple Template. After completing the search for matching tuples, one matching tuple is selected from the set of matching tuples nondeterministically using the \texttt{rand}\textsuperscript{12} function. The \texttt{rand} function returns a random integer that is used as an index into the Matching Tuple Table. After selecting a single tuple, the Matching Tuple Table is discarded, and the Tuple List Element of the selected tuple is removed from the List of Tuples (see Figure 8).

\textsuperscript{11} To optimise the matching process, a compiler directive is available that changes the matching semantics so that the first matching tuple is selected. If the first matching tuple is selected there is no need to locate all the matching tuples, and therefore this data structure is not built.

\textsuperscript{12} \texttt{rand} is declared in the ANSI C standard \texttt{stdlib.h} header file.
The matching algorithm is described now. Two tuples match until two corresponding fields do not match. As actual fields always match formal fields, and formal fields never match formal fields, only the values of corresponding actual fields are compared. This is done with the `memcmp` function.

When the Tuple Server cannot locate a tuple that matches the tuple template specified by a `linda_in` or `linda_rdp` operator it send a Null Tuple to the Client to indicate failure. Null Tuples have no format string and no values string. When the Tuple Server cannot locate a tuple that matches the tuple template specified by a `linda_in` or `linda_rd` call, the Client that called the operator is left waiting until a matching tuple is placed into the tuple space. The Tuple Server stores the request by lodging it, and continues processing other requests.

\[13\] `memcmp` is declared in the ANSI C standard `string.h` header file.
The Tuple Server stores lodged requests in the following data structure:

![Diagram of Tuple Server](image)

**Figure 11: Set of Lodged Requests for Tuple Signature**

A Request List Element is created for each lodged request. The Request List Element contains:

- a field that identifies the waiting DOS-C-Linda operator (either a `linda_in` or `linda_rd`),
- a pointer to the Tuple Template for which a matching tuple has to be found,
- a pointer to the session record in the Client Table (see Figure 1) that identifies the session with the Client Shell that is running the Client that called the DOS-C-Linda operator,
- a pointer to the next Request List Element,
- and a pointer to the previous Request List Element.

When tuples are placed into the tuple space by `linda_out` calls, the Tuple Server...
must determine whether these new tuples match any lodged requests. Tuple
Signatures are also used here to limit the number of lodged requests that must be
checked when a linda_out call is received. A lodged request is stored in a list
hanging off the Signature Record that was located when searching for tuples with the
same signature. Therefore a Signature Records has two lists: (a) a list of tuples, and
(b) a list of requests for tuples with that signature.

When the Tuple Server receives a new tuple, it uses a binary search to locate the
position where the Signature Record for the new tuple should be in the Signature
Table (see 5.2.1.4). If there is a list of lodged requests, the Tuple Server searches it
to locate any lodged requests that match the new tuple. If there are any lodged
requests that match the new tuple, the Tuple Server must send the new tuple to the
Client, using the Session Record, to satisfy the request. When lodged requests are
satisfied, the Request List Element is removed from the list and discarded, along with
the Tuple Template.

Since linda_rd calls do not remove tuples from the tuple space, multiple lodged
request may be satisfied. If the new tuple is not consumed by a lodged request
created by a linda_in call, the new tuple must still be placed into the tuple space.

Linda's matching semantics present two issues that should be discussed. The first
issue regards the matching semantics, the second issue relates to optimising the
matching process by using the matching semantics.

**Formals can match Formals**

As discussed in section 2.2.3, formal fields do not match formal fields. Jellinghaus
(1990, p. 77) points out that the reason for this is because "it hasn't been possible for
them to do so". The Eiffel-Linda by Jellinghaus represents formals as references;
they are NULL pointers. As formals now have the quasi-value NULL, Jellinghaus
(1990, p. 77) argues that "there is now no reason why formals cannot match

Edith Cowan University

Stephan Bettermann
formals'. DOS-C-Linda offers a compiler directive that changes the matching semantics so that formal tuple fields match formal tuple fields.

**Optimising Tuple Matching**

The matching of tuples that contain tuple fields that are structures (*structure tuple fields*) may be optimised by considering the matching semantics. Tuple field specifications of structure tuple fields begin, like all tuple field specifications, with a polarity character (see 4.2.2.2). Internally the polarity character of structure tuple fields may be used to optimise the matching process.

If formal fields do not match formal fields, structure tuple fields that contain any corresponding formal fields cannot match. The following example demonstrates this.

Assuming the following definition exists:

```c
struct
{
    int integer;
    double floating_point_number;
} structure_tuple_field = {42, 0};
```

The two tuples:

```c
( "%d %d %f", 42, &structure_tuple_field )
( "%d %d %f", 42, &structure_tuple_field )
```

do not match because the second fields in the structure tuple fields are both formal. Therefore structure tuple fields with any formal fields are indicated to be formal. This eliminates the need to match the values of any actual fields within such structures as the structure tuple fields already do not match. Only structure tuple fields that contain no formal fields are indicated to be actual. In this case the values of corresponding actual fields of structure tuple fields must be matched.
The advantage of this optimisation becomes negligible when formals do match formals. In this case the need to match the values of any actual fields within structure tuple fields is eliminated when structure tuple fields contain only formal fields, as the structure tuple fields already match. However structure tuple fields with only formal fields obviously do not contain any actual fields anyway.

5.2.2.2 Compressing Matching Tuples

When the Tuple Server finds a tuple in the tuple space that matches tuple template from a Client, the tuple must be sent to the Client. To reduce the amount of traffic on the network such tuples are compressed. Tuples are compressed by removing values of actual fields from values strings that will not have to be mapped onto corresponding formal fields of tuple templates.

5.2.2.3 Mapping Tuples

When Clients receive matching tuples from the tuple space, actual fields are mapped onto corresponding formal fields. Each formal field in a tuple template is accompanied by an address in the variable length argument list (see 4.2.2.1). Each address points to a memory location where the value of an actual field from a matching tuple is to be stored. The value of the actual field is copied from the values string into the memory location pointed to by the address in the variable length argument list. The complex types, arrays, structures, and pointers, introduce some complexity into this operation. The mapping of the complex types is discussed below.
References to data objects are mapped onto the memory location pointed. The following example illustrates this. Assume the following definitions exist:

```c
int integer = 42;
int *pointer = &integer;
int **ppointer = &pointer;
int ***pppointer = &ppointer;
```

The following tuple:

```c
("%*****d", 42 )
```

is a tuple with one field. The field is a pointer to a pointer to pointer to a pointer to a signed integer of value 42. It is removed with the following linda_in call:

```c
linda_in( "%*****d", &pppointer );
```

The integer value 42 is ultimately placed into the memory location occupied by the variable integer. Therefore the pointer chain depicted in Figure 6 must be traversed, as the variable length argument list contains the address of ppppointer, not integer.

DOS-C-Linda does not allocate memory for the value 42 and place the address into pointer, as the Client would have to deallocate the space. This could clearly cause garbage creation.
Mapping Structures

Following is an example of mapping values of actual fields to fields of structures.
Assume the following declaration and definitions exist:

```c
typedef struct
{
    int integer;
    float floating_point_number;
} STRUCTURE_TYPE;

STRUCTURE_TYPE out_structure = {42, 3.14f};
STRUCTURE_TYPE in_structure = {42, 0.0f};
```

The following tuple:

```
( "%{d %hf}", &out_structure )
```

is removed from the tuple space with the following operation:

```
linda_in( "%{d %hf}", &in_structure );
```

The `in_structure.floating_point_number` field must be assigned the floating point value 3.14f. The address in the variable length argument list is the address of `in_structure`, and therefore the address of the first field `integer`. The address of the second field is obtained by adding the size of the `integer` field to the address of `in_structure`.

Mapping Arrays

Arrays are mapped in a manner similar to structures. Arrays of base types can be
Implementation

mapped with a single memcpy\textsuperscript{14} call as the array occupies a contiguous area of memory. Arrays of complex types however cannot, as these arrays may not occupy contiguous areas of memory. An example of such arrays are arrays of references. Arrays of structures pose the same problem as discussed in section 5.2.1.2. Therefore DOS-C-Linda always maps arrays by traversing each element.

5.2.3 The linda\_inp operator

The linda\_inp operator functions similar to the linda\_in operator discussed above in section 5.2.2. Unlike the linda\_in operator however, if the Tuple Server cannot find a matching tuple in the tuple space, it sends a Null Tuple (see 5.2.2.1) causing the linda\_inp operator to fail.

5.2.4 The linda\_rd Operator

The linda\_rd operator also functions similar to the linda\_in operator. Unlike the linda\_in operator however, the tuple matching the tuple template is not removed from the tuple space.

5.2.5 The linda\_rdp Operator

The linda\_rdp operator functions similar to the linda\_rd operator. However, unlike the linda\_rd operator, the linda\_rdp operator is not left waiting when no matching tuples are found in the tuple space. In this case the Tuple Server sends a Null Tuple, and linda\_rdp fails.

\textsuperscript{14} memcpy is declared in the ANSI C standard string.h header file.
5.2.6 The linda_eval Operator

The linda_eval operator creates processes that run in parallel to the Clients that called the operator. The only argument of the linda_eval operator is the operating system name of the executable file representing the process that must be run in another Client Shell. The executable file is sent to the Tuple Server.

Upon receipt of a buffer indicated to have been sent by a linda_eval operator, the Tuple Server checks the Client Table (see 5.1.1) to locate an available Client Shell. The Tuple Server then sends the received executable file to the available Client Shell without modification. The Client Shell receives the executable file representing a process and writes it to secondary storage. The executable file is then executed by calling the system function. When the executable file has completed execution, the Client Shell deletes the executable file from secondary storage and notifies the Tuple Server that the process, represented by the executable file, has completed.

If no Client Shell is available for the process to run on, the Tuple Server stores the executable file representing the processes until a Client Shell becomes available. When a Client Shell becomes available, the executable file is retrieved from secondary storage and sent to the available Client Shell for execution.

5.2.6.1 Sending Processes

Executable files representing processes may be larger than the 64K size limitation of NetBIOS buffers (see 2.3.1). Therefore executable files larger than 64K are sent in fragments. As well as the executable file, the filename of the executable file is sent. Therefore the first buffer consists of a Command Block, the filename of the executable file, and the first executable file fragment. Names of executable files are

\[ 15 \text{system is declared in the ANSI C standard stdlib.h header file.} \]
'\0' terminated sequence of characters. The remaining executable file fragments occupy entire buffers. Assuming an executable file Sqrt.EXE of size 129K's exists on secondary storage, the following DOS-C-Linda call:

\[ \text{linda\_eval( "Sqrt.EXE" )} \]

causes Sqrt.EXE to be sent to the Tuple Server. Sqrt.EXE is fragmented into three buffers. The first buffer will contain a Command Block, the filename "Sqrt.EXE" and the first fragment. The Command Block occupies six bytes in the buffer, and the filename occupies a further nine bytes, so the first buffer contains the first 65,521 bytes of the process. The second buffer contains the second fragment of 65,536 bytes, thus leaving a 1,039 bytes fragment for the third buffer. Because the third buffer does not occupy 65,536 bytes, the end of the fragment sequence is recognized. If the last fragment occupies an entire buffer, as would be the case if Sqrt.EXE had been 196,593 bytes (first fragment is 65,521 bytes, the remaining two fragments are both 65,536 bytes), an additional fragment of size 0 bytes is used to indicate the end of the process' executable file.

The sending of fragments to the Tuple Server introduces a problem however. As the Tuple Server uses outstanding NCB.Receive.Any commands, other Clients and Client Shells may interrupt the transmission of an executable file. Therefore after receiving the first fragment of an executable file, the Tuple Server uses NCB.Receive's to lock out other Clients and Client Shells. When the last fragment of the executable file has been received, the Tuple Server issues an outstanding NCB.Receive.Any command to again allow all Clients to communicate with the Tuple Server.

5.2.6.2 Storing Processes

When the Tuple Server receives a linda_eval request and no Client Shell is available for processes to be executed on, the executable file representing the process must be stored until a Client Shell becomes available. The Tuple Server stores the
Implementation

executable files representing processes by writing them to secondary storage as temporary files. The names of these temporary files must be unique because two Clients may linda_eval the same process. Therefore the Tuple Server allocates unique names to temporary files.

The format of names for temporary files is Linda###.TMP where # is a digit between 0 and 9, thus giving 1,000 possible unique names. The names are allocated in a round-robin fashion. The first temporary file is given the name Linda000.TMP, the second temporary file is named Linda001.TMP, etc. Once Linda999.TMP has been used, Linda000.TMP is used again if available. Therefore up to 1,000 temporary files may be created, and thus up to 1,000 processes may be outstanding at any one time, assuming that there is enough space on secondary storage for the executable files of 1,000 outstanding processes. If no more temporary names are available, the DOS-C-Linda Machine is shut down. This is reasonable as the DOS-C-Linda Machine is obviously overloaded.
The following data structure is used to keep track of outstanding processes:

![Diagram of Outstanding Processes]

**Figure 12: List of Outstanding Processes**

### 5.2.7 The linda_evalp operator

The linda_evalp operator functions in the same manner as the linda_eval operator. It also creates processes that run in parallel to the Clients that call the operator. Unlike the linda_eval operator, when no Client Shells are available for processes to be run on, linda_evalp fails.

The linda_evalp operator first asks the Tuple Server whether a Client Shell is available. This is done by sending a buffer containing only a command block identifying the calling operator as linda_evalp. Upon receipt of a buffer indicated to have been sent by a linda_evalp operator, the Tuple Server checks the Client Table (see 5.1.1) to determine if a Client Shell is available. The Tuple Server then notifies the Client by sending a buffer with a command block indicating whether or not a Client Shell is available.
Implementation

When the Client receives notification that there is no available Client Shell, the linda_evalp operator fails. If notification is received that a Client Shell is available for the process to run on, it sends the executable file representing the process to the Tuple Server in the same manner as described in section 5.2.6.1. Unlike the linda_eval however, where the Tuple Server issues an NCB.Receive to lock out other Clients and Client Shells after receiving the first executable file fragment (see 5.2.6.1), an NCB.Receive is issued before receiving the first fragment. This prevents other Clients and Client Shells from interrupting before the first fragment is sent.
6 Validation and Evaluation

This chapter consists of three sections. The first section describes the validation phase of the development of DOS-C-Linda. The second section compares DOS-C-Linda to two other implementations, and evaluates DOS-C-Linda’s capabilities. This serves to prove the hypothesis that a Linda implementation running in a LAN of PC’s can show the same performance gains Linda implementations in expensive and specialised environments. The third section describes two applications implemented using DOS-C-Linda: (a) the dining philosophers problem posed by Dijkstra, and (b) a parallel genetic algorithm. The two applications demonstrate that DOS-C-Linda can be used as a parallel programming tool.

6.1 Validation

The validation of DOS-C-Linda consisted of unit testing and system testing stages. Unit testing was performed during the implementation phase, and tested the individual components making up DOS-C-Linda. System testing was performed once DOS-C-Linda was fully implemented.

To ensure that

- all the memory allocated was deallocated,
- only memory allocated was freed,
- memory being resized using the remalloc function was previously allocated,
a memory management library was written. It consists of the following four functions:

- `x_malloc`
- `x_free`
- `x_realloc`
- `check_memory`

The `x_malloc`, `x_free`, and `x_realloc` functions replace the `malloc`, `free`, and `realloc` functions respectively supplied by the `stdlib` library.

The `x_malloc` function operates similar to `malloc`, however unlike `malloc`, `x_malloc` maintains an Allocation Record for each allocated memory block. An Allocation Record contains:

- the address and size of the allocated memory block,
- the name of the source file from which `x_malloc` was called to allocate the memory block,
- and the line number in the source file that it was called from.

When the `x_free` function is called, it checks the Allocation Records to ensure the memory block to be deallocated was previously allocated. If the memory block was allocated, the memory block and its Allocation record are deallocated. If the memory block was not allocated, `x_free` prints an error message and returns an error code.

When the `x_realloc` function is called, it checks the Allocation Records to ensure that the memory block to be resized was previously allocated. If the memory block was not previously allocated, `x_realloc` prints an error message and returns `NULL`. If the memory block was previously allocated, `x_realloc` resizes the memory block and modifies its Allocation Record accordingly.
The `check_memory` function is a garbage collector. `check_memory` is explicitly called by the programmer before program termination to ensure that all allocated memory blocks have been deallocated. If an allocated memory block has not been deallocated, `check_memory` will deallocate the memory block and its Allocation Record, and display a warning message. The warning message details the size of the memory block, and the source file and line number from where the call was made to allocate or resize the memory block.

All `malloc`, `free`, and `realloc` function calls were replaced with `x_malloc`, `x_free`, and `x_realloc` calls respectively during the validation phase. The `check_memory` function was called before program termination to ensure all memory was deallocated. Upon completing the validation phase, all `x_malloc`, `x_free`, and `x_realloc` calls were replaced with `malloc`, `free`, and `realloc` calls respectively, and the `check_memory` call was removed.

The source code of this library is included on the disk incorporated as Appendix A.

### 6.1.1 Unit Testing

DOS-C-Linda was extensively tested using the debugging facilities available in Turbo C++ Version 1.0, under which DOS-C-Linda was developed. Upon completion of a component, such as building of data tuples (see 5.2.1.2), every execution path of the component was stepped through using the trace facility. The inspector facility was used to validate that the data structures used by the component were manipulated correctly.

### 6.1.2 System Testing

Upon completion of the implementation phase, system testing was performed by running a series of test programs. These programs were designed to test every
Validation and Evaluation

aspect of DOS-C-Linda. The test programs are included on the disk incorporated as Appendix A.

6.2 Evaluation

The objectives of this section are to compare DOS-C-Linda to other implementations described in the literature, and to evaluate DOS-C-Linda's performance. A complete performance analysis is out of the scope for this study.

Two Linda implementations are available for DOS-C-Linda to be compared against: (a) the Modula-Linda system described by Börmann and Herdieckerhoff (1989), and (b) the S/Net Linda Kernel described in Carriero and Gelernter (1986). These two implementations are chosen for the comparison because performance data is available for them. The methods used to obtain the performance data is described in the literature and may therefore be obtained for the DOS-C-Linda system.

Börmann and Herdieckerhoff (1989) use three methods to evaluate the performance of their Modula-Linda: (a) the execution delays of the Linda operators measured in real time, (b) a comparison of the Linda operators to message passing, and (c) a synthetic benchmark called Simpl. Carriero and Gelernter (1986) use more general means to measure the performance of the S/Net Linda Kernel: (a) the time taken to transfer a simple tuple between processes, and (b) the time taken to multiply various sized matrices using both sequential and parallel programs.

The matrix multiplication problem is used to establish the capability of the Linda system. The execution time a sequentially running program takes to multiply two matrices is compared with the time taken by a program using multiple processes to multiply the same matrices. The comparison can be used to determine the speedup of a parallel implementation of an application over the sequential application. It can also be used to determine the performance of the Linda implementation.
The synthetic benchmark Simp1 is too complex for the purposes of this section. Therefore the evaluation of DOS-C-Linda is limited to:

- measuring the execution delays of the seven DOS-C-Linda operators
- tuple transfer time between processes
- a comparison of the DOS-C-Linda operators to direct NetBIOS message passing
- measuring the execution time of the matrix multiplication program described in Carriero and Gelernter (1986)

Before the results are presented, the hardware and software environments of the three implementation must be described. These environments differ in capacity, and are therefore the variables in the comparison.

### 6.2.1 Hardware and Software Environments

The performance evaluation of DOS-C-Linda was performed on a local area network of four PC's accessed using the session layer provided by IBM NetBIOS 1.1. The local area network was an IBM Token Ring Network running at 4 Mbits per second. The four PC's connected to the LAN where a 33 Mhz 80386, a 10 Mhz 80286, and two 80286 IBM AT's running at 8 Mhz. None of the PC's had Maths-coprocessors. The 33 Mhz 80386 ran MS-DOS 5.0, the two AT's and the 10 Mhz 80286 ran MS-DOS 3.3.

Bormann and Herdieckerhoff's Modula-Linda runs on a "Parawell-1, a 37-node distributed memory multiprocessor" (Bormann and Herdieckerhoff, 1989, p. 6). Each node consists "of a Motorola processor pair 68020/68881, up to four MByte of local, dual-ported memory, two 32-bit bus connections and an address translation logic" (Bormann and Herdieckerhoff, 1989, p. 6). The capacity of the network bus is not reported.
Carriero and Gelernter's S/Net Linda Kernel runs on an S/Net. The S/Net "consisted of 8 MC-68000's with local memory ranging from 650 to 1200 kbytes, and a VAX 11/750, all connected via a word-parallel bus whose capacity is about 80 Mbits per second" Carriero and Gelernter (1986, p. 118).

The execution delays of the DOS-C-Linda operators are now discussed. Execution delays of three Modula-Linda operators are reported in Borrmann and Herdieckerhoff (1989, p. 6) and are included in the discussion.

### 6.2.2 DOS-C-Linda Operator Execution Delays

The execution delays of the seven DOS-C-Linda operators were measured using the \texttt{time}\textsuperscript{16} function. The Tuple Server providing the tuple space ran on the 33 Mhz 80386, the timing program ran in a Client Shell on an 8 Mhz 80286. No other PC's were connected to the LAN.

As the \texttt{time} function returns seconds, the execution delays of the data tuple and the failing \texttt{linda\_evalp} operators were obtained by timing the execution delay of 1,000 successive executions. The execution delay of a single call was obtained by dividing the execution delay of the 1,000 successive calls by 1,000, giving millisecond accuracy. The \texttt{linda\_eval} operator was executed 50 times successively, making the data tuple and the \texttt{linda\_evalp} operator measurements 20 times more precise. It turns out that the \texttt{linda\_eval} operator has an execution delay 20 times that of the \texttt{linda\_in}, the longest data tuple execution delay, and therefore the imprecision is inconsequential.

\textsuperscript{16} \texttt{time} is declared in the ANSI C standard \texttt{time.h} header file.
The data operators manipulated the following tuple:

\[
(\text{"%s"}, \text{"Ping"})
\]

Characters occupy one byte, and the Command Block preceding all buffers used for tuple transmission is 6 bytes long, so the size of the buffer needed for NetBIOS communication is 14 bytes. The \texttt{linda\_out} operator placed the above tuple into the tuple space. The \texttt{linda\_in} operator read and removed the tuple. The \texttt{linda\_inp} and \texttt{linda\_rdp} operators provide execution delays of failing predicates. As succeeding \texttt{linda\_inp} and \texttt{linda\_rdp} operators function the same as the \texttt{linda\_in} and \texttt{linda\_rd} operators respectively, the execution delays of succeeding \texttt{linda\_inp} and \texttt{linda\_rdp} operators are identical to the execution delays of the \texttt{linda\_in} and \texttt{linda\_rd} operators respectively.

The \texttt{linda\_eval} and \texttt{linda\_evalp} operators created processes 32,768 bytes and 65,536 bytes in size. With a buffer size of 60K, one buffer is required for 32,768 byte processes, and two buffers for 65,536 byte processes. The Tuple Server had no Client Shells available where the processes could be run on and therefore stored the processes on secondary storage; the \texttt{linda\_evalp} operator was failed.
The table below presents the execution delays. The execution delay of the linda_in operator shows two figures. The execution delay marked random refers to a linda_in operating on a tuple space that selects tuples from sets of matching tuples nondeterministically (see 5.2.2.1). The execution delay marked FIFO refers to a linda_in operating on a tuple space that selects the first matching tuple.

Execution delays of three Modula-Linda operators are also shown:

<table>
<thead>
<tr>
<th>Linda Operator</th>
<th>C-Linda</th>
<th>Modula-Linda</th>
</tr>
</thead>
<tbody>
<tr>
<td>out</td>
<td>7 ms</td>
<td>0.44 ms</td>
</tr>
<tr>
<td>in</td>
<td>35 ms (random)</td>
<td>1.4 ms (local)</td>
</tr>
<tr>
<td></td>
<td>12 ms (FIFO)</td>
<td>4.8 ms (global)</td>
</tr>
<tr>
<td>inp / rip (failure)</td>
<td>11 ms</td>
<td>5.3 ms</td>
</tr>
<tr>
<td>rd</td>
<td>12 ms</td>
<td>N/A</td>
</tr>
<tr>
<td>eval</td>
<td>300 ms (32,768 bytes)</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>340 ms (65,536 bytes)</td>
<td>N/A</td>
</tr>
<tr>
<td>evalp (failure)</td>
<td>10 ms</td>
<td>N/A</td>
</tr>
</tbody>
</table>

The asynchronous nature of the linda_out operator explains why linda_out has the lowest execution delay. The tuple is built from the argument list, and transmitted to the Tuple Server. Placing the tuple into the tuple space does not delay Clients, and therefore the operator completes after the tuple's transmission.

The failing linda_eval operator comes second as no tuples are build, transmitted, searched for, or mapped. Clients sends a request to the Tuple Server, the Tuple Server checks the Client Table and sends an answer to the Clients (see 5.2.7). Because no free Client Shells are available, the linda_evalp operator fails immediately. Succeeding linda_evalp's are not measured because an available Client Shell is required for each succeeding linda_evalp call. As the execution
Validation and Evaluation

delays for the \texttt{linda\_evalp} operator is measured by calling it 50 times, 50 available Client Shells would be required. Only 3 Client Shells are available.

The failing \texttt{linda\_inp} and \texttt{linda\_rdp} operators come next because tuples have to be built, sent, searched for, and Null Tuples have to be returned. These operators are faster than the \texttt{linda\_in} and \texttt{linda\_rd} operators because no matching tuples exist. The Tuple Server determines that no matching tuples exist, as no sets of tuples with same Tuple Signatures exist. Therefore Null Tuples are returned, resulting in no mapping.

The \texttt{linda\_rd} operator is faster than the \texttt{linda\_in} operator as the tuple does not have to be removed from the tuple space. Matching tuples are simply transmitted to Clients. The \texttt{linda\_in} operator however requires that the tuple be removed from the tuple space, therefore the execution delay of this operator is longer than that of the \texttt{linda\_rd} operator.

The delay caused by the \texttt{linda\_eval} operator is affected by two parameters: (a) the size of the process, and (b) the availability of Client Shells. The transmission delay caused by the smaller processes is:

\[ \frac{32,768 \times 8}{4,000,000} = 65.5 \text{ ms} \]

and the larger process caused delays of:

\[ \frac{65,536}{4,000,000} = 131.1 \text{ ms} \]

Reading and writing these processes to and from secondary storage caused the greatest delay. A program was written that measures the time spend by the Tuple Server reading, and the time spent by Client Shells writing files 32,768 and 65,536 bytes in size in the same manner as the Tuple Server and Client Shells do. The program writes and reads these files 50 times to obtain an average time. It

---

Edith Cowan University

Stephan Bettermann
determined that the smaller files take 420 ms to write on the PC's used to run the Client Shells, and 640 ms for the larger files. The PC used to run the Tuple Server takes 80 ms to read the smaller files, and 120 ms for the larger files.

To determine the time required to transfer tuples between processes a simple "ping-pong" program as described in Carriero and Gelernter (1986, p. 120) was implemented and run.

### 6.2.3 Tuple Transfer Time

It consists of two processes named *Ping* and *Pong* that exchange the following tuples:

```
( "%s", "Ping" )
( "%s", "Pong" )
```

Carriero and Gelernter (1986) use this method to provide performance data for their S/Net Linda Kernel. Borrmann and Herdieckerhoff (1989) also use this method to "render a comparison of Linda to message passing" (Borrmann and Herdieckerhoff, 1989, p. 6). Thus this experiment allows the comparison of DOS-C-Linda to both the S/Net Linda Kernel and Modula-Linda. The DOS-C-Linda Machine consisted of a 33 Mhz 80386 Tuple Server, and two 8 Mhz 80286 Client Shells.
The two processes Ping and Pong are described by Carriero and Gelernter (1986, p. 120) as follows:

PING:  count = 0;
       while (TRUE)
       {
           in( "ping" );
           if (++count == LIMIT) break;
           out( "pong" );
       }
       print elapsed time

PONG:  while (TRUE)
       {
           out( "pong" );
           in( "ping" );
       }

The number of ping-pong pairs transmitted per second is measured. The time taken to transmit a tuple between two Linda programs is half the time taken to exchange a ping-pong pair. DOS-C-Linda exchanged 30.675 ping-pongs per second, therefore the time taken to exchange one tuple between two processes is:

\[
\frac{1,000}{(30.675 \times 2)} = 16.3 \text{ ms}
\]

6.2.4 Communication Costs

The communication costs of DOS-C-Linda may be obtained by comparing a tuple’s transfer time using DOS-C-Linda to the time required to exchange the data contained in the tuple directly using NetBIOS. Two programs were implemented to provide the communication costs of exchanging the ping-pong tuple pair described in section 6.2.3 above directly using NetBIOS. When run these two programs exchanged 100 ping-pong pairs per second. This figure represents the capability of the message
passing model. The time taken for one tuple transfer is therefore:

\[
\frac{1,000}{(100 \times 2)} = 5 \text{ ms}
\]

Table 5 shows the performance of the three systems. The Modula-Linda implementation provides two figures for each measurement. The tuple space in this implementation is distributed into Clusters of Clients. Local refers to communication between Clients within a common Cluster; global refers to inter-cluster communication (see Borrmann and Herdieckerhoff (1989) for a more detailed discussion). Performance data for the S/Net Linda Kernel is limited to the Linda calls, for which two figures are provided; message passing data is not available. The first measurement refers to a slower protocol, the second to a faster protocol. For a more detailed discussion the reader is referred to Carriero and Gelernter (1986).

<table>
<thead>
<tr>
<th></th>
<th>DOS-C-Linda</th>
<th>S/Net</th>
<th>Modula-Linda</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linda</td>
<td>16.3 ms</td>
<td>1.38 ms</td>
<td>2.7 ms (local)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.30 ms</td>
<td>3.4 ms (global)</td>
</tr>
<tr>
<td>Message Passing</td>
<td>5 ms</td>
<td>N/A</td>
<td>3.1 ms (local)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4.3 ms (global)</td>
</tr>
</tbody>
</table>

As can be seen, direct message passing in the environments used by DOS-C-Linda and by Modula-Linda requires approximately the same time, yet DOS-C-Linda is about 5.5 times slower than Modula-Linda. Modula-Linda is however not built on top of the message passing system used for the comparison; this is exemplified by the lower amount of time required by Modula-Linda. Therefore a relative comparison of Modula-Linda and DOS-C-Linda cannot be presented.

A relative comparison of DOS-C-Linda to the S/Net Linda Kernel can however be
presented. DOS-C-Linda is about 12 times slower than the S/Net Linda Kernel in real time. Considering however that the S/Net Linda Kernel used a LAN that had a capacity approximately 20 times that of the LAN used by DOS-C-Linda, DOS-C-Linda performs reasonably.

A more general analysis of DOS-C-Linda's performance is obtained by an examination of the matrix multiplication problem.

6.2.5 Matrix Multiplication Problem

A matrix multiplication program based on the matrix multiplication program described in Carriero and Gelernter (1986, p. 114) was implemented using DOS-C-Linda. It consists of three processes: an initialisation process, multiple worker processes, and a cleanup process.

Assuming that matrices A and B are to be multiplied, the initialisation process places the rows of matrix A, each row being one tuple, and the columns of matrix B, each column being one tuple, into the tuple space. Multiple workers are then created using the linda_eval operator. Each worker computes one element of the product matrix using the rows and columns of matrices A and B in the tuple space, and places the result into the tuple space. Upon calculation of all the elements in the product matrix the workers terminate and the cleanup process extracts the product matrix and matrices A and B from the tuple space.

The matrix multiplication program was run on a DOS-C-Linda Machine consisting of a 33 Mhz Tuple Server, a 10 Mhz initialisation and cleanup process, and 8 Mhz
workers. Figure 13 shows the execution times of multiplying two 64x64 matrices using one and two workers. The execution times of the same program running on the S/Net Linda Kernel is also shown, here with one to four workers. The execution times of multiplying the matrices in sequentially running programs are also shown. The sequential version was run on the PC's running the workers:

![Graph showing execution times](image)

**Figure 13: Execution time of multiplying 64x64 matrices**

As can be seen, the DOS-C-Linda program takes longer than the sequential version even with two workers. The S/Net Linda Kernel version with two workers beats the sequential version. The same applies to the multiplication of the smaller matrices of size 32x32. The execution times under the S/Net Linda Kernel and under DOS-C-Linda are shown in Figure 14.
In order to determine the number of workers required by the DOS-C-Linda program to beat the sequential version, more Client Shells are required. Because a sufficient number of Client Shells were not available, the number of workers required to beat the sequential version cannot be determined. What however can be deduced is that the grain size of the DOS-C-Linda program is too fine. Grain size refers to the "duration of computation between two communications" Bormann and Herdieckerhoff (1989, p. 7). To prove that this is the case the grain size of the matrix multiplication program was increased. Grain size was increased by getting the Clients to calculate an entire row of the product matrix instead of a single element. This increases the granularity as the amount of computation between communications increases.

Carriero and Gelernter (1986) also increased the grain size of their matrix multiplication program to study the effect on execution time using this method. This provides the necessary data to further compare DOS-C-Linda to the S/Net Linda...
A second matrix multiplication program was implemented. Here the Clients calculated an entire row of the product matrix. Table 6 shows the execution times of the DOS-C-Linda and S/Net Linda Kernel programs with different granularities:

**Table 6: Execution Times of multiplying 32x32 matrices**

<table>
<thead>
<tr>
<th>System</th>
<th>Sequential</th>
<th>1 Worker</th>
<th>2 Workers</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Fine</td>
<td>Coarse</td>
</tr>
<tr>
<td>DOS-C-Linda</td>
<td>20 sec</td>
<td>180 sec</td>
<td>58 sec</td>
</tr>
<tr>
<td>S/Net</td>
<td>14 sec</td>
<td>22 sec</td>
<td>15 sec</td>
</tr>
</tbody>
</table>

As can be seen, the effect of a coarser granularity on the execution times of the S/Net Linda Kernel program is not as substantial as on the DOS-C-Linda program. The execution time of the DOS-C-Linda program multiplying a 32x32 matrix using one worker for example, decreases by a factor of:

\[
\frac{180}{58} = 3.103
\]
Figure 15 shows the effects of increased granularity in terms of decreased execution times on the DOS-C-Linda program and the S/Net Linda Kernel program:

![Effect of Grain Size on Execution Times]

**Figure 15: Effect of Grain Size on Execution Times**

As Figure 15 shows, increasing the granularity of programs running under DOS-C-Linda has a more significant effect than on programs running under the S/Net Linda Kernel. It can be deduced that the communication costs of DOS-C-Linda are significantly higher than those of the S/Net Linda Kernel.

### 6.2.6 Conclusions

DOS-C-Linda does not perform as well as the S/Net Linda Kernel and Modula-Linda in terms of real time. However the hardware and software used by the S/Net Linda Kernel is faster than the hardware and software used by DOS-C-Linda. Taking these factors into consideration, DOS-C-Linda performs reasonably against the S/Net Linda Kernel in relative terms.
The more detailed analysis and comparison of DOS-C-Linda using the matrix multiplication problem in section 6.2.5 reveals that DOS-C-Linda’s communication costs are relatively high compared to the S/Net Linda Kernel’s communication costs. Consequently it can be deduced that applications using DOS-C-Linda to achieve significant speedups over sequential implementations need to be coarse grained. The effects of increased granularity on the execution times of the matrix multiplication programs support this conclusion.

6.3 Implemented Applications

Two application were successfully implemented using DOS-C-Linda, demonstrating that DOS-C-Linda works for real applications. The dining philosophers problem was chosen for a reason best described by Ben-Ari (1982, p. 109) as:

The dining philosophers problem (posed by Dijkstra) is of great importance in concurrent programming research. The problem allows all of the pitfalls of concurrent programming to be demonstrated in a vividly graphical situation. It is a challenge to proposers of new primitives for concurrent programming.

The second application is a parallel genetic algorithm implemented using DOS-C-Linda.

6.3.1 The Dining Philosophers

each seat. A Philosopher eats by sitting in a seat and picking up the two forks immediately next to him/her, the right fork first, the left fork second. After eating he/she puts the forks back onto the table (hopefully washing them beforehand) and leaves the seat to think.

A problem exists when all philosophers sit down and pick up the right forks at the same time. All philosophers will have one fork each. To eat, two forks are required. Because philosophers place the forks back onto the table only after eating, no philosopher will ever get the two forks required for eating. This situation is called a *deadlock*.

Gehani (1989, p. 158) describes two methods in which deadlocks can be avoided:

1. philosophers pick up both forks when both are available, therefore philosophers never end up with just one fork, or
2. never allowing all five philosophers to simultaneously pick up forks.

Limiting the number of philosophers picking up forks to four means that at least one philosopher eats.

Both methods have been implemented using DOS-C-Linda. The source code and executable files are on the disk included as Appendix A.

### 6.3.1.1 First Method

The process of picking up two forks can cause deadlock, as all philosophers can end up with one fork. The deadlock can be avoided by not allowing philosophers to pick up only one fork. Philosophers either pick up both forks, or pick up no forks.

This implementation introduces the philosophers to a fork rack. The fork rack contains all the five forks. When a philosopher wants to eat, he/she picks up the
fork rack, removes the left and right forks, and places the fork rack back onto the table. When a philosopher picks up a fork rack that does not contain both the left and right forks, the philosopher places the fork rack back onto the table without removing any forks. When philosophers finish eating, the left and right forks are put back into the fork rack.

Because forks are now obtained in a monotonic fashion a deadlock cannot occur. Philosophers get either both, or no forks.

### 6.3.1.2 Second Method

The second method introduces a ticket system that prevents the philosophers from all eating at once. When a philosopher wants to eat, a ticket must be obtained. There are only four tickets therefore only four philosophers may eat simultaneously. Because there are five forks, at least one philosopher will have the required left and right forks to eat. As there is at least one philosopher eating, a deadlock cannot occur. The ticket is returned after eating.

### 6.3.2 Parallel Genetic Algorithm

The genetic algorithm consists of two Client types: (a) an *Initialising Client*, and (b) multiple *Worker Clients*. The Initialising Client creates the *gene pool* to be manipulated by the Worker Clients. The gene pool is a collection of *genes* akin to the tuple space which is a collection of tuples. The gene pool is created by placing the genes in the form of tuples into the tuple space. A gene is an array of bits.

The Worker Clients manipulate the gene pool by reading two gene tuples (parent genes) from the tuple space, from which two child genes are produced. The child genes are produced by splicing the parent genes at a random position, and crossing them over. After producing the child genes, two further genes, called *suckers*, are extracted from the tuple space. The first child gene is compared to the first sucker.
gene using a *goodness* function. The gene with the better goodness is then inserted into the gene pool. The same applies to the second child and second sucker genes.

The goodness function used in this algorithm compares the two bit streams representing the genes to a bit stream of zeros. Whichever bit stream is closer to a bit stream of zeros is of the higher degree of goodness, and therefore wins. The goal of the gene pool manipulation in this algorithm is therefore to obtain a gene pool of genes that are all bit streams of zeros.
7 Summary

7.1 The Linda Paradigm

The Linda paradigm is a new parallel model. It promotes a highly uncoupled programming style, the advantages of which are well known. The time and space uncoupling properties of Linda make it very suitable for fault tolerant applications (see 3.1) as revealed by Kambhata (1990).

Linda has been embedded in a wide variety of languages, operating in a wide variety of environments, making Linda programs very portable. Most of these environments are however expensive and specialised.

7.2 The Implementation

DOS-C-Linda is a complete implementation of Linda. The Linda operators are supplied via a library; the tuple space and PC's where processes created by the linda_eval and linda_evalp operators may be run, are provided by the DOS-C-Linda Machine. DOS-C-Linda operates in a LAN of PC's, the most inexpensive and widely available computing resource.

In contrast to other C-Linda implementations, DOS-C-Linda supplies the Linda operators via a Linda library. All other C-Linda implementations supply the Linda operators via a Linda compiler or precompiler. The advantages of a library over a compiler or precompiler are that of orthogonality and usability (see 4.3).
7.3 Performance Analysis

The analysis of DOS-C-Linda's performance reveals that DOS-C-Linda is slower, in real time, than the two other implementation that it was compared to. However, considering the hardware used by the two other implementations, DOS-C-Linda performs reasonably.

The evaluation phase also reveals that the communication costs of DOS-C-Linda are high compared to the communication costs of the S/Net Linda Kernel. It can be concluded that the major component affecting the performance of DOS-C-Linda is the capacity of the LAN.

7.4 Future Research Directions

DOS-C-Linda currently suffers two weaknesses as discussed in section 5.2.1.2: (a) arrays of pointers are only expressible as arrays of structures with a pointer field, and (b) structures must be passed into the DOS-C-Linda operators by reference. The first weakness requires the modification of the parser used by DOS-C-Linda. Overcoming the second weakness is more difficult, requiring the extension of the va_arg function to include the retrieval of a structure's individual fields from the run-time stack.

The literature review suggests several areas of research. Currently DOS-C-Linda is not fault tolerant. If DOS-C-Linda detects that it is in an unstable state (see 5.1.3.5), it shuts down. Xu and Liskov (1989) and Kambhatla (1990) suggest mechanisms of introducing fault tolerance into Linda systems (see 3.1). Xu and Liskov (1989) achieve fault tolerance through a distributed and replicated tuple space. The secondary advantage of this approach is that it "can also prevent the tuple space from becoming a bottleneck" (Xu and Liskov, 1989, p. 205). Taking the high communication costs of DOS-C-Linda into consideration, distributing the tuple space...
is appealing. Replicating the tuple space would provide for fault tolerance.

The performance analysis of DOS-C-Linda in section 6.2 is brief (see 1.4). A more detailed analysis of DOS-C-Linda's performance is required to completely determine the factors that affect DOS-C-Linda's performance.

7.5 Conclusions

The research has been successful. The objective to embed Linda into the C programming language for a LAN of PC's running DOS was met. The validation phase (see 6.1) confirmed that DOS-C-Linda is a working and complete implementation of Linda for that environment.

The evaluation phase (see 6.2) proves the hypothesis that applications using DOS-C-Linda in a LAN of PC's under DOS can show the same performance gains that applications using Linda running on specialised hardware and software have shown. The performance gains are limited to coarse grained applications however (see 6.2.5). The granularity required to achieve the performance gains can be controlled by manipulating the communication costs: a faster LAN means finer granularity.
8 Bibliography


---

Edith Cowan University
Stephan Bettermann


Edith Cowan University

Stephan Bettermann
Appendix A - Floppy Disk
# Table of Contents

1 Introduction .................................................. 116

2 Installing DOS-C-Linda .......................................... 117
   2.1 Requirements and Specifications .......................... 117
      2.1.1 Local Area Network .................................. 117
      2.1.2 Interrupt 60h ........................................ 118
      2.1.3 C Compilers .......................................... 119
      2.1.4 Outstanding Processes ............................... 119
   2.2 The Tuple Server ........................................... 119
      2.2.1 Starting ............................................... 119
      2.2.2 The Tuple Server Shell ............................... 121
      2.2.3 Shutting Down ........................................ 122
   2.3 The Client Shells ........................................... 122
      2.3.1 Logging In ............................................ 122
      2.3.2 Running Processes ................................... 125
      2.3.3 Logging Out .......................................... 125

3 The Linda Operators ........................................... 126
   3.1 Tuple Specification ........................................ 127
      3.1.1 Basic Types .......................................... 128
      3.1.2 Complex Types ....................................... 130
         3.1.2.1 Arrays ........................................... 130
         3.1.2.2 Structures ....................................... 131
         3.1.2.3 Arrays and Structures ........................... 133
         3.1.2.4 Pointers to Values ............................... 134
   3.2 linda_out .................................................. 137
   3.3 linda_in .................................................... 138
   3.4 linda_inp ................................................... 142
   3.5 linda_rd .................................................... 142
1 Introduction

DOS-C-Linda is an implementation of Linda for the C language in an environment of Personal Computers (PC's) connected by a Local Area Network. This document is useful to the programmer already familiar with Linda and the C programming language.

Chapter 2 explains how the DOS-C-Linda Machine is used. The requirements both hardware and software need to fulfil when using DOS-C-Linda are listed. The specifications and limitations of DOS-C-Linda are also described. A step through guide is given on starting the Tuple Server and the Client Shells.

Chapter 3 introduces the DOS-C-Linda operators and how to use them. Specifically it shows how tuples are described in DOS-C-Linda.

Chapter 4 is a short guide to using DOS-C-Linda. It explains how the DOS-C-Linda library is accessed, and how DOS-C-Linda programs are run.

Appendix A lists DOS-C-Linda's syntax in Backus Naur Form. This is useful for quick referencing.

Appendix B shows all the return codes of the DOS-C-Linda functions. Two tables are included listing the meaning of the return codes, what functions return what return codes. An explanatory list of these codes is also included showing the codes' message's meanings, causes, and any suggested corrective actions.

Appendix C lists the Tuple Server messages and their causes, and where appropriate, actions are suggested.

Appendix D lists the Client Shell messages and their causes, and where appropriate, actions are suggested.

Stephan Bettermann
2 Installing DOS-C-Linda

Before any of the DOS-C-Linda operators can be used the DOS-C-Linda Machine must be installed. The DOS-C-Linda Machine consists of a Tuple Server and multiple Client Shells, each on a separate PC connected to a local area network. The Tuple Server provides and manages the Tuple Space and the DOS-C-Linda Machine. Client Shells are DOS command shells \textit{logged} into the Tuple Server. DOS-C-Linda programs that are run from Client Shells have access to the Tuple Space provided by the Tuple Server. Client Shells are also used to run processes created by the DOS-C-Linda operators \texttt{linda} \_ \texttt{eval} and \texttt{linda} \_ \texttt{evalp}.

2.1 Requirements and Specifications

Before describing how the Tuple Server and the Client Shells are started, the requirements and specifications of DOS-C-Linda are clarified.

2.1.1 Local Area Network

The DOS-C-Linda Machine uses a Local Area Network for communication between processors. The interface to the network is NetBIOS which needs to be installed before DOS-C-Linda can be run. NetBIOS is available for the Token-Ring, PC Network Broadband, PC Network Baseband, and Ethernet\textsuperscript{1} network environments.

\textsuperscript{1} Ethernet\textsuperscript{\textregistered} is a registered trademark of the Xerox Corporation.

Stephan Bettermann
To install NetBIOS for a 4Mbps IBM\textsuperscript{2} Token Ring Network, for example, the following lines are added to the config.sys file:

\begin{verbatim}
files=30
buffers=30
FCBS=10,0
device = c:\NetBIOS\DXMAOMOD.sys 001
device = c:\NetBIOS\DXMC0MOD.sys
device = c:\NetBIOS\DXMT0MOD.sys
\end{verbatim}

In the case above the NetBIOS drivers are placed in the directory c:\NetBIOS. For other networks please consult your NetBIOS manual.

\section*{2.1.2 Interrupt 60h}

DOS-C-Linda requires the use of interrupt 60h for information transferal between DOS-C-Linda programs and Client Shells. The Tuple Server also uses interrupt 60h to prevent the starting of another Tuple Server or a Client Shell on the PC. If a DOS-C-Linda program detects that interrupt 60h is not set, it assumes that DOS-C-Linda is not installed. Similarly if interrupt 60h is set when starting a Client Shell or the Tuple Server, these programs will not continue as they believe that either a Client Shell or a Tuple Server are already running on the PC. The utility Kill60h.EXE, supplied with DOS-C-Linda, can be used to set interrupt 60h to NULL.

\footnote{IBM\textsuperscript{\texttrademark} is a trademark of the International Business Machines Corporation.}

Stephan Bettermann
2.1.3 C Compilers

Currently DOS-C-Linda compiles under both Turbo C++\(^3\), Version 1.0, and Microsoft C\(^4\), Version 6.0. The Turbo C++ predefined symbol \_TURBOC\_ is used to control compiler directives needed to overcome differences between the two compilers. DOS-C-Linda programs can be developed in either environments.

2.1.4 Outstanding Processes

When a DOS-C-Linda process is created but no Client Shell is available to run the new process, the Tuple Server stores the new process until a Client Shell becomes available. The maximum number of processes that may be outstanding is limited to 1000, disk space permitting.

2.2 The Tuple Server

2.2.1 Starting

The Tuple Server is the executable file TServer.EXE that comes with DOS-C-Linda. To start the Tuple Server type TServer at the command prompt as shown below, assuming that Tserver.EXE is in the path:

```
C:\>Tserver
```

If interrupt 60\(_h\) is already used the following message will appear:

```
Message ==> Linda is already installed because interrupt 60h is not NULL.
```

\(^3\) Turbo C\(^\circ\) is a registered trademark of Borland International, Inc.

\(^4\) Microsoft C\(^\circ\) is a registered trademark of the Microsoft Corporation.
The Tuple Server cannot be run until interrupt 60h is free. Remove any process that is using this interrupt. If no process is using the interrupt either run the utility Kill60h.EXE or reboot the machine, and try again.

The Tuple Server will first access the local area network. This will take about 30 seconds. If the network cannot be accessed the following message will be displayed:

DOS-C-Linda, Version 1.0
(C) 1991 Stephan Bettermann
Edith Cowan University, Western Australia, Australia
Message ==> Accessing network, please wait.
Message ==> NetBIOS has not been installed.

After accessing the network the following is displayed:

DOS-C-Linda, Version 1.0
(C) 1991 Stephan Bettermann
Edith Cowan University, Western Australia, Australia
Message ==> Accessing network, please wait.
Tuple Server :

The Tuple Server prompt asks for the Tuple Server's name. This name is used by the network to uniquely identify the Tuple Server. A name under NetBIOS can be up to 16 characters long of any character including spaces. A name of * (asterix) is
not legal, and the name must be unique to the network. After entering a name such as Tuple Server the next prompt appears asking for the name of a temporary directory:

DOS-C-Linda, Version 1.0
(C) 1991 Stephan Bettermann
Edith Cowan University, Western Australia, Australia
Message --> Accessing network, please wait.
Tuple Server : Tuple Server
Linda work directory:

Names of directories follow the usual format of DOS directory names. The directory is created by the Tuple Server and used to store processes temporarily. When the Tuple Server is shut down the directory is removed. After entering a directory name the Tuple Server Shell is entered:

DOS-C-Linda, Version 1.0
(C) 1991 Stephan Bettermann
Edith Cowan University, Western Australia, Australia
Message --> Accessing network, please wait.
Tuple Server : Tuple Server
Linda work directory: Temp
Linda Tuple Server >

Client Shells may now log into the Tuple Server.

2.2.2 The Tuple Server Shell

The Tuple Server Shell is a DOS shell. Any DOS command can be run from here. The maximum command length is 60 characters. It should however be noted that while a DOS command is running the Tuple Server is unable to deal with DOS-C-Linda requests from DOS-C-Linda programs.
Appendix B: DOS-C-Linda User Manual

A list of messages the Tuple Server can display, and their causes is included as Appendix C.

2.2.3 Shutting Down

To exit the Tuple Server the command shutdown is entered at the Tuple Server prompt. The Tuple Space is discarded and all Client Shells are logged out. If a Client Shell cannot be logged off because the Client Shell is not responding, the network session is simply closed. If any processes are still outstanding the Tuple Server will ask for permission to delete them. If no outstanding processes exist or all outstanding processes were deleted the temporary directory is removed.

2.3 The Client Shells

Logging a Client Shell into the Tuple Server takes place by running the Client Shell program and naming the Tuple Server to log into. The Client Shell then establishes communication, a network session, with the Tuple Server through the local area network.

2.3.1 Logging In

The Client Shell is the executable file Login.EXE that comes with DOS-C-Linda. To log a Client Shell into a Tuple Server type Login at the command prompt as shown below, assuming that Login.EXE is in the path:

C:\>Login

If interrupt 60h is already used the following message will appear:

Message ==> Linda is already installed because interrupt 60h is not NULL.
The Client Shell cannot be run until interrupt 60h is free. Remove any process that is using this interrupt. If no process is using the interrupt either run the utility Kill60h.EXE or reboot the machine, and try again.

If the network cannot be accessed the following message will be displayed:

DOS-C-Linda, Version 1.0
(C) 1991 Stephan Bettermann
Edith Cowan University, Western Australia, Australia
Message ==> Accessing network, please wait.
Message ==> NetBIOS has not been installed.

After accessing the Local Area Network which takes about 30 seconds, the program will ask for the Client Name:

DOS-C-Linda, Version 1.0
(C) 1991 Stephan Bettermann
Edith Cowan University, Western Australia, Australia
Message ==> Accessing network, please wait.
Client Name:

The Client Name is used by NetBIOS and the Tuple Server to uniquely identify the Client. A name under NetBIOS can be up to 16 characters long of any character including spaces. A name of * (asterix) is not legal, and the name must be unique to the network. After entering a name, such as Client 1, the next prompt that appears asks for the name of a temporary directory:

DOS-C-Linda, Version 1.0
(C) 1991 Stephan Bettermann
Edith Cowan University, Western Australia, Australia
Message ==> Accessing network, please wait.
Client Name : Client 1
Linda work directory:
Names of directories follow the usual format of DOS directory names. The temporary directory is created by the Client Shell and used to store processes temporarily. The temporary directory is removed when DOS-C-Linda is shut down. After entering a temporary directory name, such as Temp, the next prompt that appears asks for the name of the Tuple Server:

```plaintext
DOS-C-Linda, Version 1.0
(C) 1991 Stephan Bettermann
Edith Cowan University, Western Australia, Australia
Message ==> Accessing network, please wait.
Client Name : Client 1
Linda work directory: Temp
Tuple Server:
```

Type the name of the Tuple Server to complete the login process. The Tuple Server must have previously been installed. For instruction on how to install the Tuple Server see section 2.2. After entering the Tuple Server's name, such as Tuple Server, the Client Shell will establish a network session with the Tuple Server. After communication has been established the Client Shell is entered:

```plaintext
DOS-C-Linda, Version 1.0
(C) 1991 Stephan Bettermann
Edith Cowan University, Western Australia, Australia
Message ==> Accessing network, please wait.
Client Name : Client 1
Linda work directory: Temp
Tuple Server: Tuple Server
Linda >
```
2.3.2 Running Processes

The Client Shell is a DOS command shell. The maximum command length is 60 characters. DOS-C-Linda programs running in the Client Shell have access to the Tuple Space provided by the Tuple Server.

A list of the messages the Client Shell can produce and their causes is included as Appendix D.

2.3.3 Logging Out

Type `exit` at the Client Shell's command prompt to log out. The Tuple Server will either grant or deny permission to log out. Permission is denied if the Client Shell is expecting a process from the Tuple Server.
3 The Linda Operators

All Linda operators and a predicate form of the `eval` operator have been implemented as C functions. To eliminate conflicts with predefined functions, `linda_` precedes all operators. Thus the DOS-C-Linda operators are:

```c
linda_out
linda_in
linda_inp
linda_rd
linda_rdp
linda_eval
linda_evalp
```

The DOS-C-Linda operators dealing with data tuples are syntactically similar to the predefined C function `printf`\(^3\). Following are the function declarations of these operators:

```c
int linda_out( char *format_string, ... );
int linda_in( char *format_string, ... );
int linda_inp( char *format_string, ... );
int linda_rd( char *format_string, ... );
int linda_rdp( char *format_string, ... );
```

The process creation operators have the following function declarations:

```c
int linda_eval( char *process_name );
int linda_evalp( char *process_name );
```

All functions return an integer value that can be used by the caller to determine the

\(^3\) `printf` is declared in the ANSI C standard `stdio.h` header file.

Stephan Bettermann
completion status of the function call. The return codes and their meanings, as well as which functions can return them, are detailed in Appendix B.

The full syntax of the linda operators is specified in BNF in Appendix A.

### 3.1 Tuple Specification

The values of the tuple fields, or pointers to the values, are passed to the operators as arguments after the format string. The format string consists of a sequence of *field type specifications*, specifying the types of the following arguments. Field type specifications begin with a *polarity character* and end with a *field type specifier*. The polarity character is either % to indicate that the field is actual, or ? for formal fields. Between the polarity character and the field type specifier, optional *pointer* fields are possible. This indicates that the argument needs to be dereferenced one or more times to obtain the field's value. The pointer fields adopt their format from C, that is * for a pointer, ** for a pointer to a pointer and so on. The field type specifier is a character sequence defining the type of the next argument. A field type specifier can be either complex or basic. All C types are expressible, including the complex types arrays and structures.
3.1.1 Basic Types

Below is a table of the basic types and the corresponding argument type.

Table 1: Field Type Specifiers for Basic Types

<table>
<thead>
<tr>
<th>Field Type Specifier</th>
<th>Argument Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>d, i</td>
<td>int; decimal integer</td>
</tr>
<tr>
<td>u, o, x, X</td>
<td>unsigned int; unsigned integer</td>
</tr>
<tr>
<td>f, e, E, g, G</td>
<td>double;</td>
</tr>
<tr>
<td>c</td>
<td>char; character</td>
</tr>
<tr>
<td>b</td>
<td>unsigned char; character</td>
</tr>
<tr>
<td>p</td>
<td>void *; address</td>
</tr>
<tr>
<td>s</td>
<td>char *; character string. Null terminated.</td>
</tr>
</tbody>
</table>

Following is an example of specifying a tuple with seven actual fields. The fields are a signed integer of value 42, unsigned integer of value 42, a double of value 3.14, the character 'C', a byte of value 0, a pointer of value NULL, and the NULL terminated string "This is a string":

```
( "%d %u %f %c %b %p %s",
  42, 42u, 3.14, 'C', 0x00, NULL, "This is a string" )
```
Type modifiers are used to extend these basic types. Below is a table of the type modifiers, the basic types they can be applied to, and the corresponding argument type:

**Table 2: Type Modifiers for Basic Types**

<table>
<thead>
<tr>
<th>Type Modifier</th>
<th>Applied to</th>
<th>Argument Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>p</td>
<td>void _far *, far pointer</td>
</tr>
<tr>
<td>N</td>
<td>p</td>
<td>void _near *, near Pointer</td>
</tr>
<tr>
<td>h</td>
<td>d, i</td>
<td>short int; short signed integer</td>
</tr>
<tr>
<td></td>
<td>o, u, x, X</td>
<td>short unsigned int; short unsigned integer</td>
</tr>
<tr>
<td></td>
<td>f, e, E, g, G</td>
<td>float</td>
</tr>
<tr>
<td>I</td>
<td>d, i</td>
<td>long int; long signed integer</td>
</tr>
<tr>
<td></td>
<td>o, u, x, X</td>
<td>long unsigned; long unsigned integer</td>
</tr>
<tr>
<td>L</td>
<td>f, e, E, g, G</td>
<td>long double</td>
</tr>
</tbody>
</table>

Following is an example using the types listed in the table above. Assume the following definitions exist:

```c
void _far * far_pointer;
void _near * near_pointer;
short int short_integer;
short unsigned short_unsigned;
float floating_point_number;
long int long_integer;
long unsigned long_unsigned;
long double long_double;
```
The tuple specification for a tuple containing the variables defined above is as follows:

( "%Fp %Np %hd %hu %hf %ld %lu %Lf",
  far_pointer,
  near_pointer,
  short_integer,
  short_unsigned,
  floating_point_number,
  long_integer,
  long_unsigned,
  long_double )

3.1.2 Complex Types

3.1.2.1 Arrays

Arrays are indicated by the presence of an array size field after the optional pointer field in the format string. An array size field is either an integer or a #. If a # is used, the array size is taken from the next argument in the tuple field list, and used to replace the # in the format string. Assuming the following C variable definition exists:

```c
int array[100];
```

The following is a tuple specification for the array:

( "%100d", array )

Alternatively the specification could be:

( "%#d", 100, array )
Appendix B: DOS-C-Linda User Manual

DOS-C-Linda replaces the # with the array size, here 100. Thus the following two tuples are equivalent:

( "%100d", array )
( "%#d", 100, array )

3.1.2.2 Structures

Structures pose a special problem. DOS-C-Linda needs to access the individual elements in structures. C, however, does not define the physical layout of structures or unions. The layout is defined by the implementation and is often affected by any enabled optimisations such as Word Alignment. Therefore it is up to the programmer to define the internal layout of structures. The C pack pragma is used in Microsoft C to ensure the internal layout matches that specified. In Turbo C++ this pragma is ignored, and thus the Word Alignment option must be turned off. This is done by turning off the Word Alignment option in the Options.Compiler Code Generation menu.

Structures are defined by enclosing the structure’s field definitions in open and closing braces ( { }) as is done in C. A { as the first character of the field type specifier indicates that the following is a structure. The structure itself can be indicated to be either formal or actual. As all fields within the structure have polarity characters the polarity character of the entire structure is only used to indicate the beginning of the structure’s field type specification. Either polarity character is acceptable. Note that in DOS-C-Linda structures must be passed by reference not by value, and hence the & before the variable structure in the following example.

---

6 Word Alignment refers to the alignment of data items to even-byte addresses.

7 The facility to pass structures by value is not implemented in DOS-C-Linda. The reason is that the structure must be retrieved not as an entire structure but as individual fields. This has proven to be very difficult. It will be included in future versions.
The tuple specification for the following definitions:

```c
struct {
    int   integer;
    float pi;
    char * string;
} structure = {42, 3.14f, "This is a string"};
```

would be:

```c
( "%d %.2f %s", &structure )
```

Structures may contain structures as fields. For example, assume the following declaration and definition exist:

```c
typedef struct {
    int     integer;
    long double   long_double;
    void _far *   far_pointer;
} STRUCTURE_TYPE;

struct {
    int     integer;
    STRUCTURE_TYPE structure;
    unsigned char byte;
} structure = {42, {43, 3.14L, NULL}, 0x00};
```

Following is a tuple specification for a tuple containing one single field of type STRUCTURE_TYPE:

```c
( "%c %d %.2f %p %b", &structure )
```
3.1.2.3 Arrays and Structures

Naturally arrays and structures can be combined. For the following declaration and definition:

```c
typedef unsigned char BYTE_TYPE;

struct {
    unsigned short unsigned_short;
    BYTE_TYPE byte_stream[10];
} structure;
```

the tuple specification is:

```c
("%hu %10b", &structure )
```

An array of structures containing one array as defined below:

```c
typedef unsigned char BYTE_TYPE;

struct {
    unsigned short unsigned_short;
    BYTE_TYPE byte_stream[10];
} structure_array[20];
```

is specified as follows:

```c
("%20%hu %10b", structure_array )
```

The use of #’s as array size fields require care. Any #’s within structures are replaced before the structure is parsed. Hence all array sizes need to be in the
argument list before the structure’s address. For the following declaration and definition:

```c
typedef unsigned char BYTE_TYPE;

struct
{
    BYTE_TYPE first_array[10];
    float second_array[20];
    long double third_array[30];
} structure_array[40];
```

the specification is as follows:

```c
( "%1{%1b %#hf %1Lf)",
  40, 10, 20, 30, structure_array )
```

### 3.1.2.4 Pointers to Values

DOS-C-Linda allows pointers to be used as either addresses or as a reference to a value. An address is specified using the type field specifier p or the extended forms Fp and Np. The optional pointer field, *, specifies that the next argument is a pointer to a value. The type of the value is specified with a type field specifier. This allows both values being referenced and addresses, to be expressed.

For the following definition:

```c
int integer = 42;
```

a tuple specification could look like this:

```c
( "%*d", &integer )
```
This notation can be used with any of the basic and complex types. For the following declaration and definitions:

```c
int the_answer = 42;

typedef struct
{
    int * the_answer;
    char * the_question;
} STRUCTURE_TYPE;

STRUCTURE_TYPE structure =
{
    &the_answer,
    "What is the answer?"
};
```

the specification is:

```c
("%{d %s}", &structure)
```

Note that the value of the first tuple field is 42, the value of the integer being pointed to. The tuple is not isomorphic to structure's type of STRUCTURE_TYPE.

Similarly pointers to pointers may be used as shown in the following example:

```c
int integer = 42;
int *pointer = &integer;
int **ppointer = &pointer;
int ***pppointer = &ppointer;
```
The tuple specification for a pointer to a pointer to a pointer to an integer value 42 is:

( "%****d", &pppointer )

Pointers to arrays as defined below:

```c
int array[20];
int *pointer = array;
```

are defined as follows:

( "%*20d", pointer )

Arrays of pointers need the array element defined as a structure. Thus the following data structure:

```c
int *array_of_pointers[20];
```

is defined as:

( "%20{%*d}”, array_of_pointers )

Also note that %s is not the same as %*c. The %s refers to a '0' terminated sequence of characters, whereas the latter refers to a pointer to a single character.

---

8 Arrays of pointers are not directly expressible in DOS-C-Linda Version 1.0. They can however be expressed by defining the array as an array of structures containing a pointer.

Stephan Bettermann
For the following definitions:

```c
char character = 'C';
char *character_pointer = &character;
char *string = "This is string";
```

the correct tuple specification containing all these fields is:

```c
("%c %c %s", character, character_pointer, string )
```

### 3.2 linda_out

The `linda_out` operator places the specified tuple into the tuple space. Below is an example of outing a tuple. Assume the following declaration and definitions exist:

```c
typedef struct {
    int    integer;
    float * float_number;
    char * string;
} STRUCTURE_TYPE;
```

```c
float pi = 3.14;
STRUCTURE_TYPE structure = {42, &pi, "This is a string"};
```

The function call to out the structure defined above as a tuple is:

```c
linda_out( "%d %f %s", &structure );
```

DOS-C-Linda allows the placement of tuples containing formals into the tuple space. Even though no values are needed for formal fields, the corresponding argument...
must still be supplied in the form of an address. NULL may be used as an address. Following is an example of placing a tuple containing two actuals and one formal into the tuple space:

```c
int first = 1, second = 2, third = 3;

linda_out( "%d %d %d", first, &second, first );
```

### 3.3 linda_in

The `linda_in` function will remove a matching tuple from the tuple space and map any actual fields in the tuple onto the specified formal fields in the `linda_in` call. For this DOS-C-Linda needs the addresses of the formal field variables that the matching actual fields are to be mapped onto.

Assume that the following tuple exists in the tuple space. The notation `<no value>` is used to specify the field is formal:

```c
( "%d %d %d", 1, <no value>, 3 )
```

and the following definitions exist:

```c
int first = 0, second = 0, third = 0;
```

The following function call removes the tuple from the tuple space and maps the actual fields onto the specified formal fields:

```c
linda_in( "%d %d %d", &first, second, &third );
```

And thus the values of the variables `first`, `second`, and `third` will be 1, 0, 3 respectively.
Note that DOS-C-Linda does not allocate any memory for pointer fields such as strings. The following example illustrates this. Assume that the following tuple exists in the tuple space:

```
( "%d %*d %s", 42, 43, "The answer is" );
```

and the following definitions exist:

```c
int integer = 0;
int * integer_pointer = NULL;
char * string = NULL;
```

The following function call:

```c
linda_in( "?d ?*d ?*s",
       &integer, &integer_pointer, &string );
```

will cause errors. The reason is that the value 43 will be placed at NULL, as is the case with the third field, where the string "The answer is" will be copied into memory starting at NULL.

As arrays are already passed in by reference & is not used on arrays. Structures are
treated the same as in linda_out, that is they also require a & here. The following example illustrates this:

```c
int array[100];

struct
{
    int integer;
    double a_double;
} structure;

linda_in( "?100d %{d ?f}" , array, &structure );
```
Retrieval of structures containing pointers operates similarly to above. The pointers are dereferenced to find the memory locations where the fields' values are to be stored. Assume the following declaration and definitions exist:

```c
typedef struct
{
    int * the_answer;
    char * the_question;
} STRUCTURE_TYPE;

int the_answer = 42;

STRUCTURE_TYPE structure =
{
    &the_answer,
    "What is the answer?"
};

int retrieved_integer = 0;
char the_question[20];

STRUCTURE_TYPE retrieved_structure =
{
    &retrieved_integer,
    the_question
};

and the following tuple exists in the tuple space:

( "%d", &structure )
```
The following example would retrieve the above tuple into the structure 
retrieved_structure:

```c
linda_in( "?{?*d ?s}", &retrieved_structure );
```

After calling linda_in, retrieved_integer has the value of 42, and 
the_question which is an array of 20 characters has the value of "What is the 
answer?". The pointer field that is part of retrieved_structure is not 
modified. It is only used to determine where to put 42. The field 
retrieved_structure.the_question is treated similarly.

### 3.4 linda_inp

The linda_inp operator is the predicate form of the linda_in function. It also 
finds a matching tuple in the tuple space, removes it, and maps all the matching 
tuple’s actual fields onto the corresponding specified tuple’s formal fields. However 
it does not block if no matching tuple exists. In this case no mapping occurs, and 
the function returns the return code PREDICATE_FAILS. If a matching tuple is 
found then mapping occurs and the function returns SUCCESSFUL_COMPLETION.

### 3.5 linda_rd

The linda_rd operator operates similarly to the linda_in operator. It also finds 
a matching tuple in the tuple space and maps all the matching tuple’s actual fields 
onto the corresponding specified tuple’s formal fields. It however does not remove 
the matching tuple from the tuple space.
3.6 \texttt{linda\_rdp}

The \texttt{linda\_rdp} is the predicate form of the \texttt{linda\_rd} operator. If no matching tuple exists no mapping occurs, and the return code \texttt{PREDICATE\_FAILS} is returned. If there is a matching tuple then \texttt{SUCCESSFUL\_COMPLETION} is returned.

3.7 \texttt{linda\_eval}

Processes can be created using the \texttt{eval} operator or the predicate form \texttt{evalp}. The process to be created must be encased in an executable file, and thus the process name is the system name of the executable file. Assuming that the process \texttt{Function.EXE} exists on the local processor's disk in the current directory, the function call is:

\begin{verbatim}
linda_eval( "Function.EXE" );
\end{verbatim}

The process \texttt{Function.EXE} is then run in parallel on another processor. If no other processor is available the process is stored in the temporary directory until a processor does become available.

Unlike the original Linda paradigm, where process creation occurs by placing \textit{live tuples} into the tuple space DOS-C-Linda does not create \textit{live tuples} as such. In the original paradigm a live tuple may contain any number of fields. Separate processes are created to evaluate each field. Upon completion of all processes the live tuple turns into an ordinary data tuple indistinguishable from all other data tuples. In DOS-C-Linda processes are executable files. Since programs under DOS can only return integer, DOS-C-Linda does not allow processes to return values. Therefore live tuples cannot turn into data tuples as the fields cannot evaluate to anything. As there is no resultant data tuple DOS-C-Linda limits the \texttt{linda\_eval} operator to one single field and thus process.
3.8 linda_evalp

The predicate form of the linda_eval operator is linda_evalp. This function works similar to the linda_eval operator except when no machine is available. Then linda_evalp returns PREDICATE_FAILS and the process is not created.
4 Using DOS-C-Linda

4.1 Writing a DOS-C-Linda Program

When writing programs that use the DOS-C-Linda functions, the DOS-C-Linda library must be included. Two header files are supplied, Linda.h and LindaMSG.h. Linda.h is the header file for the DOS-C-Linda operators, and LindaMSG.h is the header file for LindaMSG.c that can be used to display the DOS-C-Linda operators' return codes.

LindaMSG.h also defines an enumerated data type RETURN_CODES defining the return codes of the DOS-C-Linda functions. These types should be used to check the return codes. Appendix B lists the return codes that may be expected from DOS-C-Linda functions.

Four examples of DOS-C-Linda programs are supplied. The first example is a simple program illustrating how two processes can communicate using DOS-C-Linda. The second example is the classic Dining Philosophers problem as formulated by E. W. Dijkstra. The third example is a matrix multiplication program using DOS-C-Linda. The fourth example is a parallel version of a genetic algorithm, written by Geoff Sutcliffe of Edith Cowan University.

4.2 Running a DOS-C-Linda Program

DOS-C-Linda Programs are run in a Client Shell. A Client Shell is installed by running the program Login.EXE that comes with DOS-C-Linda. Consult section 2.3 for instructions on how to start a Client Shell.
Appendix A - DOS-C-Linda Syntax in Backus Naur Form

Return_Code ::= int

Character ::= <any character in the ASCII character set>

Digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Number ::= Digit(Digit)

String ::= "Character(Character)"

Constant ::= Number | String

Tuple_Field ::= Constant | C_Variable | Address_of_C_Variable

Actuality_Indicator ::= %

Formality_Indicator ::= ?

Polarity ::= Actuality_Indicator | Formality_Indicator

Pointer ::= p

Byte ::= b

Character ::= c

Double ::= f | e | E | g | G

Unsigned_Integer ::= u | o | x | X

Stephan Bettermann
Signed_Integer ::= d | i

Double_Modifier ::= L
Long_Integer_Modifier ::= l
Short_Type_Modifier ::= h
Far_Pointer_Modifier ::= F
Near_Pointer_Modifier ::= N

Modified_Float_Type ::= Double_Modifier Double

Integer_Type ::= 
  Signed_Integer | Unsigned_Integer

Long_Integer_Type ::= 
  Long_Integer_Modifier Integer_Type

Shortable_Type ::= 
  Signed_Integer | Unsigned_Integer | Double

Short_Type ::= Short_Type_Modifier Shortable_Type

Far.Pointer ::= Far_Pointer_Modifier Pointer

Near.Pointer ::= Near_Pointer_Modifier Pointer
Unmodified_Types ::= 
  Pointer | 
  Byte | 
  Character | 
  Double | 
  Unsigned_Integer | 
  Signed_Integer 

Base_Type ::= 
  Modified_Float_Type | 
  Long_Integer_Type | 
  Short_Type | 
  Near_Pointer | 
  Far_Pointer | 
  Unmodified_Types 

Open_Structure ::= { 
  Close_Structure ::= } 

Structure ::= Open_Structure {Field_Format} Close_Structure 

Array_Size_in_Parameter_List ::= # 

Array_Size ::= Array_Size_in_Parameter_List I Number 

Array_Aggregate ::= Base_Type I Structure 

Array ::= Array_Size Array_Aggregate 

Complex_Type ::= Array I Structure
Field_Type_Specifier ::= Base_Type | Complex_Type

Address_of_Modifier ::= \*{\*}

Conversion_Specification ::= 
Polarity [Address_of_Modifier] Field_Type_Specifier

Field_Format ::= 
White_Space | 
Conversion_Specification

Format_String ::= "[Field_Format]"

Data_Tuple_Operator ::= 
  linda_out | 
  linda_in | 
  linda_inp | 
  linda_rd | 
  linda_rdp

Data_Operator ::= 
  Return_Code Data_Tuple_Operator ( Format_String[, Tuple_Field] )

Live_Tuple_Operator ::= 
  linda_eval | 
  linda_evalp

Process_Name ::= "<system file name>"

Process_Operator ::= Return_Code Live_Tuple_Operator ( Process_Name )

Linda_Operator ::= Data_Operator | Process_Operator

Stephan Bettermann
### Appendix B - Return Codes

#### B.1 DOS-C-Linda Function Return Codes

Below is a table of all the return codes as defined in the header file LindaMSG.h and their meaning. A more detailed list of the codes’ explanations, causes, and suggested actions is included as B.3.

<table>
<thead>
<tr>
<th>Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUCCESSFUL_COMPLETION</td>
<td>The DOS-C-Linda function completed successfully.</td>
</tr>
<tr>
<td>PREDICATE_FAILS</td>
<td>The predicate form of the linda operator failed.</td>
</tr>
<tr>
<td>OUT_OF_MEMORY</td>
<td>Out of memory.</td>
</tr>
<tr>
<td>INTERNAL_ERROR</td>
<td>An internal DOS-C-Linda error has been detected.</td>
</tr>
<tr>
<td>NETBIOS_NOT_INSTALLED</td>
<td>NetBIOS has not been installed.</td>
</tr>
<tr>
<td>LOST_COMMUNICATION</td>
<td>Communication has been lost.</td>
</tr>
<tr>
<td>NETWORK_OVERLOAD</td>
<td>The network is overloaded and data has been lost.</td>
</tr>
<tr>
<td>TS_TOO_BUSY</td>
<td>The tuple space is too busy to accept any more requests.</td>
</tr>
<tr>
<td>PARTNER_OUT_OF_RESOURCES</td>
<td>The session partner is out of networking resources.</td>
</tr>
<tr>
<td>NETWORK_NAME_IN_USE</td>
<td>The name is already used by another network station.</td>
</tr>
<tr>
<td>NETWORK_DEAD</td>
<td>The network is dead.</td>
</tr>
<tr>
<td>UNKNOWN_NETWORK_ERROR</td>
<td>An unrecognised NetBIOS error has occurred.</td>
</tr>
<tr>
<td>EXPECTED_FIELD_TYPE_INDICATOR</td>
<td>Invalid character in format string. Expected only ‘%’, ‘?’ or white space character only.</td>
</tr>
<tr>
<td>Code</td>
<td>Meaning</td>
</tr>
<tr>
<td>-------------------------------------------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>FAR_POINTER_APPLIED_WRONG</td>
<td>The type modifier 'F' was applied to a type other than 'p'.</td>
</tr>
<tr>
<td>NEAR_POINTER_APPLIED_WRONG</td>
<td>The type modifier 'N' was applied to a type other than 'p'.</td>
</tr>
<tr>
<td>TYPE_MODIFIER_h_APPLIED_WRONG</td>
<td>The type modifier 'h' was applied to a type other than 'd', 'i', 'o', 'u', 'x', 'X', 'f', 'e', 'E', 'g', or 'G'.</td>
</tr>
<tr>
<td>TYPE_MODIFIER_l_APPLIED_WRONG</td>
<td>The type modifier 'l' was applied to a type other than 'd', 'i', 'o', 'u', 'x', or 'X'.</td>
</tr>
<tr>
<td>TYPE_MODIFIER_L_APPLIED_WRONG</td>
<td>The type modifier 'L' was applied to a type other than 'f', 'e', 'E', 'g', or 'G'.</td>
</tr>
<tr>
<td>INVALID_TUPLE_FIELD_SPECIFIER</td>
<td>There is an invalid tuple field type specifier in the format string.</td>
</tr>
<tr>
<td>NO_MATCHING_CLOSING_BRACE</td>
<td>An aggregate in the format string is missing the closing brace ('}').</td>
</tr>
<tr>
<td>CLOSING_SESSION</td>
<td>Closing Session.</td>
</tr>
<tr>
<td>PERMISSION_GRANTED</td>
<td>Permission granted by the Tuple Server.</td>
</tr>
<tr>
<td>PERMISSION_DENIED</td>
<td>Permission denied by the Tuple Server.</td>
</tr>
<tr>
<td>SHUT_DOWN_LINDA</td>
<td>DOS-C-Linda is being shut down by the Tuple Server operator.</td>
</tr>
<tr>
<td>LINDA_CORRUPTED</td>
<td>DOS-C-Linda is corrupt and there is no point in continuing.</td>
</tr>
<tr>
<td>INTERRUPT_ALREADY_USED</td>
<td>DOS-C-Linda is already installed because interrupt 60h is not NULL.</td>
</tr>
<tr>
<td>LINDA_NOT_INSTALLED</td>
<td>DOS-C-Linda has not yet been installed.</td>
</tr>
<tr>
<td>TUPLE_TOO_BIG</td>
<td>The given tuple is too big to be sent across the network.</td>
</tr>
<tr>
<td>PROCESS_NOT_FOUND</td>
<td>The process to be created cannot be found.</td>
</tr>
</tbody>
</table>
### Appendix B: DOS-C-Linda User Manual

<table>
<thead>
<tr>
<th>Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAN_NOT_OPEN_FILE</td>
<td>The session partner cannot open the file for the process.</td>
</tr>
<tr>
<td>PROCESS_LOST</td>
<td>The process was lost.</td>
</tr>
</tbody>
</table>

Stephan Bettermann
### B.2 DOS-C-Linda Function Return Codes

Below is a table of the DOS-C-Linda return codes and the DOS-C-Linda functions that return them.

<table>
<thead>
<tr>
<th>Code</th>
<th>DOS-C-Linda Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUCCESSFUL_COMPLETION</td>
<td>out x in x inp x rd x rdp x eval x evalp x</td>
</tr>
<tr>
<td>PREDICATE_FAILS</td>
<td></td>
</tr>
<tr>
<td>OUT_OF_MEMORY</td>
<td>x</td>
</tr>
<tr>
<td>INTERNAL_ERROR</td>
<td>x x x x x x x</td>
</tr>
<tr>
<td>NETBIOS_NOT_INSTALLED</td>
<td>x</td>
</tr>
<tr>
<td>LOST_COMMUNICATION</td>
<td>x</td>
</tr>
<tr>
<td>NETWORK_OVERLOAD</td>
<td>x x x x x</td>
</tr>
<tr>
<td>TS_TOO_BUSY</td>
<td>x</td>
</tr>
<tr>
<td>PARTNER_OUT_OF_RESOURCES</td>
<td>x</td>
</tr>
<tr>
<td>NETWORK_NAME_IN_USE</td>
<td>x</td>
</tr>
<tr>
<td>NETWORK_DEAD</td>
<td>x</td>
</tr>
<tr>
<td>UNKNOWN_NETWORK_ERROR</td>
<td>x</td>
</tr>
<tr>
<td>EXPECTED_FIELD_TYPE_INDICATOR</td>
<td>x</td>
</tr>
<tr>
<td>FAR_POINTER_APPLIED_WRONG</td>
<td>x</td>
</tr>
<tr>
<td>NEAR_POINTER_APPLIED_WRONG</td>
<td>x x x x</td>
</tr>
<tr>
<td>TYPE_MODIFIER_h_APPLIED_WRONG</td>
<td>x</td>
</tr>
<tr>
<td>TYPE_MODIFIER_l_APPLIED_WRONG</td>
<td>x</td>
</tr>
<tr>
<td>TYPE_MODIFIER_l2_APPLIED_WRONG</td>
<td>x</td>
</tr>
<tr>
<td>INVALID_TUPLE_FIELDSpecifier</td>
<td>x</td>
</tr>
<tr>
<td>NO_MATCHING_CLOSING_BRACE</td>
<td>x</td>
</tr>
<tr>
<td>LINDA_NOT_INSTALLED</td>
<td>x x x x</td>
</tr>
<tr>
<td>TUPLE_TOO_BIG</td>
<td>x</td>
</tr>
<tr>
<td>PROCESS_NOT_FOUND</td>
<td>x</td>
</tr>
</tbody>
</table>

Stephan Bettermann
B.3 Return Code Explanations

Following is a list of all the error code's explanations, causes, and suggested actions. Where causes and or actions are inappropriate or not valid they are omitted.

**Code SUCCESSFUL_COMPLETION**

**Explanation:** The DOS-C-Linda function completed successfully.

**Cause:** As above.

**Action:** None.

**Code PREDICATE_FAILS**

**Explanation:** The Linda function failed.

**Cause:** The Linda function would have to block until it could succeed. User chose to use predicate form to prevent blocking.

**Action:** None.
Appendix B: DOS-C-Linda User Manual

**Code** OUT_OF_MEMORY

**Explanation:** DOS-C-Linda has run out of memory.

**Cause:** DOS-C-Linda tried to allocate some memory and failed. All memory available has been used.

**Action:** Increase the amount of memory available to DOS-C-Linda. If any memory resident programs are currently loaded, remove them.

---

**Code** INTERNAL_ERROR

**Explanation:** An internal error has been detected. This is a catch all error code.

**Cause:** Something happened internally to DOS-C-Linda that should not have happened.

**Action:** Retry the call. If the situation persists call user support.

---

**Code** NETBIOS_NOT_INSTALLED

**Explanation:** NetBIOS has not been installed.

**Cause:** As above.

**Action:** Install NetBIOS.
Appendix B: DOS-C-Linda User Manual

Code LOST_COMMUNICATION

Explanation: Communication with a session partner has been lost.

Cause: Communication with a session partner failed because the session partner did not respond within a certain amount of time.

Action: Shut DOS-C-Linda down as the Tuple Space is now in an unstable state.

Code NETWORK_OVERLOAD

Explanation: The network is overloaded and data has been lost.

Cause: The network is overloaded. Too much traffic.

Action: Reduce the amount of traffic on the network.

Code TS_TOO_BUSY

Explanation: The Tuple Server is too busy to accept any more requests.

Cause: The Tuple Server cannot process requests as fast as it is getting them.

Action: Reduce the number of Tuple Server requests.
Appendix B: DOS-C-Linda User Manual

Code PARTNER_OUT_OF_RESOURCES

Explanation: The session partner is out of networking resources.

Cause: The session partner has too many network requests to deal with. The partner cannot deal with the network requests as fast as they are arriving.

Action: Reduce the number of networking requests to the session partner.

Code NETWORK_NAME_IN_USE

Explanation: The name is already used by another network station.

Cause: As above.

Action: Choose another name.

Code NETWORK_DEAD

Explanation: The network is dead.

Cause: The network could not be accessed.

Action: Restore the network and DOS-C-Linda.
Code UNKNOWN_NETWORK_ERROR

Explanation: An unrecognised NetBIOS error has occurred.

Cause: A network call returned a code not recognised as an error code.

Action: Shut DOS-C-Linda down.

Code EXPECTED_FIELD_TYPE_INDICATOR

Explanation: There is an invalid character in the format string. Only '%', '?' or white space characters are expected.

Cause: A character other than '%', '?', space or a TAB character was found in the format string.

Action: Remove the offending character.

Code FAR_POINTER_APPLIED_WRONG

Explanation: The type modifier 'F' was applied to a type other than 'p'.

Cause: The type modifier 'F' can only be applied to 'p' to result in a far pointer. 'F' was applied to something other than 'p'.

Action: Apply the type modifier 'F' only to 'p'.

Stephan Bettermann
Code NEAR_POINTER_APPLIED_WRONG,

Explanation: The type modifier 'N' was applied to a type other than 'p'.

Cause: The type modifier 'N' can only be applied to 'p' to result in a near pointer. 'N' was applied to something other than 'p'.

Action: Apply the type modifier 'N' only to 'p'.

Code TYPE_MODIFIER_h_APPLIED_WRONG

Explanation: The type modifier 'h' was applied to a type other than 'd', 'i', 'o', 'u', 'x', 'X', 'f', 'e', 'E', 'g', or 'G'.

Cause: The type modifier 'h' may only be applied to integers, unsigned integers, and floats. It was applied to a type other than those.

Action: Apply the type modifier 'h' only to integers, unsigned integers, and doubles.

Code TYPE_MODIFIER_l_APPLIED_WRONG

Explanation: The type modifier 'l' was applied to a type other than 'd', 'i', 'o', 'u', 'x', or 'X'.

Cause: The type modifier 'l' may only be applied to integers and unsigned integers. It was applied to a type other than those.

Action: Apply the type modifier 'l' only to integers, and unsigned integers.

Code TYPE_MODIFIER_L_APPLIED_WRONG
Explanation: The type modifier 'L' was applied to a type other than 'f', 'e', 'E', 'g', or 'G'.

Cause: The type modifier 'L' may only be applied to doubles. It was applied to a type other than double.

Action: Apply the type modifier 'L' only to doubles.

Code INVALID_TUPLE_FIELD_SPECIFIER

Explanation: There is an invalid tuple field type specifier in the format string.

Cause: One of the following characters was expected:

- d, i - for signed integers
- u, o, x, X - for unsigned integers
- f, e, E, g, G - for doubles
- c - for characters
- b - for bytes (defined as unsigned character)
- p - for pointers
- s - for string, '0' terminated character strings.

DOS-C-Linda found a character other than these in the format string.

Action: Replace the offending character with the proper one.
Appendix B: DOS-C-Linda User Manual

Code NO_MATCHING_CLOSING_BRACE

Explanation: A structure is missing the closing brace (').

Cause: DOS-C-Linda could not find the closing brace of a structure.

Action: Add the missing brace.

Code CLOSING_SESSION

Explanation: The session is being closed.

Cause: The Tuple Server is closing the session.

Action: None.

Code PERMISSION_GRANTED

Explanation: The Tuple Server granted permission to perform the request.

Cause: The Client is not expecting any processes.

Action: None.
Appendix B: DOS-C-Linda User Manual

**Code** PERMISSION_DENIED

**Explanation:** The Tuple Server denied permission to perform the request.

**Cause:** The Client is expecting a process.

**Action:** Retry the request at a later time.

---

**Code** SHUT_DOWN_LINDA

**Explanation:** The Tuple Server operator is shutting down DOS-C-Linda.

**Cause:** As above.

**Action:** None.

---

**Code** LINDA_CORRUPTED

**Explanation:** The Tuple Space is corrupted and there is no point in continuing.

**Cause:** The Tuple Server has detected a lost client, or a message was received from a client no session has been established with.

---

Stephan Bettermann
**Code INTERRUPT_ALREADY_USED**

**Explanation:** DOS-C-Linda is already installed.

**Cause:** The DOS interrupt 60₈ was not NULL. DOS-C-Linda uses interrupt 60₈. If the interrupt is used either DOS-C-Linda has already been installed, or another process is using the interrupt.

**Action:** If DOS-C-Linda has not yet been installed another process is using the interrupt. Remove the invading process. If no other process is using the interrupt reboot the machine.

**Code LINDA_NOT_INSTALLED**

**Explanation:** DOS-C-Linda has not yet been installed.

**Cause:** As above.

**Action:** Install DOS-C-Linda before calling the DOS-C-Linda operator.

**Code TUPLE_TOO_BIG**

**Explanation:** The given tuple is too big to be sent across the network.

**Cause:** The specified tuple was larger than the buffer size of 60k’s.

**Action:** Reduce the size of the specified tuple so the tuple will fit into the 60k buffer.
Appendix B: DOS-C-Linda User Manual

Code PROCESS_NOT_FOUND

**Explanation:** The process to be created cannot be found.

**Cause:** The executable file of the process created by the DOS-C-Linda operators linda_eval or linda_evalp could not be opened for reading. A possible cause might be that the process resides in another path. In that case specify the path.

**Action:** Specify the process name so that the process can be opened for reading.

Code CAN_NOT_OPEN_FILE

**Explanation:** The session partner cannot open a file for the process.

**Cause:** The session partner tried to open a file when receiving the process. Possible causes are:

- no disk space
- the file already exists as a read-only file

**Action:** Depending on the cause

- increase the disk space
- resolve the filename clash

Stephan Bettermann
Code PROCESS_LOST

Explanation: The process was lost.

Cause: The process was lost either during transmission, or because the receiver's disk was full.

Action: Resend the process.
Appendix C - Tuple Server Messages

Following is a list of messages that the Tuple Server can display on its screen, and their causes. Actions are suggested where appropriate.

Message: Unable to execute system command.

Cause: The system command could not be executed because

- it does not exist
- there was not enough memory to execute the command

Another cause for this message can be the command returning an error code.

Action: Try again at a later time.

Message: Granted 'Client Name' permission to log out.

Cause: A client requested permission to log out and permission was granted.

Action: None.

Message: Denied 'Client Name' permission to log out.

Cause: A client requested permission to log out and permission was denied because the Tuple Server has already sent a process.

Action: None.
Message: Granted 'Client Name' permission to execute process.

Cause: A client requested permission to execute a process and permission was granted.

Action: None.

Message: Denied 'Client Name' permission to execute process.

Cause: A client requested permission to execute a process and permission was denied because a process has already been sent to the client for execution.

Action: None.

Message: Starting process on 'Client Name'.

Cause: A process was created by a client and either a machine was available, or a machine has become available.

Action: None.

Message: Client process on 'Client Name' has completed.

Cause: A client process has completed. This may either be a process created by another client process, or a process created by the Client Shell Operator.

Action: None.
Message: Received login request from 'Client Name'.

Cause: A client has logged in.

Action: None.

Message: Closing session with 'Client Name'.

Cause: Session is being closed with a client. Occurs when the Tuple Server is being shut down by the operator, or when a session is explicitly being shut down because an unexpected message has been received.

Action: None.

Message: Unable to close session with 'Client Name'. Just deleted client.

Cause: An attempt was made to close the session with the client. The attempt failed because the client did not respond. The session was simply closed.

Action: None.

Message: DOS-C-Linda is already installed because interrupt 60h is not NULL.

Cause: The DOS interrupt 60h was not NULL. DOS-C-Linda uses interrupt 60h. If the interrupt is used either DOS-C-Linda has already been installed, or another process is using the interrupt.

Action: If DOS-C-Linda has not yet been installed another process is using the interrupt. Remove the invading process. If no other process is using the interrupt, use the utility Kill60h.EXE to reset interrupt 60h, or reboot the machine.

Stephan Bettermann
**Message:** Out of Memory!

**Cause:** DOS-C-Linda tried to allocate some memory and failed. All memory available has been used.

**Action:** Increase the amount of memory available to DOS-C-Linda. If any memory resident programs are currently loaded, remove them.

**Message:** An internal error has been detected.

**Cause:** Something happened internally to DOS-C-Linda that should not have happened. This is a catch all error code.

**Action:** Shut DOS-C-Linda down.

**Message:** NetBIOS has not been installed.

**Cause:** As above.

**Action:** Install NetBIOS.

**Message:** Communication with a session partner has been lost.

**Cause:** Communication with a session partner failed because the session partner did not respond within a certain amount of time.

**Action:** Shut DOS-C-Linda down.
**Message:** The network is overloaded and data has been lost.

**Cause:** The network is overloaded. Too much traffic.

**Action:** Shut DOS-C-Linda down.

**Message:** The session partner is out of networking resources.

**Cause:** The session partner has too many network requests to deal with.

**Action:** Reduce the number of networking requests to the session partner.

**Message:** The name is already used by another network station.

**Cause:** As above.

**Action:** Choose another name.

**Message:** The network is dead.

**Cause:** The network could not be accessed.

**Action:** Shut DOS-C-Linda down, reboot the machine, restore the network and DOS-C-Linda.

**Message:** An unrecognised NetBIOS error has occurred.

**Cause:** A network call returned a code not recognised as an error code.

**Action:** Shut DOS-C-Linda down.
Appendix D - Client Shell Messages

Following is a list of messages that the Client Shell can produce and their meanings. Actions are suggested where appropriate.

**Message:** Could not call the Tuple Server.

**Cause:** A Tuple Server with the given name does not respond to login requests. A Tuple Server of the specified name probably does not exist.

**Action:** Retype the name of the Tuple Server correctly.

**Message:** Starting process ‘Process Name’.

**Cause:** The Tuple Server is using this free Client to run a process created by a linda_eval or linda_evalp call.

**Action:** This Client Shell cannot be used until the process has completed.

**Message:** Could not execute the system command.

**Cause:** The system command could not be executed because

- it does not exist
- there was not enough memory to execute the command

Another cause for this message can be the command returning an error code.

**Action:** Try again at a later time.

Stephan Bettermann
Message: Received defect message from the Tuple Server. Explicitly closing session.

Cause: When a message was received from the Tuple Server NetBIOS reported an error.

Action: Restart DOS-C-Linda and log in.

Message: Permission denied by Tuple Server.

Cause: The Tuple Server has already sent a process to be run on this client.

Action: Retry after the process has completed.

Message: Permission granted by Tuple Server.

Cause: The Tuple Server has not sent any processes to be run on this client.

Action: None.

Message: DOS-C-Linda is already installed because interrupt 60h is not NULL.

Cause: The DOS interrupt 60h was not NULL. DOS-C-Linda uses interrupt 60h. If the interrupt is used either DOS-C-Linda has already been installed, or another process is using the interrupt.

Action: If DOS-C-Linda has not yet been installed another process is using the interrupt. Remove the invading process. If no other process is using the interrupt, use the utility Kill60h.EXE to reset interrupt 60h, or reboot the machine.
Appendix B: DOS-C-Linda User Manual

Message: Out of Memory!

Cause: DOS-C-Linda tried to allocate some memory and failed. All memory available has been used.

Action: Increase the amount of memory available to DOS-C-Linda. If any memory resident programs are currently loaded, remove them.

Message: An internal error has been detected.

Cause: Something happened internally to DOS-C-Linda that should not have happened. This is a catch all error.

Action: Shut DOS-C-Linda down.

Message: NetBIOS has not been installed.

Cause: As above.

Action: Install NetBIOS.

Message: Communication with a session partner has been lost.

Cause: Communication with a session partner failed because the session partner did not respond within a certain amount of time.

Action: Shut DOS-C-Linda down.

Stephan Bettermann
Message: The network is overloaded and data has been lost.

Cause: The network is overloaded. Too much traffic.

Action: Shut DOS-C-Linda down.

Message: The session partner is out of networking resources.

Cause: The session partner has too many network requests to deal with.

Action: Reduce the number of networking requests to the session partner.

Message: The name is already used by another network station.

Cause: As above.

Action: Choose another name.

Message: The network is dead.

Cause: The network could not be accessed.

Action: Shut DOS-C-Linda down, reboot the machine, restore the network and DOS-C-Linda.

Message: An unrecognised NetBIOS error has occurred.

Cause: A network call returned a code not recognised as an error code.

Action: Shut DOS-C-Linda down.