BEHAVIOURAL MODEL DEBUGGING

IN

LINDA

THESIS

Submitted in fulfilment of the
requirements for the Degree of
DOCTOR OF PHILOSOPHY
of Rhodes University

by

DAVID ANDREW SEWRY

January 1994

Abstract

This thesis investigates event-based behavioural model debugging in Linda. A study is presented of the Linda parallel programming paradigm, its amenability to debugging, and a model for debugging Linda programs using Milner's CCS. In support of the construction of expected behaviour models, a Linda program specification language is proposed. A behaviour recognition engine that is based on such specifications is also discussed.

It is shown that Linda's distinctive characteristics make it amenable to debugging without the usual problems associated with parallel debuggers. Furthermore, it is shown that a behavioural model debugger, based on the proposed specification language, effectively exploits the debugging opportunity.

The ideas developed in the thesis are demonstrated in an experimental Modula-2 Linda system.

Keywords: Concurrent programming, debugging aids, Linda, CCS, parallel debugger.

Acknowledgements

My supervisors, Peter Clayton and Peter Wentworth, are to be specially thanked for the contributions they have made to my research. I have valued, and will continue to value, their knowledge of the field of research, advice, criticism, and motivational support. In the true spirit of supervisors, they have given much. They also deserve special mention for reading draft copies of this thesis in the meticulous way they did.

The Parallel Processing Group (PPG) at Rhodes University has provided a stimulating environment in which to work. It serves not only as a willing forum in which to present ideas, but also as a pool of collective wisdom, quite happy to be tapped. Thank you, PPG.

I am also very grateful for the time, opportunity, financial support and plentiful words of encouragement given to me by Prof Pat Terry, the Head of the Department of Computer Science, and general mentor.

My family rightly deserve applause too. They have tolerated my absence, "remoteness", and lack of attention, not to mention bad holidays, without complaint.

I also acknowledge the financial support of the Joint Research Committee, Rhodes University, and the Foundation for Research Development, South Africa.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

The development of software for sequential, uni-processor machines is well-supported by programming environments that include, as a matter of course, program debuggers. Few programmers can claim to write fault-free software the first time round, and sooner or later most programmers find a need to utilise program debuggers to aid the debugging process.

The ANSI/IEEE standard glossary of software engineering terms defines debugging as , "the process of locating, analyzing and correcting suspected faults", where fault is defined to be an accidental condition that causes a program to fail to perform its required function.

Sequential debugging (the process of debugging a sequential program) is essentially a cyclic process. Given a faulty program, the user develops and tests a variety of hypotheses to determine the cause of a program fault. Once a suspicion is confirmed, a repair is attempted and then validated. If the fault is eliminated, all well and good. If it is not eliminated, the user repeats the process. In the development of hypotheses, the user typically suspends program execution at various points in the program, known as breakpoints. At such points the system state (contents of program variables, memory and registers) is examined for anomalies. If no anomalies are detected, the execution is continued to a breakpoint later on in the code. Alternatively, the program is re-executed and suspended at a breakpoint earlier on in the code. This enables the user to home-in on suspected faults. The single most important contributing factor to the success of cyclic debugging is the single thread of control of sequential programs - using the same input data, the exact same program execution path can be faithfully reproduced each time the program is executed.

The development of software for parallel, multi-processor machines lags behind its sequential counterpart, especially in the area of support tools [Pan91b]. Whilst there may be an ever-increasing number of parallel machines, parallel programming paradigms, and languages, support tools like

parallel debuggers (debuggers for parallel programming languages) have not attracted the same degree of attention.

Debugging parallel programs is not an easy task, and is complicated by the following factors:

- Parallel program execution lacks reproducibility. A parallel program is composed of multiple threads of control each executing asynchronously. Processes may communicate with other processes or access shared variables. Where two or more processes attempt to access a shared variable simultaneously (one to read and the other to write), a race condition exists as a result of which either the new or the old value contained in the shared variable is accessed. Should the parallel program be re-executed, even with the same input data, there is no guarantee that, with race conditions, the same path of execution will be followed.

- Where such race conditions exist, the setting of breakpoints or the simple insertion of write statements in the parallel program can alter a critical race and consequently the path of execution of the parallel program. As attempts are made to learn more about the faulty behaviour, the fault may disappear or reappear in another form. This "probe effect" [Gai86] severely hampers the debugging process.

- Parallel program non-determinism that arises from race conditions is difficult to counter. It may depend on CPU load or network traffic, neither of which the user can control.

- The order of occurrence of events in a parallel program is difficult to determine. The lack of a global state [Lam78] complicates and confounds attempts to order the events of concurrently executing processors.

A variety of techniques have been adopted by current parallel debuggers [McD89], each of which contains limitations:

- Cyclic or breakpoint-based parallel debuggers employ traditional cyclic debugging techniques. For each active process, a debugger is spawned. For programs that do exhibit race conditions, the technique is inadequate. Faults can be suppressed whilst the debugger is applied only to manifest themselves again when the debugger is removed.

- Event-based debuggers view parallel programs as parallel sequences of events. Such event sequences are generated, stored in event histories and then analysed. The parallel program must be instrumented directly, or indirectly at the run-time support system level, to generate event information which might then render it subject to the

"probe effect".

- Visual debuggers, based on program events, attempt to provide a display of the flow of control and the distributed data structures associated with the parallel program. The volume of information generated by the parallel program that must be organised by the debugger is overwhelming.

- Static analysis debuggers perform a dataflow analysis on the parallel program without executing the parallel program. The class of error that can be detected is limited to synchronisation errors (deadlock) and data usage errors (uninitialised program variables).

A further technique that builds on the event-based technique is the behavioural model technique. The event-based technique concentrates attention on the generation and analysis of event sequences. In the behavioural model technique, program execution is preceded by a specification of the expected behaviour either of the program as a whole, or of part of the program. The specification is used to construct a model of the program's expected behaviour which is then compared with the actual, run-time behaviour. The behavioural model is constructed in terms of program events, and validated in terms of sequences of actual program events, in which case the term event-based behavioural model technique is applicable. Behavioural model debuggers improve on event-based debuggers: a structured approach to debugging is adopted, and violations between expected and actual behaviour are detected automatically.

Linda[1] [Gel85] is one of many parallel programming paradigms. It is not a programming language on its own but rather a collection of primitives which, when implemented in a standard sequential language, gives rise to a new parallel programming language or Linda dialect. Linda is based on a central content-addressable store, known as tuple space, in which data, known as tuples, are stored and removed by the constituent processes of a parallel program. Typically, a master program places work into tuple space, and slaves remove the work, act on it, and then return the results to tuple space for extraction by the master. Processes always succeed in adding tuples to tuple space but may be blocked on removal of a tuple pending tuple availability. Furthermore, processes are both spatially and temporally decoupled: spatially decoupled in that processes produce tuples without knowledge or care of which process/es may consume them, and temporally decoupled in that communication takes place via tuple space and not directly or explicitly between processes (both processes need not simultaneously engage in the communication). There is no notion of explicit message passing between processes, message passing timeouts, shared variables, synchronisation, or timing considerations of any kind. Processes simply add tuples to, or attempt to remove tuples from, tuple space. Processes may make no assumptions about the time at which a tuple space operation takes place or its duration. The duration of a tuple space operation is defined as non-deterministic. Linda not only exhibits

[1] Linda is a trademark of Scientific Computing Associates.

characteristics that favour successful debugging, but also presents an attractive environment in which to apply behavioural model techniques of debugging: by virtue of the Linda primitives, Linda processes decompose naturally into parallel and sequential components, spatial and temporal decoupling promote a process-oriented approach to debugging, program events can be based on Linda primitives, and the non-deterministic duration of tuple space operations provides an appropriate niche for debugger activity.

This thesis investigates Linda's amenability to debugging using an event-based behavioural model technique of debugging.

1.2 Summary of Results

The investigation shows that Linda programs are amenable to debugging using an event-based behavioural model technique of debugging. The programming paradigm upon which Linda is based lends itself to debugging, and promises some respite in respect of the problems that beset parallel program debugging.

The investigation includes:

- A study of the Linda parallel programming paradigm, its amenability to debugging, and a model for debugging Linda programs using Milner's Calculus of Communicating Systems (CCS) [Mil89].
- The definition of an experimental Linda program specification language.
- The use of the specification language to construct models of expected behaviour of Linda programs.
- The use of an event-based behavioural model technique of debugging to debug Linda programs.

A Linda program is composed of a suite of processes each executing asynchronously but communicating with each other indirectly via tuple space. An individual process is composed of Linda primitives (the parallel component or coordination component) and other host language code (the sequential component or computation component). This separation or layered composition of Linda primitives and other host language code conveniently demarcates the area of interest of a parallel debugger. Whilst the computation component is crucial from an individual process point of view, it is the Linda primitives and their consequent interaction with tuple space that is the province of a Linda debugger. Tuple space represents the core of the Linda system. It sanctions all tuple space activity, and is the sole keeper of tuple space information. Tuple space alone embodies the (parallel) state

space of a Linda program. The primitives and tuple space interactions serve as the atomic events of a Linda program.

To debug the parallel aspects of Linda programs, attention need merely be focussed on Linda primitives and their consequent interaction with tuple space, and tuple space itself.

Tuple space interaction time and the duration of the ensuing interaction is defined as non-deterministic. This implies, that, at any stage in the program's execution, tuple space interaction can be suspended and the state space examined, with impunity. Processes continue their execution until they attempt to execute a Linda primitive at which stage execution is suspended until tuple space interaction is again permitted. Interaction can be suspended at any time (between, during, just after, and just before completion of a tuple space request). Whilst tuple space interaction remains suspended, a consistent state space prevails, and any number of checks and analyses can be performed.

The Linda program specification language proposed by this thesis, based on Milner's CCS, enables the user to specify accurately the actions of a Linda program, in terms of Linda primitives and their interactions with tuple space. It excludes all computation detail, and forces the parallel component of a Linda program into the limelight. The specification phase precedes any coding, and naturally forms an integral part of the program design effort.

Program specifications are used to construct models of the expected behaviour of Linda programs. Models are constructed that represent the expected behaviour of Linda programs (as a collection of the behaviour of individual processes), at various levels of abstraction and from various points of view.

The models of expected behaviour are used as the basis of an event-based behavioural model debugger. The debugger forms an integral part of the Linda system, requires no program instrumentation and is not subject to the "probe effect". The models also serve as a basis for static analysis of the Linda programs.

To test the effectiveness of the technique, an experimental Modula-2 [Wir85] Linda system with debugger has been implemented. It conforms to the event-based behavioural model approach in which Linda programs are specified, "fleshed out" in the Modula-2 Linda dialect, and then executed. At run-time, the expected behaviour is compared with the actual behaviour, and inconsistencies are reported.

Experience shows that:

- the specification phase forces the programmer initially to pay attention to the parallel component of the program and to delay consideration of the sequential component until later,

- the specification language is able to specify the parallel component of Linda programs

at varying levels of specificity and abstraction, and from various points of view,

- debugging is performed in a more structured manner, and on the basis of a formal model of expected behaviour, than is the case when an intuitive, "seat of the pants" approach is adopted,

- fault detection is automated,

- the debugger and the Linda system are well-integrated - after program specification, the system functions, from an external point of view, as a standard Linda system.

1.3 Thesis Organisation

This thesis is organised as follows:

Chapter 2 outlines previous work related to event-based and behavioural model debuggers.

Chapter 3 describes the Linda parallel programming paradigm and an experimental Modula-2 Linda system. A number of example Modula-2 Linda programs can be found in Appendix E, and a discussion of the implementation of the Modula-2 Linda system can be found in Appendix F.

Chapter 4 investigates Linda's amenability to debugging, and presents a model for debugging Linda programs. The investigation makes use of the formal specification language CCS. A brief discussion of the CCS notation is contained in section 4.2, and a glossary of symbols can be found in Appendix A. CCS specifications of the full Linda system and the full Linda system with debugger can be found in Appendix B. An alternative model for debugging Linda programs that approaches the debugging problem from a different angle, but that maintains the same advantages inherent in the first model is also developed. A discussion of it, together with the corresponding CCS specifications, can be found in Appendix C.

Chapter 5 introduces an experimental specification language in which Linda programs can be specified. The syntax of the specification language can be found in Appendix D. A number of example Linda program specifications can be found in Appendix E.

Chapter 6 indicates how the specification language is used to construct models of the expected behaviour of Linda programs. The models are then used as the basis of a behavioural model debugger for Linda. A discussion of the implementation of the Modula-2 Linda system with debugger can be found in Appendix G.

Chapter 7 presents concluding remarks and topics for future research.

Chapter 2

Related Work

The topic of parallel debuggers has, and will continue to be, the focus of attention of many researchers [Pan91a], [Pan93]. The problem has been approached from a variety of angles ranging from the application of traditional sequential debugging techniques in the parallel domain, to event-based debuggers, to behavioural model debuggers, to visual debuggers, and, finally, to debuggers that perform static analyses on parallel programs, each heralding varying degrees of success (good reviews are contained in [McD89] and [Moe92]).

The state of parallel debugger research can be described succintly as follows: The literature contains relatively few reports on traditional, cyclic-based parallel debuggers, for example [Ada86], [Gri88], most of which are set aside for their inability to deal with the multiple threads of control, and the "probe effect". Debuggers that perform a variety of static analyses on parallel programs are increasing in popularity, expecially when the "probe effect" stifles all progress using other techniques, but the unacceptable time and space requirements of the technique retards their widespread adoption (a broad review is contained in [Net91]). Visual debuggers [Utt89] gained popularity by reducing the complexity of the program by the use of graphics but they remain hamstrung by the volume of information that must be processed. Interesting attempts have been made to construct debuggers that operate on automatically parallelised code but give the user the impression that the target program is still the original sequential program [Pin91], [Coh91]. Transformations from parallelised code back to sequential code cause problems. Novel use of sound to debug parallel programs (the execution of the program is translated into a sequence of sounds by associating each program construct with a distinct sound) has also attracted passing interest [Fra91]. Event-based parallel debuggers (see section 2.1) that view parallel programs as parallel sequences of events and analyse them accordingly, capture the lion's share of the parallel debugger market, whilst behavioural model debuggers (see section 2.2) that construct expected behaviour models of the program and compare them to actual behaviour, are, for the most part, still on the periphery.

This chapter describes research efforts related to the implementation of event-based parallel debuggers and behavioural model parallel debuggers, most of which are event-based. The study attempts to

formulate a framework within which behavioural model debuggers can be examined and in terms of which they can be constructed.

(Event-based behavioural model techniques of debugging are applied in sequential debuggers. Although this thesis addresses parallel debuggers, a short review of such sequential debuggers is included for completeness.)

2.1 Event-Based Debuggers

2.1.1 Introduction

Event-based debuggers cover a broad spectrum of parallel debuggers, and are distinguished by their event-specification and recognition capabilities, as well as the actions that are taken as a result of the occurrence of an event. Some debuggers consider every statement in the parallel program an event, whilst others would consider interprocess communication only as an event. Still others provide the user with sophisticated mechanisms by which patterns of events or compound events can be specified and recognised. They are, however, bound by their common view of parallel programs as parallel sequences of events. Once an event has occurred, it can be disregarded, recorded for later analysis or replay, used to support a graphic display of program execution, or used as a reason to suspend program execution and to transfer control to the user. (It is a moot point whether the highly-sophisticated event-based debuggers with advanced pattern recognition and reporting facilities are event-based or event-based behavioural model debuggers, and hence the need for a study of event-based debuggers.)

2.1.2 Issues

McDowell [McD89] raises a number of issues relating to event-based debuggers:

- What constitutes an event?
- How are events defined? Is it necessary to instrument/annotate either the target program source code or run-time support system to generate events?
- Can compound events be defined in terms of primitive events?
- How are events generated? To what extent is the "probe effect" present in the event generation process?
- How are compound events recognised?
- Are any events regarded as unimportant? Is the recognition process capable of ignoring or filtering-out unimportant information?
- What happens to an event after it has been recognised? Is event information recorded and later analysed at the user's convenience, or is it given graphical interpretation and displayed

on a monitor?

2.1.3 Discussion

An event refers to the occurrence of some interesting activity during the execution of a program. The term "interesting activity" has attracted broad definition in the literature. Following the trend established by sequential debuggers, the execution of each and every program statement is an event ([Ols91a], [Rub89], [Zer91]). Calls to system and user-defined procedures and functions are events ([Bat89], [Hse89]). Whenever the status of program entities, for example, procedures, functions, program variables, expressed in terms of and coupled with certain boolean conditionals and boolean guards, known as commands, becomes true, it constitutes an event ([Lop89], [Rub89], [Zer91]). Changes in these entities, without any coupling, [Gai85], and a simple program variable and label trace ([Gol89], [Hse89]) constitute an event. Debuggers based on program annotations (see below) provide access to all, or parts of, the source program in terms of which events may be described ([Bru83], [Luc91], [Ros91]). Process creation, termination and interprocess communication are events ([Bai86], [Car91], [Els89], [Gai85], [Gol89], Hou89], [Hse89], [LeB85a], [Ros91], [Rub89], [Smi85], [Ven89], [Zer91]). For low-level hardware-based debuggers ([Laz86], [Rub89], [Zer91]), bus activity at the memory address and I/O channel level are events. There is broad consensus that activities related to processes are important events that should command priority. There is a measure of support for regarding procedure and function entry/exit as important, whilst most argue that should events be described at a lower level of granularity than that, the number of events that would be generated would be voluminous and too detailed. The nature of the implementation of the debugger (hardware/software) may promote a higher or lower level of granularity. There is a need to isolate a critical set of activities for which primary events are generated that provides an accurate picture of the program's behaviour and that is neither too detailed nor too succint.

Most debuggers provide a core set of primitive events that usually relate to process management and system utilities ([Els89], [Hou89], [Smi85], [Ven89]). [Lin89], [Ven89] only provide access to a fixed set of events. These events are defined implicitly and require no user intervention. User-defined events are normally indicated by source code instrumentation/annotation ([Bru83], [Gai85], [Gol89], [Hse89], [Kil91], [Luc91], [Rub89], [Zer91]). The process could be as simple as the insertion of a special sequence of characters prior to a program statement ([Luc91], [Ros91]) or as complex as event commands that are fed to the debugger prior to or at run-time ([Bai86], [Bat89], [Laz86], [LeB85a], [Lop89], [Ols91a]). Compound events are also defined. Compound events attempt to relate a number of primitive events into a single event. They are usually defined in terms of primitive events, and may subscribe to a hierarchy in which further compound events may be defined in terms of primitive events and newly-defined compound events ([Bai86], [Bat89], [Els89], [Hse89], [Ols91a], [Ros91], [Rub89], [Smi85], [Zer91]). Path rules are also used to define compound events [Bru83]. Path rules contain the specification of the compound event as a generalised path expression - a path expression [Cam74] with predicates, history variables, and path functions.

Event generation is almost exclusively performed by extra code added to the target program or run-time support system. Target programs that have been instrumented to include event definitions are submitted to a preprocessor that converts the definitions into calls to debug routines that generate event messages. These messages contain information regarding the nature of the event and some form of timestamp to facilitate the generation of a partial order of events. Compound events are generated as a result of the occurrence of its constituent primitive events, and also manifest themselves in the form of messages. It has become increasingly popular to restrict the set of events to a few primitive events that are generated by the run-time support system. The target program does then not require any form of manipulation to ready it for debugging. With the possible exception of hardware-based debuggers, all debuggers fall foul of the "probe effect" during event generation. The instrumentation of the target program code and/or the run-time support system alters the execution path of the program and precludes reproducibility. Most systems make a concerted effort to reduce the impact of the intrusion by restricting and improving the efficiency of the event-generating code, and by reducing the event message size. There is a need to investigate debugging methods and programming paradigms in which debugging can be performed that are non-intrusive.

Vast numbers of events are generated during the execution of parallel programs - in strict quantitative terms, it represents many megabytes of information. In event-based debuggers, events are merely routed to some sink (file, console) and the user is left to interpret the data. Most effort is concentrated on the intelligent usage of the data. Whilst merely browsing through the event information using a standard editor may suffice in a sequential environment, the complexity of the information collected in a parallel environment necessitates a far more sophisticated approach. All event information can be viewed on a scrollable display ([Gai85], [Laz86], [Lop89]) but the user is quickly overcome with detail. Intricate windowing systems with folding capabilities that permit movement up and down in the event information hierarchy provide some respite [Ven89]. Window-based execution environments are also used ([Luc91], [Ros91]). The use of graphical interpretations of the underlying event information greatly reduces the complexity of the data. User-defined or system-allocated views are associated with certain data, and acts as an aid to extract deeper meaning from the data. The views can be dynamic animations, scrollable graphs, or static diagrams ([Rub89], [Zer91]). The idea is to reduce the textual information into a 2- or 3-D picture that represents time, process and code. In other systems, for each active process, a demon process ([Bai86], [Smi85]) is fired that collects event information as it is generated, and takes predefined action automatically. Research is also focussed on the recording of event information in a history file for later replay ([Car91], [For89], [Gol89], [LeB87], [Lin89], [Lop89], [Smi85]). Whilst the program executes, all event information is recorded. On program termination, the user enters a replay system based on the recorded data. The system replicates the execution path just followed, independent of all other paths that may be possible, in an environment that usually permits variable execution speed and traditional debugging facilities, for example, breakpoints, program variable inspection.

The introduction of event-based debuggers has aided the development of parallel programs immensely. Such debuggers are capable of detecting most program faults but do encounter difficulty in dealing

with faults that are masked by timing perturbations. Most debuggers attempt to minimise the intrusiveness of their technique. Replay systems attempt to outflank the issue entirely. Where no or very limited "probe effect" is crucial, passive-monitoring of bus and channel activity has reported success. Of primary concern with event-based debuggers is the necessity for the user to infer or deduce from event streams the nature and underlying cause of program faults. The process is not automated.

2.1.4 Details of Event-Based Debuggers

The following are a representative selection of event-based debuggers (some are referred to in the preceding discussion). A short summary of each debugger is provided in which details of their structure and operation are discussed. The intention of this section is to provide the reader with an overall view of a particular debugger, and can be skipped without loss of continuity.

The following system implements an **event-based debugger**:

CBUG [Gai85] is a window-based parallel debugger for C [Ker78]. It targets attention at both low-level (program variables) and high-level (interprocess communication) aspects of the program. Debugging tools include: snapshot dumps, conditional breakpointing, execution tracing, single-stepping, interactive breaks, process creation, and interprocess communication.

The C source code is annotated prior to compilation to install debug hooks (line numbers, labels, jumps to CBUG entry point) that generate event information. At run-time, each process is associated with a window, and a command entry window for debug commands.

The window-based environment provides the user with a "lively" display of the program as a whole, its component processes, and its process interactions. The debug commands are mostly low-level, cannot be structured into a compound command, and are excessively intrusive - "with slow tracing the probe effect is very obvious".

Rather than executing on the same processor as the target program, the following **event-based debuggers** are implemented on **separate hardware**:

DISDEB [Laz86] is an event-based debugger for the Mara system [Mar81]. Parts of DISDEB execute on separate hardware to that on which the program being debugged executes.

The system is composed of a command interpreter and a number of separate software modules that execute on individual Programmable Debugging Aid boards. The programmable boards monitor bus

and I/O channel activity, the occurrence of which is expressed in terms of events and sequences of events as specified by the programmer via the global command interpreter.

At run-time, the occurrence of events is displayed on the operator's console.

DISDEB operates at a particularly low level - memory location addresses and I/O channel activity, and counters. No higher level, abstract view of the program under scrutiny is permitted. It is claimed that the use of additional, dedicated debug hardware precludes the "slowing down [of] the processes being debugged", and that the user can control program execution, "while, in most cases, maintaining the real-time operation of the target system". The lack of debugger intrusiveness is not convincing.

The Makbilan machine consists of two (almost identical) parallel machines, one on which the target program executes, and the other on which the MAD (Monitoring, Animating, Debugging) system [Rub89], [Zer91] executes.

Parallel programs are instrumented to generate notice of so-called interesting events (IE). Events are categorised into user's simple IE, for example, loading and storing into a variable, breakpoints, systems's simple IE, for example, process creation and termination, processor load, and hardware simple IE, for example, a new value of various status registers. Compound events are "boolean predicates interrelating the state of several simple events". The language in which the compound events are defined has the full power of path rules [Bru83]. Events are generated, filtered to ignore unimportant events and to trigger compound events (as described by the user), and finally used to add to and to manipulate a shared database of debug information. Various views are associated with specific events, and the data associated with the event are given multiple graphical interpretations on a graphics display.

Simple and compound events specify possible scenarios of program execution. That certain events (primitive or compound) do not occur, and hence are not displayed graphically, only indicates that the system is not behaving as expected or as specified. There is no indication of where or why. The utilisation of the second processor for the MAD system must reduce the level of debugger intrusiveness but that amount which remains, together with the required program instrumentation, must provoke the "probe effect". At the time of writing the article, the authors indicate that "[since] the Makbilan parallel machine is currently under construction, ... we have yet to witness the full power of non-intrusiveness".

The following **event-based debuggers** include a **replay** component that facilitates varying degrees of post-execution analysis:

SPIDER [Smi85] provides mechanisms for accessing and controlling interprocess activities in a multi-process system. It operates at a high level and specifically precludes program source code and data access.

The SPIDER system is composed of a kernel that implements the multi-process model. At run-time, a variety of separate processes are active: a kernel process, a user command interface process, zero or more programs (one for each process), and zero or more fired debug demon processes. The kernel is also modified to execute the debug demon processes and to manage the collection of process transcriptions.

The command interface gives the user access to a variety of aspects of the system: interprocess objects, interprocess events, programs in processes, and the transcription of interprocess events. The kernel responds to such user commands, and either acknowledges the command and acts on it, or informs the user of the occurrence of a particular event. Groups of commands can be housed together in user-programmed demons. The commands are triggered when a predicate, a boolean expression defined over information available in the kernel about processes and events, becomes true.

Interprocess events can also be recorded for replay. The replay process generates the recorded events in the same way as processes would normally do under normal execution, except that the kernel is not involved. These events can then be manipulated by the user or a demon as if under normal execution.

User interactions and demons interfere with the normal sequence of interprocess events. The set of events is also rigid, and demons provide a limited mechanism for the construction of compound events or higher levels of program abstraction.

PDME [Lop89] is both a sequential and parallel program debugging and measuring environment that supports widely used debugging techniques and performance evaluation.

PDME operates in one of two modes: break mode where the user interacts with PDME via a console to display the state of the target program, and to define an experiment, and run mode where target programs are executed under the control of PDME, subject to an experiment, with results displayed on the console or recorded in a file.

Experiments are defined in terms of commands that accurately describe an event and the action to be taken as a result of the event. An event specification is composed of one or more accesses (a specification of target program entities - procedures, variables, constants, program statements, and an access mode - read, write, start, complete), and a conditional (a boolean expression over target

program variables, literals, constants). A guard (a boolean expression over target program variables, literals, constants) may precede a command, in which case, the command is active (opened) or deactive (closed). At run-time, the guards are evaluated, events are triggered when the execution reaches an access point, and pursuant actions are taken. Results are routed to the console and/or recorded in a file for later analysis.

The complexity of the overall experiment is determined by the complexity of the constituent command specifications. Events are not predefined, nor rigid, and only limited by the ingenuity of the user. A hierarchy of event specification is not permitted. For parallel debugging, the debugging process is conducted at a very low-level, fraught with detail, and close to the source code. It is difficult to see how a high-level, abstract view of process activity would be easily defined, if at all.

Goldszmidt et al [Gol89] provide an event-based debugger for occam [Inm84] programs.

The debugger is composed of two categories of tools: a language specific category in which the program is instrumented to generate event information, and a language independent category in which functions are provided that display, query, replay, and analyse event information.

The user inserts event-generating directives in the occam source code to trigger a program variable and/or label trace, to set breakpoints, and to schedule priorities for the process. (Process creation, termination, and interprocess communication is recorded automatically.) On invocation, the debugger compiles the annotated program, and executes the code. During execution, event information is recorded in a database which is analysed later. Entries in the database are timestamped using an algorithm that reflects causality. Analyses on the event information contained in the database include queries on the state of the computation at a particular point, and temporal assertions to be checked by the assertion checker. Prolog [Clo84] is used to implement the analytic tools and the database.

The debugger successfully separates information gathering (language specific) from information analysis (language independent) but is intrusive and generates vast amounts of information.

Clouds [Lin89] is an event-based debugger constructed as part of the support tools for the clouds system [Das85], [Das87], [LeB85b] that centres on execution replay.

The system supports a fixed set of basic events based on object requests and responses. At run-time, events are caught by the kernel interface, timestamped, packaged into event records, and stored in a database. The database is then used to support execution replay under which the user views execution at various, typically two, levels of detail, for example, inter-object communication and then detailed object execution.

Filtering techniques applied during both event generation and replay reduce the volume of event information. User-defined events are not permitted.

Carver et al [Car91] and Tai and Carver [Tai91] describe a debugger based on deterministic execution debugging. The objective is to generate an instrumented version of a program such that on repeated executions of the program, the same execution path is followed. The user can then debug that specific execution with impunity.

The target program (P) is instrumented to generate event information, based on synchronisations known as synchronisation sequences - "SYN-sequences". The instrumented program (P') is executed, with certain input data (I), and event information is recorded that represents a particular execution path (E). The original program (P) is then instrumented again, this time in the light of the recorded event information, so that, on execution of the second instrumented program (P"), using the input data (I), the same execution path (E) always results.

The technique does not require manipulation of the compiler, run-time system or operating system. The debugger is language-based in that the program itself is instrumented, and it remains a program written in the original language. It does incur a run-time overhead. Its value is found in its ability to reproduce program behaviour. (Instant Replay [LeB87] is a similar replay effort centred on an implementation-based instrumentation scheme. Agora [For89] is similar to Instant Replay.)

The following **event-based debugger** adopts an overt **monitoring** rather than debugging stance:

GRIP [Ven89] is a graphics-based, real-time monitor/debugger for occam programs [Inm84] that makes use of a watchdog to monitor channel activity.

The basic unit of event is an interprocess communication or channel communication. A pre-processor transforms the occam program so that channel run-time behaviour can be mapped to source code channel names. At run-time, the channel watchdog captures channel activity information which is then used to provide a graphical display of: the state of the "sending" and "receiving" processes connected to the channel, the percentage activity of a specified channel in relation to all other channels, channel latency, channel function, global channel activity, and a history reflecting the last few values communicated over the channel.

The user can fold or unfold the screen display for a particular channel, and alter the execution speed of GRIP to permit "a more controlled observance of channel behaviours".

The user has no control over the specification of events, is required to deduce errors from event traces, and could be overcome by the volume of information that is produced by the watchdog. Selection of

a slower GRIP execution speed alters execution sequence - "timing perterbations (sic) are compensated for", although the compensating mechanisms are not discussed.

2.2 Behavioural Model Debuggers

2.2.1 Introduction

Behavioural model debuggers build models of the expected behaviour of parallel programs prior to program execution, and then check that the actual program behaviour, captured at run-time, matches the expected behaviour. The program can then be said to be faulty or fault-free with respect to the expected behavioural model. The automatic detection of program faults is a distinguishing characteristic of behavioural model debuggers. Behavioural model debuggers are usually accompanied by a specification language or specification mechanism in which the expected behaviour of the program is specified.

2.2.2 Issues

A study of behavioural model debuggers reveals a number of basic issues:

- How is the model of the expected behaviour of the target program specified? Is a special-purpose language used that is dependent on the target programming language, or is a generic specification language used that is independent of the target programming language? Are the specifications expressed in terms of underlying program events? Is a separate file used to contain the specifications or are specifications merely annotations to the target program?
- What is the relationship, if any, between the syntactic and semantic model of the specification language and that of the target programming language?
- Can the specifications be used for anything else other than to check program behaviour? Can they be used to support or be part of the program design phase?
- Is a model constructed of the expected behaviour of the program as a whole or only selected parts of the program that the user deems important?
- Does the specification process support the construction of higher levels of event abstraction?
- What internal model or formalism is used to represent the behavioural model?
- Other than operational semantics, does the model embody any further information regarding the target program?
- Does the debugger execute in its own space or that of the target program? Does the model have access to the state-space or data-space or a subset of these spaces of the target program?
- What mechanisms are used to recognise and match the actual and expected program behaviour? To what extent is the "probe effect" present in the recognition process?
- What information does the recognition process provide to the user during the debugging

process?

2.2.3 Discussion

Expected behaviour is specified using program annotations or a specification langauge. Most specification languages are special-purpose and target a particular programming language and paradigm (Bat89], [Bai86], [Bru83], [Kil91], [LeB85a], [Ols91a]). Whilst they may share common aspects, they remain very distinct. A close relationship exists between the semantic model of the debugger and that of the target programming language and paradigm. For systems that specify behaviour by way of program annotations ([Luc91], [Ros91]), the specifications are frequently provided in the target programming language itself. Where the underlying semantic constructs for parallelism are distinct, applicability of a specification language to a wide diversity of environments is unlikely.

Specification languages are subjected to the same level of rigour as any normal programming lanaguge, that is, formal syntactic and semantic definition.

Specifications are expressed in terms of underlying events that are either primitive or user-defined. Where specifications are defined by way of program annotations, any number of activities and program entities can be included as the basis of events.

Specifications are stored in a separate file from which either program annotations are automatically made, or entirely separate debug processes or demons are generated that embody the specifications. Programs are also annotated directly with specifications. The submission of specifications in a separate file to that of the target program is gaining in acceptability.

Most systems use the specifications for debugging alone. Some systems that require the user to annotate the target program manually use the specifications to define module interfaces, even before any code is developed [Luc91]. It seems reasonable to use the specifications early on in the software design and development phase. Authors refer to the usefulness of such ideas but go no further.

It is unusual that debuggers require that the entire program be specified. Most require that the user specify that part of the program that is believed to be at fault, and then to whatever degree of detail. Multiple specifications of the same piece of code, each from different semantic angles, are sometimes permitted. In some debuggers, a detailed approach is required whereby all processes are specified individually, and in full [Bai86]. Researchers argue that programmers ought to be able to concentrate attention on suspect code rather than on the entire program. Full specification is also regarded as onerous but this depends heavily on what is specified.

A variety of mechanisms are used to specify events, both primitive and compound: regular expressions, constrained expressions [Hou89], extended regular expressions ([Bat89], [Els89], [Hou89], [Kil91]),

path expressions, predicate path expressions, generalised path expressions [Bru83], data path expressions [Hse89], and path rules (a generalised path expression and a path action) ([Bru83], [Rub89], [Zer91]). The progression of formalisms represents an increased desire to capture ever more detailed events, to maintain greater past information upon which to base decisions about current event validity, and to take action pursuant to the occurrence of an event. Such specification formalisms find realisation in finite-state automata ([Bru83], [Els89]), shuffle automata [Bat89], predecessor automata [Hse89], graphs and trees (where leaf nodes represent primitive events and internal nodes represent compound events) [Ols91a].

Early debuggers tend to operate solely on the current event and information pertinent to it alone. The event is verified, and the next event awaited. For more efficient debugging, it is apparent that more information is required about the target program's execution than just the current event. History variables, counts of specific events, and previous events are now maintained (and hence, the increased complexity of internal model formalism) ([Bai86], [Hse89]). Some debuggers permit access to the target program's state space or data space ([Bai86], [Bru83], [Els89]) but most maintain a subset of these spaces from which information is gleaned.

Debuggers that are implemented as separate processes ([Bai86], [Ols91a]) execute in their own state space. Debuggers that are implemented in the form of program annotations execute in the same state space as the target program [Luc91] and, of course, have access to those spaces (program annotations, scattered throughout the target program, are converted into target program source code and compiled together with the target program). The latter technique runs the risk of the target program corrupting the debugging code. Separate state spaces are preferable but do provide some hindrance, if the state space of the target program must be navigated.

The "probe effect" is as present in behavioural model debuggers as it is in event-based debuggers. Events are generated, compound events constructed/recognised, and then tested against behavioural models, all of which require an element of time. For systems that suspend process execution whilst an event is tested, the "probe effect" is more manifest.

The information that is presented to the user during execution is dependent on the nature of the specifications. They may be specifications of behaviour that may not violated (and if they are, a message is generated) ([Bat89], [Els89], Hou89], [LeB85a], [Ols91a]), or specifications of events that may occur (and that on occurrence cause a message to be generated) ([Bai86], [Bru83], Hse89], [Luc91], [Ros91]). Where selected sections of the target program are specified as events, the debugger may generate no information - no primitive or compound event is recognised. The debugger may inform the user of the occurrence of an event or execute certain commands without user intervention [Bru83]. For debuggers that require a full specification of the target program, each event is checked against the behavioural model, and mismatches may result in target process suspension or an alert to the user [Bai86]. Some debuggers provide the user with partial match information [Bat89]. Essentially, if the specification is partial, the debugger informs the user directly or indirectly, via some

predefined action, of the occurrence of some event. If the specification is in full, each event is checked and mismatches are reported. It is possible, that should the specification be in full and certain state information be maintained that some form of analysis be performed on the internal model, and answers to questions of deadlock be provided.

The primary contribution of behavioural model debuggers is their formal approach to debugging, the possible use of the behavioural models as an aid to software design and development, and the automatic detection of violations between expected and actual behaviour.

2.2.4 Details of Behavioural Model Debuggers

The following are a representative selection of behavioural model debuggers (some are referred to in the preceding discussion). A short summary of each debugger is provided in which details of their structure and operation are discussed. The intention of this section is to provide the reader with an overall view of a particular debugger, and can be skipped without loss of continuity.

2.2.4.1 Sequential

The following debuggers are **event-based behavioural model debuggers** for **sequential** environments:

Dalek [Ols91a], [Ols91b] is an event-based behavioural model debugger for sequential C programs, although the authors express an intention to extend the model of debugging to concurrent programs. The authors state that current sequential debuggers have limited control over actions taken at breakpoints, and have no mechanisms by which "several logically related breakpoints [can be correlated] into a single, more abstract occurrence". Dalek attempts to overcome these problems.

The user is provided with a general-purpose debugging language, based on the GDB debugger language [Sta89], in which events are defined, raised, recognised, and combined. High-level events are defined in terms of primitive events, which in turn are defined to capture the occurrence of any interesting activity in the code's execution. A directed graph is used to store event specifications, in which leaf nodes represent primitive events, and nodes "higher-up" in the graph (interior nodes) represent high-level events.

At run-time, Dalek executes as a separate process to that of the target program being debugged. Control is transferred to the Dalek process at user-specified breakpoints in the target program's execution at which time events are raised which might be used to raise higher-level events.

[San93] describes extensions to this work. Instead of using in-line code to perform the checks, separate tasks are spawned to check the target program code.

Analyzer [Luc91] is a specification-based debugger for Anna (annotated Ada) [Luc87].

Using the Anna specification language, Ada [Geh84] programs are annotated to include specifications, in terms of program requirements, of the expected behaviour of the program. The Analyzer converts the specifications into code which, together with the program code, is then compiled in the usual way.

At run-time, the target program is executed under the control of the debugger to which control is transferred whenever the program violates a specification.

It is not a requirement of the system that the entire program is annotated - the user uses annotations to home-in on faulty code. The automatic detection of violations of specifications, the structured debugging process, and its usefulness in earlier stages of program development (as formal specifications of module interfaces) are attractive components.

2.2.4.2 Parallel

The following are typical **event-based behavioural model debuggers** for **parallel** environments:

Baiardi et al [Bai83b], [Bai86] describe an event-based behavioural model debugger for a CSP-based [Hoa78] language ECSP [Bai81],[Bai83a].

The behaviour of the processes constituting the parallel program are each specified in terms of atomic event specifications that describe either channel communication, process termination, or the connection/disconnection of one process to/from a set of input ports of another process. An in-house specification language uses these events to define one or more partial orders of events for each process. The partial order of events represents the allowed sequences of process interactions for a particular process. For each process, a debugger process (implemented as an ECSP process) is launched that embodies these specifications. Atomic event generation is performed by an instrumented run-time support system that would otherwise normally handle the actions upon which the atomic events are based. At run-time, the debugger processes are informed of events, and these are then checked against the expected behaviour. Whilst an event is being checked, the process that generated the event is suspended. If the event is permitted, the process is resumed, otherwise control is returned to the user at which stage, the process can be terminated, the whole program can be aborted, or the values of certain variables can be changed and the process resumed at a later point.

The model of the expected behaviour of the program includes aspects of both a denotational and axiomatic model: denotational in that it describes process state and possible sequences of events that may occur, for example, communication with this process then that process, and so on, and axiomatic in that it is expressed in terms of events and the number of occurrences of each event, for example, a particular event must occur after a number of occurrences of some other event.

A special delay operator in the specification language ensures that whilst a process is suspended on a check, all other processes that are involved in a non-deterministic event choice together with the suspended process are delayed.

The debugger is significant in that the semantic model of the debugger follows the ECSP language closely, that the behaviour of individual processes is specified and not the program as a whole, and that the specifications form a hierarchy from event specifications to behaviour specifications. However, the instrumented run-time support system, process delay and the detailed specifications that are required mitigate against the debugger.

Bruegge and Hibbard [Bru83] discusses the use of path rules to specify the expected behaviour of sequential and parallel programs and to define the action to be taken when expected and actual behaviour differ. (The paper addresses the use of path rules as a means of debugging primarily sequential programs but concludes with a discussion of its applicability to parallel programs. Its importance lies more in the parallel than the sequential domain and so is included here.)

Path rules consist of an event recognition part (a generalised path expression) and an path action (a function that is called on match/mismatch of an event). Processes are instrumented to include such path rules, and the component path functions and path actions are executed as the computation proceeds.

Path expressions are a convenient formalism for the specification of the execution path, control flow, of a parallel program. The development of generalised path expressions can be traced back to basic path expressions. Basic path expressions are regular expressions, with operands known as path functions, that can be recognised by finite-state automata. Andler [And79] developed predicate path expressions by extending basic path expressions to include history variables, TERM and ACT, for descibing the history of computation, and predicates. Generalised path expressions extend predicate path expressions by enhancing the nature of the path function, by permitting the inclusion of any program variable in a predicate, and by providing a pre-defined path function, to refer to the change in state of a variable.

Bruegge notes that path rules automatically detect behavioural violations (the user need not make any deductions from traces), reduce the volume of output data necessary to monitor the system, and permit observation of the system at a level of abstraction in which the user is currently thinking. The use of path expressions as the basis of a parallel debugger is discussed briefly but the success or possible success of an implementation is not specified. It is not understood how the level of intrusion would not be significant.

Bates describes work on behavioural abstraction [Bat83a] and event-based behavioural abstraction [Bat83b], [Bat89] that forms the basis of an event-based behavioural model debugger.

Characteristic atomic program behaviours are described in the Event Definition Language (EDL) [Bat82], [Bat87]. Each description is divided into three sections: the event class (primitive or clustered/compound), for compound events, filtering conditions that describe relationships between event classes constituting the compound event, and a set of expressions that bind values to event instance attributes. Compound events are described as a combination of primitive events using temporal and relational constraints. Each event has a time and location attribute associated with it. These descriptions constitute the model of the expected behaviour of the program.

The run-time support system is instrumented to generate atomic events, for example, task creation, open-file. The events are routed to an event recogniser that matches these events/sequence of events with the primitive or compound event specifications. On a successful match, an event instance record is generated and is transmitted to the user for notification. Not all atomic events are of interest to primitive and compound event specifications, and unimportant atomic events are filtered from the event stream.

The need to filter event information to gather only important events and the need to recognise patterns of events (for compound events) rules out the use of a finite-state automaton behaviour recognition formalism, and necessitates the use of a shuffle-automaton (a shuffle-automaton is a finite-state automaton-like formalism that fires on transitions based on groups of symbols rather than on single symbols).

The generation of atomic event information and its distribution to an abstraction node of the toolset does effect the order of process execution. Given that a program must be debugged, the user is not required to provide models of its total behaviour but just models of those aspects that strike the user as important for the purpose of debugging. Whilst this reduces the volume of specification that is required, the user may never chance upon the faulty section of the program. Indeed, "when user models match actual system behaviour, the user has attained an understanding of some aspect of the system and might need to shift viewpoints or focus more closely on suspected components". The degree to which the program will be debugged seems unnecessarily dependent on the programmer's ability to specify ever more complicated compound events. A measure of relief is afforded by the event recogniser that produces "information regarding the goodness of fit for models yet to be satisfied", but the nature of this information is not specified.

Hseush and Kaiser [Hse89] describe a debugger that is based both on a data flow and a control flow model, and that employs data path expressions as the debugging formalism. In data-oriented debugging, problem or fault behaviour and program behaviour are described in terms of data, whilst in control-oriented debugging, the behaviour or abstract entities are described in terms of control

constructs which are bound to the syntax and semantics of the base language.

Data path expressions extend generalised path expressions [Bru83] by permitting data events to appear as path functions. An event is an assertion that a particular program state, point in the execution of the program, or activity has been reached. Three categories of events are presented in data path expressions: a data event, for example, an event is generated when a program variable contains a particular value or is in a particular range of values, a control event, for example, an event is generated when a particular procedure, function, or group of named statements is called, and a message event, for example, an event is generated when a message is sent or received. The authors envisage the construction of a partial ordering graph [Lam78] to represent the execution of the program, and against which the data path expressions will be checked.

In [Hse90], their work is continued, and they describe a hierarchy of data path expressions using a hierarchy of extended Petri net models [Pet81], and the use of predecessor automata as an implementation vehicle for safe data path expressions (data path expressions that express general bounded parallelism). A predecessor automaton is a finite-state automaton that fires on transitions that are based both on the current event and predecessor events. It can thus recognise or generate partial ordering graphs and strings.

The user specifies expected program behaviour in the form of data path expressions, and these are then translated into a predecessor automaton. Hardware and software is instrumented to generate primitive events, the occurrence of which are channelled to a data path expression recogniser. The recogniser filters unimportant events, and regulates the stream of events to preserve partial ordering, before using them as the basis for a transition in the predecessor automaton. At run-time, the system indicates whether the data path expression was matched or not.

[Pon91] is a more recent, although not much changed, account of the research.

Rosenblum [Ros91] describes the use of the Task Sequencing Language (TSL) [Hel85] to specify, by way of program annotations, the correct behaviour of a concurrent program in terms of patterns of events [Bat83b].

The TSL compiler transforms the annotations into calls to appropriate run-time procedures that organise event information. Basic, user-defined and compound events are permitted.

At run-time, patterns of events are compared against the specification. If a specification violation occurs, the program is suspended and control is transferred to the user interface. The interface provides the functionality of a traditional symbolic debugger, for example, single-stepping and breakpointing, but at a higher level of abstraction.

Execution **replay** is emphasised in the following **event-based behavioural model debuggers**:

Radar [LeB85a] is a monitor/debugger that provides a post-mortem view of the execution of Pronet [Mac82] programs. Pronet is significant in that it is composed of two sub-languages: a network specification language (NETSLA) that is used to initiate process execution and to control the communications environment, and a process description language (ALSTEN).

Primitive events, for example, process creation, termination, interprocess communication, and user-defined events are permitted. User-defined events are specified in the network specification, and include details of actions that must be carried out on the occurrence of events.

At run-time, event information is generated by special communication libraries and recorded, together with an event number (to provide a partial order of events), and replayed later using a graphical display.

The facilities that are provided for user-defined events are limited, and no facility is provided to define higher level abstract events. That event information is logged to a file, does reduce the level of debugger intrusiveness but not completely. The authors argue that since message receipt is non-deterministic, suspending one process out of a group from which messages may be received (whilst data is logged for it) does not preclude receipt of messages from the others in the group that are not suspended.

Belvedere [Hou89] is a pattern-oriented, trace-based, post-mortem debugger that attempts to match communication patterns with user-defined events.

Primitive events include GETs and PUTs of messages, and process and channel creation and deletion. Patterns are described as abstract events using constrained expressions [Bat89]. At run-time, as primitive and abstract events are identified, they are placed in a relational database where they become the target of queries, and the basis for the animation of process interactions.

Whilst the graphical component provides an improved understanding of process interaction, the authors cite a number of problems: the lack of generality of the automatic animation displays, the modelling and query language, and the behavioural model, and the limited system feedback provided for missing or extraneous behaviour.

The Amoeba debugger [Els89] is an event-based debugger that, in addition to the provision of facilities for event specification and capture, also provides many facilities that are found in sequential debuggers, for example, memory inspection and modification, and breakpoints.

Primitive events are generated by an annotated run-time support system each time a process invokes a system service. The user provides the debugger with patterns (sequences of events), filters (mechanisms by which unimportant events are ignored), and recognisers (sequences of events are specified making use of extended regular expressions that are then implemented as finite-state automata for recognition purposes) directly or via a file. At run-time, events are generated by the system, fed into the event stream where they are checked to determine whether they match any of the patterns. On recognition of an event sequence, a list of user-specified commands is executed. Commands that insert a new event into the stream can be executed at this stage. This affords the user the opportunity of specifiying hierarchies of event abstractions based on the occurrence of sub-events.

The breakpoint facility and the annotation of the run-time support system disturbs the execution sequence of the program. The debugger resides in the same dataspace as the process being debugged (the debugger is implemented as a special library), and runs the risk of its data structures being corrupted by the debugged process. The mechanism by which higher levels of event abstraction is implemented, is unnatural.

The following systems **build models of expected program behaviour** not necessarily for debugging purposes, but for **monitoring, performance monitoring,** or **static analysis:**

ChaosMON [Kil91] is a system designed to capture and display program performance information. Whilst it is not a debugger, it incorporates a number of issues pertinent to the specification and construction of behavioural models of programs. Essentially, the user specifies models of the application program and its performance, and views performance accordingly.

An attribute language is used to create a description of the high-level application program. Model components are mapped to program components (processes, objects), and attributes of components to program variable expressions. A view language, based on Bates' work [Bat89], is used to create performance models, specified as performance views. Primitive views are mapped to attributes of the high-level program model, and abstract views are specified in terms of primitive views. These views are used to introduce probes into the target program which generate information on which the graphic displays of the program's performance are based.

Formal, operational models of the behaviour of target programs are the basis of an event-trace monitoring system described by [Dor92]. An abstract model of the behaviour of the target system is constructed prior to system execution. It, together with event information that is produced by the system, is then used to monitor the system.

Graph-grammars with grouped rules form the basis of the model formalism. In the formalism, transitions are related to event instances, transition rules are related to an event, which is comprised

of a collection of event instances, and a sequence of transitions is related to an execution of the system. The model also supports the addition of attributes and attribute evaluation rules to measure system performance. The authors note that "the abstract behaviour of the system is described in terms of abstract system views and transition rules, [whilst] the concrete behaviour of the running system is formulated in terms of states and transitions".

Once the model is constructed, it is used to manage the collection of information that is generated by an instrumented system. As events occur, they each represent a step of the monitored system which can be interpreted by performing an operation on the system's model.

The system reduces the amount of information that must be collected in the event-trace by recording events and not the event-instances that comprise the event.

TOTAL [Sha90] is an Ada program static-analysis toolkit in which Ada programs are expressed as Petri nets and analysed accordingly.

The toolkit is comprised of two subsystems: a translator subsystem that transforms Ada programs into different representations, and an information display subsystem that provides the interface between the user and the query analysis system. The translator subsystem, and the intermediate program forms are of particular interest.

Although the ultimate goal of the subsystem is to produce a Petri net representation of the original program, an intermediate form is used in which the program is expressed in the Ada Tasking Language (ATL). The ATL is a formalism in which all aspects of an Ada program are specified that effect its tasking behaviour. In the translation process, a special utility takes the original program and filters out all program statements that do not effect its tasking behaviour, and expresses the remainder in ATL. This representation accurately reflects the behaviour of the parallel component of the original program. At this stage, rather than use it as the basis for a run-time debugger, it is used as the basis for static analysis.

2.3 Problem In Context

A number of event-based and behavioural model debuggers are available, each of which approach the problem from different perspectives. Whilst none of the debuggers solve the debugging problems entirely, the individual techniques employed make valuable contributions to debugger technology.

Problems that continue to frustrate implementors include: the "probe effect" (in all its guises), simplicity of event specification (both primitive, compound, and at various levels of abstraction), simplicity and sufficiency of behaviour specification, and smooth debugger/target system integration.

Linda dialects exist for a variety of languages but studies of the underlying Linda paradigm, its amenability to debugging, and, indeed, Linda debuggers are scarce.

DR.PAL (Distributed Real-time Program Animation Language) [Bus89] is an event-based viewer/debugger that enables the user to develop, debug and view Linda programs. The user focusses attention on specific aspects or "significant occurrences" of the algorithm by setting graphics flags that are embedded within program comments. The flags trigger views or "graphical windows", generated by code written in DR.PAL, that animate aspects of the algorithm. DR.PAL commands control speed of program execution, view selection, and the variables that must be traced. The authors state that program development effort and time are reduced, and that program data structures are more readily understood. "Significant occurrences" are not detailed, nor is the nature of the DR.PAL language.

TupleScope [Ber90a] is an event-based monitor and debugger for Linda programs developed at Yale University. It is graphics-based and provides the user with a clear view of tuple space activity. Tuple space operations serve as events and potential breakpoints. At such breakpoints, the entire Linda program is suspended. The user is also provided with the TupleScope Debugging Language to aid the debugging process. Commands are constructed in the language that test tuple fields, tuple space operations, and process numbers. The commands are formulated as boolean conditionals followed by actions that are executed when the conditionals are true. Actions include Linda program suspension, activation of a display filter, display colour change, and storage of the current contents of tuple space to a file. A post-mortem replay facility is also provided. Whilst TupleScope ranks with the more advanced event-based debuggers, the debugging process is not automated, suffers from a volume of detail problem, and although the TupleScope Debugging Language does provide a higher level of event abstraction mechanism, it is insufficiently powerful to describe complex behaviours.

This thesis addresses these particular (Linda) concerns and those described in the preceding sections, in the context of an event-based behavioural model debugger for Linda.

- Unlike the hardware-based debuggers of ([Laz86],[Rub89],[Zer91]), most debuggers are subject to the "probe effect". A study is made of the Linda parallel programming paradigm to determine its amenability to debugging, and the extent to which the implementation of the proposed debugger is susceptible to the "probe effect".
- A limited, controllable, primitive event set is considered.
- Like ([Bai86],[Bat89],[Bru83]), a separate specification language is defined.
- Specification of expected behaviour by way of program annotation ([Luc91],[Ros91]) is not deemed desirable (the code and data space of the debugger and the target program is usually mixed), and a separate file is used to house the specifications.
- Despite some opposition to full specification of all participating processes [McD89] (undue burden on programmers, volume of specification), Linda is receptive to such a strategy without the reported disadvantages.
- A "probe-effect"-free event generation and collection mechanism is explored.

- A simplistic behaviour recognition formalism is sought. The automata described by ([Bat89],[Bru83],[Hse89]) serve as starting points.
- A more simplified history file and replay system than [Tai91] is investigated.
- The debugger both utilises and produces information. Greater use of all this information, in the software development cycle, is explored.
- The success of the debugger is partly based on transparency of the integration of the debugger with the base Linda system. This issue is considered at all times.

Chapter 3

Linda

In this chapter the Linda parallel programming paradigm is discussed. The Linda model, including the basic primitives, certain implementation issues, as well as programming imperatives are presented. An experimental Linda system is also discussed that represents an implementation of a Modula-2 Linda dialect.

3.1 Linda

3.1.1 A Brief History

Linda is a parallel programming paradigm that, in recent years, has aroused great interest in the research community, not least of which for its pure simplicity.

First described in [Gel85], and then in [Ahu86], [Gel88], [Car89a], it is based on the notion of a central content-addressable store, known as tuple space, and a handful of primitives that manipulate tuple space. Linda programs are composed of a number of processes that use these primitives to add and to remove information from tuple space. Linda comes to life when tuple space and the primitives are implemented in some standard sequential programming language, and in so doing, gives rise to a new Linda dialect. Distinctive to the paradigm is the intermediary role played by tuple space - Linda processes communicate indirectly with each other via tuple space. As a result, processes are temporally and spatially decoupled.

The Linda model has been used to implement efficient solutions for parallel searches for DNA sequences, the travelling salesman problem, matrix applications, and database applications [Car88], [Car90b].

The basic Linda paradigm has not escaped modification, and a number of variants have appeared in

the literature: Polymorphic Linda [But91a], Persistent Linda [And91], Multiple Tuple Space Linda [Cia91], and Kernel Linda [Lel90].

Today, numerous Linda dialects exist across a broad spectrum of host languages:

C	[Cla92],[Bjo87],[Bus89],[She93]
Concurrent Smalltalk	[Mat88]
Fortran	[Sci93]
Lisp	[Yue90]
Modula-2	[Bor88]
ProSet	[Has91]
Prolog	[Mac90]
Scheme	[Dah90]

to name but a few, and implementation platforms: transputers (a popular platform), Sun, HP/Apollo, Sequent, Encore, Cray, Convex, Intel iPSC, nCube, NeXT, Mac and i860 computer servers. Commercially, Linda and Network Linda are available from Scientific Computing Associates, and Tuplex is available from Torque Systems Inc.

Apart from new Linda dialects and implementations of Linda on different platforms, current research targets more efficient implementations (tuple space organisation, tuple match strategies) [Car90a], [Car93], and programming environments, for example, the Linda Program Builder [Ahm91a], [Ahm91b].

3.1.2 The Basic Paradigm

The central components of Linda are tuple space and the Linda primitives.

Tuple space is a content-addressable store in which tuples are added and removed by Linda processes. A bag, in which multiple copies of the same tuple may exist, implements tuple space.

Tuples are ordered lists of typed fields or elements, for example:

(3, 'hello', 7.2)

is a tuple of 3 elements: an integer, a string, and a float.

Tuples are added to tuple space using the **out** primitive, for example:

out(3, 'hello', 7.2)

The process that executes the **out** primitive does not block - it adds the tuple to tuple space and then continues to execute.

Tuples are removed from tuple space using the **in** or **read** primitives. They both specify a template that is used to describe the tuple that must be removed, for example:

```
in(3, 'hello', 7.2)
read(3, 'hello', 7.2)
```

Fields in the template must be matched identically but formal parameters (known as "formal" tuple elements) may be used, for example:

```
in(?I, 'hello', ?F)
read(?I, 'hello', ?F)
```

Here, the range of possible matching values is broadened to any integer for **?I** (instead of just **3**), and any float for **?F** (instead of just **7.2**). On tuple match, the corresponding integer and float are assigned to **I** and **F** respectively. In the case of **in**, the matching tuple is then removed from tuple space, and in the case of **read**, a copy of the matching tuple is removed from tuple space. If more than one tuple matches the template, one is chosen non-deterministically. This non-deterministic tuple selection is an important trademark of the Linda paradigm. If no tuple matches the template, the issuing process is blocked pending the arrival of a suitable matching tuple[1].

Predicate forms of **in** and **read** are also provided:

```
inp(3, 'hello', 7.2)
readp(3, 'hello', 7.2)
```

that entail the same semantics as **in** and **read** but do not block on tuple template mismatch. They both return TRUE, if a tuple match is found, otherwise FALSE, for example:

```
if inp(3, 'hello', 7.2)
        then    (* do something *)
        else    (* do something else *)
```

[1] Linda does not prescribe any particular policy with respect to the action that must ensue on the arrival of a suitable matching tuple. For any tuple that is awaited, a list of associated **in** and **read** requests may exist. Possible actions include: service all **read** requests, and choose one **in** request at random; service a subset of **read** requests, and choose one **in** request at random. [Mif92a] and [Mif92b] suggests that rather than blocking any unsuccessful requests, "blocked" requests are merely re-submitted together with all new requests. This policy allows healthy tuple space competition to dictate the policy, and hence, the greatest non-determinism.

So far, tuples represent passive data structures that are added to or removed from tuple space based on tuple templates. So-called active data structures, or live data structures, are also permitted. Active data structures represent some form of computation which, on completion, results in a passive data structure. The **eval**[2] primitive implements such active data structures. For example:

eval('square root', 25, sqrt(25))

Tuple space recognises the active data structure, implicitly creates a process to compute the value for each tuple element, and awaits the results. On completion, the active data tuple

('square root', 25, sqrt(25))

is replaced by the passive data tuple

('square root', 25, 5)

which is then available for removal or copy.

A number of important features characterise tuples and tuple space operations:

1. Tuples are not labelled or identified in any manner. On addition to tuple space, they become anonymous and addressable or removable by content only. Tuple space requests are NOT of the form

   out(ProducerProcess, ConsumerProcess, <Tuple>)
   in(ProducerProcess, ConsumerProcess, <Tuple>)

   To produce a tuple, the name of the consumer is not required. To consume a tuple, the name of the producer is not required. Linda processes are spatially decoupled.

   Furthermore, consumer and producer processes need not execute simultaneously. Producer processes can terminate before the consumer starts to execute, and requests for tuples can be made before the relevant producer starts to execute. Linda processes are temporally decoupled.

   If point to point communication is desired, it can be implemented easily. Extra addressing

[2] The implementation of the **eval** primitive is the most problematic of all Linda primitives, and has given rise to the largest number of interpretations. The problem relates to the values that are bound to the arguments of eval, and the environment and order in which the tuple elements are evaluated. Typically, arguments inherit bindings from the environment of the process that executes the eval only for whatever names are cited explicitly, and no bindings for any free variables. A non-deterministic argument evaluation order is best applied.

fields are merely added to the tuple, for example

out('prodprocname', 'conprocname', <Tuple>)
in('prodprocname', 'conprocname', <Tuple>)

2. Tuple space operations are not bound by a specific time frame. Tuple space requests are NOT of the form

in(<Tuple>, TimeOutTime)

where **TimeOutTime** might indicate the maximum length of time the issuing process is prepared to wait for **<Tuple>**. The paradigm does not place any upper or lower bound on the length of time it takes to begin to service and to complete a request. The duration of tuple space operations is non-deterministic.

3. Tuple space operations are atomic. Tuple space operations are dealt with completely before new requests are considered.

3.1.3 Programming in Linda

Linda programming techniques are described in [Car86], [Car88], [Car89b], and [Car90b]. The basic styles of parallelism, as well as the basic Linda application program data structures are presented. A brief review follows:

1. Styles of Parallelism

a) result parallelism

The parallel computation is constructed in terms of the result that must be produced. Many separate but identical processes are spawned that generate part of the result. For example, determine all prime numbers between 2 and some limit: for each number in the range spawn a process that determines whether the number is prime. Characteristic of result parallelism is the vast number of processes that are spawned.

b) agenda parallelism

The parallel computation is structured on the master-slave model in which the master generates tasks and the slaves do the work. Normally the computation is initiated by a master who spawns a group of processes that then scavenge for work. The master generates tasks, the slaves seize them, complete the associated work, and then submit the results to the master. For example, determine all prime numbers between 2 and some limit: the range is divided into sections, each section of which is converted into a task, which are then attended to by slaves. Characteristic of agenda parallelism is the

limited number of processes that are spawned, limited tuple space interaction and the implicit workload balancing that occurs ("fast" slaves, or slaves that attend to a task of limited work, get more work rather than idling).

c) specialist parallelism

The parallel computation is structured in terms of independent components each of which conducts a specialised task. For example, determine all prime numbers between 2 and some limit: structure the solution as a series of specialist sieves (from the Sieve of Eratosthenes), that is, a 2-sieve that removes multiples of 2, then a 3-sieve that removes multiples of 3, and so on. Characteristic of specialist parallelism is the pipelined approach, and the possibility of an uneven distribution of workload amongst specialist processes.

Under Linda, agenda parallelism achieves greatest success.

2. Data Structures

Data is categorised as follows:

a) live data structures: a process represents the portion of the data structure that it will create (appropriate for result parallelism).
b) distributed data structures: many processes share direct access to many data objects (appropriate for agenda parallelism).
c) message passing: no data objects are shared but processes communicate information by message passing (appropriate for specialist parallelism).

Distributed data structures include:

a) Semaphores

To execute a V on semaphore **'sem'**:

out('sem')

To execute a P on **'sem'**:

in('sem')

To initialise a semaphore's value to **n**, execute:

out('sem')

n times.

b) Tasks

Task creation:

out('task', TaskDescriptor)

Task withdrawal:

in('task', ?NewTask)

The set of task descriptors normally includes a "poison pill" - a task descriptor that represents "no more tasks to be performed". Slaves recognise the "poison pill" and commit suicide.

c) Name-accessed structures

Structures are given a distinct name, for example, **'count'**:

```
in('count', ?count)
(* some computation on Count *)
out('count', count)
```

Note that the **in** provides mutually-exclusive access to the **'count'** structure. The juxtaposed use of **in** and **out**, as exemplified in the above example, is commonly found in Linda programs.

d) Barriers

Barriers are mechanisms by which the overall computation can be synchronised:

out('barrier', n)

At the barrier, individual processes modify the barrier (they indicate that they have reached the barrier):

```
in('barrier', ?val)
out('barrier', val-1)
```

and await all other processes' arrival at the barrier:

read('barrier', 0)

e) Position-accessed structures

- distributed array:

  ('A', 1, 1, <element>)
  ('A', 1, 2, <element>)
  ('A', 1, 3, <element>)
  ('A', 2, 1, <element>)

  and so on.

- distributed table:

  ('primes', 1, 2)
  ('primes', 2, 3)
  ('primes', 3, 5)
  and so on

- streams:

  ('stream', 1, val1)
  ('stream', 2, val2)
  ('stream', 3, val3)
  and so on.

  The head and tail of the stream are controlled by two further tuples:

  ('stream', 'head', 1)
  ('stream', 'tail', 3)

  To read from the stream:

  in('stream', 'head', ?index)
  out('stream', 'head', index + 1)
  in('stream', index, ?val)

  To write to the stream:

  in('stream', 'tail', ?index)
  out('stream', 'tail', index + 1)
  out('stream', index + 1, newvalue)

3.2 An Experimental Modula-2 Linda System

An experimental Modula-2 Linda system[3] was constructed so that the ideas developed in this thesis could be tested in a real environment.

3.2.1 Modula-2 Linda

Modula-2 Linda restricts the user to the use of integers and strings as tuple element types.

The usual semantics are ascribed to **out**, **in**, **read**, **inp**, and **readp**, but **eval** has the following semantics:

> **eval** takes a single string tuple element that names an executable code file. The tuple space server forks a new process that executes the code contained in the file. No active data tuple is maintained nor does a passive data tuple result from the execution.

Users may also create processes independently of the **eval** primitive by merely executing the appropriate Linda process code file.

3.2.2 An Example Modula-2 Linda Program

The following Linda program[4] implements the cross-product of two matrices. The solution is modelled on the agenda style of parallelism.

```
Algorithm for the master process:
begin
  start a number of worker processes,
  add the relevant rows and columns of the two matrices to tuple space,
  add the first work seed to tuple space, and
  await the results
end.
```

[3] Details of the implementation can be found in Appendix F.

[4] Further examples can be found in Appendix E.

Algorithm for a worker process:

```
begin
 loop
       extract a work seed from tuple space,
       place the "next" work seed in tuple space,
       if the work seed is poisoned
       then    terminate
       else    get the relevant row,
               get the relevant column,
               compute the cross product,
               add the result to tuple space
 end
end.
```

```
MODULE mastercrossp;
(*----------------*)
(*  Modula-2 Linda program:  Cross product of two matrices
    Process:                 master

    Implementation of the cross product of two matrices.
*)

FROM EasyInOut IMPORT WriteString, WriteLn, WriteInt;

TYPE
  MATRIX = ARRAY [1..3] OF ARRAY [1..3] OF INTEGER;
VAR
  M1,M2,M3 : MATRIX;
  Index, I, J, Value : INTEGER;

      PROCEDURE PrintMatrix (Matrix : MATRIX);
      VAR
        I, J : INTEGER;
      BEGIN
        FOR I := 1 TO 3 DO
          FOR J := 1 TO 3 DO
            WriteInt(M2[I,J], 4)
          END;
          WriteLn
        END
      END PrintMatrix;

BEGIN
  (* initialise the matrices *)
  M1[1,1] := 1; M1[1,2] := 2; M1[1,3] := 3;
  M1[2,1] := 1; M1[2,2] := 2; M1[2,3] := 3;
  M1[3,1] := 1; M1[3,2] := 2; M1[3,3] := 3;

  M2[1,1] := 4; M2[1,2] := 5; M2[1,3] := 6;
  M2[2,1] := 4; M2[2,2] := 5; M2[2,3] := 6;
  M2[3,1] := 4; M2[3,2] := 5; M2[3,3] := 6;
```

```
  (* start 3 workers *)
  eval('crossp');
  eval('crossp');
  eval('crossp');

  (* add all rows to tuple space *)
  FOR I := 1 TO 3 DO
    out('r', I, M1[I,1], M[I,2], M[I,3]);
  END;

  (* add all columns to tuple space *)
  FOR I := 1 TO 3 DO
    out('c', I, M[1,I], M[2,I], M[3,I]);
  END;

  (* add work seed to tuple space *)
  out('next', 0);

  (* get the answers back - in any order *)
  FOR Index := 1 TO 9 DO
    in(?I, ?J, ?Value);
    M3[I,J] := Value
  END;

  (* print results *)
  WriteString('First matrix:'); WriteLn;
  PrintMatrix(M1); WriteLn;
  WriteString('Second matrix:'); WriteLn;
  PrintMatrix(M2); WriteLn;
  WriteString('Cross Product:'); WriteLn;
  PrintMatrix(M3); WriteLn
END mastercrossp.


MODULE crossp;
(*----------*)
(*  Modula-2 Linda program:  Cross product of two matrices
    Process:                 worker

    Implementation of the cross product of two matrices.
*)

TYPE
  VECTOR     = ARRAY[1..3] OF INTEGER;
VAR
  Seed, I, J : INTEGER;
  Row, Col   : VECTOR;

BEGIN
  LOOP
    (* get element number to compute *)
    in('next', ?Seed);

    (* set up next piece of work *)
    out('next', Seed + 1);

    IF Seed >= 9
      THEN (* no more work *)
           EXIT
    END;
```

```
        I := (Seed DIV 3) + 1;
        J := (Seed MOD 3) + 1;

        (* get the appropriate row *)
        read('r', I, ?Row[1], ?Row[2], ?Row[3]);

        (* get the appropriate column *)
        read('c', J, ?Col[1], ?Col[2], ?Col[3]);

        (* compute the result *)
        out(I, J, Row[1]*Col[1] + Row[2] * Col[2] + Row[3] * Col[3]);
    END
END crossp.
```

3.3 Conclusion

Linda is a simple model of parallel programming that employs a central content-addressable store, known as tuple space, and a handful of primitives (**out**, **in**, **read**, **inp**, **readp**, and **eval**) with which processes add and remove information, known as tuples.

Linda is not a language, but when tuple space and the associated primitives are implemented in any sequential language, a powerful new parallel programming language results. In this way, many so-called Linda dialects have been developed.

Modula-2 Linda is an experimental Linda implementation. It supports tuple space and the six Linda primitives (albeit with a restricted semantics for the **eval**-primitive) for a constrained set of tuple element types (strings and integers). Notwithstanding these constraints, it demonstrates the Linda programming philosophy adequately.

Chapter 4

A Model for Debugging Linda Programs

The previous chapter detailed the Linda parallel programming paradigm. This chapter presents a model for debugging Linda programs that is based on behavioural model techniques of debugging. An informal discussion of the model is followed by a formal approach wherein the basic Linda model and the Linda debugging model are expressed in the Calculus of Communicating Systems (CCS) [Mil89]. Observations are made and important properties are derived, in the Modal mu-Calculus [Koz83]:

1. of the basic model that show that Linda programs are amenable to debugging, and
2. of the debugging model that show that the debugger avoids certain problems inherent in other debuggers.

4.1 The Debugging Model

Behavioural model techniques of debugging require that the programmer specify the expected behaviour of the parallel program prior to program execution. At run-time the actual behaviour is compared with the expected behaviour, and inconsistencies are reported. Any deviation from the expected behaviour is deemed to constitute a program execution fault. The debugging process, as such, begins with the specification of the expected behaviour (the static phase), is followed at run-time by the comparison of the actual versus the expected behaviour (the dynamic phase), and culminates with the declaration that the program satisfied its expected behaviour or with the detection of comparison anomalies somewhere along the way.

Behavioural model techniques of debugging represent a much changed strategy with regard to debugging than has been adopted previously (whether in the sequential or parallel programming domain). Traditional techniques embrace an approach wherein the user sets breakpoints in the program's execution path at which the program state is examined. If, in the sole opinion of the user,

all "looks OK", execution is allowed to proceed to the next breakpoint or to conclusion. Alternatively, the user detects what is considered to be anomolous behaviour for which an appropriate solution can be generated, and terminates the execution of the program. The user is required to deduce, from the program state, reasons for correct or faulty behaviour. Whilst a plethora of facilities may exist that aid the deductions, they remain mere tools. The success of the entire process is a direct function of the user's deductive powers, and the use, as opposed to misuse or abuse, of the appropriate tools at the appropriate time.

Behavioural model techniques attempt to remove much of the burden of the debugging process from the user, and to locate the entire process in a more formal domain. The user is required to follow a rigid sequence of steps that is aimed at a structured debugging process, rather than an ad hoc approach that is based on useful tricks. In essence, the user plays a much more passive role.

Expected program behaviour is defined in terms of program events. A wide variety of program actions can constitute a program event, and their definition varies from system to system. In the simplest case, the execution of every program statement, every memory location access, indeed any program execution action, is an event. Such a scenario is unquestionably all-embracing, but deficient in that it makes no discrimination between useful/interesting events and events that are of marginal significance. The volume of events that are generated is also quite overwhelming. A more useful strategy is to regard event occurrence at a higher level of abstraction; for example, to regard the execution of a procedure/function call as an event. So, given the following program fragment (*pf*):

```
.
.
.
ReadData(a);
(* program code that excludes any procedure calls *)
ProcessData(a);
(* program code that excludes any procedure calls *)
PrintResults(a);
.
.
.
```

the corresponding expected behaviour, based on procedure/function calls, is:

$$Expected\_Behaviour_{pf} = ReadData(a).ProcessData(a).PrintResults(a)$$

Here, should the program fragment be executed, the actual behaviour would match the expected behaviour. If the expected behaviour was specified as follows:

$Expected\_Behaviour_{pf}$ = *ReadData(a).PrintResults(a).ProcessData(a)*

a program execution fault would be reported at *PrintResults(a)*.

The use of procedure/function calls as events allows the behaviour of a program to be considered in more abstract terms and, on a practical note, reduces the volume of events to a more manageable level. Unfortunately it does not discriminate between "important" and "unimportant" procedures and functions.

In a parallel programming context, a discriminator could separate calls to procedures and functions charged with the responsibility of process creation, process termination, and interprocess communication from calls to other procedures and functions. Event lists generated as a result of the occurrence of such calls would paint a picture of reasonable resolution of the parallel activity of the program under inspection. But what of Linda?

Linda programs are composed of a suite of processes that interact/communicate indirectly with each other via tuple space. Processes themselves are divided logically into a parallel or coordination component, wherein the Linda primitives are found, and a sequential or computation component that glues together the Linda primitives and the process as a whole. If processes are described purely in terms of Linda primitives, a clear picture is obtained of the parallel nature of the processes. Should the Linda primitives be the sole source of Linda program events, an equally clear picture is revealed to the debugger of the parallel nature of the processes.

This work utilises the Linda primitives

out, in, read, inp, readp, eval

as the source of Linda program events. For example, given the following Linda program fragment (*lpf*):

```
.
.
.
in(?a);
(* computation code *)
in(?b);
(* computation code *)
out(a*b);
.
.
.
```

the corresponding expected behaviour, based on Linda events, is:

$$Expected\_Behaviour_{lpf} = in(?a).in(?b).out(a*b)$$

Should the program fragment be executed, the actual behaviour would match the expected behaviour. (It is worth noting that the Linda primitives, in all likelihood, find realisation in the form of procedures and functions in the host language. In practical terms, the use of Linda primitives as the source of program events translates to nothing more than the use of procedures and functions as the source of program events, but discriminating against all procedures and functions that do not implement a Linda primitive.)

In general, the expected behaviour of a Linda process is defined as an ordered collection of Linda events. The expected behaviour of a Linda program is then defined as the sum of the expected behaviour of the individual processes of which the Linda program is composed:

$$Expected\_Behaviour_{Linda\ program} = \Sigma_{p \in P}\ Expected\_Behaviour_p$$

where: 1. P = set of all process identifiers
2. $p \in P$

Given that some form of internal model of expected Linda program behaviour exists, run-time data (actual behaviour) must be obtained against which to perform the behavioural comparisons. A Linda environment is composed of a suite of processes and tuple space to which all processes, by way of Linda primitives, appeal for attention. Since Linda events are based on Linda primitives, tuple space, on attending to a Linda primitive, signals the occurrence of the corresponding Linda event, and initiates a behavioural comparison. In practical terms, as each process ($Process_p$) requests tuple space attention by means of a Linda primitive, and gains attention, the associated Linda event is generated and compared with the next expected event in $Expected\_Behaviour_p$. The debugger compares the actual event with the expected event, and posts a reply to tuple space. If the events compared unfavourably, the debugger also signals a process fault to the "environment" (in an actual implementation, a tuple space monitor constitutes the "environment"). Tuple space receives the result of the comparison, and continues execution.

The debugging model is simple, and is based on a limited set of well-defined program events. It is also independent of the expected behavioural model construction process and the nature of the behavioural model itself.

4.2 CCS

The Calculus of Communicating Systems (CCS) is a formal specification language that is frequently

used to specify parallel systems and their implementations. The resultant definitions are examined, compared for equivalences, and generally used to reason about the systems so defined, all within the confines of a formal methodology.

CCS models individual computation as an agent that changes state via inter-agent "actions". Complex systems and agents are defined by building complex agent expressions in CCS. Agent expressions, in turn, are composed of "actions", agents, and a set of operators over agents.

Agents are entities within a system that synchronise their activities through complementary named ports/labels that are drawn from the set *Act* of actions:

$$Act = A \cup \bar{A} \cup \{\tau\}$$

where: 1. A and $\bar{A}$ are sets of observable actions between which there is a one-to-one correspondence via $\bar{}$.

2. τ is the silent action which is not observable.

For example, some $Agent_1$ could synchronise activity or communicate with some other $Agent_2$ via the complementary labels $\overline{in}$ and in, where the overbar designates an output label. Furthermore, output labels may be parameterised by an expression, and input labels parameterised by a variable.

The following operations are defined on agents:

Given the agents P and Q:

1. action prefix:

   $a.P \xrightarrow{a} P$

   The agent $a.P$ performs the action a and evolves into P.

2. exclusive selection (summation) (+):

   if $P \xrightarrow{a} P'$ and $Q \xrightarrow{b} Q'$

   then $(P + Q) \xrightarrow{a} P'$

   or

   $(P + Q) \xrightarrow{b} Q'$

   The agent $P + Q$ can perform either the action a or b and evolve into either P' or Q' respectively.

3. composition (|):
parallel action

$$\text{if } P \xrightarrow{a} P' \quad \text{then} \quad P|Q \xrightarrow{a} P'|Q$$
$$\text{if } Q \xrightarrow{\bar{a}} Q' \quad \text{then} \quad P|Q \xrightarrow{\bar{a}} P|Q'$$
$$\text{if } P \xrightarrow{a} P' \text{ and } Q \xrightarrow{\bar{a}} Q' \quad \text{then} \quad P|Q \xrightarrow{\tau} P'|Q'$$

The agent $P|Q$ can perform either the action a, $\bar{a}$ or τ and evolve into $P'|Q$, $P|Q'$, or $P'|Q'$ respectively. Note how τ is used to represent communication on complementary named ports, and is not observable.

4. restriction (\):
restricted communication on labels

$$\text{if } P \xrightarrow{a} P' \text{ and } a, \bar{a} \notin L \quad \text{then} \quad P\backslash L \xrightarrow{a} P'\backslash L$$

the agent P may only communicate on a, if a and $\bar{a}$ are not contained in the sort (collection of labels) L. Note that the silent action (τ) can not be restricted.

5. relabelling ([f]):
label renaming function

$$\text{if } P \xrightarrow{a} P' \quad \text{then} \quad P[f] \xrightarrow{f(a)} P'[f]$$

label a in agent P is renamed.

The agent incapable of any action is represented by **0**.

Agents are usually defined in the following manner:

$$agent \overset{def}{=} agentexpression$$

Recursive definitions are also permitted, that is, *agentexpression* may contain *agent*. For example, an agent that repeatedly communicates on label $in1$ and then on $in2$ is modelled as:

$$Agent1 \overset{def}{=} \overline{in1}.\overline{in2}.Agent1$$

The $\rightarrow$ transition system states that given the agents P and Q and some action a then the interpretation

of $P \xrightarrow{a} Q$ is that P may evolve into Q by performing the observable action a.

For silent (τ) actions, a second transition system, $\Rightarrow$, is derived. The set of transitions is $\{\stackrel{a}{\Rightarrow} \mid a \in A \cup \{\xi\}\}$ where $\stackrel{\xi}{\Rightarrow}$ is the transitive reflexive closure of $\xrightarrow{\tau}$, so that $P \stackrel{\xi}{\Rightarrow} Q$ if P may evolve into Q by performing zero or more silent actions, and that $\stackrel{a}{\Rightarrow} = \stackrel{\xi}{\Rightarrow} \xrightarrow{a} \stackrel{\xi}{\Rightarrow}$ if P may evolve into Q by performing zero or more silent actions, followed by a, followed by zero or more silent actions.

Equivalence relations are defined between agents that are based on action capabilities. One such equivalence is observational equivalence. Intuitively, two complex systems are observationally equivalent, if they always exhibit the same observable behaviour (silent (τ) activity aside), that is, if an observed action of one expression can be matched by an observed action of the other expression so that the resulting states are themselves observationally equivalent.

Observational equivalence ($\approx$) is defined between agents P and Q such that:

$P \approx Q$, iff for each action a:

a) if P' is such that $P \xrightarrow{a} P'$, then either 1) there is a Q' such that $Q \stackrel{a}{\Rightarrow} Q'$ and $P' \approx Q'$, or 2) $a = \tau$ and $P' \approx Q$, and conversely,

b) if Q' is such that $Q \xrightarrow{a} Q'$, then either 1) there is a P' such that $P \stackrel{a}{\Rightarrow} P'$ and $P' \approx Q'$, or 2) $a = \tau$ and $P \approx Q'$.

(Consult [Wal87] for an introduction to CCS, and [Mil89] for a comprehensive discussion of CCS.)

4.3 Modal Logic and the Modal Mu-Calculus

The properties of parallel systems can be described in modal and temporal logics. In a generalisation of Hennessy-Milner logic [Hen80], [Hen85], so-called logic formulae are constructed from boolean connectives and the modal operators $[K]$ and $\langle K \rangle$, where K is a set of actions (in Hennessy-Milner logic, only single actions are permitted in the modalities). The abstract syntax definition for these formulae are:

$$\phi \quad ::= \quad \mathtt{tt} \mid \mathtt{ff} \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid [K]\phi \mid \langle K \rangle \phi$$

All processes have the property `tt`; no process has the property `ff`; a process has the property $\phi_1 \wedge \phi_2$, if it has both the property ϕ_1 and ϕ_2; a process has the property $\phi_1 \vee \phi_2$, if it has either the property ϕ_1 or ϕ_2; a process has the property $[K]\phi$ (necessity), if after every performance of any action in K, each resultant process has the property ϕ; and a process has the property $\langle K \rangle \phi$ (possibility), if after the performance of at least one action in K, the resultant process has the property ϕ.

The logic does not accord any special status to silent (τ) activity. Silent activity can, however, be

introduced by two new modalities:

[[ ]] - weak necessity
<< >> - weak possibility

wherein the occurrence of zero or more silent actions ($\stackrel{\tau}{\Rightarrow}$) is embodied. Further modalities can then be introduced:

$[[K]] ::= [[\ ]]\ [K]\ [[\ ]]$
$\langle\langle K \rangle\rangle ::= \langle\langle\ \rangle\rangle\ \langle K \rangle\ \langle\langle\ \rangle\rangle$

where: K is a subset of observable actions.

The modal logic is able to express local capabilities (the system is able to perform some sequence of action/s) and immediate necessities (the system must perform some sequence of action/s) but not enduring capabilities (the system must always be able to perform some sequence of action/s) or long-term inevitabilities (the system must eventually be able to perform some sequence of action/s). So that such temporal properties may be expressed, the modal mu-calculus [Koz83], in an extended form [Sti91], is used that extends the modal logic to include propositional variables and fixed point operators. The above abstract syntax is augmented as follows:

$\phi ::= \ldots \mid Z \mid \nu Z.\phi \mid \mu Z.\phi$

where 1. Z is a propositional variable
2. $\nu Z.\phi$ is the maximal fixed point operator (ν) in the modal equation Z
3. $\mu Z.\phi$ is the minimal fixed point operator (μ) in the modal equation Z

The modal logic and modal mu-calculus are used to define properties that processes exhibit. It is also frequently the case that equivalence relations are defined between processes that are based on the properties that they do or do not possess.

(Consult [Sti92] for an informative discussion of modal and temporal logics for processess.)

4.4 Linda

4.4.1 Properties of Linda

Chapter 3 detailed the Linda parallel programming paradigm. The six primitives were introduced, as were the spacial and temporal decoupling of processes, and the non-deterministic duration of tuple space operations. Prior to embarking on any formal specification, the properties embodied in the

primitives, the decoupling, and the non-determinism merit highlight.

A distinguishing characteristic of the Linda primitives is their commitment to interaction with tuple space, once interaction is initiated. When a process makes a request on tuple space, it does not "back-off" or "time-out" until the request is serviced.

Allied to this commitment is the time duration of tuple space operations. In [Mar90], a time and event-action paradigm is introduced for the study of debugging tools for parallel and distributed software. A variety of terms are introduced, a few of which are relevant to this discussion:

- event (e) occurs (o) at time t_e^o (Linda equivalent: tuple space interaction requested)
- event (e) is recognised (r) at time t_e^r (Linda equivalent: tuple space attention gained)
- initiates (i) action (a) at time $t_{e,a}^i$ (Linda equivalent: tuple space starts processing request)
- terminates (t) action (a) at time $t_{e,a}^t$ (Linda equivalent: tuple space completes processing request)

The following values are derived:

event recognition latency (t_e^r - t_e^o)
action enabling latency ($t_{e,a}^i$ - t_e^r)
duration of action ($t_{e,a}^t$ - $t_{e,a}^i$)
event processing time ($t_{e,a}^t$ - t_e^o)

In Linda, the sum of the values constitutes the duration of a tuple space interaction. This duration is, however, defined as non-deterministic. This provides tuple space with widespread licence to conduct any number of time-variant activities, possibly related to debugging, without violating the underlying paradigm.

Linda processes are both spatially and temporally decoupled. Processes interact indirectly via tuple space - processes do not name the process with whom they ultimately interact, indeed the relevant processes may not even execute simultaneously.

4.4.2 Formal Specifications

4.4.2.1 Previous Work

Researchers have specified the Linda parallel programming paradigm in a number of formalisms, notably CCS, both the basic and the full calculus [Mil89], and Z [Hay87], [Spi92]. A distinguishing characteristic of the attempts that have been made is the level of abstraction at which the paradigm is modelled which seems to reflect the relative "distance" from actual implementation at which the

specification is made. In general, the more abstract is the specification, the greater is the degree of non-determinism that is permitted.

Mifsud [Mif92a], [Mif92b] uses CCS to explore the semantics of Linda with special emphasis on the integration of Linda into a suitable host language. Careful consideration is taken of the order of argument tuple evaluation, tuple element assignment (actual to formal), and the integration of Linda into a sequential imperative language. A high level of abstraction is adopted in the specification of tuple space and the individual Linda primitives. Agents are defined that specify the behaviour of tuple space, and the six Linda primitives (**in, read, inp, readp, out, eval**). Of importance, is the treatment of unsuccessful **in** and **read** operations. In Mifsud's model, if no match can be found between the request template and a free tuple, the corresponding **in** or **read** agent resubmits the request anew, in competition with all other tuple space requests. This effectively implements a "busy wait" policy. The resubmission policy is demonstrated in the following extract from the overall definition:

$$TupleSpace(M) \stackrel{def}{=} intuple(u).InTuple(M, u) + rdtuple(u).RdTuple(M, u) + addtuple(u).AddTuple(M, u)$$

$$InTuple(M, u) \stackrel{def}{=} \text{if } (match\ M\ u = \emptyset) \text{ then } \overline{fail}.TupleSpace(M) \text{ else } \overline{gettuple(u')}.TupleSpace(M')$$

$$In(u) \stackrel{def}{=} \overline{intuple(u)}.(fail.In(u) + gettuple(u').AssignTuple(u, u'))$$

where: 1. Tuple space (M) is defined as a multiset over which the operations multiset union ($\uplus$) and multiset difference ($\biguplus$) are defined.

2. $M = M' \uplus \{u\}$

3. $u' \in match\ M\ u$
   $match\ M\ u = \{u' \mid matchtuple(u, u')\}$
   $matchtuple(u, u')$ compares template u for equality with free tuple u'

4. $AssignTuple(u, u')$ takes a template and a free tuple, and assigns actual values to formal fields.

Repeated requests can be thought of as "internal chatter" - since the process can not engage in any further activity, the semantics of **in** are preserved. The absence of any form of explicit blocking on unsuccessful requests obviates the necessity for any form of policy on the following issues:

- which blocked **in(u)** is serviced by an appropriate **out(u')**?
- are all **read(u)**'s that are waiting on a common **u'** serviced by the corresponding **out(u')**? In which order are they serviced? Is some **in(u)** serviced as well? In which case, which

in(u)?

Tuples are added to tuple space with no further considerations - the policy is inherent in the definition. **read**'s, **in**'s, and repeated **read**'s and **in**'s all compete for tuple space attention on an equal footing.

Hazelhurst [Haz90] uses CCS to specify the semantics of Linda (tuple space and Linda primitives). Agents are defined that specify the behaviour of tuple space and the six Linda primitives. A far more explicit approach is adopted where aspects of the Linda implementation are reflected. Of importance is the use of a blocking mechanism for unsuccessful tuple space requests. Tuple space is modelled as a triple:

$TS\langle R,I,T\rangle$

where: 1. T = free tuples
2. I = processes blocked on **in(u)**
3. R = processes blocked on **read(u)**
4. I and R are sets each element of which is composed of the process name and the tuple template.

Whenever tuple space is unable to find a match between a tuple template and a free tuple, the process and tuple template are added to I or R. (Arguments to the primitives include the tuple template as well as the originating process to which a reply can be later sent.) Each time a new tuple (**u'**) is added to tuple space, a subset of those processes blocked on R pending addition of such a tuple (**u'**) are serviced, and any one of those processes blocked on I pending addition of such a tuple (**u'**) are serviced. The blocking mechanism is demonstrated in the following extract from his overall definition:

$$TS(R,\ I,\ T) \stackrel{def}{=} tgive(x,p).$$
$$\text{if } p(x,T) = \varnothing$$
$$\text{then} \quad TS(R,\ I \cup \{(x,p)\},\ T)$$
$$\text{else} \quad \Sigma_{y \in p(x,T)}\ \overline{tget_p}(y).TS(R,\ I,\ T - \{y\})$$
$$\ldots$$
$$+$$

tadd(x).
$\Sigma_{N \subseteq p(x,R)} P_N$.
if $p(x,I) = \varnothing$
then $TS(R\text{-}N, I, T \uplus \{x\})$
else $\Sigma_{(y,p) \in p(x,I)} \overline{tget_p}(y).TS(R\text{-}N, I\text{-}\{(y,p)\}, T)$

...

$In_p(x) \stackrel{def}{=} \overline{tgive}(x,p).tget_p(y)$

where: 1. $p(x,I) = \{(y,p) \in I \mid match(x,y)\}$ for any $x \in$ Tuples
2. $p(x,R) = \{(y,p) \in R \mid match(x,y)\}$ for any $x \in$ Tuples
3. $p(x,T) = \{y \in T \mid match(x,y)\}$ for any $x \in$ Tuples
4. $match(x,y)$ is a boolean function that performs a match between tuples x and y.
5. $P_N = \Pi_{(y,p) \in N} \overline{tget_p}(y)$

Essentially Hazelhurst provides explicit definition for the actions pursuant to **read**, **in**, and **out** operations, whereas Mifsud allows healthy competition for tuple space attention to determine the action policy.

Jensen [Jen90] uses CCS to explore the semantics of tuple space and the correctness of an implementation. The semantics of a Linda language, a Linda language with respect to tuple space, and tuple space are defined. Use is made of inference (transition) rules to specify the behaviour of tuple space and processes.

Ciancarini et al (the article is co-authored by Jensen) [Cia92] specify Linda semantics in SOS [Plo81], CCS, Petri Nets, and the Chemical Abstract Machine [Ber90b], and then compare the specifications. The basis of the CCS specification is a translation from the Linda calculus to CCS and agents.

Butcher [But91b] and Hasselbring [Has92] use Z to specify the semantics of Linda-2 and ProSet-Linda respectively. [But91b] concentrates attention on the semantics of the Linda model and makes as few assumptions about the host language as possible. [Has92], like [Mif92b], provides a more all-embracing definition of the host language and the Linda model.

None of the specifications place any form of restriction on timing (wait times before tuple space attends to a request or duration of tuple space operations), or fairness (given two or more processes awaiting the addition of the same tuple, there is no guarantee that all requests will be serviced

eventually). This affords the broadest non-determinism and least constraint on implementations.

4.4.2.2 A Formal Specification of Linda

The following specification defines the semantics of the Linda parallel programming paradigm. The actions of tuple space as well as those of individual processes are provided. Like [But91b] as few assumptions as possible are made about the host language, save that it has a type system, and each specific type supports a set of values:

$$\boldsymbol{T} \quad - \quad \text{set of all types}$$
$$\forall\, T \in \boldsymbol{T},\ \exists\, V_T = \{v_{T_1}, v_{T_2}, \ldots, v_{T_n}\} \wedge \exists\, \perp_T$$

where: 1. v_{T_i} is a value of type T
2. $\perp_T$ is a formal of type T

Furthermore, $\boldsymbol{T}$ is a set of distinct types so for any $T_i, T_j \in \boldsymbol{T}$, V_{T_i} and V_{T_j} are disjoint.

A tuple element is a pair

$$(v{:}T) \mid T \in \boldsymbol{T} \wedge\ ((v \in V_T) \vee (v = \perp_T))$$

The element is either an actual value or a formal ($\perp$) of a particular type (T).

A tuple (u) is composed of either any number of tuple elements or a process identifier:

$$u \quad = \quad (v_1{:}T_1, v_2{:}T_2, \ldots, v_n{:}T_n) \vee \text{process identifier}^1$$

Tuple space (M) is defined as a multiset.

Given two tuples, u and u':

$$u \quad = \quad (v_1{:}T_1, v_2{:}T_2, \ldots, v_n{:}T_n)$$
$$u' \quad = \quad (v'_1{:}T'_1, v'_2{:}T'_2, \ldots, v'_m{:}T'_m)$$

[1] The **eval**-primitive takes a single tuple element (a process identifier) which identifies the process that must be spawned.

a boolean tuple match function is defined as follows:

$$\begin{aligned} &\textit{matchtuple}\ (u, u') \\ &\quad = \quad m = n \ \ \bigwedge_{i=1}^{n} \textit{matchvalue}\ (v_i{:}T_i,\ v_i'{:}T_i') \\ &\qquad\ \ m \neq n \ \ \text{false} \end{aligned}$$

$$\begin{aligned} &\textit{matchvalue}(v_i{:}T_i,\ v_i'{:}T_i') \\ &\quad = \quad \textit{true} \ \ ((v_i \neq \perp_{T_i}) \wedge (v_i = v_i')) \vee ((v_i = \perp_{T_i}) \wedge (T_i = T_i')) \\ &\qquad\ \ \textit{false} \ \ \text{otherwise} \end{aligned}$$

A further function:

$$\textit{match}(M, u) \quad = \quad \{u' \mid u' \in M \ \wedge \ \textit{matchtuple}(u, u')\}$$

returns a set of tuples that match *u*.

In this work, the Linda primitives are defined as follows:

out(u) - add (passive) tuple (**u**) to tuple space

in(u) - if matching tuple (**u'**) in tuple space
then extract tuple **u'** from tuple space
else wait until tuple **u'** is available

read(u) - if matching tuple (**u'**) in tuple space
then extract a copy of tuple **u'** from tuple space
else wait until tuple **u'** is available

inp(u) - if matching tuple (**u'**) in tuple space
then extract tuple **u'** from tuple space
else return false

readp(u) - if matching tuple (**u'**) in tuple space
then extract a copy of tuple **u'** from tuple space
else return false

eval(u) - instantiate process (**u**)

The definition of Linda comprises two major components: tuple space and processes. Linda processes are modelled by individual agents, and communicate with tuple space via a number of ports that

represent the Linda primitives. For **in**, **read**, **inp**, and **readp**, dual ports are provided that represent a request for tuple information and a reply from tuple space. For **out**, a single port is provided, since it does not solicit reply information. Unsuccessful **inp** and **readp** operations are informed of their failure by communication on a failure port. Special signals are generated, via appropriately named ports, on operation completion and process termination - they constitute interaction with the environment (observers) and not interaction with any specific agent of the (Linda) system. Unsuccessful **in** and **read** requests are not blocked within tuple space pending arrival of appropriate tuples but are rejected by tuple space and re-submitted by the relevant process, as is done in [Mif92b]. It is important to note that a different communication port is used to re-submit requests. In the debugger, it is necessary to distinguish between original and re-submitted requests. Since all tuple space requests are treated with equal priority, no violation of the underlying paradigm is experienced by different communication ports for original and re-submitted requests.

The definition resembles that found in [Mif92b] but differs in the following respects: explicit Linda process agents are defined that communicate with tuple space on labels indexed by process identifier (the debugger requires that a process identifier accompany all tuple space requests that it checks), the assignment of actual values to formal tuple fields is not specified, extra agents and ports have been utilised. [Haz90]'s rigorous treatment of unsuccessful tuple space requests over-constrains what are generally regarded as acceptable Linda semantics.

Tuple space is modelled by the *TS(M)* agent:

$$
\begin{aligned}
TS(M) \quad &\stackrel{\text{def}}{=} \quad out_p(u).TS(M \uplus \{u\}) + \\
&\qquad inreq_p(u).TSinreq(M, u, p) + \\
&\qquad repinreq_p(u).TSinreq(M, u, p) + \\
&\qquad rdreq_p(u).TSrdreq(M, u, p) + \\
&\qquad reprdreq_p(u).TSrdreq(M, u, p) + \\
&\qquad inpreq_p(u).TSinpreq(M, u, p) + \\
&\qquad rdpreq_p(u).TSrdpreq(M, u, p)
\end{aligned}
$$

$$
\begin{aligned}
TSinreq(M, u, p) \quad &\stackrel{\text{def}}{=} \quad \textit{if } match(M, u) = \varnothing \\
&\qquad \textit{then} \quad \overline{fail}.TS(M) \\
&\qquad \textit{else} \quad \overline{in_p}(u').TS(M - \{u'\})
\end{aligned}
$$

$$
\begin{aligned}
TSrdreq(M, u, p) \quad &\stackrel{\text{def}}{=} \quad \textit{if } match(M, u) = \varnothing \\
&\qquad \textit{then} \quad \overline{fail}.TS(M) \\
&\qquad \textit{else} \quad \overline{rd_p}(u').TS(M)
\end{aligned}
$$

$$
\begin{aligned}
TSinpreq(M, u, p) \quad &\stackrel{\text{def}}{=} \quad \textit{if } match(M, u) = \varnothing \\
&\qquad \textit{then} \quad \overline{fail}.TS(M) \\
&\qquad \textit{else} \quad \overline{inp_p}(u').TS(M - \{u'\})
\end{aligned}
$$

$$TSrdpreq(M, u, p) \stackrel{def}{=} \text{if } match(M, u) = \emptyset \text{ then } \overline{fail}.TS(M) \text{ else } \overline{rdp_p}(u').TS(M)$$

Processes are modelled by the $Process_p$ agent:

$$Process_p \stackrel{def}{=} \overline{out_p}(u).ProcessOut_p + \overline{inreq_p}(u).ProcessIn_p(u) + \overline{rdreq_p}(u).ProcessRd_p(u) + \overline{inpreq_p}(u).ProcessInp_p + \overline{rdpreq_p}(u).ProcessRdp_p + \overline{term_p}.\mathbf{0}$$

$$ProcessOut_p \stackrel{def}{=} \overline{done_p}.Process_p$$

$$ProcessIn_p(u) \stackrel{def}{=} in_p(u').\overline{done_p}.Process_p + fail.\overline{repinreq_p}(u).ProcessIn_p(u)$$

$$ProcessRd_p(u) \stackrel{def}{=} rd_p(u').\overline{done_p}.Process_p + fail.\overline{reprdreq_p}(u).ProcessRd_p(u)$$

$$ProcessInp_p \stackrel{def}{=} fail.\overline{res_p}(false).Process_p + inp_p(u').\overline{res_p}(true).Process_p$$

$$ProcessRdp_p \stackrel{def}{=} fail.\overline{res_p}(false).Process_p + rdp_p(u').\overline{res_p}(true).Process_p$$

In $TS(M)$ and $Process_p$:

1. $u \in$ set of all tuples
2. $u' \in match(M, u)$
3. P = set of all process identifiers
4. $p \in P$

Note how:

1. $\overline{repinreq}$ and $\overline{reprdreq}$ are used to re-submit unsuccessful **in** and **read** requests,
2. $\overline{done}$ is used to signal **out**, **in**, and **read** operation completion,
3. $\overline{res}$ is used to signal successful or otherwise **inp** and **readp** operation completion, and

4. $\overline{term}$ is used to signal process termination.

The Linda system is then specified as follows:

$$Linda \quad \stackrel{def}{=} \quad (TS(M)|Process_1|Process_2| \ ... \ | \ Process_n)\backslash L$$

where: $L = \{\bigcup_{p \in P} (out_p, inreq_p, repinreq_p, in_p, rdreq_p, reprdreq_p, rd_p, inpreq_p, inp_p, rdpreq_p, rdp_p) \cup fail\}$

The specification so far does not model the **eval**-primitive. Rather, it models the action of a Linda system in terms of tuple space, all the processes of which the Linda program is composed, and certain tuple space actions. No consideration is taken of process instantiation - all processes are considered instantiated *ab initio*. Whilst the specification does not accurately reflect Linda (there is no dynamic process creation), it does possess the very desirable property of a finite state space. All processes and process actions are known in advance, and can be analysed accordingly.

The introduction of dynamic process creation is now considered.

It is true of most Linda implementations that a single process, usually a master or distinguished process, is instantiated whenever the Linda system is started. Thereafter all further processes, known as spawned processes, are instantiated using the **eval**-primitive. To include such an **eval** mechanism in the specification, individual processes must be accorded the capability of spawning processes. Tuple space must also be informed each time a process spawns a process. (Strictly speaking, tuple space need not be informed - it is merely a synchronisation that does not impact on tuple space. However, when the debugger is introduced, all process behaviour, including process creation, is checked from within tuple space, and it needs to know of all process activity.) Tuple space (*TS*) and processes ($Process_p$) are augmented with an *eval* label:

$$TS(M) \quad \stackrel{def}{=} \quad .$$
$$.$$
$$.$$
$$eval_p(u).TS(M)$$

$$Process_p \quad \stackrel{def}{=} \quad .$$
$$.$$
$$.$$
$$\overline{eval_p}(u).(\overline{done_p}.Process_p \mid Process_u)$$

Here, the process informs tuple space of its intention to spawn a new process after which it goes ahead and does so.

The Linda system is then specified as follows:

$$Linda \stackrel{def}{=} (TS(M) | Process_1) \backslash L$$

where: 1. $Process_1$ is the distinguished process.

2. $L = \{\bigcup_{p \in P} (out_p, inreq_p, repinreq_p, in_p, rdreq_p, reprdreq_p, rd_p, inpreq_p, inp_p, rdpreq_p, rdp_p, eval_p) \cup fail\}$

$Process_1$ spawns new processes that themselves are able to spawn new processes. The modified specification certainly caters for dynamic process creation but also introduces an undesirable infinite state space - no constraint is placed on the number of processes that may be created. As a result, many useful analyses of the system are precluded.

The specification of a Linda system is now explored that provides for a limited form of dynamic process creation within a finite state space.

The specification models the action of a Linda system in terms of tuple space, all the processes of which the Linda program is composed, and tuple space actions (as in the original specification). Again, a distinguished process is utilised that is instantiated whenever the Linda system is started (as is the case in the first attempt at the inclusion of **eval**). However, so that all processes, except the distinguished process, are not capable of action until they are actually spawned, all spawned processes are forced to wait on a signal ($start_{sp}$), pursuant to an appropriate *eval*, after which they become active. The specification is modified as follows:

$$TS(M) \stackrel{def}{=} .$$
$$.$$
$$.$$
$$eval_p(u).TS(M)$$

$$Process_1 \stackrel{def}{=} \overline{out_1}(u).ProcessOut_1 +$$
$$\overline{inreq_1}(u).ProcessIn_1(u) +$$
$$\overline{rdreq_1}(u).ProcessRd_1(u) +$$
$$\overline{inpreq_1}(u).ProcessInp_1 +$$
$$\overline{rdpreq_1}(u).ProcessRdp_1 +$$
$$\overline{eval_1}(u).\overline{start_u}.ProcessEval_1$$
$$\overline{term_1}.\mathbf{0}$$

$$ProcessEval_1 \stackrel{def}{=} \overline{done_1}.Process_1$$

$$Process_{sp} \stackrel{def}{=} \overline{start_{sp}}.ProcessSt_{sp}$$

$$ProcessSt_{sp} \stackrel{def}{=} \overline{out_{sp}}(u).ProcessStOut_{sp} + \overline{inreq_{sp}}(u).ProcessStIn_{sp}(u) + \overline{rdreq_{sp}}(u).ProcessStRd_{sp}(u) + \overline{inpreq_{sp}}(u).ProcessStInp_{sp} + \overline{rdpreq_{sp}}(u).ProcessStRdp_{sp} + \overline{eval_{sp}}(u).\overline{start_u}.ProcessStEval_{sp} \overline{term_{sp}}.\mathbf{0}$$

$$ProcessStEval_{sp} \stackrel{def}{=} \overline{done_{sp}}.ProcessSt_{sp}$$

where: 1. $Process_1$ is the distinguished process.

2. $Process_{sp}$ represents a spawned process that awaits a start signal on $\overline{start_u}$, where u is a process identifier, and $sp \in (P - \{1\})$.

3. $ProcessOut_1$, $ProcessIn_1$, $ProcessRd_1$, $ProcessInp_1$, $ProcessRdp_1$, and $ProcessStOut_{sp}$, $ProcessStIn_{sp}$, $ProcessStRd_{sp}$, $ProcessStInp_{sp}$, $ProcessStRdp_{sp}$ are the same as $ProcessOut_p$, $ProcessIn_p$, $ProcessRd_p$, $ProcessInp_p$, $ProcessRdp_p$ but for a change in agent name (inclusion of *St*) and index (*1*).

4. $\overline{done}$ is used to signal **eval**-operation completion.

The Linda system is then specified as follows[2]:

$$Linda \stackrel{def}{=} (TS(M)|Process_1|Process_2| \ldots | Process_n)\backslash L$$

where: 1. $Process_1$ is the distinguished process.

2. $Process_2$ .. $Process_n$ are spawned processes.

3. $L = \{\bigcup_{p \in P} (out_p, inreq_p, repinreq_p, in_p, rdreq_p, reprdreq_p, rd_p, inpreq_p, inp_p, rdpreq_p, rdp_p, eval_p, start_p) \cup fail\}$

No single process is started more than once - every process that forms part of the Linda program, even multiple instances of the same process, is represented by a different $Process_p$. Since all processes are represented, the state space is finite. Essentially, dynamic process creation is provided within the confines of a finite state space, but subject to the upper limit of the *n* process "slots" defined in the system.

As is demonstrated above, CCS is quite able to model systems that can increase unboundedly. Indeed, [Mil89] notes that "this takes us out of the realm of direct descriptions of physical systems, and opens up the possibility of more abstract descriptions such as the generation of tasks in a parallel

[2] A full specification of the Linda system can be found in Appendix B.

programming language". However, he also indicates the disadvantages of such "unboundedness" - observation equivalence is undecidable. The proposed bounded approach constrains the system to a finite state in terms of which analyses may be performed ([Hoa78] also concentrates attention on bounded process activation). Similar approaches are adopted in implementations of concurrency, for example, the `PAR` construct of occam [Bur88], and the `Do in parallel` of HUL [Cla89], both of which need to know the maximum number of processes at compile time.

It should be noted that each process request of tuple space is composed of a sequence of events, the sum of which constitutes the request, and falls within the bounds of the time duration of the tuple space request:

Request	Response from TS	Process Reaction	Request Completed	Completion Signal
$\overline{out_p(u)}$	-	-	*yes*	$\overline{done_p}$
$\overline{inreq_p(u)}$	$\overline{fail}$	*resubmit*	*no*	-
	or			
	$\overline{in_p(u')}$	-	*yes*	$\overline{done_p}$
$\overline{rdreq_p(u)}$	$\overline{fail}$	*resubmit*	*no*	-
	or			
	$\overline{rd_p(u')}$	-	*yes*	$\overline{done_p}$
$\overline{inpreq_p(u)}$	$\overline{fail}$	-	*yes*	$\overline{res_p(false)}$
	or			
	$\overline{inp_p(u')}$	-	*yes*	$\overline{res_p(true)}$
$\overline{rdpreq_p(u)}$	$\overline{fail}$	-	*yes*	$\overline{res_p(false)}$
	or			
	$\overline{rdp_p(u')}$	-	*yes*	$\overline{res_p(true)}$
$\overline{eval_p(u)}$	-	$\overline{start_u}$	*yes*	$\overline{done_p}$

It is important to note that, in any real implementation of Linda, processes are not connected directly to tuple space but are decoupled by some form of tuple space library. The library accepts process requests, passes them on to tuple space, awaits replies which it then routes back to the process. Most importantly, it re-submits unsuccessful **in** and **read** requests - the process itself is not responsible for the implementation of the re-submission process. Similar action applies to the **eval**-primitive - the library is responsible for spawning processes, not the parent process.

A separate tuple space library is coupled to each process so that a typical Linda system is composed of tuple space, multiple tuple space libraries, and as many processes. A single tuple space library through which all process requests are routed does not implement the Linda paradigm - the moment any process executes an unsuccessful **in** or **read** request, the tuple space library does not entertain any

further process requests until the unsuccessful request succeeds (which it never will, since no new tuples can be added to tuple space).

4.4.2.3 Observations and Properties

CCS is a particularly apposite formalism in which to specify the Linda Parallel Programming Paradigm. The basic calculus provides a sufficient set of operators (and from which further operators can be derived), and the full calculus (which includes value-passing communication) provides adequately for the data component of Linda. CCS also provides for the natural expression of the non-deterministic duration of tuple space operations.

Fundamental to CCS is the communication that takes place between agents through complementary-named ports, for example, a and $\bar{a}$. Such communication takes place between agents whenever they are capable of performing the complementary actions, for example:

$$A \stackrel{def}{=} a.m.A$$
$$B \stackrel{def}{=} \bar{a}.n.B$$
$$System \stackrel{def}{=} (A|B)\backslash\{a\}$$

Here A is capable of receiving information on a, and B is capable of delivering information on $\bar{a}$. In the context of *System*, communication takes place between A and B on these ports. Given the following additional definitions:

$$C \stackrel{def}{=} a.q.C$$
$$System1 \stackrel{def}{=} (A|B|C)\backslash\{a\}$$

communication is possible between A and B, and between C and B, using the complementary ports a - $\bar{a}$. A choice is made between the two, at which stage the process not selected to engage in communication is forced to wait until communication is possible.

In the context of *Linda*,

$$Linda \stackrel{def}{=} (TS(M)|Process_1|Process_2| \ldots |Process_n)\backslash L$$

many processes are capable of communicating with tuple space but tuple space is only capable of communicating with any one of the processes at a time - the others are forced to wait their turn. In this way, the actual action of Linda processes is mirrored perfectly.

So that the evolution of an agent may be analysed, the expansion law ([Mil89] page 67) is used.

For the *Linda* agent defined above, the expansion law provides for a full derivation of all Linda programs. For the agent:

$$(TS(M)|Process_1|Process_2)\backslash L$$

the expansion law provides for a full derivation of all the actions in which $Process_1$ and $Process_2$ may engage.

In the following observations, specific actions, as opposed to all possible actions, are examined for the given processes. That is, the agent is examined subject to some sequence of actions, for example:

$Process_1$: **eval**$_1$**(2), out**$_1$**("a"), out**$_1$**("b")** <terminate>
$Process_2$: **in**$_2$**("a")** <terminate>

These particular actions represent specific instances of Linda processes. Derivations based on these actions alone permit observations to be made that are applicable to a subset of all possible actions. A suitable interpretation for "subset" would be "a particular Linda program".

This approach could well be thought of as a shorthand version of:

$$(TS(M)|Process_1|Process_2)\backslash L$$

where: $Process_1 \stackrel{def}{=} \overline{eval_1}(2).\overline{start_2}.\overline{out_1}("a").\overline{done_1}.\overline{out_1}("b").\overline{done_1}.\overline{term_1}.\mathbf{0}$
$Process_2 \stackrel{def}{=} start_2.\overline{inreq_2}("a").(fail.Process_2 + in_2("a").\overline{done_2}.\overline{term_2}.\mathbf{0})$

In other CCS definitions of Linda ([Haz90], [Cia92]), Linda primitives are modelled individually. Process activity is then modelled as a sequence of actions selected from these models, for example, in [Haz90], the following agent represents a Linda system in which two processes and tuple space interact:

$$(TS\langle R,I,T\rangle \mid Out_1("a").Out_1("b").\mathbf{0} \mid In_2(c).\mathbf{0})\backslash L$$

where: $Out_p(x) \stackrel{def}{=} \overline{tadd_p}(x)$
$In_p(x) \stackrel{def}{=} \overline{tgive_p}(x,p).tget_p(y)$

The expansion rule is then used to demonstrate the evolution of the system. The approaches are essentially the same: in this work, the proposed approach selects specific actions, from the set of all possible actions, and in a specific order, whilst the approach that is adopted by [Haz90] selects actions, in their own right, and forms a sequence. A similar course of action is followed by [Cia92].

Given that tuple space is initially empty ($M = \varnothing$), a number of observations[3] can made. The following laws are used in the derivations:

Law 1: Expansion Law

Let $P = (P_1[f_1] \| \ldots \| P_n[f_n])\backslash L$, with $n \geq 1$

Then

$$P = \Sigma \{ f_i(\alpha).(P_1 [f_1] \mid \ldots \mid P_i' [f_i] \mid \ldots \mid P_n [f_n])\backslash L : P_i \xrightarrow{\alpha} P_i',\ f_i(\alpha) \notin L \cup \bar{L}\}$$

$$+ \Sigma \{\tau.(P_1 [f_1] \mid \ldots \mid P_i' [f_i] \mid \ldots \mid P_j' [f_j] \mid \ldots \mid P_n [f_n])\backslash L : P_i \xrightarrow{l_1} P_i',\ P_j \xrightarrow{l_2} P_j',\ f_i(l1) = \overline{f_j(l2)},\ i \neq j\}$$

Where f_i is the identity function Id, and using $P[Id] = P$:

Then

$$P = \Sigma \{\alpha.(P_1 \mid \ldots \mid P_i' \mid \ldots \mid P_n)\backslash L : P_i \xrightarrow{\alpha} P_i',\ \alpha \notin L \cup \bar{L}\}$$

$$+ \Sigma \{\tau.(P_1 \mid \ldots \mid P_i' \mid \ldots \mid P_j' \mid \ldots \mid P_n)\backslash L : P_i \xrightarrow{l} P_i',\ P_j \xrightarrow{\bar{l}} P_j',\ i \neq j\}$$

Law 2: The case of n = 1 of the Expansion Law relating prefix with restriction

$$(\alpha Q)\backslash L = \begin{cases} \mathbf{0} & \text{if } \alpha \in L \cup \bar{L} \\ \alpha Q\backslash L & \text{otherwise} \end{cases}$$

Law 3: A derived law relating prefix with restriction

Let

$$S = (\alpha_1.P_1 \mid \alpha_2.P_2 \mid \ldots \mid \alpha_n.P_n)\backslash L, \text{ with } n \geq 1$$

---

[3] Terms in the expanded agents are numbered for easy reference. For example,

3.1

refers to a term in step *3* of an expansion that is descended from term *1* in the immediately preceding step. Where the immediately preceding term evolves into more than one, say three, terms, the new terms are numbered:

3.1.1
3.1.2
3.1.3

If reference is made to: *3.** it refers to all terms in step *3*.

Then

$$S = \begin{cases} \mathbf{0} & \text{if } (\alpha_1 \in L \cup \bar{L}) \wedge (\alpha_2 \in L \cup \bar{L}) \wedge \ldots \\ & \quad \wedge (\alpha_n \in L \cup \bar{L}) \\ (\alpha_1.P_1 \mid \alpha_2.P_2 \mid \ldots \mid \alpha_n.P_n)\backslash L & \\ \quad \text{otherwise} & \end{cases}$$

Law 4: Monoid Law

$$P + P = P$$

Law 5: Composition Law

$$P|\mathbf{0} = P$$

1. Tuple space preserves tuples.

For a given Linda program that consists of a single process that adds a tuple to tuple space and then retrieves the tuple, the process terminates:

$Process_1$: $\mathbf{out_1}$("a"), $\mathbf{in_1}$("a") <terminate>

	$(TS(M)\|Process_1)\backslash L$	
=	$\tau.(TS(M \uplus \{"a"\})\|ProcessOut_1)\backslash L$	*1.1*
=	$\tau.\overline{done_1}.(TS(M \uplus \{"a"\})\|Process_1)\backslash L$	*2.1*
=	$\tau.\overline{done_1}.\tau.(TSinreq(M \uplus \{"a"\},"a",1)\|ProcessIn_1("a"))\backslash L$	*3.1*
=	$\tau.\overline{done_1}.\tau.\tau.(TS(M)\|\overline{done_1}.Process_1)\backslash L$	*4.1*
=	$\tau.\overline{done_1}.\tau.\tau.\overline{done_1}.(TS(M)\|Process_1)\backslash L$	*5.1*
=	$\tau.\overline{done_1}.\tau.\tau.\overline{done_1}.\overline{term_1}(TS(M)\|\mathbf{0})\backslash L$	*6.1*

using Law 5:

=	$\tau.\overline{done_1}.\tau.\tau.\overline{done_1}.\overline{term_1}(TS(M))\backslash L$	*7.1*

using Law 3:

=	$\mathbf{0}$	*8.1*

Note how the identity, in terms of the originating process, of free tuples is not preserved in

tuple space - free tuples are anonymous.

2. Tuple space does not create tuples.

For a given Linda program that consists of a single process that attempts to retrieve a tuple from tuple space, the process does not termninate but indulges in infinite "internal chatter" (diverges):

$Process_1$: **in**$_1$("a"), **out**$_1$("a") <terminate>

(TS(M)|Process$_1$)\L

= $\tau.(TSinreq(M,"a",1)|ProcessIn_1("a"))\backslash L$ *1.1*

= $\tau.\tau.(TS(M)|\overline{repinreq_1}("a").ProcessIn_1("a"))\backslash L$ *2.1*

= $\tau.\tau.\tau.(TSinreq(M,"a",1)|ProcessIn_1("a"))\backslash L$ *3.1*

= $\tau.\tau.\tau.\tau.(TS(M)|\overline{repinreq_1}("a").ProcessIn_1("a"))\backslash L$ *4.1*

= ...

All further requests to extract tuple (*"a"*) are met with equal failure. This also demonstrates the implicit "busy wait" mechanism of the **in** request. A similar derivation exists for the **read** request with the same result.

3. The specification of tuple space does not guarantee fairness.
For a given Linda program that consists of two processes, one that adds a tuple and one that attempts retrieve the same tuple, both processes terminate, if both are guaranteed tuple space attention, regardless of the order of attention, otherwise not:

$Process_1$: **eval**$_1$(2), **out**$_1$("a") <terminate>
$Process_2$: **in**$_2$("a") <terminate>

3.1 If action **out**$_1$("a") precedes **in**$_2$("a"), the processes terminate:

$(TS(M)|Process_1|Process_2)\backslash L$

= $\tau.(TS(M)|\overline{start_2}.ProcessEval_1|Process_2)L$ *1.1*

= $\tau.\tau.(TS(M)|ProcessEval_1|ProcessSt_2)L$ *2.1*

= $\tau.\tau.\overline{done_1}.(TS(M)|Process_1|ProcessSt_2)L$ *3.1*

$= \quad \tau.\tau.\overline{done_1}.\tau.(TS(M \uplus \{"a"\})|ProcessOut_1|ProcessSt_2)\backslash L$ *4.1*

$= \quad \tau.\tau.\overline{done_1}.\tau.\overline{done_1}.(TS(M \uplus \{"a"\})|Process_1|ProcessSt_2)\backslash L +$ *5.1.1*

$\tau.\tau.\overline{done_1}.\tau.\tau.(TSinreq(M \uplus \{"a"\},"a",2)|ProcessStOut_1|$
$ProcessIn_2("a"))\backslash L$ *5.1.2*

$= \quad \tau.\tau.\overline{done_1}.\tau.\overline{done_1}.\overline{term_1}.(TS(M \uplus \{"a"\})|\mathbf{0}|ProcessSt_2)\backslash L +$ *6.1.1*

$\tau.\tau.\overline{done_1}.\tau.\overline{done_1}.\tau.(TSinreq(M \uplus \{"a"\},"a",2)|Process_1|$
$ProcessStIn_2("a"))\backslash L +$ *6.1.2*

$\tau.\tau.\overline{done_1}.\tau.\tau.\overline{done_1}.(TSinreq(M \uplus \{"a"\},"a",2)|Process_1|$
$ProcessStIn_2("a"))\backslash L +$ *6.2.1*

$\tau.\tau.\overline{done_1}.\tau.\tau.\tau.(TS(M)|ProcessOut_1|\overline{done_2}.ProcessSt_2)\backslash L$ *6.2.2*

$= \quad \tau.\tau.\overline{done_1}.\tau.\overline{done_1}.\overline{term_1}.\tau.(TSinreq(M \uplus \{"a"\},"a",2)|\mathbf{0}|$
$ProcessStIn_2("a"))\backslash L +$ *7.1*

$\tau.\tau.\overline{done_1}.\tau.\overline{done_1}.\tau.\overline{term_1}.(TSinreq(M \uplus \{"a"\},"a",2)|\mathbf{0}|$
$ProcessStIn_2("a"))\backslash L +$ *7.2.1*

$\tau.\tau.\overline{done_1}.\tau.\overline{done_1}.\tau.\tau.(TS(M)|Process_1|\overline{done_2}.ProcessSt_2)\backslash L +$ *7.2.2*

$\tau.\tau.\overline{done_1}.\tau.\tau.\overline{done_1}.\overline{term_1}.(TSinreq(M \uplus \{"a"\},"a",2)|\mathbf{0}|$
$ProcessStIn_2("a"))\backslash L +$ *7.3.1*

$\tau.\tau.\overline{done_1}.\tau.\tau.\overline{done_1}.\tau.(TS(M)|Process_1|\overline{done_2}.ProcessSt_2)\backslash L +$ *7.3.2*

$\tau.\tau.\overline{done_1}.\tau.\tau.\tau.\overline{done_1}.(TS(M)|Process_1|\overline{done_2}.ProcessSt_2)\backslash L +$ *7.4.1*

$\tau.\tau.\overline{done_1}.\tau.\tau.\tau.\overline{done_2}.(TS(M)|ProcessOut_1|ProcessSt_2)\backslash L$ *7.4.2*

$= \quad \tau.\tau.\overline{done_1}.\tau.\overline{done_1}.\overline{term_1}.\tau.\tau.(TS(M)|\mathbf{0}|\overline{done_2}.ProcessSt_2)\backslash L +$ *8.1*

$\tau.\tau.\overline{done_1}.\tau.\overline{done_1}.\tau.\overline{term_1}.\tau.(TS(M)|\mathbf{0}|\overline{done_2}.ProcessSt_2)\backslash L +$ *8.2*

$\tau.\tau.\overline{done_1}.\tau.\overline{done_1}.\tau.\tau.\overline{term_1}.(TS(M)|\mathbf{0}|\overline{done_2}.ProcessSt_2)\backslash L +$ *8.3.1*

$\tau.\tau.\overline{done_1}.\tau.\overline{done_1}.\tau.\tau.\overline{done_2}.(TS(M)|Process_1|ProcessSt_2)\backslash L +$ *8.3.2*

$\tau.\tau.\overline{done_1}.\tau.\tau.\overline{done_1}.\overline{term_1}.\tau.(TS(M)|\mathbf{0}|\overline{done_2}.ProcessSt_2)\backslash L +$ *8.4*

$\tau.\tau.\overline{done_1}.\tau.\tau.\overline{done_1}.\tau.\overline{term_1}.(TS(M)|\mathbf{0}|\overline{done_2}.ProcessSt_2)\backslash L +$ *8.5.1*

$\tau.\tau.\overline{done_1}.\tau.\tau.\overline{done_1}.\tau.\overline{done_2}.(TS(M)|Process_1|ProcessSt_2)\backslash L +$ *8.5.2*

$\tau.\tau.\overline{done_1}.\tau.\tau.\tau.\overline{done_1}.\overline{term_1}.(TS(M)|\mathbf{0}|\overline{done_2}.ProcessSt_2)\backslash L +$ *8.6.1*

$\tau.\tau.\overline{done_1}.\tau.\tau.\tau.\overline{done_1}.\overline{done_2}.(TS(M)|Process_1|ProcessSt_2)\backslash L +$ *8.6.2*

$\tau.\tau.\overline{done_1}.\tau.\tau.\tau.\overline{done_2}.\overline{done_1}.(TS(M)|Process_1|ProcessSt_2)\backslash L +$ *8.7.1*

$\tau.\tau.\overline{done_1}.\tau.\tau.\tau.\overline{done_2}.\overline{term_2}.(TS(M)|ProcessOut_1|\mathbf{0})\backslash L$ *8.7.2*

$= \quad \tau.\tau.\overline{done_1}.\tau.\overline{done_1}.\overline{term_1}.\tau.\tau.\overline{done_2}.(TS(M)|\mathbf{0}|ProcessSt_2)\backslash L +$ *9.1*

$\tau.\tau.\overline{done_1}.\tau.\overline{\text{done}_1}.\tau.\overline{term_1}.\tau.\overline{\text{done}_2}.(TS(M)|\mathbf{0}|ProcessSt_2)\backslash L +$ *9.2*

$\tau.\tau.\overline{done_1}.\tau.\overline{\text{done}_1}.\tau.\tau.\overline{term_1}.\overline{done_2}.(TS(M)|\mathbf{0}|ProcessSt_2)\backslash L +$ *9.3*

$\tau.\tau.\overline{done_1}.\tau.\overline{\text{done}_1}.\tau.\tau.\overline{done_2}.\overline{term_1}.(TS(M)|\mathbf{0}|ProcessSt_2)\backslash L +$ *9.4.1*

$\tau.\tau.\overline{done_1}.\tau.\overline{\text{done}_1}.\tau.\tau.\overline{done_2}.\overline{term_2}.(TS(M)|Process_1|\mathbf{0})\backslash L +$ *9.4.2*

$\tau.\tau.\overline{done_1}.\tau.\tau.\overline{done_1}.\overline{term_1}.\tau.\overline{done_2}.(TS(M)|\mathbf{0}|ProcessSt_2)\backslash L +$ *9.6.5*

$\tau.\tau.\overline{done_1}.\tau.\tau.\overline{done_1}.\tau.\overline{term_1}.\overline{done_2}.(TS(M)|\mathbf{0}|ProcessSt_2)\backslash L +$ *9.6*

$\tau.\tau.\overline{done_1}.\tau.\tau.\overline{done_1}.\tau.\overline{done_2}.\overline{term_1}.(TS(M)|\mathbf{0}|ProcessSt_2)\backslash L +$ *9.7.1*

$\tau.\tau.\overline{done_1}.\tau.\tau.\overline{done_1}.\tau.\overline{done_2}.\overline{term_2}.(TS(M)|Process_1|\mathbf{0})\backslash L +$ *9.7.2*

$\tau.\tau.\overline{done_1}.\tau.\tau.\tau.\overline{done_1}.\overline{term_1}.\overline{done_2}.(TS(M)|\mathbf{0}|ProcessSt_2)\backslash L +$ *9.8*

$\tau.\tau.\overline{done_1}.\tau.\tau.\tau.\overline{done_1}.\overline{done_2}.\overline{term_1}.(TS(M)|\mathbf{0}|ProcessSt_2)\backslash L +$ *9.9.1*

$\tau.\tau.\overline{done_1}.\tau.\tau.\tau.\overline{done_1}.\overline{done_2}.\overline{term_2}.(TS(M)|Process_1|\mathbf{0})\backslash L +$ *9.9.2*

$\tau.\tau.\overline{done_1}.\tau.\tau.\tau.\overline{done_2}.\overline{done_1}.\overline{term_1}.(TS(M)|\mathbf{0}|ProcessSt_2)\backslash L +$ *9.10.1*

$\tau.\tau.\overline{done_1}.\tau.\tau.\tau.\overline{done_2}.\overline{done_1}.\overline{term_2}.(TS(M)|Process_1|\mathbf{0})\backslash L +$ *9.10.2*

$\tau.\tau.\overline{done_1}.\tau.\tau.\tau.\overline{done_2}.\overline{term_2}.\overline{done_1}.(TS(M)|Process_1|\mathbf{0})\backslash L$ *9.11*

$= \quad \tau.\tau.\overline{done_1}.\tau.\overline{done_1}.\overline{term_1}.\tau.\tau.\overline{done_2}.\overline{term_2}.(TS(M)|\mathbf{0}|\mathbf{0})\backslash L +$ *10.1*

$\tau.\tau.\overline{done_1}.\tau.\overline{\text{done}_1}.\tau.\overline{term_1}.\tau.\overline{\text{done}_2}.\overline{term_2}.(TS(M)|\mathbf{0}|\mathbf{0})\backslash L +$ *10.2*

$\tau.\tau.\overline{done_1}.\tau.\overline{\text{done}_1}.\tau.\tau.\overline{term_1}.\overline{done_2}.\overline{term_2}.(TS(M)|\mathbf{0}|\mathbf{0})\backslash L +$ *10.3*

$\tau.\tau.\overline{done_1}.\tau.\overline{\text{done}_1}.\tau.\tau.\overline{done_2}.\overline{term_1}.\overline{term_2}.(TS(M)|\mathbf{0}|\mathbf{0})\backslash L +$ *10.4*

$\tau.\tau.\overline{done_1}.\tau.\overline{\text{done}_1}.\tau.\tau.\overline{done_2}.\overline{term_2}.\overline{term_1}.(TS(M)|\mathbf{0}|\mathbf{0})\backslash L +$ *10.5*

$\tau.\tau.\overline{done_1}.\tau.\tau.\overline{done_1}.\overline{term_1}.\tau.\overline{done_2}.\overline{term_2}.(TS(M)|\mathbf{0}|\mathbf{0})\backslash L +$ *10.6*

$\tau.\tau.\overline{done_1}.\tau.\tau.\overline{done_1}.\tau.\overline{term_1}.\overline{done_2}.\overline{term_2}.(TS(M)|\mathbf{0}|\mathbf{0})\backslash L +$ *10.7*

$\tau.\tau.\overline{done_1}.\tau.\tau.\overline{done_1}.\tau.\overline{done_2}.\overline{term_1}.\overline{term_2}.(TS(M)|\mathbf{0}|\mathbf{0})\backslash L +$ *10.8*

$\tau.\tau.\overline{done_1}.\tau.\tau.\overline{done_1}.\tau.\overline{done_2}.\overline{term_2}.\overline{term_1}.(TS(M)|\mathbf{0}|\mathbf{0})\backslash L +$ *10.9*

$\tau.\tau.\overline{done_1}.\tau.\tau.\tau.\overline{done_1}.\overline{term_1}.\overline{done_2}.\overline{term_2}.(TS(M)|\mathbf{0}|\mathbf{0})\backslash L +$ *10.10*

$\tau.\tau.\overline{done_1}.\tau.\tau.\tau.\overline{done_1}.\overline{done_2}.\overline{term_1}.\overline{term_2}.(TS(M)|\mathbf{0}|\mathbf{0})\backslash L +$ *10.11*

$\tau.\tau.\overline{done_1}.\tau.\tau.\tau.\overline{done_1}.\overline{done_2}.\overline{term_2}.\overline{term_1}.(TS(M)|\mathbf{0}|\mathbf{0})\backslash L +$ *10.12*

$\tau.\tau.\overline{done_1}.\tau.\tau.\tau.\overline{done_2}.\overline{done_1}.\overline{term_1}.\overline{term_2}.(TS(M)|\mathbf{0}|\mathbf{0})\backslash L +$ *10.13*

$\tau.\tau.\overline{done_1}.\tau.\tau.\tau.\overline{done_2}.\overline{done_1}.\overline{term_2}.\overline{term_1}.(TS(M)|\mathbf{0}|\mathbf{0})\backslash L +$ *10.14*

$\tau.\tau.\overline{done_1}.\tau.\tau.\tau.\overline{done_2}.\overline{term_2}.\overline{done_1}.\overline{term_1}.(TS(M)|\mathbf{0}|\mathbf{0})\backslash L$ *10.15*

using Law 5:

$= \quad \tau.\tau.\overline{done_1}.\tau.\overline{done_1}.\overline{term_1}.\tau.\tau.\overline{done_2}.\overline{term_2}.(TS(M))\backslash L +$ *11.1*

$\tau.\tau.\overline{done_1}.\tau.\overline{\text{done}_1}.\tau.\overline{term_1}.\tau.\overline{\text{done}_2}.\overline{term_2}.(TS(M))\backslash L +$ *11.2*

$\tau.\tau.\overline{done_1}.\tau.\overline{\text{done}_1}.\tau.\tau.\overline{term_1}.\overline{done_2}.\overline{term_2}.(TS(M))\backslash L +$ *11.3*

$\tau.\tau.\overline{done_1}.\tau.\overline{\text{done}_1}.\tau.\tau.\overline{done_2}.\overline{term_1}.\overline{term_2}.(TS(M))\backslash L +$ *11.4*

$\tau.\tau.\overline{done_1}.\tau.\overline{\text{done}_1}.\tau.\tau.\overline{done_2}.\overline{term_2}.\overline{term_1}.(TS(M))\backslash L +$ *11.5*

$\tau.\tau.\overline{done_1}.\tau.\tau.\overline{done_1}.\overline{term_1}.\tau.\overline{done_2}.\overline{term_2}.(TS(M))\backslash L +$ *11.6*

$\tau.\tau.\overline{done_1}.\tau.\tau.\overline{done_1}.\tau.\overline{term_1}.\overline{done_2}.\overline{term_2}.(TS(M))\backslash L +$ *11.7*

$\tau.\tau.\overline{done_1}.\tau.\tau.\overline{done_1}.\tau.\overline{done_2}.\overline{term_1}.\overline{term_2}.(TS(M))\backslash L +$ *11.8*

$\tau.\tau.\overline{done_1}.\tau.\tau.\overline{done_1}.\tau.\overline{done_2}.\overline{term_2}.\overline{term_1}.(TS(M))\backslash L +$ *11.9*

$\tau.\tau.\overline{done_1}.\tau.\tau.\tau.\overline{done_1}.\overline{term_1}.\overline{done_2}.\overline{term_2}.(TS(M))\backslash L +$ *11.10*

$\tau.\tau.\overline{done_1}.\tau.\tau.\tau.\overline{done_1}.\overline{done_2}.\overline{term_1}.\overline{term_2}.(TS(M))\backslash L +$ *11.11*

$\tau.\tau.\overline{done_1}.\tau.\tau.\tau.\overline{done_1}.\overline{done_2}.\overline{term_2}.\overline{term_1}.(TS(M))\backslash L +$ *11.12*

$\tau.\tau.\overline{done_1}.\tau.\tau.\tau.\overline{done_2}.\overline{done_1}.\overline{term_1}.\overline{term_2}.(TS(M))\backslash L +$ *11.13*

$\tau.\tau.\overline{done_1}.\tau.\tau.\tau.\overline{done_2}.\overline{done_1}.\overline{term_2}.\overline{term_1}.(TS(M))\backslash L +$ *11.14*

$\tau.\tau.\overline{done_1}.\tau.\tau.\tau.\overline{done_2}.\overline{term_2}.\overline{done_1}.\overline{term_1}.(TS(M))\backslash L$ *11.15*

using Law 3:

$= \quad \mathbf{0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0 + 0}$ *12.1-12.15*

using Law 4:

$= \quad \mathbf{0}$

3.2 If action $\mathbf{in_2}(\text{"a"})$ precedes action $\mathbf{out_1}(\text{"a"})$, and $Process_1$ is guaranteed tuple space attention, the processes terminate:

$(TS(M)|Process_1|Process_2)\backslash L$

$= \quad \tau.(TS(M)|\overline{start_2}.ProcessEval_1|Process_2)L$ *1.1*

$= \quad \tau.\tau.(TS(M)|ProcessEval_1|ProcessSt_2)L$ *2.1*

$= \quad \tau.\tau.\overline{done_1}.(TS(M)|Process_1|ProcessSt_2)L$ *3.1*

$= \quad \tau.\tau.\overline{done_1}.\tau.(TSinreq(M,"a",2)|Process_1|ProcessStIn_2("a"))\backslash L$ *4.1*

$= \quad \tau.\tau.\overline{done_1}.\tau.\tau.(TS(M)|Process_1|\overline{repinreq_2}("a").ProcessStIn_2("a"))\backslash L$ *5.1*

$= \quad \tau.\tau.\overline{done_1}.\tau.\tau.\tau.(TS(M \uplus \{"a"\})|ProcessOut_1|$
$\overline{repinreq_2}("a").ProcessStIn_2("a"))\backslash L$ *6.1*

$= \quad \tau.\tau.\overline{done_1}.\tau.\tau.\tau.\overline{done_1}.(TS(M \uplus \{"a"\})|Process_1|$
$\overline{repinreq_2}("a").ProcessStIn_2("a"))\backslash L +$ *7.1.1*
$\tau.\tau.\overline{done_1}.\tau.\tau.\tau.\tau.(TSinreq(M \uplus \{"a"\},"a",2)|ProcessOut_1|$
$ProcessStIn_2("a"))\backslash L$ *7.1.2*

$= \quad \tau.\tau.\overline{done_1}.\tau.\tau.\tau.\overline{done_1}.\overline{term_1}.(TS(M \uplus \{"a"\})|\mathbf{0}|$
$\overline{repinreq_2}("a").ProcessStIn_2("a"))\backslash L +$ *8.1.1*
$\tau.\tau.\overline{done_1}.\tau.\tau.\tau.\overline{done_1}.\tau.(TSinreq(M \uplus \{"a"\},"a",2)|Process_1|$
$ProcessStIn_2("a"))\backslash L +$ *8.1.2*
$\tau.\tau.\overline{done_1}.\tau.\tau.\tau.\tau.\tau.(TS(M)|ProcessOut_1|\overline{done_2}.ProcessSt_2)\backslash L +$ *8.2.1*
$\tau.\tau.\overline{done_1}.\tau.\tau.\tau.\tau.\overline{done_1}.(TSinreq(M \uplus \{"a"\},"a",2)|Process_1|$
$ProcessStIn_2("a"))\backslash L$ *8.2.2*

$= \quad \tau.\tau.\overline{done_1}.\tau.\tau.\tau.\overline{done_1}.\overline{term_1}.\tau.(TSinreq(M \uplus \{"a"\},"a",2)|\mathbf{0}|$
$ProcessStIn_2("a"))\backslash L +$ *9.1*
$\tau.\tau.\overline{done_1}.\tau.\tau.\tau.\overline{done_1}.\tau.\overline{term_1}.(TSinreq(M \uplus \{"a"\},"a",2)|\mathbf{0}|$
$ProcessStIn_2("a"))\backslash L +$ *9.2.1*
$\tau.\tau.\overline{done_1}.\tau.\tau.\tau.\overline{done_1}.\tau.\tau.(TS(M)|Process_1|\overline{done_2}.ProcessSt_2)\backslash L +$ *9.2.2*
$\tau.\tau.\overline{done_1}.\tau.\tau.\tau.\tau.\overline{done_1}.\overline{term_1}.(TSinreq(M \uplus \{"a"\},"a",2)|\mathbf{0}|$
$ProcessStIn_2("a"))\backslash L +$ *9.4.1*
$\tau.\tau.\overline{done_1}.\tau.\tau.\tau.\tau.\overline{done_1}.\tau.(TS(M)|Process_1|\overline{done_2}.ProcessSt_2)\backslash L +$ *9.4.2*
$\tau.\tau.\overline{done_1}.\tau.\tau.\tau.\tau.\tau.\overline{done_1}.(TS(M)|Process_1|\overline{done_2}.ProcessSt_2)\backslash L +$ *9.3.1*
$\tau.\tau.\overline{done_1}.\tau.\tau.\tau.\tau.\tau.\overline{done_2}.(TS(M)|\overline{done_1}.Process_1|ProcessSt_2)\backslash L$ *9.3.2*

$= \quad \tau.\tau.\overline{done_1}.\tau.\tau.$*(see derivative 3.1 (7.*))*

3.3 If both processes gain tuple space attention non-deterministically, the processes may or may not terminate.

Informally, after process instantiation, the processes behave in the following way (even though some actions are τ-actions, action names from the perspective of the process are used):

either $\overline{out_1}("a").A.B.C.D.E$ end
or $\overline{inreq_2}("a").fail.$ goto Z

Z. either $\overline{out_1}("a").A.B.C.D.E$ end
or $\overline{repinreq_2}("a").fail.$ goto Z

where: $A \stackrel{def}{=} \overline{done_1}$

$B \stackrel{def}{=} \overline{term_1}$

$C \stackrel{def}{=} \overline{inreq_2}("a").in_2("a")$

$D \stackrel{def}{=} \overline{done_2}$

$E \stackrel{def}{=} \overline{term_2}$

A,B,C,D,E can occur in any order as long as A precedes B, and C precedes D precedes E.

The derivations are:

$(TS(M)|Process_1|Process_2)\backslash L$

$= \tau.(TS(M)|\overline{start_2}.ProcessEval_1|Process_2)L$ *1.1*

$= \tau.\tau.(TS(M)|ProcessEval_1|ProcessSt_2)L$ *2.1*

$= \tau.\tau.\overline{done_1}.(TS(M)|Process_1|ProcessSt_2)L$ *3.1*

$= \tau.\tau.\overline{done_1}.\tau.(TS(M \uplus \{"a"\})|ProcessOut_1|ProcessSt_2)\backslash L +$ *4.1.1*

$\tau.\tau.\overline{done_1}.\tau.(TSinreq(M,"a",2)|Process_1|ProcessStIn_2("a"))\backslash L$ *4.1.2*

$=$ *(see derivative 3.1 (4.1)) +* *5.1*

$\tau.\tau.\overline{done_1}.\tau.\tau.(TS(M)|Process_1|\overline{repinreq_2}("a").ProcessStIn_2("a"))\backslash L$ *5.2*

$=$ *(see derivative 3.1 (4.1)) +* *6.1*

$\tau.\tau.\overline{done_1}.\tau.\tau.\tau.(TS(M \uplus \{"a"\})|ProcessOut_1|$

$\overline{repinreq_2}("a").ProcessStIn_2("a"))\backslash L +$ *6.2.1*

$\tau.\tau.\overline{done_1}.\tau.\tau.\tau.(TSinreq(M,"a",2)|Process_1|ProcessStIn_2("a"))\backslash L$ *6.2.2*

$=$ *(see derivative 3.1 (4.1)) +* *7.1*

(see derivative 3.2 (6.1)) + *7.2*

$\tau.\tau.\overline{done_1}.\tau.\tau.$*(see derivative 3.3 (4.1.2))* *7.3*

In more succint terms, zero or more unsuccessful requests to remove tuple (*"a"*) are followed by a single request to add tuple (*"a"*), followed by a single request to remove tuple (*"a"*):

$\{\tau\ \tau\}^{0 \text{ or more}}\ \{\tau\}^1\ \{\tau\ \tau\}^1$

with tuple operation completion ($\overline{done_2}$, $\overline{done_1}$) and process termination ($\overline{term_1}$,

$\overline{term_2}$) signals occurring at the relevant places.

In this case, it is possible that $Process_1$ is never able to gain tuple space attention - $Process_2$ "beats" it to it. However, the moment action $\overline{out_1}("a")$ succeeds, that is, $Process_1$ gains tuple space attention, both actions succeed. It is in this scenario that the more deterministic specification found in [Haz90], in which unsuccessful tuple space requests are blocked (and in which the associated process is suspended), precludes starvation.

4. Tuple space is safe.

No two **in** requests can be met on the same tuple.

For a given Linda program that consists of two processes, both of which attempt to remove the same tuple from tuple space, only one process terminates.

$Process_1$: **out**$_1$("a"), **eval**$_1$(2), **in**$_1$("a") <terminate>
$Process_2$: **in**$_2$("a") <terminate>

$(TS(M)|Process_1|Process_2)\backslash L$

$= \tau.(TS(M \uplus \{"a"\})|ProcessOut_1|Process_2)\backslash L$ 1.1

$= \tau.\overline{done_1}.(TS(M \uplus \{"a"\})|Process_1|Process_2)\backslash L$ 2.1

$= \tau.\overline{done_1}.\tau.(TS(M \uplus \{"a"\})|\overline{start_2}.ProcessEval_1|Process_2)\backslash L$ 3.1

$= \tau.\overline{done_1}.\tau.\tau.(TS(M \uplus \{"a"\})|ProcessEval_1|ProcessSt_2)\backslash L$ 4.1

$= \tau.\overline{done_1}.\tau.\tau.\overline{done_1}.(TS(M \uplus \{"a"\})|Process_1|ProcessSt_2)\backslash L$ 5.1

$= \tau.\overline{done_1}.\tau.\tau.\overline{done_1}.\tau.(TSinreq(M \uplus \{"a"\},"a",1)|ProcessIn_1("a")|$
$ProcessSt_2)\backslash L +$ 6.1.1
$\tau.\overline{done_1}.\tau.\tau.\overline{done_1}.\tau.(TSinreq(M \uplus \{"a"\},"a",2)|Process_1|$
$ProcessStIn_2("a"))\backslash L$ 6.1.2

$= \tau.\overline{done_1}.\tau.\tau.\overline{done_1}.\tau.\tau.(TS(M)|\overline{done_1}.Process_1|ProcessSt_2)\backslash L +$ 7.1
$\tau.\overline{done_1}.\tau.\tau.\overline{done_1}.\tau.\tau.(TS(M)|Process_1|\overline{done_2}.ProcessSt_2)\backslash L$ 7.2

$$= \quad \tau.\overline{done_1}.\tau.\tau.\overline{done_1}.\tau.\tau.\overline{done_1}.(TS(M)|Process_1|ProcessSt_2)\backslash L + \qquad 8.1.1$$

$$\tau.\overline{done_1}.\tau.\tau.\overline{done_1}.\tau.\tau.\tau.(TSinreq(M,"a",2)|\overline{done_1}.Process_1|$$

$$ProcessStIn_2("a"))\backslash L + \qquad 8.1.2$$

$$\tau.\overline{done_1}.\tau.\tau.\overline{done_1}.\tau.\tau.\overline{done_2}.(TS(M)|Process_1|ProcessSt_2)\backslash L \qquad 8.2.1$$

$$\tau.\overline{done_1}.\tau.\tau.\overline{done_1}.\tau.\tau.\tau.(TSinreq(M,"a",1)|ProcessIn_1("a")|$$

$$\overline{done_2}.ProcessSt_2)\backslash L \qquad 8.2.2$$

At this stage, it can be seen that one of the processes has successfully retrieved *"a"* and will terminate, and the other process has already requested or will request *"a"* from an empty tuple space, and will indulge in infinite internal chatter.

The general capability of tuple space is best described as an infinite ability to perform any action selected from a group of actions, coupled with a total inability to perform any other actions. Having engaged in a certain action, tuple space may be obligated to perform certain sub-actions. This general capability is exemplified in the following definition and formula:

$$TSset \quad \stackrel{def}{=} \quad \bigcup_{p \in P} \{out_p, inreq_p, repinreq_p, rdreq_p, reprdreq_p, inpreq_p, rdpreq_p, eval_p\}$$

$$TS(M) \quad \models \quad \nu(X. \quad [-TSset]\mathtt{ff} \quad \wedge$$

$$\bigwedge_{p \in P} \quad (\langle out_p(u)\rangle X \quad \wedge$$

$$\langle inreq_p(u)\rangle$$

$$(\langle \overline{fail}\rangle X \vee \langle \overline{in_p}(u')\rangle X) \quad \wedge$$

$$\langle repinreq_p(u)\rangle$$

$$(\langle \overline{fail}\rangle X \vee \langle \overline{in_p}(u')\rangle X) \quad \wedge$$

$$\langle rdreq_p(u)\rangle$$

$$(\langle \overline{fail}\rangle X \vee \langle \overline{rd_p}(u')\rangle X) \quad \wedge$$

$$\langle reprdreq_p(u)\rangle$$

$$(\langle \overline{fail}\rangle X \vee \langle \overline{rd_p}(u')\rangle X) \quad \wedge$$

$$\langle inpreq_p(u)\rangle$$

$$(\langle \overline{fail}\rangle X \vee \langle \overline{inp_p}(u')\rangle X) \quad \wedge$$

$$\langle rdpreq_p(u)\rangle$$

$$(\langle \overline{fail}\rangle X \vee \langle \overline{rdp_p}(u')\rangle X) \quad \wedge$$

$$\langle eval_p(u)\rangle X))$$

Similarly, the capability of processes is an infinite ability to perform any action selected from a group of actions, coupled with a total inability to perform any other actions. Processes need not perform any particular action, but having engaged in a particular action, the process is obligated to perform certain sub-actions, for example, the repetition that models the capability of the **in** and **read** operations. Processes are also permitted to terminate after which they are incapable of any further action. The

capability of the distinguished process differs from that of spawned processes in respect of its immediate ability to act, versus having to wait until instantiation. The following definitions and formulae exemplify the capability of both the distinguished process and spawned processes ($Process_p$, where $p = 1$ is the distinguished process, and $p > 1$ represents all other processes):

$$
\begin{array}{lcll}
Pset_p & \stackrel{def}{=} & \{\overline{out_p}, \overline{inreq_p}, \overline{rdreq_p}, \overline{inpreq_p}, \overline{rdpreq_p}, \overline{eval_p}, \overline{term_p}\} & \\
PStarted_p & \stackrel{def}{=} & \nu(X. \quad [\text{-}Pset_p]\mathtt{ff} & \wedge \\
 & & \quad <\overline{out_p}(u)><\overline{done_p}>X & \wedge \\
 & & \quad <\overline{inreq_p}(u)> (\nu(Y.(<in_p(u')><\overline{done_p}>X) \vee & \\
 & & \qquad (<fail><\overline{repinreq_p}(u)>Y))) & \wedge \\
 & & \quad <\overline{rdreq_p}(u)> (\nu(Y.(<rd_p(u')><\overline{done_p}>X) \vee & \\
 & & \qquad (<fail><\overline{reprdreq_p}(u)>Y))) & \wedge \\
 & & \quad <\overline{inpreq_p}(u)>((<inp_p(u')><\overline{res_p}(true)>X) \vee & \\
 & & \qquad (<fail><\overline{res_p}(false)>X) & \wedge \\
 & & \quad <\overline{rdpreq_p}(u)>((<rdp_p(u')><\overline{res_p}(true)>X) \vee & \\
 & & \qquad (<fail><\overline{res_p}(false)>X) & \wedge \\
 & & \quad <\overline{eval_p}(u)><\overline{start_u}><\overline{done_p}>X & \wedge \\
 & & \quad <\overline{term_p}>[\text{-}]\mathtt{ff}) & \\
 & & & \\
Process_p & \models & \forall\, p > 1 \qquad <start_p>\, PStarted_p \wedge [\text{-}]\mathtt{ff} & \\
 & & p = 1 \qquad PStarted_p \wedge [\text{-}]\mathtt{ff} &
\end{array}
$$

Once committed to a tuple space operation, that is, communication has taken place on any of the complementary ports (as in, out_p-$\overline{out_p}$, $inreq_p$-$\overline{inreq_p}$, $rdreq_p$-$\overline{rdreq_p}$, $inpreq_p$-$\overline{inpreq_p}$, $rdpreq_p$-$\overline{rdpreq_p}$ and $eval_p$-$\overline{eval_p}$), a process may not engage in any new tuple space request until the current request is completed. Similarly, tuple space may not accept any new requests from any process until it has dealt completely with the current request. This requirement is exemplified in the following definitions and formulae:

$$
\begin{array}{lcll}
Only(K) & \stackrel{def}{=} & (<K>\mathtt{tt} \wedge [\text{-}K]\mathtt{ff}) & \\
 & & & \\
TS(M) & \models & ([\text{-}TSset]\mathtt{ff} & \wedge \\
 & & \wedge_{p \in P} \quad (<out_p(u)>\, Only(TSset) & \wedge \\
 & & \quad <inreq_p(u)>((<\overline{fail}>\, Only(TSset)) \vee & \\
 & & \qquad (<\overline{in_p}(u')>\, Only(TSset))) & \wedge \\
 & & \quad <repinreq_p(u)>((<\overline{fail}>\, Only(TSset)) \vee & \\
 & & \qquad (<\overline{in_p}(u')>\, Only(TSset))) & \wedge \\
 & & \quad <rdreq_p(u)>((<\overline{fail}>\, Only(TSset)) \vee & \\
 & & \qquad (<\overline{rd_p}(u')>\, Only(TSset))) & \wedge \\
 & & \quad <reprdreq_p(u)>((<\overline{fail}>\, Only(TSset)) \vee & \\
 & & \qquad (<\overline{rd_p}(u')>\, Only(TSset))) & \wedge
\end{array}
$$

$$
\begin{aligned}
&\langle inpreq_p(u)\rangle((\langle \overline{fail}\rangle\ Only(TSset)) \vee \\
&\qquad (\langle \overline{inp_p}(u')\rangle\ Only(TSset))) \wedge \\
&\langle rdpreq_p(u)\rangle((\langle \overline{fail}\rangle\ Only(TSset)) \vee \\
&\qquad (\langle \overline{rdp_p}(u')\rangle\ Only(TSset))) \wedge \\
&\langle eval_p(u)\rangle\ Only(TSset)))
\end{aligned}
$$

$$
\begin{aligned}
PCommit_p \quad \stackrel{def}{=} \quad & ([\text{-}Pset_p]\mathtt{ff} \wedge \\
& \langle \overline{out_p}(u)\rangle[\text{-}\overline{done_p}]\mathtt{ff} \wedge \\
& \langle \overline{inreq_p}(u)\rangle\ (\nu X.(\langle in_p(u')\rangle[\text{-}\overline{done_p}]\mathtt{ff}) \vee \\
& \qquad (\langle fail\rangle\langle \overline{repinreq_p}(u)\rangle X))) \wedge \\
& \langle \overline{rdreq_p}(u)\rangle\ (\nu X.(\langle rd_p(u')\rangle[\text{-}\overline{done_p}]\mathtt{ff}) \vee \\
& \qquad (\langle fail\rangle\langle \overline{reprdreq_p}(u)\rangle X))) \wedge \\
& \langle \overline{inpreq_p}(u)\rangle((\langle inp_p(u')\rangle[\text{-}\overline{res_p}(true)]\mathtt{ff}) \vee \\
& \qquad (\langle fail\rangle[\text{-}\overline{res_p}(false)]\mathtt{ff})) \wedge \\
& \langle \overline{rdpreq_p}(u)\rangle((\langle rdp_p(u')\rangle[\text{-}\overline{res_p}(true)]\mathtt{ff}) \vee \\
& \qquad (\langle fail\rangle[\text{-}\overline{res_p}(false)]\mathtt{ff})) \wedge \\
& \langle \overline{eval_p}(u)\rangle\langle \overline{start_u}\rangle[\text{-}\overline{done_p}]\mathtt{ff} \wedge \\
& \langle \overline{term_p}\rangle[\text{-}]\mathtt{ff})
\end{aligned}
$$

$$
\begin{aligned}
Process_p \quad \models \quad & \forall\ p > 1 \quad \langle start_p\rangle\ PCommit_p \wedge [\text{-}]\mathtt{ff} \\
& p = 1 \quad PCommit_p \wedge [\text{-}]\mathtt{ff}
\end{aligned}
$$

4.5 Linda with Debugger

4.5.1 Properties of Linda with Debugger

Section 4.1 detailed the debugging model. During the static phase, a model of the expected behaviour of the program is constructed in terms of Linda program events, that is, Linda primitives. During the dynamic phase, the actual behaviour of the program is compared against the expected behaviour. Specifically, as tuple space attends to requests, the corresponding Linda event is generated and used as the basis of the comparison.

The establishment of a global state space in a sequential programming environment is trivial - the program is suspended at some point, and code and data space is available for inspection with no further effort. In a parallel programming environment, a consistent global state space is hard to establish. Individual processes execute independently, possibly on different processors, with different clocks that are not synchronised, and attempts to suspend execution on all processors simultaneously is complex. Linda presents a very different set of circumstances. Tuple space, in particular, plays a pivotal role. All parallel program activity is controlled by tuple space. Tuple space is the repository for all free tuples, and also controls, or implements, the mechanisms behind which processes are

blocked/unblocked in pursuance of particular tuples. Indeed, at any instance in the program's execution, tuple space is in possession of all relevant information concerning the execution of the program. It alone embodies the global state space, from a parallel point of view, of a Linda program. Furthermore, no extra effort is required to establish the global state - suspension of tuple space activity is sufficient. It may be argued that the suspension of tuple space activity alone, without suspension of any or all process activity, does not lead to a consistent global state space. However, an expressed tenet of the paradigm, as noted in section 4.4.1, is the non-deterministic duration of tuple space operations, where duration includes the time a process waits before tuple space begins to attend to its request. Processes continue to execute until they request tuple space interaction, at which stage they suspend activity whilst tuple space activity is suspended. Such process suspension is indistinguishable from suspension that results from wait times induced when tuple space is attending to other requests.

With the possible exception of programs that execute in environments where hardware-based parallel program debuggers are active, most parallel programs themselves, or the run-time support systems with which they interact, are instrumented to generate notice of the occurrence of program events. In so doing, different programs are produced that execute with different timing constraints and, possibly, execution paths. Since Linda primitives form Linda program events, and since notice of event occurrence is generated within the protective (timing) confines of tuple space, the original and "debugged" versions of the program are indistinguishable, both syntactically and semantically.

All program actions that result in program events are embodied in the Linda primitives, and a single mechanism within tuple space generates all events (there is no need to route events anywhere). Event collection strategies and event collection are therefore unnecessary activities - they are functions of the underlying paradigm.

A major problem faced by parallel debuggers is the determination of the order in which events occur. Elaborate mechanisms exist whereby partial orders of events are provided. For the most part, the partial order is based on causality, for example:

> $Process_A$ sends a message m to $Process_B$, and
> $Process_B$ receives a message m from $Process_A$

the order is:

> send message m
> receive message m

Linear orders of events are difficult to obtain and are normally the preserve of debuggers that operate at the hardware level. Tuple space, on the other hand, presents a natural linear order of events by the manner in which it deals with all Linda primitives one at a time. As each Linda event is generated, a copy is appended to a history file which, at any stage of the program's execution, represents a linear

order of events. Again, no extra effort is required.

4.5.2 Formal Specifications

4.5.2.1 A Formal Specification of Linda with Debugger

The specification of the Linda system that was defined in section 4.4.2.2 is modified to represent the Linda system with the debugger. It involves the introduction of a new agent to represent the debugger, as well as the development of some policy by which the result of the comparison of actual and expected behaviour is communicated to the system.

The debugger, or more specifically the comparison of actual and expected behaviour, can be included in the Linda system in a number of ways, as can the result of the comparison be utilised in many ways with different consequences.

The following factors influence the design of the debugger:

1. Point of inclusion

The protective confine that tuple space provides is fundamental to the approach that is taken in the Linda debugger. The debugger must be included as a function of tuple space (TS), where timing constraints are not prevalent.

2. Nature of the comparison request

The debugger is asked to check whether the behaviour of a particular process is consistent with the behaviour that is expected of that process. The debugger must be provided with the process name and tuple space operation (primitive and tuple). (This fact may clarify the reason why a process identifier ($_p$) is included in all tuple space requests, for example, $\overline{in req_p(u)}$, in the base Linda system - indeed, the monitor-like action of tuple space coupled with the synchronous communication obviates the necessity for process identification.)

3. Role of the debugger

The debugger compares behaviour and generates match/mismatch results. What is the system to do with these results? It can utilise the result to continue execution, on a match, or to terminate some component of the system or the whole system (debugger, tuple space, offending process, all processes), on a mismatch. Alternatively, the result, match or mismatch, can be communicated to the environment after which execution continues. In the first instance, the debugger fulfils an active, influential role, whereas in the second, the debugger fulfils the role of a passive monitor.

There is little merit in the termination of an erroneous process from a suite of processes - there is a close knit relationship between the processes that would be rendered meaningless, if one process is terminated. Either no processes or all processes are terminated. At a much more fundamental level, it is intended that the processes that constitute the Linda program be invariant on application or removal of the debugger. If processes must take cognisance of behavioural comparisons (as they would be obliged to do so if they terminate on mismatch), this would have to be reflected in the very nature of the Linda primitives and would result in physically different processes. (It is possible to argue that all Linda primitives generate some form of reply from tuple space (either an acknowledgement, a boolean result on the predicate forms, or a tuple) and a match/mismatch reply merely adds to the list. Furthermore, even if processes took cognisance of behavioural comparisons, it may be argued that processes ought to be invariant until mismatch occurs, and not thereafter.) If match results are broadcast to the environment but process execution continues regardless, processes remain invariant, and the observer is at liberty to take action or not. In concrete terms, the observer could be faced with a graphic display of tuple space and the debugger, and be prompted with match results generated by the debugger. The results could be ignored (the specification may be incorrect) or the system could be terminated on serious mismatches. From the process point of view, execution appears normal, for example, some requests may block - they unexpectedly request data never in tuple space, or unexpectedly add or remove data, but nonetheless execute Linda primitives in the manner dictated by the paradigm.

The proposed policy[4] requires that tuple space, on receipt of a tuple space request, ask the debugger to check the behaviour. The debugger checks the behaviour, the result is transmitted back to tuple space, and execution continues regardless (essentially a synchronisation event). In the event that the debugger declares a mismatch, it also broadcasts the bad news to the environment. (Bidirectional communication between the debugger and tuple space seems meaningless if tuple space disregards the result. It does, however, require that tuple space wait for the result before attending to the current and subsequent requests. During this time, the debugger can not only conduct the comparison but also gain access to an undisturbed tuple space to conduct any further tests or analyses.)

Using the specification of the base *Linda* system, a Linda system with debugger is developed.

The *Debugger* must check all requests made of tuple space, and register an objection whenever a mismatch occurs between expected and actual behaviour. A suitable sort for *Debugger*:

$$Debugger : \{\bigcup_{p \in P} (checkout_p, checkin_p, checkrd_p, checkinp_p, checkrdp_p, checkeval_p, \overline{failout_p}, \overline{failin_p}, \overline{failrd_p}, \overline{failinp_p}, \overline{failrdp_p}, \overline{faileval_p}) \cup result\}$$

[4] An alternative policy is developed in Appendix C. In this model, it is argued that processes ought to be invariant until a mismatch occurs, at which stage the offending process is terminated. A model is developed that demonstrates such invariance and controls process termination in cases of mismatch, but that still embodies the usual Linda properties.

The debugger is then defined as follows:

$$
\begin{aligned}
Debugger \quad \stackrel{def}{=} \quad & checkout_p(u). \\
& (\overline{result.Debugger} + \\
& \overline{failout_p(u).result.Debugger}) \quad + \\
& checkin_p(u). \\
& (\overline{result.Debugger} + \\
& \overline{failin_p(u).result.Debugger}) \quad + \\
& checkrd_p(u). \\
& (\overline{result.Debugger} + \\
& \overline{failrd_p(u).result.Debugger}) \quad + \\
& checkinp_p(u). \\
& (\overline{result.Debugger} + \\
& \overline{failinp_p(u).result.Debugger}) \quad + \\
& checkrdp_p(u). \\
& (\overline{result.Debugger} + \\
& \overline{failrdp_p(u).result.Debugger}) \quad + \\
& checkeval_p(u). \\
& (\overline{result.Debugger} + \\
& \overline{faileval_p(u).result.Debugger})
\end{aligned}
$$

where: 1. $u \in$ set of all tuples
2. P = set of all process identifiers
3. $p \in P$

Whilst the result is always transmitted back to tuple space, a failure message is also broadcast if a mismatch occurs.

Tuple space (*TS*) is modified to include communication with the debugger. A new tuple space agent (*TSD*) results:

$$
\begin{aligned}
TSD(M) \quad \stackrel{def}{=} \quad & out_p(u).\overline{checkout_p(u)}.result.TSD(M \uplus \{u\}) \quad + \\
& inreq_p(u).\overline{checkin_p(u)}.result.TSDinreq(M, u, p) \quad + \\
& repinreq_p(u).TSDinreq(M, u, p) \quad + \\
& rdreq_p(u).\overline{checkrd_p(u)}.result.TSDrdreq(M, u, p) \quad + \\
& reprdreq(_p(u).TSDrdreq(M, u, p) \quad + \\
& inpreq_p(u).\overline{checkinp_p(u)}.result.TSDinpreq(M, u, p) \quad + \\
& rdpreq_p(u).\overline{checkrdp_p(u)}.result.TSDrdpreq(M, u, p) \quad + \\
& eval_p(u).\overline{checkeval_p(u)}.result.TSD(M)
\end{aligned}
$$

$$TSDinreq(M, u, p) \stackrel{def}{=} \textit{if } match(M, u) = \varnothing \textit{ then } \overline{fail}.TSD(M) \textit{ else } \overline{in_p}(u').TSD(M - \{u'\})$$

$$TSDrdreq(M, u, p) \stackrel{def}{=} \textit{if } match(M, u) = \varnothing \textit{ then } \overline{fail}.TSD(M) \textit{ else } \overline{rd_p}(u').TSD(M)$$

$$TSDinpreq(M, u, p) \stackrel{def}{=} \textit{if } match(M, u) = \varnothing \textit{ then } \overline{fail}.TSD(M) \textit{ else } \overline{inp_p}(u').TSD(M - \{u'\})$$

$$TSDrdpreq(M, u, p) \stackrel{def}{=} \textit{if } match(M, u) = \varnothing \textit{ then } \overline{fail}.TSD(M) \textit{ else } \overline{rdp_p}(u').TSD(M)$$

where: 1. $u \in$ set of all tuples
2. $u' \in match(M, u)$
3. P = set of all process identifiers
4. $p \in P$

Note how, on re-submission of unsuccessful **in** and **read** requests, the debugger is not called to check the validity of the operation - the check is performed the first time the request is considered. Agent name changes (*TS* to *TSD*, and *TSinreq* to *TSDinreq*, etc.) account for the only difference between the definitions for *TSDinreq*, *TSDrdreq*, *TSDinpreq*, and *TSDrdpreq* and the definitions for *TSinreq*, *TSrdreq*, *TSinpreq* and *TSrdpreq*. The definitions of all agents in the process collection ($Process_1$, $Process_p$) remain the same.

The Linda system with debugger is then defined as follows[5]:

$$LindaD \stackrel{def}{=} ((TSD(M)|Debugger)\backslash Ld| Process_1|Process_2| \ldots |Process_n)\backslash L$$

where: 1. $Ld = \{\bigcup_{p \in P} (checkout_p, checkin_p, checkrd_p, checkinp_p, checkrdp_p, checkeval_p) \cup result\}$
2. $L = \{\bigcup_{p \in P} (out_p, inreq_p, repinreq_p, in_p, rdreq_p, reprdreq_p, rd_p, inpreq_p, inp_p, rdpreq_p, rdp_p, eval_p) \cup fail\}$

Although it is not intended that *LindaD* include facilities for the construction of a history file, it is

[5] A full specification of the Linda system with debugger can be found in Appendix B.

worthwhile to consider what mechanisms would be required to support such a venture.

A history file is merely a collection of events that took place during some execution of a program. The value of the file is vested in how accurately it reflects the actual order in which events took place. In the context of *LindaD*, a history file can be generated that reflects a linear order of events - tuple space operation indivisibility, and the fact that tuple space deals completely with the current request before it attends to any new request ensures this. *TSD(M)* and *Debugger* can be modified so that each time an event of some significance occurs, notice of the event is posted to a history agent. Significant events might include:

1. initial process request,
2. behaviour mismatch, and
3. result of tuple space operation.

The first two could be accommodated accordingly:

$$
\begin{aligned}
Debugger \quad \stackrel{def}{=} \quad & checkout_p(u). \\
& (\overline{result.Debugger} + \\
& \overline{histfailout_p}(u).\overline{failout_p}(u). \\
& \qquad \overline{result.Debugger}) \quad + \\
& checkin_p(u). \\
& (\overline{result.Debugger} + \\
& \overline{histfailin_p}(u).\overline{failin_p}(u). \\
& \qquad \overline{result.Debugger}) \quad + \\
& checkrd_p(u). \\
& (\overline{result.Debugger} + \\
& \overline{histfailrd_p}(u).\overline{failrd_p}(u). \\
& \qquad \overline{result.Debugger}) \quad + \\
& checkinp_p(u). \\
& (\overline{result.Debugger} + \\
& \overline{histfailinp_p}(u).\overline{failinp_p}(u). \\
& \qquad \overline{result.Debugger}) \quad + \\
& checkrdp_p(u). \\
& (\overline{result.Debugger} + \\
& \overline{histfailrdp_p}(u).\overline{failrdp_p}(u). \\
& \qquad \overline{result.Debugger}) \quad + \\
& checkeval_p(u). \\
& (\overline{result.Debugger} + \\
& \overline{histfaileval_p}(u).\overline{faileval_p}(u). \\
& \qquad \overline{result.Debugger})
\end{aligned}
$$

$$
\begin{aligned}
TSD(M) \quad \stackrel{def}{=} \quad & out_p(u).\overline{histout_p}(u). \\
& \quad \overline{checkout_p}(u).result.TSD(M \uplus \{u\}) \quad + \\
& inreq_p(u).\overline{histin_p}(u). \\
& \quad \overline{checkin_p}(u).result.TSDinreq(M, u, p) \quad + \\
& repinreq_p(u).\overline{histrepin_p}(u). \\
& \quad TSDinreq(M, u, p) \quad + \\
& rdreq_p(u).\overline{histrd_p}(u). \\
& \quad \overline{checkrd_p}(u).result.TSDrdreq(M, u, p) \quad + \\
& reprdreq(_p(u).\overline{histreprd_p}(u). \\
& \quad TSDrdreq(M, u, p) \quad + \\
& inpreq_p(u).\overline{histinp_p}(u). \\
& \quad \overline{checkinp_p}(u).result.TSDinpreq(M, u, p) \quad + \\
& rdpreq_p(u).\overline{histrdp_p}(u). \\
& \quad \overline{checkrdp_p}(u).result.TSDrdpreq(M, u, p) \quad + \\
& eval_p(u).\overline{histeval_p}(u). \\
& \quad \overline{checkeval_p}(u).result.TSD(M)
\end{aligned}
$$

$$
\begin{aligned}
History \quad \stackrel{def}{=} \quad \Sigma_{p \in P} \; & (histout_p(u).History \quad + \\
& \quad histin_p(u).History \quad + \\
& \quad histrepin_p(u).History \quad + \\
& \quad histrd_p(u).History \quad + \\
& \quad histreprd_p(u).History \quad + \\
& \quad histinp_p(u).History \quad + \\
& \quad histrdp_p(u).History \quad + \\
& \quad histeval_p(u).History \quad + \\
& \quad histfailout_p(u).History \quad + \\
& \quad histfailin_p(u).History \quad + \\
& \quad histfailrd_p(u).History \quad + \\
& \quad histfailinp_p(u).History \quad + \\
& \quad histfailrdp_p(u).History \quad + \\
& \quad histfaileval_p(u).History)
\end{aligned}
$$

where: *History* can be incorporated into *LindaD* which is then further restricted by the sort of *History*.

4.5.2.2 Observations and Properties

In section 4.4.2.3 a number of observations and properties were noted of *Linda*, specifically of tuple space and processes. Issues of tuple creation and preservation, lack of tuple space fairness, and tuple space safety were investigated. In *LindaD*, tuple space and processes exhibit the same properties. The

derivation sequences that may be developed differ from those in section 4.4.3.2 in so far as extra terms result from behavioural mismatches, and added silent activity that occurs as a result of hidden communication with the debugger.

For example, that tuple space preserves tuples is shown as:

$Process_1$: **out$_1$("a"), in$_1$("a")** <terminate>

$(TSD(M)|Debugger)\backslash Ld|Process_1)\backslash L$

$= \tau.((\overline{checkout_1}("a").result.TSD(M \uplus \{"a"\})|Debugger)\backslash Ld|$
$ProcessOut_1)\backslash L$ 1.1

$= \tau.\tau.((result.TSD(M \uplus \{"a"\})|(\overline{result}.Debugger +$
$\overline{failout_1}("a").\overline{result}.Debugger))\backslash Ld|ProcessOut_1)\backslash L +$ 2.1.1
$\tau.\overline{done_1}.((\overline{checkout_1}("a").result.TSD(M \uplus \{"a"\})|Debugger)\backslash Ld|$
$Process_1)\backslash L$ 2.1.2

$= \tau.\tau.\tau.((TSD(M \uplus \{"a"\})|Debugger)\backslash Ld|ProcessOut_1)\backslash L +$ 3.1.1
$\tau.\tau.\overline{failout_1}("a").((result.TSD(M \uplus \{"a"\})|\overline{result}.Debugger))\backslash Ld|$
$ProcessOut_1)\backslash L +$ 3.1.2
$\tau.\tau.\overline{done_1}.((result.TSD(M \uplus \{"a"\})|(\overline{result}.Debugger +$
$\overline{failout_1}("a").\overline{result}.Debugger))\backslash Ld|Process_1)\backslash L +$ 3.1.3
$\tau.\overline{done_1}.\tau.((result.TSD(M \uplus \{"a"\})|(\overline{result}.Debugger +$
$\overline{failout_1}("a").\overline{result}.Debugger))\backslash Ld|Process_1)\backslash L$ 3.2

$= \tau.\tau.\tau.\overline{done_1}.((TSD(M \uplus \{"a"\})|Debugger)\backslash Ld|Process_1)\backslash L +$ 4.1
$\tau.\tau.\overline{failout_1}("a").\tau.((TSD(M \uplus \{"a"\})|Debugger))\backslash Ld|$
$ProcessOut_1)\backslash L +$ 4.2.1
$\tau.\tau.\overline{failout_1}("a").\overline{done_1}.((result.TSD(M \uplus \{"a"\})|\overline{result}.Debugger))\backslash Ld|$
$Process_1)\backslash L +$ 4.2.2
$\tau.\tau.\overline{done_1}.\tau.((TSD(M \uplus \{"a"\})|Debugger)\backslash Ld|Process_1)\backslash L +$ 4.3.1
$\tau.\tau.\overline{done_1}.\overline{failout_1}("a").((result.TSD(M \uplus \{"a"\})|\overline{result}.Debugger)\backslash Ld|$
$Process_1)\backslash L +$ 4.3.2
$\tau.\overline{done_1}.\tau.\tau.((TSD(M \uplus \{"a"\})|Debugger)\backslash Ld|Process_1)\backslash L +$ 4.4.1
$\tau.\overline{done_1}.\tau.\overline{failout_1}("a").((result.TSD(M \uplus \{"a"\})|\overline{result}.Debugger)\backslash Ld|$
$Process_1)\backslash L$ 4.4.2

$= \quad \tau.\tau.\tau.\overline{done_1}.\tau.((\overline{checkin_1}("a").result.TSDinreq(M \uplus \{"a"\},"a",1)|Debugger)\backslash Ld|$
$ProcessIn_1("a"))\backslash L +$ 5.1
$\tau.\tau.\overline{failout_1}("a").\tau.\overline{done_1}.((TSD(M \uplus \{"a"\})|Debugger)\backslash Ld|$
$Process_1)\backslash L +$ 5.2
$\tau.\tau.\overline{failout_1}("a").\overline{done_1}.\tau.((TSD(M \uplus \{"a"\})|Debugger)\backslash Ld|$
$Process_1)\backslash L +$ 5.3
$\tau.\tau.\overline{done_1}.\tau.\tau.((\overline{checkin_1}("a").result.TSDinreq(M \uplus \{"a"\},"a",1)|Debugger)\backslash Ld|$
$ProcessIn_1("a"))\backslash L +$ 5.4
$\tau.\tau.\overline{done_1}.\overline{failout_1}("a").\tau.((TSD(M \uplus \{"a"\})|Debugger)\backslash Ld|$
$Process_1)\backslash L +$ 5.5
$\tau.\overline{done_1}.\tau.\tau.\tau.((\overline{checkin_1}("a").result.TSDinreq(M \uplus \{"a"\},"a",1)|Debugger)\backslash Ld|$
$ProcessIn_1("a"))\backslash L +$ 5.6
$\tau.\overline{done_1}.\tau.\overline{failout_1}("a").\tau.((TSD(M \uplus \{"a"\})|Debugger)\backslash Ld|$
$Process_1)\backslash L$ 5.7

Two distinct terms emerge (renumbered for convenience):

$((\overline{checkin_1}("a").result.TSDinreq(M \uplus \{"a"\},"a",1)|Debugger)\backslash Ld|$
$ProcessIn_1("a"))\backslash L +$ 5.1
$((TSD(M \uplus \{"a"\})|Debugger)\backslash Ld|Process_1)\backslash L$ 5.2

$= \quad \tau.((result.TSDinreq(M \uplus \{"a"\},"a",1)|(\overline{result.}Debugger +$
$\overline{failin_1}("a").\overline{result.}Debugger))\backslash Ld|ProcessIn_1("a"))\backslash L +$ 6.1
$\tau.((\overline{checkin_1}("a").result.TSDinreq(M \uplus \{"a"\},"a",1)|Debugger)\backslash Ld|$
$ProcessIn_1("a"))\backslash L +$ 6.2

$= \quad \tau.\tau.((TSDinreq(M \uplus \{"a"\},"a",1)|Debugger)\backslash Ld|ProcessIn_1("a"))\backslash L +$ 7.1.1
$\tau.\overline{failin_1}("a").((result.TSDinreq(M \uplus \{"a"\},"a",1)|\overline{result.}Debugger))\backslash Ld|$
$ProcessIn_1("a"))\backslash L +$ 7.1.2
$\tau.\tau.((result.TSDinreq(M \uplus \{"a"\},"a",1)|(\overline{result.}Debugger +$
$\overline{failin_1}("a").\overline{result.}Debugger))\backslash Ld|ProcessIn_1("a"))\backslash L$ 7.2

$= \quad \tau.\tau.\tau.((TSD(M)|Debugger)\backslash Ld|\overline{done_1}.Process_1)\backslash L +$ 8.1
$\tau.\overline{failin_1}("a").\tau.((TSDinreq(M \uplus \{"a"\},"a",1)|Debugger)\backslash Ld|$
$ProcessIn_1("a"))\backslash L +$ 8.2
$\tau.\tau.\tau.((TSDinreq(M \uplus \{"a"\},"a",1)|Debugger)\backslash Ld|ProcessIn_1("a"))\backslash L +$ 8.3.1
$\tau.\tau.\overline{failin_1}("a").((result.TSDinreq(M \uplus \{"a"\},"a",1)|\overline{result.}Debugger))\backslash Ld|$
$ProcessIn_1("a"))\backslash L$ 8.3.2

$= \quad \tau.\tau.\tau.\overline{done_1}.((TSD(M)|Debugger)\backslash Ld|Process_1)\backslash L +$ *9.1*

$\tau.\overline{failin_1}("a").\tau.\tau.((TSD(M)|Debugger)\backslash Ld|\overline{done_1}.Process_1)\backslash L +$ *9.2*

$\tau.\tau.\tau.\tau.((TSD(M)|Debugger)\backslash Ld|\overline{done_1}.Process_1)\backslash L +$ *9.3*

$\tau.\tau.\overline{failin_1}("a").\tau.((TSDinreq(M \uplus \{"a"\},"a",1)|Debugger)\backslash Ld|$
$ProcessIn_1("a"))\backslash L$ *9.4*

$= \quad \tau.\tau.\tau.\overline{done_1}.\overline{term_1}.((TSD(M)|Debugger)\backslash Ld|\mathbf{0})\backslash L +$ *10.1*

$\tau.\overline{failin_1}("a").\tau.\tau.\overline{done_1}.((TSD(M)|Debugger)\backslash Ld|Process_1)\backslash L +$ *10.2*

$\tau.\tau.\tau.\tau.\overline{done_1}.((TSD(M)|Debugger)\backslash Ld|Process_1)\backslash L +$ *10.3*

$\tau.\tau.\overline{failin_1}("a").\tau.\tau.((TSD(M)|Debugger)\backslash Ld|\overline{done_1}.Process_1)\backslash L$ *10.4*

$= \quad \tau.\tau.\tau.\overline{done_1}.\overline{term_1}.((TSD(M)|Debugger)\backslash Ld|\mathbf{0})\backslash L +$ *11.1*

$\tau.\overline{failin_1}("a").\tau.\tau.\overline{done_1}.\overline{term_1}.((TSD(M)|Debugger)\backslash Ld|\mathbf{0})\backslash L +$ *11.2*

$\tau.\tau.\tau.\tau.\overline{done_1}.\overline{term_1}.((TSD(M)|Debugger)\backslash Ld|\mathbf{0})\backslash L +$ *11.3*

$\tau.\tau.\overline{failin_1}("a").\tau.\tau.\overline{done_1}.((TSD(M)|Debugger)\backslash Ld|Process_1)\backslash L$ *11.4*

$= \quad \tau.\tau.\tau.\overline{done_1}.\overline{term_1}.((TSD(M)|Debugger)\backslash Ld|\mathbf{0})\backslash L +$ *12.1*

$\tau.\overline{failin_1}("a").\tau.\tau.\overline{done_1}.\overline{term_1}.((TSD(M)|Debugger)\backslash Ld|\mathbf{0})\backslash L +$ *12.2*

$\tau.\tau.\tau.\tau.\overline{done_1}.\overline{term_1}.((TSD(M)|Debugger)\backslash Ld|\mathbf{0})\backslash L +$ *12.3*

$\tau.\tau.\overline{failin_1}("a").\tau.\tau.\overline{done_1}.\overline{term_1}.((TSD(M)|Debugger)\backslash Ld|\mathbf{0})\backslash L$ *12.4*

using Law 5:

$= \quad \tau.\tau.\tau.\overline{done_1}.\overline{term_1}.((TSD(M)|Debugger)\backslash Ld)\backslash L +$ *13.1*

$\tau.\overline{failin_1}("a").\tau.\tau.\overline{done_1}.\overline{term_1}.((TSD(M)|Debugger)\backslash Ld)\backslash L +$ *13.2*

$\tau.\tau.\tau.\tau.\overline{done_1}.\overline{term_1}.((TSD(M)|Debugger)\backslash Ld)\backslash L +$ *13.3*

$\tau.\tau.\overline{failin_1}("a").\tau.\tau.\overline{done_1}.\overline{term_1}.((TSD(M)|Debugger)\backslash Ld)\backslash L$ *13.4*

using Law 3:

$= \quad \mathbf{0} + \mathbf{0} + \mathbf{0} + \mathbf{0}$ *14.1-14.4*

using Law 4:

$= \quad \mathbf{0}$

It should be noted that the properties hold, regardless of any possible inconsistency between actual and expected behaviour.

The general capability of tuple space remains invariant under *Linda* and *LindaD*. Tuple space's ability to perform certain activities and its inability to perform any other activities is maintained. The assertion that certain sub-actions be performed consequent to initial communications is also preserved.

$$TSset \stackrel{def}{=} \bigcup_{p \in P} \{out_p, inreq_p, repinreq_p, rdreq_p, rerdreq_p, inpreq_p, rdpreq_p, eval_p\}$$

$$\begin{aligned} TSD(M) \models \nu(X.\ & [\text{-}TSset]\mathtt{ff} \wedge \\ & \bigwedge_{p \in P} (\langle out_p(u)\rangle \langle\tau\rangle \langle\tau\rangle X \vee \\ & \langle inreq_p(u)\rangle \langle\tau\rangle \langle\tau\rangle \\ & \quad (\langle \overline{fail}\rangle X \vee \langle \overline{in_p}(u')\rangle X) \wedge \\ & \langle repinreq_p(u)\rangle \\ & \quad (\langle \overline{fail}\rangle X \vee \langle \overline{in_p}(u')\rangle X) \wedge \\ & \langle rdreq_p(u)\rangle \langle\tau\rangle \langle\tau\rangle \\ & \quad (\langle \overline{fail}\rangle X \vee \langle \overline{rd_p}(u')\rangle X) \wedge \\ & \langle reprdreq_p(u)\rangle \\ & \quad (\langle \overline{fail}\rangle X \vee \langle \overline{rd_p}(u')\rangle X) \wedge \\ & \langle inpreq_p(u)\rangle \langle\tau\rangle \langle\tau\rangle \\ & \quad (\langle \overline{fail}\rangle X \vee \langle \overline{inp_p}(u')\rangle X) \wedge \\ & \langle rdpreq_p(u)\rangle \langle\tau\rangle \langle\tau\rangle \\ & \quad (\langle \overline{fail}\rangle X \vee \langle \overline{rdp_p}(u')\rangle X) \wedge \\ & \langle eval_p(u)\rangle \langle\tau\rangle \langle\tau\rangle X)) \end{aligned}$$

The definition for tuple space maintains the necessity to complete all the processing that is related to the current request before a new request is attempted:

$$Only(K) \stackrel{def}{=} (\langle K\rangle \mathtt{tt} \wedge [\text{-}K]\mathtt{ff})$$

$$\begin{aligned} TSD(M) \models ([\text{-}TSset]\mathtt{ff} \wedge & \\ \bigwedge_{p \in P} (\langle out_p(u)\rangle \langle\tau\rangle \langle\tau\rangle & Only(TSset) \wedge \\ \langle inreq_p(u)\rangle \langle\tau\rangle \langle\tau\rangle & \\ (\langle \overline{fail}\rangle & Only(TSset) \vee \\ \langle \overline{in_p}(u')\rangle & Only(TSset)) \wedge \\ \langle repinreq_p(u)\rangle & \\ (\langle \overline{fail}\rangle & Only(TSset) \vee \\ \langle \overline{in_p}(u')\rangle & Only(TSset)) \wedge \\ \langle rdreq_p(u)\rangle \langle\tau\rangle \langle\tau\rangle & \\ (\langle \overline{fail}\rangle & Only(TSset) \vee \\ \langle \overline{rd_p}(u')\rangle & Only(TSset)) \wedge \end{aligned}$$

$$
\begin{aligned}
&\langle reprdreq_p(u)\rangle \\
&\qquad (\langle \overline{fail}\rangle\ Only(TSset) \vee \\
&\qquad \langle \overline{rd_p}(u')\rangle\ Only(TSset)) \qquad \wedge \\
&\langle inpreq_p(u)\rangle\ \langle\tau\rangle\ \langle\tau\rangle \\
&\qquad (\langle \overline{fail}\rangle\ Only(TSset) \vee \\
&\qquad \langle \overline{inp_p}(u')\rangle\ Only(TSset)) \qquad \wedge \\
&\langle rdpreq_p(u)\rangle\ \langle\tau\rangle\ \langle\tau\rangle \\
&\qquad (\langle \overline{fail}\rangle\ Only(TSset) \vee \\
&\qquad \langle \overline{rdp_p}(u')\rangle\ Only(TSset)) \qquad \wedge \\
&\langle eval_p(u)\rangle\ \langle\tau\rangle\ \langle\tau\rangle\ Only(TSset))
\end{aligned}
$$

Tuple space alone generates notice of the occurrence of all program events. Immediately after tuple space receives a request (excluding re-submitted requests) from a process, notice of the event is generated. An event is never generated at any other time:

$$
\begin{aligned}
&EventOut \quad \stackrel{def}{=} \quad \bigcup_{p \in P} \{\overline{checkout_p}, \overline{checkin_p}, \overline{checkrd_p}, \overline{checkinp_p}, \overline{checkrdp_p}, \overline{checkeval_p}\} \\
&EventGenerators \quad \stackrel{def}{=} \quad \bigcup_{p \in P} \{out_p, inreq_p, rdreq_p, inpreq_p, rdpreq_p, eval_p\} \\
&RepeatRequests \quad \stackrel{def}{=} \quad \bigcup_{p \in P} \{repinreq_p, reprdreq_p\} \\
&TSD(M) \quad \models \quad \nu(X. \quad (\langle\text{->}\mathtt{tt} \qquad \wedge \\
&\qquad [EventGenerators]\langle EventOut\rangle\ X \qquad \wedge \\
&\qquad [EventOut]\mathtt{ff} \qquad \wedge \\
&\qquad [RepeatRequests]\ X \qquad \wedge \\
&\qquad [\text{-}(EventGenerators \cup RepeatRequests \cup EventOut)]\ X))
\end{aligned}
$$

Neither the debugger nor any process is capable of generating notice of an event:

$$
\begin{aligned}
&Debugger \quad \models \quad \nu(X. \quad ([EventOut]\mathtt{ff} \wedge [\text{-}]X)) \\
&Process_p \quad \models \quad \nu(X. \quad ([EventOut]\mathtt{ff} \wedge [\overline{term_p}][\text{-}]\mathtt{ff} \wedge [\text{-}\overline{term_p}]X))
\end{aligned}
$$

Notice of the occurrence of an event is communicated to the debugger, where it is used to check expected versus actual behaviour:

$$
EventIn \quad \stackrel{def}{=} \quad \bigcup_{p \in P} \{checkout_p, checkin_p, checkrd_p, checkinp_p, checkrdp_p, checkeval_p\}
$$

$$
\begin{aligned}
\textit{Debugger} \quad \models \quad \nu(X. \quad & (<\textit{EventIn}> (<\overline{result}>X \vee \\
& \bigvee_{p \in P} \; (<\overline{failout_p}><\overline{result}> X \vee \\
& \quad <\overline{failin_p}><\overline{result}> X \vee \\
& \quad <\overline{failrd_p}><\overline{result}> X \vee \\
& \quad <\overline{failinp_p}><\overline{result}> X \vee \\
& \quad <\overline{failrdp_p}><\overline{result}> X \vee \\
& \quad <\overline{faileval_p}><\overline{result}> X))))
\end{aligned}
$$

Neither tuple space (obviously) nor any process is capable of receiving notice of the event:

$$\textit{TSD(M)} \quad \models \quad \nu(X. \quad ([\textit{EventIn}]\mathtt{ff} \wedge [-]X))$$

$$\textit{Process}_p \quad \models \quad \nu(X. \quad ([\textit{EventIn}]\mathtt{ff} \wedge [\overline{term_p}][-]\mathtt{ff} \wedge [-\overline{term_p}]X))$$

4.6 A Comparison of the Behaviour of Linda and Linda with Debugger

[Mil89] describes equivalence relations (particularly observational equivalence) between agents that is based, intuitively, on an equivalence between agents in terms of observable action capabilities. It is instructive to explore observational equivalence ($\approx$) between *Linda* and *LindaD*, that is:

$$\textit{Linda} \approx \textit{LindaD}$$

The two agents are defined as:

$$\textit{Linda} \quad \stackrel{\text{def}}{=} \quad (TS(M)|Process_1|Process_2| \ldots | Process_n)\backslash L$$

$$
\begin{aligned}
\textit{LindaD} \quad \stackrel{\text{def}}{=} \quad & ((TSD(M)|Debugger)\backslash Ld| \\
& Process_1|Process_2| \ldots |Process_n)\backslash L
\end{aligned}
$$

where: 1. $Ld = \{\bigcup_{p \in P} (checkout_p, checkin_p, checkrd_p, checkinp_p, checkrdp_p, checkeval_p) \cup result\}$

2. $L = \{\bigcup_{p \in P} (out_p, inreq_p, repinreq_p, in_p, rdreq_p, reprdreq_p, rd_p, inpreq_p, inp_p, rdpreq_p, rdp_p, eval_p) \cup fail\}$

An essential and desirable property of the specification of the Linda system and the Linda system with debugger is the invariance of the specification of processes (both the distinguished process and spawned processes). Since composition and restriction preserve bisimilarity ([Mil89] page 113), it is sufficient to consider:

$$TS(M) \approx (TSD(M)|Debugger)\backslash Ld$$

instead of *Linda* and *LindaD*.

Analysis (restrictions are omitted for brevity):

1. $TS(M)$ - left member

Initial actions ($\overline{out_p}$ $\overline{inreq_p}$ $\overline{repinreq_p}$ $\overline{rdreq_p}$ $\overline{reprdreq_p}$ $\overline{inpreq_p}$ $\overline{rdpreq_p}$ $\overline{eval_p}$) of the left member:

2. $TS(M \uplus \{u\})$
3. $TSinreq(M, u, p)$
4. $TSinreq(M, u, p)$
5. $TSrdreq(M, u, p)$
6. $TSrdreq(M, u, p)$
7. $TSinpreq(M, u, p)$
8. $TSrdpreq(M, u, p)$
9. $TS(M)$

1. $TSD(M)|Debugger$ - right member

The actions ($\overline{out_p}$ $\overline{inreq_p}$ $\overline{repinreq_p}$ $\overline{rdreq_p}$ $\overline{reprdreq_p}$ $\overline{inpreq_p}$ $\overline{rdpreq_p}$ $\overline{eval_p}$) are matched in the right member by:

2. $\overline{checkout_p}(u).result.TSD(M \uplus \{u\})|Debugger$
3. $\overline{checkin_p}(u).result.TSDinreq(M, u, p)|Debugger$
4. $TSDinreq(M, u, p)|Debugger$
5. $\overline{checkrd_p}(u).result.TSDrdreq(M, u, p)|Debugger$
6. $TSDrdreq(M, u, p)|Debugger$
7. $\overline{checkinp_p}(u).result.TSDinpreq(M, u, p)|Debugger$
8. $\overline{checkrdp_p}(u).result.TSDrdpreq(M, u, p)|Debugger$
9. $\overline{checkeval_p}(u).result.TSD(M)Debugger$

If no mismatch is encountered, the right member engages in silent activity (τ - request to check; τ - reply), which is matched in the left member by ξ (no activity):

10. $TS(M \uplus \{u\})$
11. $TSinreq(M, u, p)$
12. $TSinreq(M, u, p)$
13. $TSrdreq(M, u, p)$
14. $TSrdreq(M, u, p)$
15. $TSinpreq(M, u, p)$

16. $TSrdpreq(M, u, p)$
17. $TS(M)$

10. $TSD(M \uplus \{u\})|Debugger$
11. $TSDinreq(M, u, p)|Debugger$
12. $TSDinreq(M, u, p)|Debugger$
13. $TSDrdreq(M, u, p)|Debugger$
14. $TSDrdreq(M, u, p)|Debugger$
15. $TSDinpreq(M, u, p)|Debugger$
16. $TSDrdpreq(M, u, p)|Debugger$
17. $TSD(M)|Debugger$

If a mismatch is encountered, the right member engages in a silent action (τ - request to check), a failure signal, and a further silent action (τ - reply). The silent activity is matched in the left member by ξ but there is no equivalent for the failure signal.

From *10, 11, 12, 13, 14, 15, 16,* and *17,* equivalent patterns of behaviour are exhibited that are matched by both the left and right members.

Since the failure signal of the right member is not matched by the left member:

$$TS(M) \not\approx (TSD(M)|Debugger)\backslash Ld$$

and hence:

$$Linda \not\approx LindaD$$

However, if the possibility of failure is excluded from the system:

$$\begin{aligned} Debugger \quad \stackrel{def}{=} \quad & checkout_p(u).\overline{result}.Debugger + \\ & checkin_p(u).\overline{result}.Debugger + \\ & checkrd_p(u).\overline{result}.Debugger + \\ & checkinp_p(u).\overline{result}.Debugger + \\ & checkrdp_p(u).\overline{result}.Debugger + \\ & checkeval_p(u).\overline{result}.Debugger \end{aligned}$$

and $\overline{result}$ only transmits a favourable reply, then the following bisimulation holds:

$$TS(M) \approx (TSD(M)|Debugger)\backslash Ld$$

Intuitively, if no mismatch can occur, that is, the processes behave as expected, the Linda system

exhibits the same behaviour as the Linda system with debugger.

Alternatively, if the broadcast is internalised:

$$(TSD(M)|Debugger|Error)\backslash Ld \cup Lf$$

where: 1. $Error \stackrel{def}{=} \Sigma_{p \in P}\ (failout_p(u).Error +$
$failin_p(u).Error +$
$failrd_p(u).Error +$
$failinp_p(u).Error +$
$failrdp_p(u).Error +$
$faileval_p(u).Error)$

2. $Lf = \{\cup_{p \in P}\ (failout_p, failin_p, failrd_p, failinp_p, failrdp_p, faileval_p)\}$

then the following bisimulation holds:

$$TS(M) \approx (TSD(M)|Debugger|Error)\backslash Ld \cup Lf$$

Intuitively, if failure signals are not broadcast to the environment (are not observable) but dealt with internally, the Linda system exhibits the same behaviour as the Linda system with the debugger.

4.7 Conclusion

A model for debugging Linda programs has been presented that is based on an event-based behavioural model technique of debugging.

The technique requires the user to construct a model of the expected program behaviour prior to program execution. At run-time, actual program behaviour is compared with expected program behaviour. Any inconsistencies between actual and expected behaviour are considered to be program faults. Expected behaviour is defined in terms of program events, which in the context of Linda programs, relates to Linda primitives (or tuple space operations).

Since the behavioural model reflects the parallel component of the Linda program, the technique demands that the user pay specific attention to the parallel component during the model construction process. The overall approach that is adopted in the technique is formal, is composed of a set sequence of steps, and, due to its automated character, requires that the user play a far more passive role during program execution, but a far more active role during program development than is the case when traditional breakpoint styles of debugging are employed.

A CCS specification of both the basic Linda model and the Linda model with debugger has also been

presented. A number of observations have been made, and properties derived of the basic model and of the debugging model. Linda's commitment to tuple space interaction after the initiation of interaction, non-deterministic duration of tuple space operations, and the spatial and temporal decoupling of processes contribute to a general amenability to debugging. The Linda system that includes the debugger is able to establish a global state, generate events, and record a linear order of events without adversely effecting the execution of the Linda program to the extent that the "probe effect" is demonstrated. Linda programs are invariant on application or removal of the debugger. Observational equivalence is shown between the basic model and the debugging model when no behavioural mismatches occur, or when mismatch signals are caught by an internal error handler.

Chapter 5

A Specification Language for Linda Programs

Fundamental to behavioural model techniques of debugging is the construction of a model of the expected behaviour of the program under review. The previous chapter proposed a model of debugging that requires a specification of the actions of all the processes that constitute the Linda program. The specification must embody the parallel component (tuple space interaction) of the individual processes without the added baggage of all the other host language sequential code. To this end, some mechanism must be developed whereby this behaviour may be specified, and from which an appropriate model may be constructed. This chapter presents an experimental Linda program specification language.

5.1 A Mechanism for the Specification of Behaviour

Prior to the development of any specification mechanism, a clear idea must be established of what is meant by the term "behaviour", and the exact nature of that which must be specified.

The proposed Linda debugger targets the coordination component of the Linda program[1]. Its domain of interest is the Linda primitives and their interaction with tuple space. For the following example:

```
.
.
.
in('counter', ?Counter);
Temp := Counter;
```

[1] The computation component is considered by a standard sequential debugger that may be applied to the appropriate Linda process.

```
WriteString('Old value of Counter: ');
WriteInt(Counter, 4);
out('counter', Counter + 1);
WriteString('New value of Counter: ');
WriteInt(Counter + 1, 4);
.
.
.
```

only those statements in boldtype concern the Linda debugger. Indeed emphasis is greater than that: only those statements that appear in boldtype are subject to the scrutiny of the debugger. The behaviour of a Linda program refers specifically to the Linda primitives that are executed. A behaviour specification is an ordered collection of Linda primitives that are expected to be executed.

The specification attempts to capture the expected operational behaviour of a Linda program - what is the program expected to "do"? In terms of exactly what must be specified, a number of levels of concern can be identified.

A global level specifies the behaviour of a process as an ordered list of all the primitives that the process will execute at run-time, for example:

```
in(?a)
in(?b)
out(a*b)
```

At this level, emphasis is placed on the need to specify a complete list of primitives. That is, for the given process, no other primitives will be executed other than those specified and in that order. This reflects the general behaviour specification of the process in terms of what it must execute.

On a specific level, process activity may be constrained further by other specific behavioural patterns or requirements. For example, a sequence of primitives may be repeated a minimum or maximum number of times, or the number of times a certain primitive is executed may not exceed the number of times another primitive is executed.

The specification language and the specifications themselves form part of a debugger, and the specification of behavioural patterns that are of interest to users debugging their programs is crucial. On a debugging level, a user may wish to ascertain whether the behaviour of a process includes some highly-specific sequence of actions or sub-behaviour, for example:

```
in(37, 'hello')
out('too many hello')
```

within the lifetime of a process.

Furthermore, whilst any individual specification at any of these levels constitutes a behavioural specification of some aspect of the process, there is no reason why a multitude of specifications, at any level, could not constitute the behavioural specification of the process. It is quite conceivable that, as the characteristics of a particular Linda program are better understood, further behavioural specifications are added to enhance the overall specification, and to guard against undesirable behaviour. ("View" is a good metaphor for the various program specifications.) The Linda process would then have to satisfy each specification simultaneously for it to be said to be executing according to its expected behaviour.

The specification mechanism must be able to give expression to the program's behaviour at all of these levels.

A simple specification language, designed specifically for Linda programs, is proposed that is able to give expression to the program's behaviour at a multiplicity of levels, to varying degrees of detail, and that can handle more than one specification per process.

It is important to note that such program specifications can impact Linda program development beneficially. Following sound software development practices, program specification should precede any program implementation. (Indeed, such specification can form an integral part of the software development cycle.) For Linda programs, this means that the coordination component enjoys full attention without any competition from the excessive detail of the computation component. The strategy forces the programmer to consider:

- all the individual processes,
- process interaction via tuple space, and
- the composition of each tuple

all within the confines of a structured specification language prior to program implementation.

5.2 Previous Work

Section 2.2.3.2 contains a full report on behavioural model debuggers. This discussion summarises the attempts that have been made to specify program behaviour.

All behavioural model debuggers include some formal mechanism with which to specify behaviour. It usually takes the form of a special-purpose programming language or command language that requires behaviour to be specified in a rigid format. [Bai86], [Bat83b], [Bru83] and [Hse89] require a specification file, separate from the parallel program file, to be generated, whilst [Ros91] and

[Luc91] require that the parallel program be annotated. The specifications normally consist of a multitude of event (primitive and compound) specifications that each have sub-sections for the actual event specification, guards or constraining expressions that control firing conditions, and action-clauses that indicate what must ensue as a result of the occurrence of the event.

[Bai86] describes a special-purpose specification language in which is defined a partial order on the events of a process. For each process, all interactions in which it takes part are defined. Operators are: sequencing, non-determinism, iteration, reversing (the definition of future behaviour as a function of past behaviour), parallelism, and user interaction. Assertions are permitted that define predicates on the value of the variables of a process and the debugger state. [Bat89] also uses a special-purpose specification language to define events. Each event specification is composed of three sections: the event specification as a function of primary events, a series of expressions that constrain event occurrence, and a series of bindings that indicate how values must be bound to event instance attributes upon event occurrence. An extended form of regular expression is used to specify events that includes a shuffle operator. All operand events that are connected to the shuffle must occur but the order of occurrence is unimportant. The shuffle operator permits the expression of concurrency amongst participating events in the shuffle. [Hou89] uses the same formalism. [Els89] uses an extended form of regular expression to specify events. The extension is in the form of a permutation operator that is akin to the shuffle operator of [Bat89]. [Bru83] uses a modified version of path expressions known as path rules to specify behaviour. A path rule consists of an event recognition part (a generalised path expression) and a path action (a function that is called on event match/mismatch). Generalised path expressions evolved from path expressions (a regular expression with repetition, sequencing, and exclusive selection operators, and operands (path functions) that are the names of functions defined on the data types). Path expressions express restrictions on the allowed sequences of operations and the flow of information. Predicate path expressions extend basic path expressions to include history variables and predicates that are associated with path functions. Generalised path expressions extend predicate path expressions to include any program variable in predicates, and predefined path functions. [Hse89] uses an extended form of generalised path expressions known as data path expressions to specify behaviour. Data path expressions permit data events to appear as path functions.

It is frequently the case that a regular expression or a extension of a regular expression is used as the base formalism in which to specify behaviour. The basic regular expression controls the sequence of valid symbols whilst the extensions add further operators (shuffle operator), constraints on firing conditions, access to counters and program variables, and ever more elaborate specification of actions that ensue on event occurrence.

Specification languages are characteristically small, in terms of syntactic constructs, and reflect a minimalist mindset.

It is not standard practice that the behaviour of the entire program be specified ([Bai86] does, however,

require full specification). More usually, the user is able to specify as many events as desired which, collectively, may reflect some abstraction of the behaviour of the whole program.

5.3 An Experimental Linda Program Specification Language

5.3.1 Introduction

Section 5.1 detailed a number of base requirements that the specification language must satisfy, namely, that Linda primitives form the base components of the language, and that facilities be provided for multiple specifications or views of Linda processes. Using these as non-negotiable basics, a language is constructed that combines the Linda primitives in simple and more complex alliances to provide a useful set of forms with which to specify program behaviour.

Central to the proposed way in which Linda programs must be specificied, is the necessity to specify the complete behaviour of all participating processes. This does, however, conflict with the desire to provide specific and debugging level specifications where isolated process behaviour might be targetted. The language must make allowance for this.

It is also anticipated that once the global level specification is complete, further specific and debugging level specifications will be added as the program is debugged. The language should facilitate the easy inclusion of additional specifications.

It is also desired that the specifications, especially the global level specifications be used for more than just the basis of the behavioural models. If a similar semantic, and syntactic, base is employed by the specification language that matches or approximates the host Modula-2 Linda dialect, the specification could well be used as a starting point for final program implementation (the specification could be "fleshed-out"). (It ought to be noted that such a strategy may propagate errors from the specification into the final implementation. A measure of safety is, of course, provided by the specific and debugging level specifications.)

Issues of language expressivity and sufficiency are difficult to quantify. Example specifications provide some indication of these qualities.

5.3.2 Design Foundations

Section 5.2 detailed a variety of behaviour specification techniques. Most are special-purpose languages in which behaviour is expressed in some form of regular expression. In the design of a specification formalism, both the Linda paradigm and the underlying model of the proposed debugger exert considerable influence.

Linda processes are both spatially and temporally decoupled - processes need not know of the existence of other processes, nor do the processes with whom they interact indirectly have to execute simultaneously. As a result, process behaviour specifications do not include any reference to other processes, and essentially specify a straightforward sequence of events. A parallel operator is not required.

The model of debugging dictates that the debugger compare behaviour and provide tuple space with the results of the comparison (which tuple space ignores). If a mismatch in behaviour results, the debugger also informs the environment. The notion of user-defined actions executable on event match/mismatch is not applicable. Path actions or similar constructs are not necessary. Behaviour specifications that take the form of program annotations normally provide access to target program entities, for example, procedures, functions, variables and parameters. The proposed model of debugging does not make use of annotations but uses a separate specification that exists as part of the tuple space management system. As part of the tuple space management system, it does not have access to target program entities but is privy to tuple space information. Such information may be used to develop more comprehensive specifications. For example, if a tagged value is removed from tuple space, incremented, and then replaced in tuple space, a tentative specification might be:

in('counter', ?int)
out('counter', int)

where: a type specifier only is used.

A more useful specification is:

in('counter', ?Counter)
out('counter', Counter + 1)

where: the correctness of the *out*-operation is dependent on the value assigned to *Counter* during the *in*-operation.

The specification formalism must also be able to accept multiple specifications per process.

Based on past research, first consideration is given to regular expressions as the specification formalism. They are adequate for all requirements except for the representation of changing tuple space information. Multiple regular expressions cater for multiple specifications.

As an alternative to strict regular expressions, the use of CCS, or a subset of CCS, as the specification formalism is explored. In chapter 4, value-passing CCS was used to specify the Linda paradigm. As would have been expected, the specification of individual processes makes use of a limited subset of CCS (action prefix (input and output of values), inactive agent, and summation) - no composition,

relabelling or restriction constructs are used. Furthermore:

- it has a conditional (if) construct for added expressivity, and
- it has all the syntactic constructs (and associated semantics) that are found in regular expressions:

	Reg. Expr.	value-passing CCS
sequencing	ab	a.b
alternation	a + b	a + b
repetition (*) (Kleene star)	(a)*b ...	A = a.A + b. ...

and the derived repetition(+) operator:

repetition (+)	$(a)^+b$ ...	A = a.B B = a.B + b. ...

The use of CCS to specify the behaviour of concurrent systems is well-known. CCS has also been used to analyse the behaviour of imperative languages [Fre90], [Hen83]. In [Fre90], programs written in Lunsen (an imperative language with constructs for concurrency and communication) are initially translated into a typed CCS[2] and then to basic CCS, after which the program's behaviour is analysed[3]. The thrust of the exercise is to examine the parallel component of the program.

The feasibility of a similar approach is apparent: in this work, the experimental Modula-2 Linda system is hosted in an imperative language; the coordination component is clearly identifiable; and the translation to a value-passing CCS is well-defined. Additionally, multiple value-passing CCS specifications cater for multiple process specifications. Consequently, the proposed specification language is based on value-passing CCS, with its syntax based on the language described by [Bru91].

5.3.3 Language Syntax

The language[4] supports the specification of the behaviour of a Linda program from a multiplicity of

---

[2] The typed CCS language is much like the language developed by [Bru91] for value-passing CCS.

[3] Both Lunsen (at the program level) and VP-CCS place restrictions on the language to make it finite-state. In Lunsen, types of variables may only contain finitely many elements, and arbitrary recursion is not permitted. In VP-CCS, types of variables may also only contain finitely many elements. Analysis is conducted with the use of the Edinburgh Concurrency Workbench [Cle88].

[4] A full listing of the language syntax (expressed in extended BNF notation) can be found in Appendix D.

behavioural angles and at a variety of levels. The underlying objective is to provide constructs that enable the user to specify accurately the behaviour of a Linda program at the level of the Linda primitives.

A specification for a single process is typically composed of a number of sub-specifications each of which define alternate behavioural specifications or views of the process:

```
spec <SpecName1>;
   <Specification>
endspec

.
.
.

spec <SpecNameN>;
   <Specification>
endspec
```

Each sub-specification is syntactically and semantically independent of the other sub-specifications.

Typically, one sub-specification specifies a general behavioural pattern, and any number of other sub-specifications specify other behavioural patterns.

Comments are permitted in the specification language, and take the following form:

```
/*  comment  */
```

<Specification> is divided into two sections: a variable declaration section, and the body of the sub-specification. The specification language provides facilities whereby variables may be declared in which actual tuple data is stored that has been matched with formal tuples:

```
var
   <Identifier1>, <Identifier2> : <TypeIdentifier>;
   <Identifier3> : <TypeIdentifier>;
```

The data is used in expressions that form part of future tuple elements and boolean expressions. It strengthens the specification base.

The body of the sub-specification is composed of any number of sub-processes:

process <ProcessName1>
= <CompoundStatement>;

.
.
.

process <ProcessNameN>
= <CompoundStatement>

that collectively specify the behaviour of the Linda process.

<CompoundStatement> is composed both of simple statements that relate to:

- reference to other *<ProcessName>*'s,
- Linda primitives, and
- process termination

and constructors. Simple statements are separated by the sequencing operator, ".", for example:

a . b c

where: *a*, *b*, and *c* are simple statements.

Processes may make reference to other processes, and themselves. Such a reference is an effective goto-statement, and is the vehicle by which iteration is implemented. For example:

```
process Arithmetic
   = /* get task */
     /* get data */
     /* perform computation */
     /* return result */
     /* now go get next task */
     Arithmetic
```

and

```
process Part1
   = .
     .
     /* now off to second part */
     Part2

process Part2
   = .
     .
     /* completed, so back to first part */
     Part1
```

The statements that relate to Linda primitives are:

out(tuple)	-	*out(<TElement>)*
in(tuple)	-	*in(<TElement>)*
read(tuple)	-	*read(<TElement>)*
inp(tuple)	-	*inp(<TElement>)*
readp(tuple)	-	*readp(<TElement>)*
eval(tuple)	-	*eval(<TElement>)*

Tuple element, *<TElement>*, detail may be provided:

<Element1>, ... ,<ElementN>

at varying levels of specificity:

<Expression>	-	an integer expression or string constant
int	-	an anonymous actual integer
str	-	an anonymous actual string
?<Identifier>	-	a named formal integer or string
?int	-	an anonymous formal integer
?str	-	an anonymous formal string

where: *int* and *str* are type identifiers.

For example:

```
var
   I : int;
   Count, Name : str;

.
.
.
in(?Count, int, I+4, ?str)
out(int, int, int)
readp('my_name', ?Name)
.
.
.
```

In line with the semantics that are attached to **eval**, its <*TElement*> is restricted to a single actual string that names an executable process.

Alternatively, a tuple may be specified using the wild tuple indicator:

```
*
```

The wild tuple indicator represents any tuple, that is, a tuple that is free of any composition constraints, for example:

```
inp(*)
read(*)
```

where: *inp(*)* represents the universal set of all **inp** primitives, and *read(*)* represents the universal set of all **read** primitives.

An even more powerful wild Linda primitive statement:

```
*
```

is also defined. The wild Linda primitive represents any one of the Linda primitives to which may be coupled any tuple. For example, any three Linda primitives that separate **out(3)** and **out(4)** can

be specified as follows:

out(3)..*.*.out(4)*

where: * represents the universal set of all Linda primitives and possible tuples.

The wild Linda primitive construct is used to ignore irrelevant behaviour. The wild Linda primitive, specified Linda primitive, wild tuple, and named and anonymous tuple elements constitute a hierarchy of specificity or degrees of "don't care".

Linda processes do not normally have infinite behaviour, and the specifications must provide a mechanism by which process termination may be signalled. (Process termination in the sense that no further tuple space operations - Linda primitives - are executed by the Linda process[5].) The *NIL*-statement reflects such a condition, for example:

```
process SomeTask
   = /* first part of task */
     /* second part of task */
     /* last part of task */
     NIL
```

A *random*-construct[6] (similar to the shuffle-operator [Bat89] and the permutation-operator [Els89])

random (<LindaPrimitive> . { <LindaPrimitive> })

provides for the specification of a group of actions that may occur in random order. For example:

```
.
.
.
random(in(?I).out(4).in(37))
.
.
.
```

[5] It is important to remember that the Modula-2 Linda system is such that processes are required to both initiate and sever tuple space interaction by a special call on the tuple space management system. The "initiate" call signals a desire to interact whilst the "sever" call signals the cessation of interaction. In this way, the tuple space management system is provided with boundaries within which interaction is still possible.

[6] The current implementation of the language supports a trivial form of the construct. A sequential order of occurrence must be followed, that is, the actions must occur in the same order as they appear in the *random*-construct.

Each action specified in the group must occur once, and once only.

Two constructors implement alternation. An internally-decidable *if*-statement:

```
if ( <Condition>
   then  <CompoundStatement>
   else  <CompoundStatement> )
```

provides for alternate specification paths based on a condition that is either one of the Linda predicate forms:

```
inp(<TElement>)
readp(<TElement>)
```

or an integer or string boolean expression. For example:

```
var
   Action : str;

process Arithmetic
   = /* get task */
      in(?Action).
      /* check for 'no more tasks' descriptor */
      if ( Action = 'no more'
         then  /* you're done */
               NIL
         else  /* get data */
               /* perform computation */
               /* return result */
               /* now go get next task */
               Arithmetic)
```

An externally-decidable *choice*-statement:

choice (*<CompoundStatement>*
| *<CompoundStatement>*

.
.
.

| *<CompoundStatement>)*

provides for multiple alternate specification paths. In this case, although only one specification path is eventually followed, more than one specification path is available for satisfaction - each of which is equally correct. The first simple statement found in each *<CompoundStatement>* serves as the trigger by which alternate paths are chosen. The trigger may be a simple statement, that is:

out(<TElement>)
in(<TElement>)
read(<TElement>)
inp(<TElement>)
readp(<TElement>)
eval(<TElement>)
NIL

or, if it is not one of the above, the following semantics apply:

<ProcessName> - the first simple statement in the sub-process indicated by *<ProcessName>*

if-statement - predicate conditional:
the predicate conditional

expression conditional:
the first simple statement in either the *then*-clause or the *else*-clause

choice-statement - nested *choice*-statements are "flattened", that is:

```
choice (    <Ca>
          | choice (    <Cb>
                      | <Cc>)
          | <Cd>)
```

is equivalent to:

```
choice (    <Ca>
          | <Cb>
          | <Cc>
          | <Cd>)
```

The search for trigger simple statements may encounter nested, contiguous *<ProcessName>*-, *if*-, and *choice*-statements, in which case, the search process is defined to be recursive.

An example of the use of the choice-statement is:

```
process MasterArithmetic
   = choice (    out('+').Addition
               | out('-').Subtraction
               | out('*').Multiplication
               | out('/').Division);

process Addition
   = /* some specification */

process Subtraction
   = /* some specification */

process Multiplication
   = /* some specification */

process Division
   = /* some specification */
```

where: the specification permits the associated Linda process only to add either '+', '-', '*', or '/' to tuple space and then to continue as specified.

5.3.4 Language Semantics

A number of semantic rules apply.

1. Sub-specification names, <*SpecName*>'s, must be distinct.

2. Sub-process names, <*ProcessName*>'s, within a particular sub-specification must be distinct. Sub-process names are further constrained by the requirement that one of the names must match the name of the Linda process that is being specified. For example, if the Linda process is named *Primes*, an acceptable specification is:

```
spec PrimesGeneral;

   process Primes
      = /* some specification */

   .
   .
   .

endspec
```

   The sub-process so named is the sub-process at which specification checking is begun - a start symbol.

3. All variable identifier names within a particular sub-specification must be distinct. Identifiers that are used in <*Expression*>'s, for example:

```
out(I+1)

if ((I = 10
   then  ...
   else  ...)
```

   must also be initialised by previous use as a formal tuple element, for example, either:

```
in(?I)
read(?I)
```

or a Linda primitive predicate form, for example:

inp(?I, int)
readp(?I, str)

that fires TRUE.

4. Actual and expected behaviour is compared in the following manner:

   a) The Linda primitive, for example, **in** or **out**, must match.
   b) Tuple arity must be equal.
   c) For each tuple element:
      - both elements must be formal or actual,
      - both elements must be of the same type,
        for an actual element:
          if both elements are defined (not anonymous), the value of the expression must be equal
      - for both formal and actual elements:
          if either is anonymous, a match is declared
   d) If the wild tuple indicator is used, the Linda primitive name, for example, **read** or **out**, alone must match.
   e) If the wild Linda primitive is used, as long as a Linda primitive occurs, a match is declared regardless.

   Note that the usual Linda tuple rules that match templates with free tuples do not apply.

5. The externally-decidable *choice*-statement offers a number of match possibilities. The particular match strategy followed is:

   a) In the event that an anonymous *int* or *str* is used, it excludes from the range of *int*'s or *str*'s all specific instances used in other tuples coupled to the same type of Linda primitive found as the trigger elsewhere in the *choice*-statement, for example:

```
choice (    out(12). ...
         |  out(int). ...)
```

   then a candidate **out(12)** does not match *out(int)*.

b) In the event that a wild tuple is used, it excludes from the universal set of tuples, all tuples coupled to the same type of Linda primitive found as the trigger elsewhere in the *choice*-statement, for example:

```
choice (    out('hello', 87). ...
        |   out(*). ...)
```

then a candidate **out('hello', 87)** does not match *out(*)*.

c) In the event that the wild Linda primitive is used, it excludes from the set of all Linda primitives, all Linda primitives and associated tuples found as the trigger elsewhere in the *choice*-statement, for example:

```
choice (    out('answer'). ...
        |   *. ...)
```

then a candidate *out('answer')* does not match *.

6. The specification must provide a complete specification of all the Linda primitives that will be executed by the Linda process. No provision is made for part-specification of processes. For example, it is insufficient to specify the behaviour of a process that executes the primitive **out(54)** sometime in its lifetime, amongst many other Linda primitives as:

```
spec ExampleIncorrect;

  process Example
    = out(54).NIL

endspec
```

The interpretation that is ascribed to the above specification is: the Linda process, *Example*, executes one, and only one, Linda primitive (**out(54)**) in its lifetime. The correct specification is:

```
spec ExampleCorrect;

  process Example
    = choice (    *.Example
              |   out(54).Example1);
```

```
process Example1
   = choice (    NIL
             |   *.Example1)

endspec
```

Note how the wild Linda primitive is used to by-pass or ignore behaviour.

7. Whilst syntactically correct, some constructs specify no behaviour - they lack any action specification. For example:

```
process A
   = A;
```

and:

```
process B
   = C;

process C
   = B;
```

and:

```
process D
   = choice (    D
             |   E);

process E
   = D;
```

The interpretation that is ascribed to processes *A, B, C, D,* and *E* is: they are all incapable of performing any Linda primitives and do not terminate.

It may, however, be the case that the process never interacts with tuple space but does terminate, in which case, the appropriate specification is:

```
process A
   = NIL
```

5.4 Specification Techniques

The facilities that are provided by the language can be used to generate a wide variety of specifications.

(In the specifications that follow, the actions (a_1, a_2, a_3, ... ,a_n) refer to Linda primitives, and *Next, Next1, Next2* are *<ProcessName>*'s.)

1. A single a_1-action:

```
process Example
    = a1.Next;
```

2. A sequence of actions (a_1, a_2, a_3):

```
process Example
    = a1.a2.a3.Next;
```

3. Internally-decidable alternation:

```
process Example
    = a1.
      if ( <Condition>
         then  a2.Next1
         else  a3.Next2);
```

4. Externally-decidable alternation:

```
process Example
    = a1.
      choice (    a2.Next1
                | a3.Next2);
```

5. Infinite iteration:

```
process Example
    = a1.a2. ... .an.Example
```

6. Deterministic iteration:

```
process Example
   = if ( <Condition>
        then  a1.a2. ... .an.Example
        else  Next)
```

7. Non-deterministic iteration:

```
process Example
   = choice (    a1.a2. ... .an.Example
               | Next)
```

8. The *if*-statement and *choice*-statement implement alternation. If a common specification must be followed after all alternates, it must be specified explicitly, for example:

```
process Example
   = if ( <Condition>
        then  a1.Example1
        else  a2.Example1)

process Example1
   = /* some specification */
```

or:

```
process Example
   = choice (    a1.Example1
               | a2.Example1)

process Example1
   = /* some specification */
```

The construction of the global level specifications may be approached in a programming-like manner, where the actions are specified as they would appear in the final Linda program. The specific level and debugging level specifications, on the other hand, are usually moulded in the form of properties that the process must satisfy.

Some common properties include:

1. Only one of a set of actions (a_1, a_2, a_3):

```
process Example
  = choice (   a1.Next
             | a2.Next
             | a3.Next)
```

2. Always one of a set of actions (a_1, a_2, a_3):

```
process Example
  = choice (   a1.Example
             | a2.Example
             | a3.Example)
```

In this case, infinite behaviour is expected. The addition of a *NIL*-statement as one of the choices permits finite behaviour.

3. A minimum number, say three, of a_1-actions:

```
process Example
  = a1.a1.a1.Example1

process Example1
  = choice (   Next
             | a1.Example1)
```

4. A maximum number, say three, of a_1-actions:

```
process Example
  = choice (   Next
             | a1.Example1);

process Example1
  = choice (   Next
             | a1.Example2);
```

process Example2
= *choice (* *Next*
| a_1*.Next);*[7]

5. Zero or more a_1-actions:

process Example
= *choice (* *Next*
| a_1*.Example)*

6. One or more a_1-actions (this is similar to 3 above):

process Example
= a_1*.Example1;*

process Example1
= *choice (* *Next*
| a_1*.Example1)*

7. One or more of a sequence of actions (a_1, a_2, a_3):

process Example
= *choice (* $a_1.a_2.a_3$*.Example1*
| $a_1.a_2$*.*.Example*
| a_1*.*.Example*
| **.Example);*

process Example1
= *choice (* **.Example1*
| *NIL)*

---

[7] The specification of minimum and maximum number of actions begs the introduction of a replicate-operator:

"**rep**" "(" [<LowerBound] "," [UpperBound] "," <Statement> { "." <Statement> } ")"

This is discussed in more detail in the final chapter.

5.5 Example

The use of the specification language is demonstrated in the following example[8]:

Problem: A number of simple arithmetic operations must be performed. Each operation is composed of an operator (+, -, *, /), and two atomic operands.

Solution: A Linda solution, in the agenda-style of parallelism, is proposed in which a master process starts a number of slaves, adds tasks to tuple space and retrieves the results, whilst slaves scavenge for work, do the work, and return the answers to tuple space. Task tuples are structured as follows:

task_id, operator

Each task tuple is associated with a data tuple that is of the form:

task_id, operand, operand

The master process adds all tasks and task data to tuple space, retrieves all the answers and then posts a poison task to tuple space. Slaves continue to extract tasks and task data until the poison task is retrieved at which stage they return the poison task and terminate.

A possible specification for the master is as follows:

```
spec Arithmetic;
/* Master process that adds tasks to tuple space and retrieves results */

var
   Answer : int;

process Master
/* start a number of slave processes */
   = choice (     addtasks
                | eval('Slave').Master)
```

[8] Further examples can be found in Appendix E.

```
process addtasks
/* continue adding tasks to tuple space until no more tasks available */
   = choice (    getresults
               | out(int, '+').addoperands
               | out(int, '-').addoperands
               | out(int, '*').addoperands
               | out(int, '/').addoperands);

process addoperands
   = /* add operands */
     out(int, int, int).
     /* organise next task */
     addtasks;

process getresults
   = choice (    /* add poison task */
                 out(int, 'end').
                 /* you're done */
                 NIL
               | /* retrieve answer */
                 in(int, ?Answer).
                 /* get more answers */
                 getresults)

endspec
```

A possible specification for the slave process is as follows:

```
spec Arithmetic
/* Slave process that scavenges for tasks and task data, performs
   computations, and returns results */

var
   Task_Id, Op1, Op2 : int;
   Operator : str;
```

```
process Slave
   = /* get task */
     in(?Task_Id, ?Operator).
     /* check if poison task */
     if (Operator = 'end'
        then  /* yes, return it */
              out(Task_Id, Operator).
              /* and terminate */
              NIL
        else  calculate)

process calculate
   = /* get task data */
     in(Task_Id, ?Op1, ?Op2).
     /* perform computation, and return result */
     if (Operator = '+'
        then  out(Task_Id, Op1+Op2).
              Slave
        else  if (Operator = '-'
              then  out(Task_Id, Op1-Op2).
                    Slave
              else  if (Operator = '*'
                    then  out(Task_Id, Op1*Op2).
                          Slave
                    else  if (Operator = '/'
                          then  out(Task_Id, Op1 DIV Op2).
                                Slave
                          else  NIL))))

endspec
```

Any number of other behavioural patterns may be specified:

```
spec AtLeastOneAddition;
/* Check that at least one addition operation is performed
   in the lifetime of the process */

var
   Operator : str;

process Slave
   = in(?int, ?Operator).
      in(*).
      out(*).
      if (Operator = '+'
         then  ignoreallelse
         else  Slave);

process ignoreallelse
   = choice (     NIL
               |  *.ignoreallelse)

endspec


spec AtLeastOneAnsweris11;
/* Check that at least one answer of 11 is computed by
   the process in its lifetime */

process Slave
   = choice (     *.Slave
               |  out(int, 11).ignoreallelse);

process ignoreallelse
   = choice (     NIL
               |  *.ignoreallelse)

endspec
```

```
spec NoTwoConsecutiveOuts;
/* Check that no two consecutive out operations are performed by
   the process in its lifetime */

process Slave
   = choice (    NIL
               | *.Slave
               | out(*).choice (    NIL
                                  | in(*).Slave
                                  | read(*).Slave
                                  | inp(*).Slave
                                  | readp(*).Slave
                                  | eval(*).Slave))

endspec
```

5.6 Conclusion

An experimental specification language for Linda programs has been presented that is based on value-passing CCS. It is used to specify the expected behaviour of the processes that constitute a Linda program, and from which a model of the expected behaviour is then constructed.

The language provides facilities whereby the parallel component of Linda programs may be specified, in terms of Linda primitives, at varying degrees of specificity and from any number of behavioural levels or views (global, specific, and debugging). The language is also able to specify the expected behaviour in terms of properties that must be satisfied by a Linda process. A process specification is typically composed of a variety of sub-specifications, each of which specifies its entire expected behaviour from a different behavioural angle. (The requirement that the entire behaviour of a process be specified is relaxed somewhat by the wild tuple and wild Linda primitive forms.) This contrasts with many other such systems in which specific events are specified that only model particular aspects of the program's overall behaviour. A multiplicity of sub-specifications is encouraged.

It is not the author's experience that Linda programs contain inordinately many tuple space operations. Since Linda primitives form the core of the specification language, the length of the specification is manageable.

Whilst the language has minimal syntactic constructs that could well be expanded, the language nonetheless contains a core set of useful constructs that demonstrate adequately the principles underpinning the debugging methodology.

Chapter 6

Behavioural Models

Chapter 4 explored a mechanism for debugging Linda programs that is based on behavioural model debugging. Central to the technique is the construction of models of the expected behaviour of programs. Chapter 5 described an experimental Linda program specification language that is used to specify expected program behaviour, and in terms of which the expected behavioural model is then constructed. This chapter describes the internal model representation, the model construction process, model/Linda system integration, and model control during program execution.

6.1 From Specifications to Models to Recognition Engines

Expected behaviour models act as recognition engines that accept actual behaviour, and produce notice

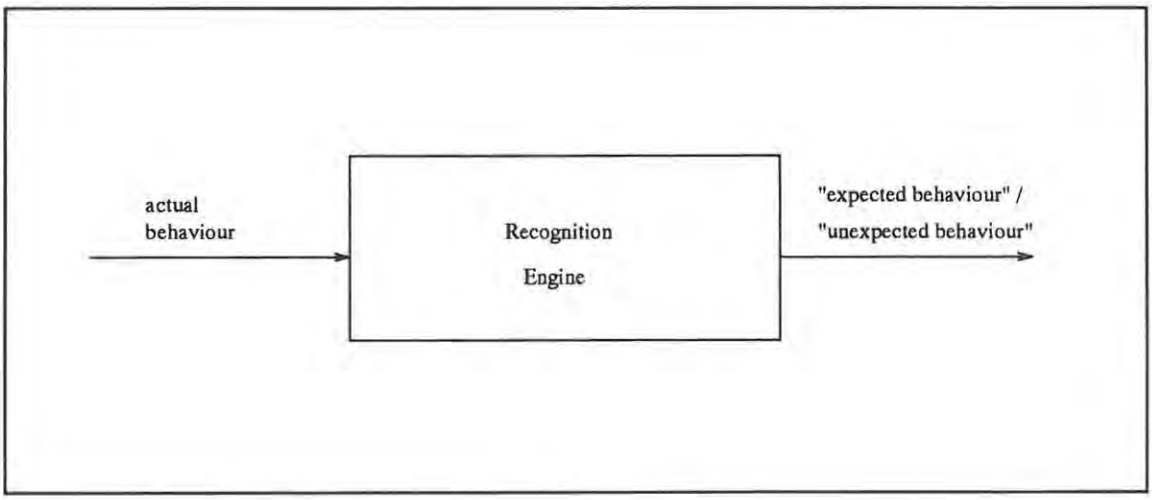

Figure 6.1 Recognition process

of whether the behaviour was expected or unexpected (see Figure 6.1).

The recognition process is cyclic - actual behaviour is composed of a number of events, each of which is checked by the model. The model recognises the actual behaviour of the target program or declares a mismatch at some point.

Chapter 5 detailed a specification language for Linda programs with which the expected behaviour of the program may be specified in terms of Linda primitives. This specification forms the source of information from which the desired model is constructed (see Figure 6.2).

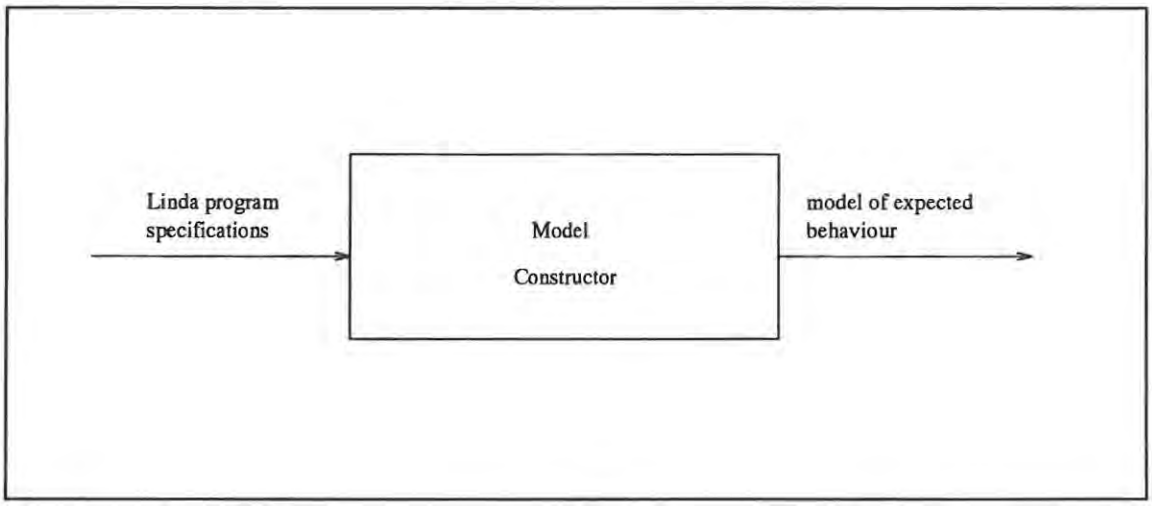

Figure 6.2 Model construction

Given that an appropriate model of the expected behaviour of the program exists, it is used as the basis of a recognition engine (see Figure 6.3).

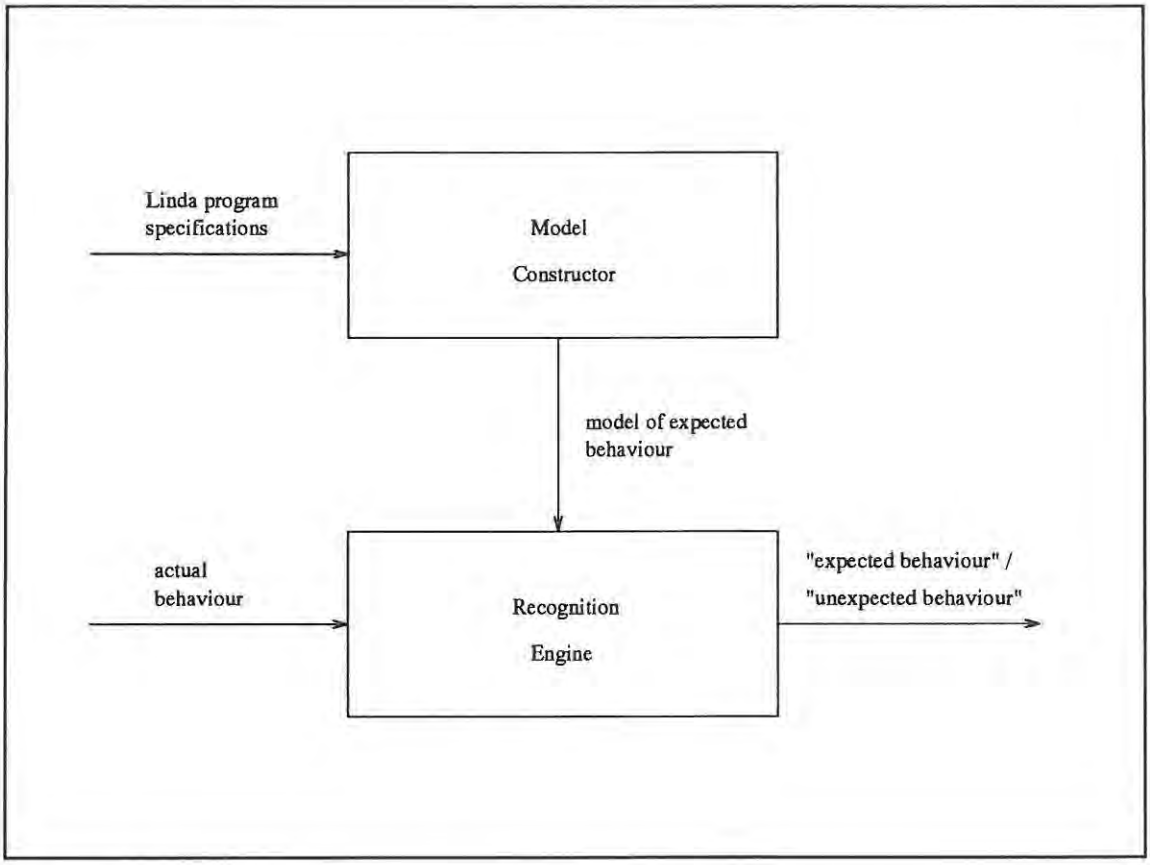

Figure 6.3 Recognition process with expected behaviour model constructor

As the target Linda program executes, the recognition engine checks actual Linda primitives with expected Linda primitives as found in the model.

6.2 Previous Work

Section 2.2.3.2 contains a full report on behavioural model debuggers. A summary is now provided of attempts that have been made to construct models of expected behaviour, recognition engines and the recognition process.

Regular expressions and variants of regular expressions are predominantly used to specify behaviour. Finite-state automata and variants of finite-state automata implement these regular expressions and form the basis of recognition engines.

[Ols91a] and [Ols91b] use directed dataflow graphs as the model in which leaf nodes define primitive events and internal nodes define compound events. [Bai86] implements specifications that define partial orders of events of a process as a process in the target programming language. The resultant process executes simultaneously with the target process, and awaits event information which it checks. [Bru83] uses a finite-state automaton to implement generalised path expressions. The automaton

resides in the address space of either the debugger or the target process where it recognises process behaviour and takes appropriate action upon event occurrence. [Bat89] uses a shuffle-automaton to implement regular expressions based on patterns of symbols. Simple finite-state automata are insufficient: to recognise sets (patterns) of symbols; to base transitions on relational expressions that are based on attributes of input symbols; and to handle concurrent pattern matching. [Els89] uses finite-state automata to implement a slightly extended form of regular expressions (they include concatenation, alternation, repetition and permutation operators). [Hse89] uses predecessor automata to implement data path expressions. Predecessor automata fire on transitions that are based on both the current event and predecessor events. They recognise partial ordering graphs as well as strings. Where specifications are provided in CCS, they are implemented as a series of transition graphs, as in the Concurrency Workbench [Cle93].

The increase in complexity of specification formalism (as noted in Chapter 5) has necessitated an increasingly complicated implementation equivalent, and recognition process.

6.3 Internal Model Representation

The task of checking actual with expected behaviour is the responsibility of a recognition engine. In this work, recognition engines are based on program specifications expressed in the Linda program specification language. This section discusses a strategy for the implementation of recognition engines.

The implementation must cater for:

1. the inclusion of information in Linda primitives that may not be static, and
2. multiple specifications for each Linda process.

In previous work, frequent use is made of finite-state automata to implement recognition engines. Expressed simplistically, behaviour recognition can be thought of as a process in which a stream of tokens is recognised as valid or not. For languages based on regular expressions, finite-state automata form an appropriate implementation for corresponding recognition engines. Finite-state automata are defined as follows:

Definition: A non-deterministic (finite-state) automaton (NFA) D is a 5-tuple (Q, T, δ, S, F), where:

1. Q is a finite non-empty set, elements of which are called states.
2. T is an alphabet.
3. δ is a function (transition function) from Q X (T ∪ {Λ}) into the set of subsets of Q.
4. S ∈ Q is a start symbol.
5. F ⊆ Q is a non-empty set of final states.

In the context of the Linda debugger, the Linda primitives are the alphabet (T), and the points in the program's execution at which a particular primitive (or set of primitives) is expected constitute the various states (Q). The alphabet does, however, require closer examination. Section 5.3.2 proposed that the specification language permit tuple information to be specified in the Linda primitives, for example:

out(Number)

Here, *Number* is not defined statically but is dependent on the particular binding that is operable at the time the primitive is encountered in the recognition process. Tokens in the proposed language are composed of values that may change from time to time - the alphabet is dynamic. Unfortunately, an NFA requires that the alphabet be static (predefined), and is therefore inadequate.

In this work, an extended NFA is proposed that differs from the standard NFA in respect of the alphabet and the transition function.

An environment, E, is defined

$$E = \bigcup_{i=1}^{n} \{(Name_i, Value_i)\}$$

where: n is the number of named variables in E.

in which a set of tuples, $(Name_i, Value_i)$, is maintained that associates a $Value_i$ with each $Name_i$. The alphabet is subject to the environment

$$T_E$$

and the transition function is modified accordingly

$$\delta = Q \times (T_E \cup \{\Lambda\})$$

The full definition of the extended NFA is as follows:

Definition: An extended non-deterministic (finite-state) automaton (ENFA) D is a 6-tuple (Q, E, T, δ, S, F), where:

1. Q is a finite non-empty set, elements of which are called states.
2. E is an environment of tuples that bind names to values.
3. T_E is an alphabet.
4. δ is a function (transition function) from Q X (T_E ∪ {Λ}) into the set of subsets of Q.
5. S ∈ Q is a start symbol.
6. F ⊆ Q is a non-empty set of final states.

For example, in Figure 6.4

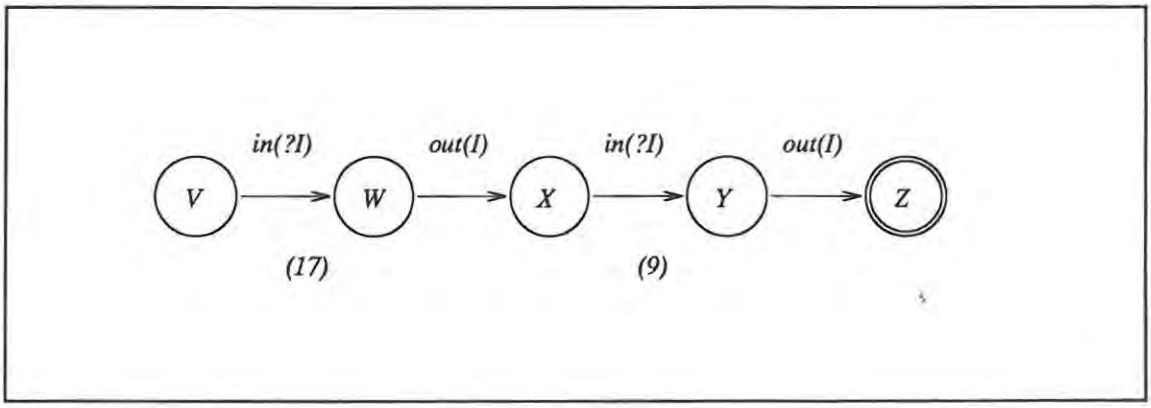

Figure 6.4 Internal model of expected behaviour

the following ENFA is depicted:

Q = {*V,W,X,Y,Z*}
δ = Q X (T_E ∪ {Λ})
S = *V*
F = {*Z*}

at the various states, E and T_E are as follows:

at *V*: T_E = *in(?I), out(I)*
E = {}

at *W,X*: T_E = *in(?I), out(17)*
E = {*(I,17)*}

at *Y,Z*: $T_E = in(?I), out(9)$
$E = \{(I,9)\}$

The internal model representation is further complicated by:

1. the requirement that a behavioural specification be provided for each Linda process, and
2. the option of multiple behavioural specifications (or sub-specifications) per Linda process.

To cater for these requirements, a compound internal model is used (see Figure 6.5).

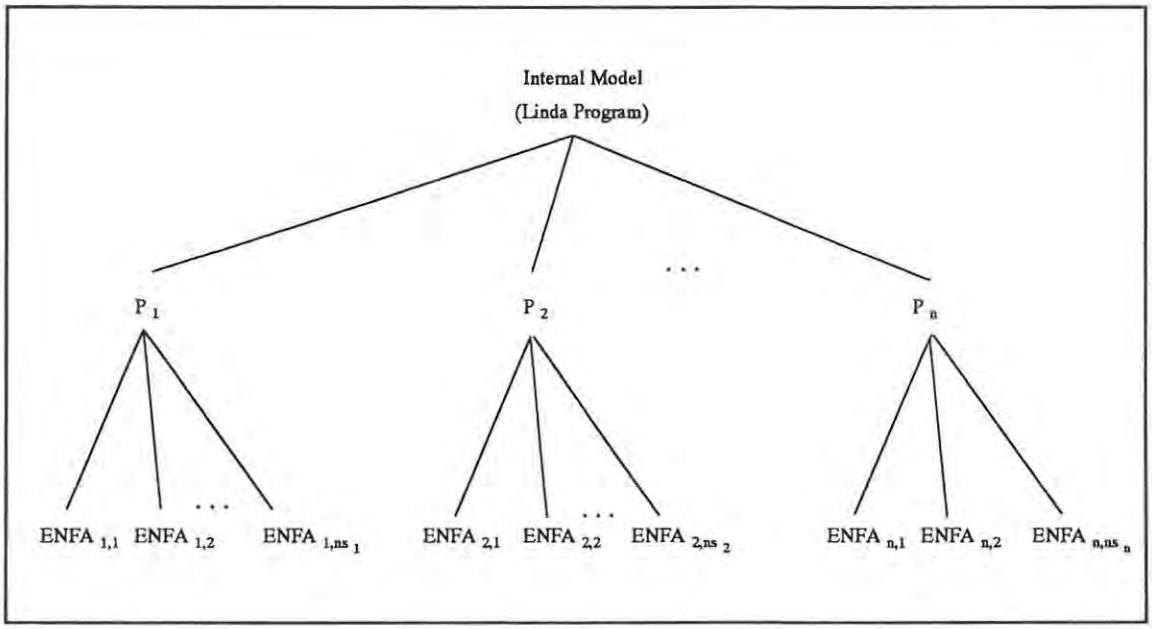

Figure 6.5 Internal model of a Linda program

For each sub-specification, a separate ENFA, and private environment, is constructed that manages a particular abstraction of the process.

In summary, the internal model of a Linda program is the summation of the internal models of all its constituent processes:

$$InternalModel_{LindaProgram} = \Sigma_{p \in P} D_p$$

where: 1. P is the set of all process identifiers
2. $p \in P$

The internal model of a single process is the summation of all the internal models that represent its

multiple sub-specifications:

$$D_p = \Sigma_{i=1,ns_p} D_{p,i}$$

where: ns_p is the number of sub-specifications for process p.

And the internal model of a sub-specification is an ENFA:

$$D_{p,i} = (Q_{p,i}, E_{p,i}, T_{E_{p,i}}, \delta, S_{p,i}, F_{p,i})$$

6.4 Model Construction

A labelled transition graph (G) is used to implement the ENFA. Coupled to G is an environment (E) in which tuples are maintained that associate a value with a name. Wherever names are referenced in G, an appropriate hook is maintained to this environment.

The implementation of the internal model of the Linda program is composed of the implementation of all the internal models of all participating processes:

$$ImplModel_{LindaProgram} = \Sigma_{p \in P} ImplModel_p$$

For each process, a graph is constructed and an environment is maintained for each sub-specification:

$$ImplModel_p = \Sigma_{i=1,ns_p} G_{p,i} E_{p,i}$$

Process specifications expressed in the Linda program specification language are parsed, and converted into a graph. Most language constructs translate into a state with a single output transition. Alternation constructs translate into a state with two output transitions (in the case of the *if*-statement), and more than one output transition (in the case of the *choice*-statement).

Some graph post-processing is performed in which nested *choice*-statements are "flattened", multiple reference to the same sub-process and termination (*NIL*-nodes) in the same *choice*-statement are removed, and attention is drawn to empty specifications, for example:

process A
= *B;*

process B
= *A;*

It is standard practice that non-deterministic finite-state automata are converted to deterministic finite-state automata prior to their use as recognition engines. It imposes greater construction time but improves recognition time. The requirement that symbols (Linda primitives) need not be fully-defined at model construction time (they are dynamic) precludes any conversion. For example, a state may be defined to have two transitions, namely, *out(I)* and *out(J)*. Dependent on the values bound to *I* and *J*, the transitions may or may not be deterministic. Furthermore, they may be placed within some cycle, in which case the state may change from deterministic to non-deterministic, or vice versa, for example:

```
process A
    = choice (   NIL
               |  out(I).in(?I).A
               |  out(J).in(?J).A)
```

As a result, the recognition engine is forced to pursue multiple paths when behaviour is checked.

6.5 The Model at Work

6.5.1 Model Control

6.5.1.1 Informal Description

Internal models are constructed for all processes participating in the Linda program, and their actual behaviour is checked against these models.

The following, iterative, checking process is defined:

1. For each internal model, the debugger sets a current state marker equal to the start state.

2. As processes interact with tuple space, the debugger is informed of the nature of the interaction and the originating process.

   For each interaction:

   - Based on the associated environment for each model, the debugger updates all next expected Linda primitives (transitions) at the current state.

   - The debugger then tests the actual behaviour against the behaviour as expected (transitions) in each sub-specification for that process.

- If the behaviour is accepted by all sub-specifications, a match is declared, otherwise a mismatch is declared.

- Based on the valid transitions, the current state marker is updated to reflect the new state.

- For each model, the associated environment is updated to reflect any new bindings.

The checking process continues until all processes terminate interaction with tuple space.

Note that the behaviour of processes is checked simultaneously, as each process requests tuple space interaction. Although tuple space requests from the various processes are interleaved in time, the requests for a single process represent a linear stream of behaviour.

6.5.1.2 Formal Description

The internal model of the target Linda program is composed of a number of ENFA for each process. The Linda program satisfies its expected behaviour, if, for every ENFA, the stream of behaviour (symbols) is accepted and a final state is reached. (For programs that have infinite behaviour, the problem is undecidable.)

A current state is maintained for each ENFA that represents a sub-specification

$$currentstate_{p,i}$$

where: 1. P is the set of all process identifiers
2. $p \in P$
3. i is the i^{th} sub-specification

Since the graphs are non-deterministic, a set is used to represent the current state (multiple paths are followed in parallel).

The standard algorithm for the recognition of a language by a deterministic machine [Bac79] is used as the basis for the algorithm that implements the matching process for any process ($p \in P$).

```
{   Algorithm to determine, for some process p (∈ P), whether a given sequence of behaviour B_p
    (∈ T_Ep,i*) is in the language recognised by the ENFA machine D_p,i = (Q_p,i, E_p,i, T_Ep,i, δ, S_p,i
    , F_p,i).}

for all i do
        currentstate_p,i := ∅ ∪ S_p,i;
behaviour := first behaviour symbol in B_p;
satisfied := true;
while B_p not exhausted do
        for all i do
                for all states in currentstate_p,i
                        update T_Ep,i;
        for all i do
                currentstate_p,i = δ(currentstate_p,i, behaviour);
                if currentstate_p,i = ∅
                        then    satisfied := false;
                                "Mismatch"
                        else    update E_p,i
        nextbehaviour(behaviour)
end;
for all i do
        if NOT currentstate_p,i ∈ F_p,i
                then    satisfied := false;
if satisfied
        then "Process satisfied expected behaviour"
        else "Process did not satisfy expected behaviour".
```

6.5.2 Model/Linda System Integration

The experimental Modula-2 Linda system is composed of a server that manages tuple space and mechanisms that enable processes to communicate with the server and to interact with other processes via tuple space (see Figure 6.6).

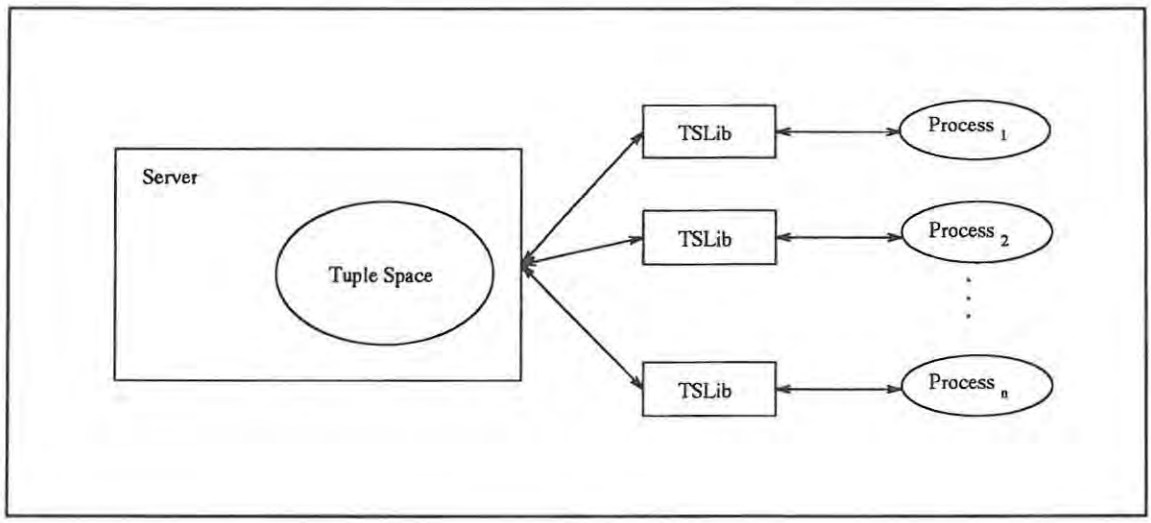

Figure 6.6 Modula-2 Linda system

A separately executing specification handler constructs and manages the model, and implements the recognition engine (see Figure 6.7).

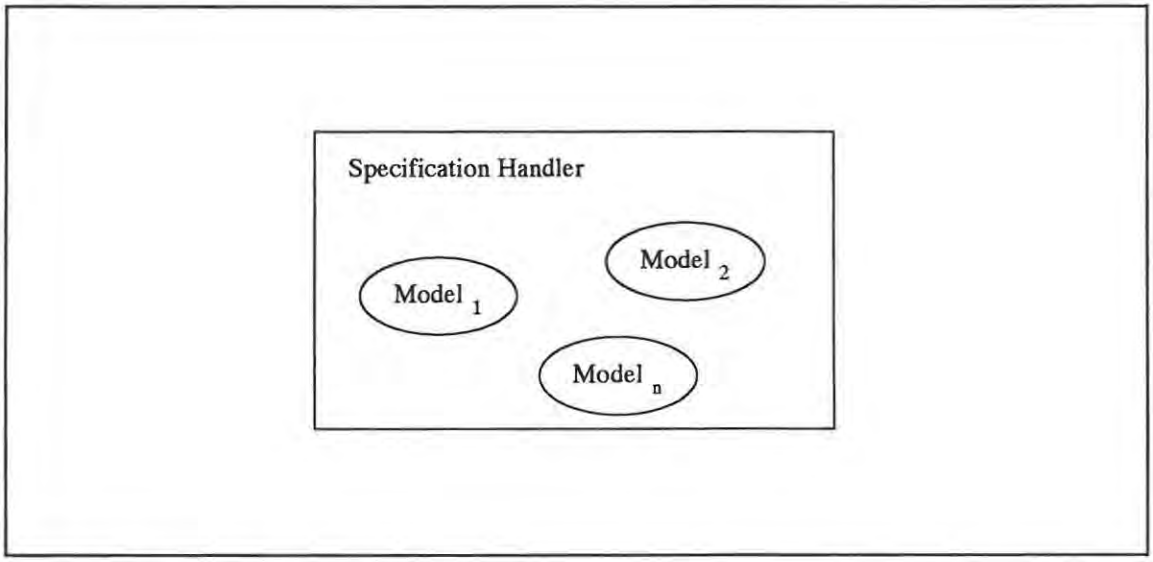

Figure 6.7 The specification handler

The server communicates with the specification handler via a single link. Information that is carried by the link includes:

1. notice of new processes that initiate communication with the server and of old processes that sever communication with the server,
2. actual process behaviour, and

3. the results of expected and actual behaviour comparisons.

Figure 6.8 illustrates the overall system.

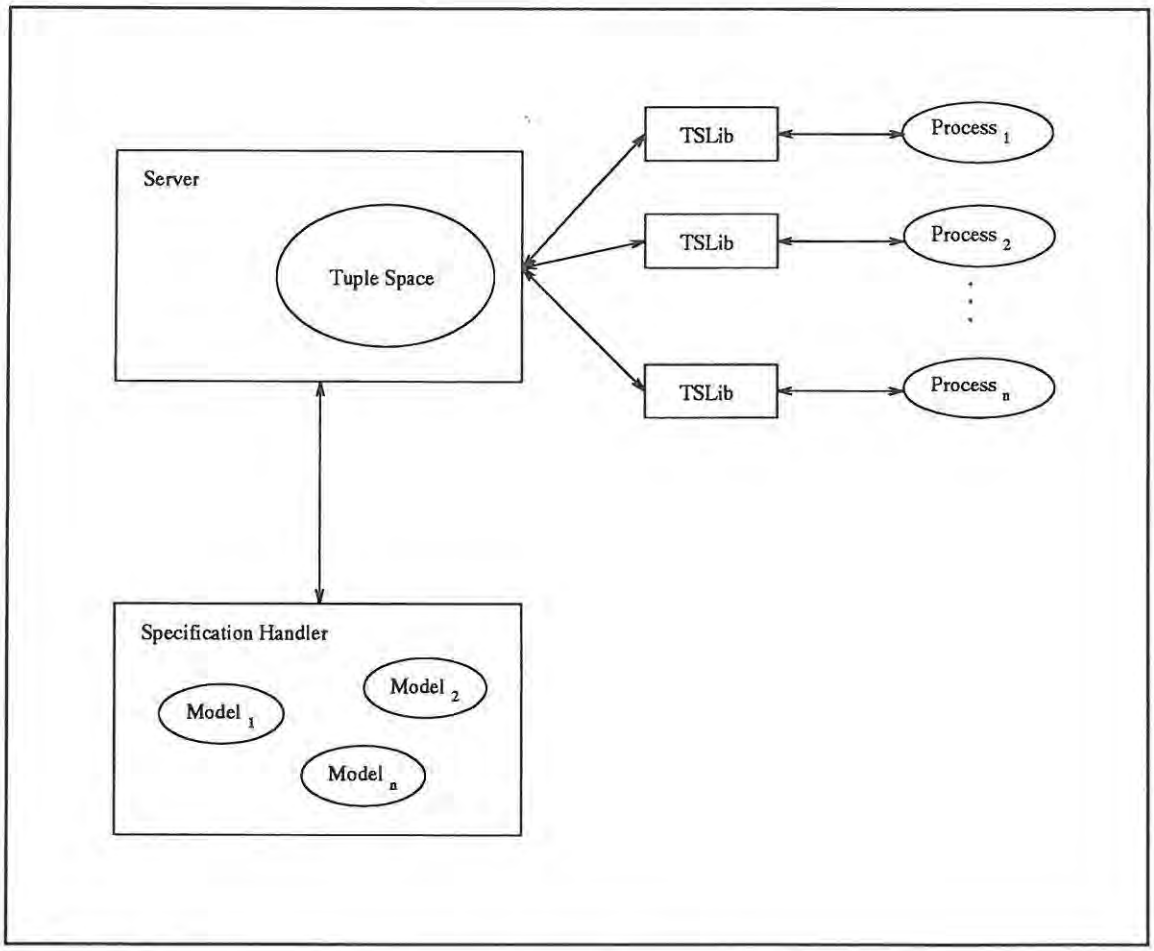

Figure 6.8 Modula-2 Linda system with specification handler

The standard Linda system requires minimal change to incorporate the specification handler[1]. The server code is modified at appropriate points to include communication with the specification handler, whilst Linda application program (process) code remains unchanged. Since the specification handler is a separate program, it executes in its own code and data space. Its execution is free of side-effects - it does not modify the state or order of computation sequence of any Linda process. Neither can it access the code or data space of any Linda process - for the data in which it is interested (tuples added/removed from tuple space), it maintains copies of its own.

[1] Details of the implementation of the debugger can be found in Appendix G.

6.6 Conclusion

An expected behaviour model, based on specifications provided in the experimental specification language for Linda programs, has been presented that acts as a recognition engine.

A new formalism, an extended non-deterministic finite-state automaton (ENFA), is proposed to represent the model. In a standard non-deterministic finite-state automaton (NFA), the alphabet is static, whilst in an ENFA the alphabet is dynamic. The evolving nature of tuples that are coupled to the Linda primitives (symbols of the language) necessitate the new formalism. The value of conversion from a non-deterministic to deterministic automaton, either at model construction time or later, is acknowledged, but made impossible by the dynamic alphabet.

Multiple ENFA's implement the facility whereby processes may be accompanied by more than one sub-specification.

A labelled transition graph implements the ENFA. So that changing values associated with named tuple elements may be recorded, a separate environment (in which values are associated with names) is maintained for each ENFA.

The behaviour recognition algorithm is based on a standard algorithm for the recognition of a language by a non-deterministic machine. A Linda program is said to satisfy its expected behaviour, if all models (ENFA's) for each process accept the stream of behaviour for the process it represents, and reaches a final state.

The recognition engine is implemented as a standalone program (independent of the Linda tuple space server and Linda program code). The server provides the engine with a stream of process behaviour which it then checks. The server code is modified slightly to include communication with the engine. Linda process code remains unchanged. The recognition engine executes in its own code and data space, separate from that of the server and all Linda processes. The execution of the recognition engine is side-effect free - it does not modify the state or execution sequence of the process computation.

Chapter 7

Conclusions and Future Research

7.1 Conclusions

7.1.1 Introduction

The development of fault-free sequential and parallel programs is widely recognised as a non-trivial exercise. Programmers of all levels of competence acknowledge that the programming process requires a whole host of skills, included in which is a touch of serendipity.

In sequential programming environments, it is commonplace to discover, amongst other utilities, program debuggers that aid the inevitable debugging process. Sequential debugging is a well-understood, yet continually improving, process. It is supported by a broad spectrum of debuggers, some of which constitute a more than comprehensive set of manipulative devices that are designed to ferret out the most stubborn of faults.

In parallel programming environments, the situation is very different. Support tools like debuggers, are less frequently encountered and are of questionable use. Debugging parallel programs is made difficult by an inability to reproduce reliably the behaviour of the program, by the influence of the "probe effect", by non-determinism, and by a difficulty in determining the order of occurrence of events in concurrently executing processes. Attempts to construct parallel debuggers have included the application of sequential debugging techniques in the parallel domain, event-based debuggers, visual debuggers, and static analysis debuggers. Their success is limited and variable, partly as a result of the suppression of faults on application of the debugger, the "probe effect", the production of excessive debug detail, or a restriction on the class of faults that can be detected.

This thesis has proposed the use of an event-based behavioural model technique of debugging to debug Linda programs.

7.1.2 Contributions of the Thesis

The investigation of the Linda parallel programming paradigm, and the model for debugging Linda programs shows that Linda programs are amenable to debugging using an event-based behavioural model technique of debugging.

The proposed model of debugging is as follows:

1. A specification, in terms of Linda primitives, is provided of the actions of each process that constitute the Linda program. An experimental specification language for Linda programs is used.
2. The specifications are used to construct models of the expected behaviour of the Linda program upon which behaviour recognition engines are based.
3. At run-time, the behavioural models are used to compare expected with actual program behaviour. Inconsistencies are reported.

It is found that

- The non-deterministic duration of tuple space operations provides the debugger with a convenient slot into which to place its activity without effecting the Linda program semantics or introducing the vagaries of the "probe effect".
- The Linda primitives form a simple, well-defined set of primitive events.
- Spatial and temporal process decoupling promote a process-specific or process oriented debugging approach.
- The design of the debugger is such that it is not necessary to write any code to implement event-generation and event-collection mechanisms. Whereas other debuggers are forced to program special event generation and collection code, the Linda primitives themselves form the only events, and the extant routing of requests to tuple space is an in-place event collection mechanism.
- A global program state space, representing the coordination component of the Linda program, is established whenever tuple space interaction is suspended.
- Tuple space provides a convenient place at which to linearize requests. As it deals, synchronously, with each request, the debugger exploits the opportunity to generate notice of event occurrence.
- Since the debugger is designed to interact with tuple space only, Linda processes are

invariant on application or removal of the debugger.

- Milner's observational equivalence is shown between the basic Linda model and the Linda debugging model when no behavioural mismatches occur, or when mismatch signals are caught by an internal error handler.

The model of debugging demonstrates a number of desirable properties:

- The explicit process specification phase forces the programmer to concentrate attention on the coordination component to the exclusion of the computation component.
- It imposes a structured approach to debugging that is based on a formal model of expected versus actual behaviour, and program transitions from valid states to valid states.
- The debugging process is automated, and requires that the user play a far more passive role during program execution but a more active role during program development than is the case with other parallel debuggers.
- It improves Linda program design by demanding that programs be specified.

Important aspects of the specification language include:

- Linda programs are specified
  - on a per process basis,
  - in terms of Linda primitives,
  - at varying degrees of specificity, and
  - from any number of behavioural levels or views.
- Unlike some systems in which specific events are specified that only model particular aspects of the program's overall behaviour, Linda processes must be specified in full (the degree of specificity may vary, but the specification must still describe full process behaviour).
- Multiple specifications of the same process from different views is facilitated and encouraged.
- The length of specifications is manageable.
- The number of syntactic constructs in the language
  - is minimal,
  - adequate to demonstrate the principles that underpin the debugging methodology, but
  - should be expanded to capture extra behavioural patterns.

Of the expected behaviour models

- They are represented by a new, yet simple, formalism, an extended non-deterministic finite-state automaton (ENFA). A labelled transition graph implements the ENFA.
- Multiple specifications of the same process are handled easily by multiple ENFA's.

The implementation of the debugger is such that

- Unlike many other parallel debuggers (especially those that annotate the target code), the Linda debugger executes in its own code and data space, and is side-effect free.

7.2 Future Research

A number of issues remain unexplored by this thesis. Future research includes the following:

- History Files and Replay

  The production of a history file was considered in chapter 4. However, it was not included in the final definition, nor was it implemented in the final experimental Modula-2 Linda system with debugger. The construction of a full replay system, based on the history file, should be considered. Since a linear order of events is available, a fully-reproducible program execution sequence is possible. The replay system could be a simple browse facility, or a full-blown reconstruction of the execution.

- Tuple Space Organisation

  The Linda paradigm is based on a single, logical tuple space to which all process requests are directed. Logical tuple space deals with all requests, in sequence, and in a monitor-like [Hoa74] fashion. On receipt of a request, tuple space generates notice of the associated event to the debugger for validation. The single stream of events generated by the single, logical tuple space is pivotal to the success of the debugger. In the event that logical tuple space is implemented as a single, physical tuple space, the proposed debugging methodology holds. However, in an attempt to improve tuple space performance, physical tuple space has taken on many new forms, namely

  - partitioned tuple space: based on requests that will be made by processes, tuple space is divided into partitions that service distinct sets of processes,

- distributed tuple space: tuple space is distributed (broken-up) into multiple tuple spaces that may reside across a network of processors, and

- replicated tuple space: tuple space is replicated across a network of processors.

The resultant individual sub-tuple spaces are autonomous but, in the case of distributed and replicated tuple spaces, may communicate with each other. The lack of a single, physical tuple space impacts negatively on the notion of a single stream of events - system performance is improved at the expense of debugging opportunity. The following needs to be assessed:

- the extent to which the proposed model of debugging may be applied to a Linda system in which tuple space is partitioned, distributed or replicated, and
- the extent to which the benefits of the proposed model of debugging and alternative implementations of logical tuple space can be derived by the development of a system that implements both.

The formal model of debugging developed in chapter 4 can be used in the assessment. In that chapter, a Linda system

$$Linda \stackrel{def}{=} (TS(M)|Process_1|Process_2| \ldots | Process_n)\backslash L$$

and a Linda system with debugger

$$LindaD \stackrel{def}{=} ((TSD(M)|Debugger)\backslash Ld| Process_1|Process_2| \ldots |Process_n)\backslash L$$

were specified. The following bisimulation was investigated:

$$Linda \approx LindaD$$

reduced to

$$TS(M) \approx (TSD(M)|Debugger)\backslash Ld$$

If tuple space in the Linda system with debugger were replaced with, for example, a replicated tuple space (*TSRepD(M)*), a *LindaRepD* system results:

$$LindaRepD \stackrel{def}{=} ((TSRepD(M)|Debugger)\backslash Ld| Process_1|Process_2| \ldots |Process_n)\backslash L$$

The following bisimulation can then be investigated:

$$LindaD \approx LindaRepD$$

reduced to

$$TSD(M) \approx TSRepD(M)$$

(This assumes that *Debugger* and all processes are invariant under *LindaD* and *LindaRepD*.)

If this bisimulation can be established, the model of debugging developed in chapter 4 can be applied to a Linda system in which tuple space is replicated. The same procedure follows for a partitioned or distributed tuple space.

- The Specification Language

Chapter 5 described an experimental specification language for Linda programs. The language contains a core set of constructs that demonstrate the model of debugging adequately, but more work needs to be done on the kinds of constructs that best express behavioural patterns. Some constructs that may prove useful include

- a replicate-operator

  "**rep**" "(" [<LowerBound>] "," [<UpperBound>] ","
  <Statement> { "." <Statement> } ")"

  that permits a sequence of actions to occur repeatedly within a lower and upper bound, and

- a predefined primitive-count function

  CNT "(" <LindaOp> ")"

  that gives the specification access to the number of times a particular Linda primitive has occurred.

At a more fundamental level, the nature of the language could also be examined. Presently, the specifications detail the series of actions that the process must carry out. It would be worthwhile to consider the incorporation of constructs that specify actions that the process must NOT do. Rather than specifying the action or range of actions that are currently

permissable, the negative form may be far more succinct.

- The Edinburgh Concurrency Workbench

The Edinburgh Concurrency Workbench (CWB) is an automated tool which caters for the manipulation and analysis of concurrent systems expressed in CCS or a modal logic. Since the specification language is based on value-passing CCS, the possibility of interaction between the debugger and the CWB should be investigated. Essentially, the process specifications could be translated to basic CCS, submitted to the CWB, and analysed to determine immediate and eventual process progress, possible deadlock, and so on. Particular process properties could also be investigated. The following issues would need attention:

- the translation of language constructs not based on value-passing CCS to basic CCS (changing tuple element information),
- the state-space explosion on translation from value-passing to basic CCS, and
- CWB and debugger integration.

- Speculative Evaluation

The debugger is only active when processes interact with tuple space (and tuple space requires the debugger to validate a process request). It is likely that the debugger will have periods in which it is inactive. During this period, the possible future behaviour of processes could be analysed - for similar purposes and in much the same way as the CWB may be utilised (it may be a good place to call on the expertise of the CWB).

- Alternate Behavioural Model Representation

The present implementation of the behavioural model is a labelled transition graph in which transitions are based on single Linda primitives. Transition graphs that represent *random*-constructs are characterised by an "explosive" structure as a result of the many permissable orders of occurrence of actions. For a large random sequence of actions, the "explosion" is dramatic, brought about mainly by the single-action transition. It is worth considering a strategy where transitions are based on a set of actions. In the simplest case, that is, where a single action constitutes the transition, a singleton set results. Then, for a particular state, if the actual action is found in the set, it is an expected action. If so, it is removed from the set, and, if the set is then empty, a transition is made to the next state. For large random sequences of actions, the set would be larger, but the transition rules identical. Manageable graph sizes would result.

- User Interface

  The widespread availability of high-technology graphics monitors and windows-based support software simplifies the construction of quality user interfaces. Not only could the action of the debugger, the process specifications (expressed as a labelled transition graph), and tuple space be depicted, but many of the ideas present in TupleScope [Ber90a] could be incorporated, for example, tuple space browse facilities, and highlight mechanisms for specific tuples.

7.3 In Closing

The ever-increasing demand for computing power places a high premium on the development of fast machines, most of which are parallel processor-based. To date, researchers have been hard-pressed to match the hardware development with software of comparable quality. New paradigms, methodologies, and indeed, ways of thinking are required. Linda combined with an event-based behavioural model technique of debugging offers a contribution to the new order.

Bibliography

[Ada86] Adams, E., Muchnick, S.S. Dbxtool: A Window-Based Symbolic Debugger for Sun Workstations. Software - Practice and Experience, 16(7), 653-669, 1986.

[Ahm91a] Ahmed, S., Gelernter, D. Program Builders as Alternatives to High-Level Languages. Technical Report: YALEU/DCS/RR-887, Department of Computer Science, Yale University, 1991.

[Ahm91b] Ahmed, S., Carriero, N., Gelernter, D. The Linda Program Builder. In: Proceedings of the 3rd Workshop on Languages and Compilers for Parallelism, Irvine, 1990. Also: Languages and Compilers for Parallel Computing II, MIT Press, Cambridge, Massachusetts, 1991.

[Ahu86] Ahuja, S., Carriero, N., Gelernter, D. Domesticating Parallelism - Linda and Friends. Computer, 19(8), 26-34, 1986.

[And79] Andler, S. Predicate Path Expressions: A High-Level Synchronisation Mechanism. Ph.D Thesis, Department of Computer Science, Carnegie-Mellon University, 1979.

[And91] Anderson, B.G., Shasha, D. Persistant Linda: Linda + Transactions + Query Processing. Lecture Notes in Computer Science, 574, 93-109, 1991.

[Bac79] Backhouse, R.C. Syntax of Programming Languages: Theory and Practice. Prentice-Hall International, London, 1979.

[Bai81] Baiardi, F., Fantechi, A., Tomasi, A., Vanneschi, M. Mechanisms for a robust distributed environment in the MuTeam kernel. In: Proceedings of 11th Fault-tolerant Computing Symposium, 24-29, 1981.

[Bai83a] Baiardi, F., Fantechi, A., Vanneschi, M. Language constructs for a robust distributed environment. In: The MuTeam Experience in Designing Distributed Systems of Microprocessors. Bologna, Italy: Tecnoprint, 25-84, 1983.

[Bai83b] Baiardi, F., De Francesco, N., Matteoli, E., Stefanini, S., Vaglini, G. Development of a Debugger for a Concurrent Language. ACM SIGPlan Notices, 18(8), 98-106, 1983.

[Bai86] Baiardi, F., De Francesco, N., Vaglini, G. Development of a Debugger for a Concurrent Language. IEEE Transactions on Software Engineering, SE-12(4), 547-553, 1986.

[Bat82] Bates, P.C., Wileden, J.C. Event Definition Language: An Aid to Monitoring and Debugging of Complex Software Systems. In: Proceedings of the 15th Hawaii International Conference on Systems Science, 1982.

[Bat83a] Bates, P.C., Wileden, J.C. An Approach to High-level Debugging of Distributed Systems (Preliminary Draft). ACM SIGPlan Notices, 18(8), 107-111, 1983.

[Bat83b] Bates, P.C., Wileden, J.C. High-Level Debugging of Distributed Systems: The Behavioural Abstraction Approach. Journal of Systems and Software, 3(4), 255-254, 1983. Reprinted in: Tutorial: Distributed Software Engineering (ed. Shatz, S.M. and Wang, J-P.), 205-214, IEEE Computer Society Press, Washington, 1989.

[Bat87] Bates, P.C. The EBBA Modelling Tool, a.k.a. Event Definition Language. Technical Report: 87-35, University of Massachusetts, 1987.

[Bat89] Bates, P.C. Debugging Heterogeneous Distributed Systems Using Event-Based Models of Behavior. ACM SIGPlan Notices, 24(1), 11-22, 1989.

[Ber90a] Bercovitz, P, Carriero, N. TupleScope: A Graphical Monitor and Debugger for Linda-Based Parallel Programs. Research Report: YALEU/DCS/RR-782, April 1990, Department of Computer Science, Yale University, 1990.

[Ber90b] Berry, G., Boudol, G. The Chemical Abstract Machine. In: Proceedings of the 17th ACM Conference on Principles of Programming Languages, 81-94, 1990.

[Bjo87] Bjornson, R. A Linda User's Manual. Scientific Computing Associates, New Haven, Connecticut, June 1987.

[Bor88] Borrman, L., Herdieckerhoff. M., Klein, A. Tuple Space Integrated in Modula-2: Implementation of the Linda Concept on a Hierarchical Multiprocessor. In: Jesshope and Reinartz (eds.), Proceedings of CONPAR '88, Cambridge University Press, 1988.

[Bru83] Bruegge, B., Hibbard, P. Generalized Path Expressions: A High-Level Debugging Mechanism (Preliminary Draft). ACM SIGPlan Notices, 18(8), 34-44, 1983.

[Bru91] Bruns, G. A Language for Value-Passing CCS. Technical Report: ECS-LFCS-91-175, Laboratory for Foundations in Computer Science, Department of Computer Science, Edinburgh University, 1991.

[Bur88] Burns, A. Programming in occam 2. Addison-Wesley, Wokingham, England, 1988.

[Bus89] Busalacchi, P.J. Linda on Transputer-based Personal Computer. In: Proceedings of the Australian Transputer and OCCAM User Group Conference, 53-57, Glasshouse Theatre, Royal Melbourne Institute of Technology, July 6-7, 1989.

[But91a] Butcher, P., Zedan, H. Lucinda - A Polymorphic Linda. Lecture Notes in Computer Science, 574, 126-146, 1991.

[But91b] Butcher, P. A Behavioural Semantics for Linda-2. Software Engineering Journal, July, 196-204, 1991.

[Cam74] Campbell, R.H., Habermann, A.N. The Specification of Process Synchronisation by Path Expressions. Lecture Notes in Computer Science, 16, 89-102, 1974.

[Car86] Carriero, N., Gelernter, D., Leichter, J. Distributed Data Structures in Linda. In: Proceedings of the 13th Symposium on Principles of Programming Languages, St Petersburg, Fla., January, 1986.

[Car88] Carriero, N., Gelernter, D. Applications Experience with Linda. ACM SIGPlan Notices, 23(9), 173-187, 1988.

[Car89a] Carriero, N., Gelernter, D. Linda in Context. Communications of the ACM, 32(4), 444-458, 1989.

[Car89b] Carriero, N., Gelernter, D. How to Write Parallel Programs: A Guide to the Perplexed. ACM Computing Surveys, 21(3), 323-357, 1989.

[Car90a] Carriero, N., Gelernter, D. Tuple Analysis and Partial Evaluation Strategies in the Linda Precompiler. In: Gelernter, D., Nicolau, A. and Padua, D. (eds) Languages and Compilers for Parallel Computing, 114-125, MIT Press, Cambridge, Massachusetts, 1990.

[Car90b] Carriero, N., Gelernter, D. How to Write Parallel Programs: A First Course. MIT Press, Cambridge, Massachusetts, 1990.

[Car91] Carver, R.H., Tai, K-C. Replay and Testing for Concurrent Programs. IEEE Software, 66-74, March, 1991.

[Car93] Carriero, N. Private Communication. 1993.

[Cia91] Ciancarini, P. Parallel Logic Programming using the Linda Model of Computation. Lecture Notes in Computer Science, 574, 110-125, 1991.

[Cia92] Ciancarini, P., Jensen, K.K., Yanklevich, D. The Semantics of a Parallel Language Based on Shared Dataspaces. Technical Report: 26/92, University of Pisa, 1992.

[Cla89] Clayton, P.G. Interrupt-Generating Active Data Objects. Ph.D Thesis, Department of Computer Science, Rhodes University, 1989.

[Cla92] Clayton, P.G., Wentworth, E.P., Wells, G.C., de-Heer-Menlah, F.K. An Implementation of Linda Tuple Space under the Helios Operating System. SACJ/SART, 6, 3-10, 1992.

[Cle88] Cleaveland, R., Parrow, J., Steffen, B. The Concurrency Workbench: Operating Instructions. Technical Note: 10, Laboratory for Foundations in Computer Science, Department of Computer Science, Edinburgh University, 1988.

[Cle93] Cleaveland, R., Parrow, J., Steffen, B. The Concurrency Workbench: A Semantics-Based Tool for the Verification of Concurrent Systems. ACM Transactions on Programming Languages and Systems, 15(1), 36-72, 1993.

[Clo84] Clocksin, W., Mellish, C. Programming in Prolog. Springer-Verlag, 1984.

[Coh91] Cohn, R. Source Level Debugging of Automatically Parallelized Code. ACM SIGPlan Notices, 26(12), 132-143, 1991.

[Dah90] Dahlen, U. Scheme-Linda. Technical Report. EPCC-TN90-06, Department of Computer and Information Science, Linkoping University, Linkoping Sweden, September, 1990.

[Das85] Dasgupta, P., LeBlanc, R.J., Spafford, E. The Clouds Project: Design and Implementation of a Fault-Tolerant Distributed Operating System. Technical Report: GIT-ICS-85/29, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, Georgia, 1985.

[Das87] Dasgupta, P., LeBlanc, R.J., Appelbe, W. The Clouds Distributed Operating System: Functional Details and Related Work. Technical Report: GIT-ICS-87/42, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, Georgia, 1987.

[Dij68] Dijkstra, E.W. Cooperating Sequential Processes. In: Genuys, F. (ed.), Programming Languages, Academia, N.Y., 43-112, 1968.

[Dor92] Dorr, H. Monitoring with Graph-Grammars as formal operational Models. In: (eds) Kuchen, R., Loogen, R. Proceedings of the 4th International Workshop on the Parallel Implementation of Functional Languages, Aachen, 1992.

[Els89] Elshoff, I.J.P. A Distributed Debugger for Amoeba. ACM SIGPlan Notices, 24(1), 1-10, 1989.

[For89] Forin, A. Debugging of Heterogeneous Parallel Systems. ACM SIGPlan Notices, 24(1), 130-140, 1989.

[Fra91] Francioni, J.M., Albright, L., Jackson, J.A. Debugging Parallel Programs using Sound. ACM SIGPlan Notices, 26(12), 68-75, 1991.

[Fre90] Fredlund, L., Jonsson, B., Parrow, J. An Implementation of a Translational Semantics for an Imperative Language. Lecture Notes in Computer Science, 458, 246-262, 1990.

[Gai85] Gait, J. A Debugger for Concurrent Programs. Software - Practice and Experience, 15(6), 539-554, 1985.

[Gai86] Gait, J. A Probe Effect in Concurrent Programs. Software - Practice and Experience, 16(3), 225-233, 1986.

[Geh84] Gehani, N. Ada: An Advanced Introduction Including Reference Manual for the Ada Programming Language. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1984.

[Gel85] Gelernter, D. Generative Communication in Linda. ACM Transactions on Programming Languages and Systems, 7(1), 81-112, 1985.

[Gel88] Gelernter, D. Getting the Job Done. Byte, 13(12), 301-308, 1988.

[Gol89] Goldszmidt, G.S., Katz, S., Yemini, S. Interactive Blackbox Debugging for Concurrent Languages. ACM SIGPlan Notices, 24(1), 2171-282, 1989.

[Gri88] Griffin, J.H., Wasserman, H.J., McGavran, L.P. A Debugger for Parallel Processes. Software - Practice and Experience, 18(12), 1179-1190, 1988.

[Has91] Hasselbring, W. On Integrating Generative Communication into the Prototyping Language ProSet. Informatik-Bericht: 05-91, Essen University, December 1991.

[Has92] Hasselbring, W. A Formal Z Specification of Pro-Set Linda. Technical Report: 04-92, Fachbereich Mathematik und Informatik, Essen University, 1992.

[Hay87] Hayes, I. Specification Case Studies. Prentice Hall, New York, 1987.

[Haz90] Hazelhurst, S. A Proposal for the Formal Specification of the sematics of Linda. Technical Report: 1990-14, Department of Computer Science, University of the Witwatersrand, 1990.

[Hel85] Helmbold, D., Luckman, D.C. TSL: Task Sequencing Language. In: Proceedings of the Ada International Conference, 255-274, Cambridge University Press, Cambridge, England, 1985.

[Hen80] Hennessy, M., Milner, R. On Observing Nondeterminism and Concurrency. Lecture Notes in Computer Science, 85, 295-309, 1980.

[Hen83] Hennessy, M., Li, W. Translating a Subset of Ada into CCS. In: Bjoener, D. (ed.), Formal Description of Programming Concepts II, 227-249, North-Holland, Amsterdam, 1983.

[Hen85] Hennessy, M., Milner, R. Algebraic Laws for Nondeterminism and Concurrency. Journal of the Association for Computing Machinery, 32, 137-162, 1985.

[Hoa74] Hoare, C.A.R. Monitors: An Operating System Structuring Concept. Communications of the ACM, 17(10), 549-557, 1974.

[Hoa78] Hoare, C.A.R. Communicating Sequential Processes. Communications of the ACM, 21(8), 666-677, 1978.

[Hou89] Hough, A.A., Cuny, J.E. Initial Experience with a Pattern-Oriented Parallel Debugger. ACM SIGPlan Notices, 24(1), 195-205, 1989.

[Hse89] Hseush, W., Kaiser, G.E. Data Path Debugging: Data-Oriented Debugging for a Concurrent Programming Language. ACM SIGPlan Notices, 24(1), 236-247, 1989.

[Hse90] Hseush, W., Kaiser, G.E. Modelling Concurrency in Parallel Debugging. ACM SIGPlan Notices, 25(3), 11-20, 1990.

[Inm84] Inmos Ltd. Occam Programming Manual. Prentice-Hall, 1984.

[Jen90] Jensen, K.K. The Semantics of Tuple Space and Correctness of an Implementation. Research Report: 788, Yale University, 1990.

[Ker78] Kerningham, B.W., Ritchie, D.M. The C Programming Language. Prentice-Hall, Englewood-Cliffs, New Jersey, 1978.

[Kil91] Kilpatrick, C., Schwan, K. ChaosMON - Application-Specific Monitoring and Display of Performance Information for Parallel and Distributed Systems. ACM SIGPlan Notices, 26(12), 5767, 1991.

[Koz83] Kozen, D. Results on the Propositional mu-Calculus. Theoretical Computer Science, 27, 333-354, 1983.

[Lam78] Lamport, L. Time, Clocks, and the Ordering of Events in a Distributed System. Communications of the ACM, 21(7), 558-565, 1978.

[Laz86] Lazzerini, B., Prete, C.A. DISDEB: An Interactive High-Level Debugging System for a Multi-Microprocessor System. Microprocessing and Microprogramming, 18, 401-408, 1986.

[LeB85a] LeBlanc, R.J., Robbins, A.D. Event-Driven Monitoring of Distributed Programs. Proceedings of the 5th International Conference on Distributed Computing Systems, Denver, Colorado, 515-522, 1985. Reprinted in: Tutorial: Distributed Software Engineering (ed. Shatz, S.M. and Wang, J-P.), 215-222, IEEE Computer Society Press, Washington, 1989.

[LeB85b] LeBlanc, R.J., Wilkes, C.T. Systems programming with Objects and Actions. In: Proceedings of the 5th International Conference on Distributed Computing Systems, Denver, Colorado, 1985.

[LeB87] LeBlanc, T.J., Mellor-Crummey, J.M. Debugging Programs with Instant Replay. IEEE Transactions on Computers, C-36(4), 471-482, 1987.

[Lel90] Leler, W. Linda Meets Unix. IEEE Computer, 23(2), 43-54, 1990.

[Lin89] Lin, C-C., LeBlanc, R.J. Event-Based Debugging of Object/Action Programs. ACM SIGPlan Notices, 24(1), 23-34, 1989.

[Lop89] Lopriore, L. A User Interface Specification for a Program Debugging and Measuring Environment. Software - Practice and Experience, 19(5), 437-460, 1989.

[Luc87] Luckman, D.C. et al. Anna: A Language for Annotating Ada Programs. Lecture Notes in Computer Science, 260, 1987.

[Luc91] Luckman, D.C., Sankar, S, Takahashi, S. Two-Dimensional Pinpointing: Debugging with Formal Specifications. IEEE Software, January, 74-84, 1991.

[Mac82] Maccabe, A.B. Language Features for Fully Distributed Processing Systems. Ph.D Thesis. Technical Report: GIT-ICS-82/12, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, Georgia, 1982.

[Mac90] MacDonald, N. Prolog-Linda. CS4 Project Report, Department of Computer Science, Edinburgh University, 1990.

[Mar81] Martin, I.R. Mara: An Overview of the Civil Implementation. Rapporto Interno Selenia, SPM 2.1, 1981.

[Mar90] Marinescu, D.C., Lumpp, J.E., Casavant, T.L., Siegel, H.J. Models for Monitoring and Debugging Tools for Parallel and Distributed Software. Journal of Parallel and Distributed Computing, 9, 171-184, 1990.

[Mat88] Matsuoka, S., Kawai, S. Using Tuple Space Communication in Distributed Object-Oriented Languages. In: Proceedings of OOPSLA '88, San Diego, September, 276-284, 1988. Also in: ACM SIGPlan Notices, 23(11), 276-284, 1988.

[McD89] McDowell, C.E., Helmbold, D.P. Debugging Concurrent Programs. ACM Computing Surveys, 21(4), 593-622, 1989.

[Mif92a] Mifsud, A. A Semantic Description of Mseq-Linda in CCS. Draft Report, 1992.

[Mif92b] Mifsud, A. Semantic Development of Distributed Linda Systems. M.Sc Thesis, Department of Computer Science, Edinburgh University, 1992.

[Mil89] Milner, R. Communication and Concurrency, Prentice Hall, New York, 1989.

[Moe92] Moe, P. PanParallacea: A System for Debugging and Monitoring Parallel Programs. Ph.D Thesis, Department of Computer Systems, Norwegian Institute of Technology, 1992.

[Net91] Netzer, R.H.B. Race Condition Detection for Debugging Shared-Memory Parallel Programs. Ph.D Thesis, Department of Computer Science, University of Wisconsin-Madison, 1991.

[Ols91a] Olsson, R.A., Crawford, R.H., Ho, W.W. A Dataflow Approach to Event-Based Debugging. Software - Practice and Experience, 21(2), 209-229, 1991.

[Ols91b] Olsson, R.A., Crawford, R.H., Ho, W.W., Wee, C.E. Sequential Debugging at a High Level of Abstraction. IEEE Software, May, 27-36, 1991.

[Pan91a] Pancake, C.M., Utter, P.S. A Bibliography of Parallel Debuggers, 1990 Edition. ACM SIGPlan Notices, 26(1), 21-37, 1991.

[Pan91b] Pancake, C.M. Software Support for Parallel Computing: Where are we headed? Communications of the ACM, 34(11), 53-64, 1991.

[Pan93] Pancake.C.M., Netzer, R.H.B. A Bibliography of Parallel Debuggers, 1993 Edition. This bibliography is available via anonymous ftp at: cs.orst.edu and wilma.cs.brown.edu

[Pet81] Peterson, J.L. Petri Net Theory and The Modelling of Systems. Prentice-Hall, Englewood-Cliffs, New Jersey, 1981.

[Pin91] Pineo, P.P., Soffa, M.L. Debugging Parallelized Code using Code Liberation Techniques. ACM SIGPlan Notices, 26(12), 108-119, 1991.

[Plo81] Plotkin, G. A Structured Approach to Operational Semantics. Technical Report: DAIMI FN-19, Department of Computer Science, Aarhus University, 1981.

[Pon91] Ponamgi, M.K., Hseush, W., Kaiser, G.E. Debugging Multithreaded Programs with MPD. IEEE Software, May, 37-43, 1991.

[Ros91] Rosenblum, D.S. Specifying Concurrent Systems with TSL. IEEE Software, May, 52-61, 1991.

[Rub89] Rubin, R.V., Rudolf, L., Zernik, D. Debugging Parallel Programs in Parallel. ACM SIGPlan Notices, 24(1), 216-225, 1989.

[San93] Sankar, S., Mandal, M. Concurrent Runtime Monitoring of Formally Specified Programs. Computer, 26(3), 32-41, 1993.

[Sci93] Fortran-Linda Update. Scientific Observations (publication of: Scientific Computing Associates Inc.), Fall 1993.

[Sha90] Shatz, S.M., Mai, K., Black, C., Tu, S. Design and Implementation of a Petri Net Based Toolkit for Ada Tasking Analysis. IEEE Transactions on Parallel and Distributed Systems, 1(4), 424-441, 1990.

[She93] Shekar, K.H., Srikant, Y.N. Linda Sub System on Transputers. Computer Languages, 18(2), 125-136, 1993.

[Smi85] Smith, E.T. A Debugger for Message-based Processes. Software - Practice and Experience, 15(11), 1073-1086, 1985.

[Spi92] Spivey, J.M. The Z Notation: A Reference Manual, Prentice Hall, New York, 1992.

[Sta89] Stallman, R.M. GDB Manual (The GNU Source Level Debugger), Third Edition, GDB version 3.1, Free Software Foundation, Cambridge, MA. January, 1989.

[Sti91] Stirling, C. An Introduction to Modal and Temporal Logics for CCS. Lecture Notes in Computer Science, 491, 2-20, 1991.

[Sti92] Stirling, C. Modal and Temporal Logics for Processes. Technical Report: ECS-LFCS-92-221, Laboratory for Foundations in Computer Science, Department of Computer Science, Edinburgh University, 1992.

[Tai91] Tai, K-C., Carver, R.H., Obaid, E.E. Debugging Concurrent Ada Programs with Deterministic Execution. IEEE Transactions on Software Engineering, 17(1), 45-63, 1991.

[Utt89] Utter, P.S., Pancake, C.M. Advances in Parallel Debuggers: New Approaches to Visualization. Theory Center Technical Report: CTC89TR18, 12/89, Cornell Theory Center, Cornell University, 1989.

[Ven89] Venables, P.J., Zedan, H. Debugging and Monitoring Highly Parallel Systems with GRIP. Microprocessing and Microprogramming, 28, 79-84, 1989.

[Wal87] Walker, D. Introduction to a Calculus of Communicating Systems. Technical Report: ECS-LFCS-87-22, Laboratory for Foundations in Computer Science, Department of Computer Science, Edinburgh University, 1987.

[Wir85] Wirth, N. Programming in Modula-2 (3rd edition). Springer-Verlag, New York, 1985.

[Yue90] Yuen, C.K., Wong, W.F. BaLinda Lisp: A Parallel List-Processing Language. In: Proceedings of the 2nd IEEE International Workshop on Tools for Artificial Intelligence, Faifax, Virginia, 618-624, November, 1990.

[Zer91] Zernik, D., Rudolf, L. Animating Work and Time for Debugging Parallel Programs Foundation and Experience. ACM SIGPlan Notices, 26(12), 46-56, 1991.

Appendix A

Glossary of Symbols

A glossary of symbols.

Tuples

u	-	tuple (template)
u'	-	free tuple that matches u
$\perp$	-	formal tuple element

Entities

l	-	label
s	-	action sequence
L	-	sort
P	-	agent
E,F	-	agent expressions
x	-	value variable
e	-	value expression
b	-	boolean expression
ϕ	-	formula of process logic
f	-	relabelling function

Set Constructions

$\varnothing$	-	empty set
$-$	-	set difference
$\cup$	-	set union
$\uplus$	-	multiset union
$\tilde{x}$	-	indexed set $\{x_i : i \in I\}$ (I understood)

Action Constructions

τ	-	silent action
$\overline{l}$	-	label complement
ξ	-	empty action sequence

Transitions

$\xrightarrow{l}$	-	l-transition
$\xrightarrow{\tau}$	-	silent transition
$\xRightarrow{\xi}$	-	transitive reflexive closure of $\xrightarrow{\tau}$
$\xRightarrow{l}$	-	$\xRightarrow{\xi} \xrightarrow{l} \xRightarrow{\xi}$

Basic Agent Constructions

$l.E$	-	prefix
$\mathbf{0}$	-	inactive agent
$E + F$	-	summation
$\Sigma_{i \in I} E_i$	-	summation over an indexing set
$E \mid F$	-	composition
$\Pi_{i \in I} E_i$	-	composition over an indexing set
$E \backslash L$	-	restriction
$E[f]$	-	relabelling

Value-Passing Agent Constructions

$l(x).E$	-	prefix (input of values)
$\overline{l}(e).E$	-	prefix (output of values)
if b then E	-	conditional
$A(\widetilde{x}) \stackrel{\text{def}}{=} E$	-	parametric agent definition

Agent Equivalence Relation

$E \approx F$	-	observational equivalence

Basic Logical Constructions

$<s>\,\phi$	-	possibility
$\bigwedge_{i \in I} \phi_i$	-	conjunction
$P \models \phi$	-	satisfaction

Derived Logical Constructions

`tt`	-	truth
`ff`	-	falsity
$[s]\,\phi$	-	necessity
$<<s>>\,\phi$	-	weak possibility
$[[s]]\,\phi$	-	weak necessity
$\bigvee_{i \in I} \phi_i$	-	disjunction

Extended Logical Constructions

Z	-	propositional variable
$\nu Z.\phi$	-	maximal fixed point operator (ν) in the modal equation Z
$\mu Z.\phi$	-	minimal fixed point operator (μ) in the modal equation Z

Appendix B

CCS Specifications

This appendix contains CCS specifications for the full Linda system and the full Linda system with debugger.

B.1 Linda System

The definition is composed of three sections, namely: tuple space (*TS(M)*), the distinguished process ($Process_1$), and spawned processes ($Process_{sp}$):

$$
\begin{aligned}
TS(M) \quad &\stackrel{def}{=} \quad out_p(u).TS(M \uplus \{u\}) + \\
&\qquad in req_p(u).TSinreq(M, u, p) + \\
&\qquad repinreq_p(u).TSinreq(M, u, p) + \\
&\qquad rdreq_p(u).TSrdreq(M, u, p) + \\
&\qquad reprdreq_p(u).TSrdreq(M, u, p) + \\
&\qquad inpreq_p(u).TSinpreq(M, u, p) + \\
&\qquad rdpreq_p(u).TSrdpreq(M, u, p) \\
&\qquad eval_p(u).TS(M)
\end{aligned}
$$

$$
\begin{aligned}
TSinreq(M, u, p) \quad &\stackrel{def}{=} \quad \text{if } match(M, u) = \emptyset \\
&\qquad \text{then} \quad \overline{fail}.TS(M) \\
&\qquad \text{else} \quad \overline{in_p}(u').TS(M - \{u'\})
\end{aligned}
$$

$$
\begin{aligned}
TSrdreq(M, u, p) \quad &\stackrel{def}{=} \quad \text{if } match(M, u) = \emptyset \\
&\qquad \text{then} \quad \overline{fail}.TS(M) \\
&\qquad \text{else} \quad \overline{rd_p}(u').TS(M)
\end{aligned}
$$

$TSinpreq(M, u, p) \stackrel{def}{=}$ *if* $match(M, u) = \emptyset$
then $\overline{fail}.TS(M)$
else $\overline{inp_p}(u').TS(M - \{u'\})$

$TSrdpreq(M, u, p) \stackrel{def}{=}$ *if* $match(M, u) = \emptyset$
then $\overline{fail}.TS(M)$
else $\overline{rdp_p}(u').TS(M)$

$Process_1 \stackrel{def}{=}$ $\overline{out_1}(u).ProcessOut_1 +$
$\overline{inreq_1}(u).ProcessIn_1(u) +$
$\overline{rdreq_1}(u).ProcessRd_1(u) +$
$\overline{inpreq_1}(u).ProcessInp_1 +$
$\overline{rdpreq_1}(u).ProcessRdp_1 +$
$\overline{eval_1}(u).\overline{start_{u'}}.ProcessEval_1$
$\overline{term_1}.\mathbf{0}$

$ProcessOut_1 \stackrel{def}{=}$ $\overline{done_1}.Process_1$

$ProcessIn_1(u) \stackrel{def}{=}$ $in_1(u').\overline{done_1}.Process_1 +$
$fail.\overline{repinreq_1}(u).ProcessIn_1(u)$

$ProcessRd_1(u) \stackrel{def}{=}$ $rd_1(u').\overline{done_1}.Process_1 +$
$fail.\overline{reprdreq_p}(u).ProcessRd_1(u)$

$ProcessInp_1 \stackrel{def}{=}$ $fail.\overline{res_1}(false).Process_1 +$
$inp_1(u').\overline{res_1}(true).Process_1$

$ProcessRdp_1 \stackrel{def}{=}$ $fail.\overline{res_1}(false).Process_1 +$
$rdp_1(u').\overline{res_1}(true).Process_1$

$ProcessEval_1 \stackrel{def}{=}$ $\overline{done_1}.Process_1$

$Process_{sp} \stackrel{def}{=}$ $start_{sp}.ProcessSt_{sp}$

$ProcessSt_{sp} \stackrel{def}{=}$ $\overline{out_{sp}}(u).ProcessStOut_{sp} +$
$\overline{inreq_{sp}}(u).ProcessStIn_{sp}(u) +$
$\overline{rdreq_{sp}}(u).ProcessStRd_{sp}(u) +$
$\overline{inpreq_{sp}}(u).ProcessStInp_{sp} +$
$\overline{rdpreq_{sp}}(u).ProcessStRdp_{sp} +$
$\overline{eval_{sp}}(u).\overline{start_{u'}}.ProcessStEval_{sp}$
$\overline{term_{sp}}.\mathbf{0}$

$ProcessStOut_{sp} \stackrel{def}{=} \overline{done_{sp}}.ProcessSt_{sp}$

$ProcessStIn_{sp}(u) \stackrel{def}{=} in_{sp}(u').\overline{done_{sp}}.ProcessSt_{sp} + fail.\overline{repinreq_{sp}}(u).ProcessStIn_{sp}(u)$

$ProcessStRd_{sp}(u) \stackrel{def}{=} rd_{sp}(u').\overline{done_{sp}}.ProcessSt_{sp} + fail.\overline{reprdreq_{sp}}(u).ProcessStRd_{sp}(u)$

$ProcessStInp_{sp} \stackrel{def}{=} fail.\overline{res_{sp}}(false).ProcessSt_{sp} + inp_{sp}(u').\overline{res_{sp}}(true).ProcessSt_{sp}$

$ProcessStRdp_{sp} \stackrel{def}{=} fail.\overline{res_{sp}}(false).ProcessSt_{sp} + rdp_{sp}(u').\overline{res_{sp}}(true).ProcessSt_{sp}$

$ProcessStEval_{sp} \stackrel{def}{=} \overline{done_{sp}}.ProcessSt_{sp}$

where: 1. $u \in$ set of all tuples
2. $u' \in match(M, u)$
3. P = set of all process identifiers
4. $p \in P$
5. $sp \in (P - \{1\})$

The Linda system is then specified as follows:

$Linda \stackrel{def}{=} (TS(M)|Process_1|Process_2| \ldots | Process_n)\backslash L$

where: 1. $Process_1$ is the distinguished process.
2. $Process_2$ .. $Process_n$ are spawned processes.
3. $L = \{\bigcup_{p \in P} (out_p, inreq_p, repinreq_p, in_p, rdreq_p, reprdreq_p, rd_p, inpreq_p, inp_p, rdpreq_p, rdp_p, eval_p, start_p) \cup fail\}$

B.2 Linda System with Debugger

The definition is composed of four sections, namely: tuple space (*TS(M)*), the debugger (*Debugger*), the distinguished process ($Process_1$), and spawned processes ($Process_{sp}$):

$$
\begin{aligned}
TSD(M) \stackrel{def}{=}\ & out_p(u).\overline{checkout_p}(u).result.TSD(M \uplus \{u\}) + \\
& inreq_p(u).\overline{checkin_p}(u).result.TSDinreq(M, u, p) + \\
& repinreq_p(u).TSDinreq(M, u, p) + \\
& rdreq_p(u).\overline{checkrd_p}(u).result.TSDrdreq(M, u, p) + \\
& reprdreq(_p(u).TSDrdreq(M, u, p) + \\
& inpreq_p(u).\overline{checkinp_p}(u).result.TSDinpreq(M, u, p) + \\
& rdpreq_p(u).\overline{checkrdp_p}(u).result.TSDrdpreq(M, u, p) + \\
& eval_p(u).\overline{checkeval_p}(u).result.TSD(M)
\end{aligned}
$$

$$
\begin{aligned}
TSDinreq(M, u, p) \stackrel{def}{=}\ & \textit{if } match(M, u) = \varnothing \\
& \quad \textit{then} \quad \overline{fail}.TSD(M) \\
& \quad \textit{else} \quad \overline{in_p}(u').TSD(M - \{u'\})
\end{aligned}
$$

$$
\begin{aligned}
TSDrdreq(M, u, p) \stackrel{def}{=}\ & \textit{if } match(M, u) = \varnothing \\
& \quad \textit{then} \quad \overline{fail}.TSD(M) \\
& \quad \textit{else} \quad \overline{rd_p}(u').TSD(M)
\end{aligned}
$$

$$
\begin{aligned}
TSDinpreq(M, u, p) \stackrel{def}{=}\ & \textit{if } match(M, u) = \varnothing \\
& \quad \textit{then} \quad \overline{fail}.TSD(M) \\
& \quad \textit{else} \quad \overline{inp_p}(u').TSD(M - \{u'\})
\end{aligned}
$$

$$
\begin{aligned}
TSDrdpreq(M, u, p) \stackrel{def}{=}\ & \textit{if } match(M, u) = \varnothing \\
& \quad \textit{then} \quad \overline{fail}.TSD(M) \\
& \quad \textit{else} \quad \overline{rdp_p}(u').TSD(M)
\end{aligned}
$$

$$
\begin{aligned}
Debugger \stackrel{def}{=}\ & checkout_p(u). \\
& \ (\overline{result}.Debugger + \\
& \ \overline{failout_p}(u).\overline{result}.Debugger) + \\
& checkin_p(u). \\
& \ (\overline{result}.Debugger + \\
& \ \overline{failin_p}(u).\overline{result}.Debugger) + \\
& checkrd_p(u). \\
& \ (\overline{result}.Debugger + \\
& \ \overline{failrd_p}(u).\overline{result}.Debugger) +
\end{aligned}
$$

$checkinp_p(u).$
$(\overline{result.Debugger} +$
$\overline{failinp_p}(u).\overline{result.Debugger})$ +
$checkrdp_p(u).$
$(\overline{result.Debugger} +$
$\overline{failrdp_p}(u).\overline{result.Debugger})$ +
$checkeval_p(u).$
$(\overline{result.Debugger} +$
$\overline{faileval_p}(u).\overline{result.Debugger})$

$Process_I \stackrel{def}{=} \overline{out_I}(u).ProcessOut_I +$
$\overline{inreq_I}(u).ProcessIn_I(u) +$
$\overline{rdreq_I}(u).ProcessRd_I(u) +$
$\overline{inpreq_I}(u).ProcessInp_I +$
$\overline{rdpreq_I}(u).ProcessRdp_I +$
$\overline{eval_I}(u).\overline{start_u}.ProcessEval_I$
$\overline{term_I}.\mathbf{0}$

$ProcessOut_I \stackrel{def}{=} \overline{done_I}.Process_I$

$ProcessIn_I(u) \stackrel{def}{=} in_I(u').\overline{done_I}.Process_I +$
$fail.\overline{repinreq_I}(u).ProcessIn_I(u)$

$ProcessRd_I(u) \stackrel{def}{=} rd_I(u').\overline{done_I}.Process_I +$
$fail.\overline{reprdreq_p}(u).ProcessRd_I(u)$

$ProcessInp_I \stackrel{def}{=} fail.\overline{res_I}(false).Process_I +$
$inp_I(u').\overline{res_I}(true).Process_I$

$ProcessRdp_I \stackrel{def}{=} fail.\overline{res_I}(false).Process_I +$
$rdp_I(u').\overline{res_I}(true).Process_I$

$ProcessEval_I \stackrel{def}{=} \overline{done_I}.Process_I$

$Process_{sp} \stackrel{def}{=} start_{sp}.ProcessSt_{sp}$

$ProcessSt_{sp} \stackrel{def}{=} \overline{out_{sp}}(u).ProcessStOut_{sp} +$
$\overline{inreq_{sp}}(u).ProcessStIn_{sp}(u) +$
$\overline{rdreq_{sp}}(u).ProcessStRd_{sp}(u) +$
$\overline{inpreq_{sp}}(u).ProcessStInp_{sp} +$
$\overline{rdpreq_{sp}}(u).ProcessStRdp_{sp} +$

$\overline{eval_{sp}}(u).\overline{start_u}.ProcessStEval_{sp}$
$\overline{term_{sp}}.\mathbf{0}$

$ProcessStOut_{sp} \stackrel{def}{=} \overline{done_{sp}}.ProcessSt_{sp}$

$ProcessStIn_{sp}(u) \stackrel{def}{=} in_{sp}(u').\overline{done_{sp}}.ProcessSt_{sp} +$
$fail.\overline{repinreq_{sp}}(u).ProcessStIn_{sp}(u)$

$ProcessStRd_{sp}(u) \stackrel{def}{=} rd_{sp}(u').\overline{done_{sp}}.ProcessSt_{sp} +$
$fail.\overline{reprdreq_{sp}}(u).ProcessStRd_{sp}(u)$

$ProcessStInp_{sp} \stackrel{def}{=} fail.\overline{res_{sp}}(false).ProcessSt_{sp} +$
$inp_{sp}(u').\overline{res_{sp}}(true).ProcessSt_{sp}$

$ProcessStRdp_{sp} \stackrel{def}{=} fail.\overline{res_{sp}}(false).ProcessSt_{sp} +$
$rdp_{sp}(u').\overline{res_{sp}}(true).ProcessSt_{sp}$

$ProcessStEval_{sp} \stackrel{def}{=} \overline{done_{sp}}.ProcessSt_{sp}$

where: 1. $u \in$ set of all tuples
2. $u' \in match(M, u)$
3. P = set of all process identifiers
4. $p \in P$
5. $sp \in (P - \{1\})$

Note that in *Linda* and *LindaD*, the definitions for the distinguished process ($Process_1$), and spawned processes ($Process_{sp}$) are the same.

The Linda system with debugger is then defined as:

$LindaD \stackrel{def}{=} ((TSD(M)|Debugger)\backslash Ld|$
$Process_1|Process_2| \ldots |Process_n)\backslash L$

where: 1. $Ld = \{\bigcup_{p \in P} (checkout_p\ checkin_p\ checkrd_p\ checkinp_p\ checkrdp_p\ checkeval_p)$
$\cup\ result\}$
2. $L = \{\bigcup_{p \in P} (out_p\ inreq_p\ repinreq_p\ in_p\ rdreq_p\ reprdreq_p\ rd_p\ inpreq_p\ inp_p$
$rdpreq_p\ rdp_p\ eval_p) \cup fail\}$

Appendix C

An Alternative Debugging Model

Section 4.5.2.1 suggested that Linda processes ought to be invariant on application or removal of the debugger. However, it may be argued that processes ought to be invariant until a mismatch occurs in actual and expected behaviour, at which stage they are terminated. This theme is now explored.

If processes must terminate as a result of behavioural inconsistencies, they must be informed of such inconsistencies, that is, a match result message must be routed back to the process. It is reasonable to suggest that such message transmission fits the current model, since all Linda primitives generate some form of reply from tuple space (either an acknowledgement, a boolean result or a tuple) and a match/mismatch reply merely adds to the list.

Essentially the debugger must transmit the result of the behavioural comparison to tuple space which must then act on the result, and then transmit it on back to the relevant process. To do this, new labels are introduced to the debugging model: $\overline{good}$ - *good* and $\overline{bad}$ - *bad* communicate comparison results between the debugger and tuple space, and $\overline{goodreq}$ - *goodreq* and $\overline{badreq}$ - *badreq* communication comparison results between tuple space and processes.

The definition is composed of four sections, namely: tuple space (*TS(M)*), the debugger (*Debugger*), the distinguished process ($Process_1$), and spawned processes ($Process_{sp}$):

$$
\begin{aligned}
TSD(M) \quad \stackrel{\text{def}}{=} \quad & out_p(u).\overline{checkout_p}(u). \\
& \quad (good.\overline{goodreq}.TSD(M \uplus \{u\}) + \\
& \quad bad.\overline{badreq}.TSD(M)) \quad + \\
& inreq_p(u).\overline{checkin_p}(u). \\
& \quad (good.\overline{goodreq}.TSDinreq(M, u, p) + \\
& \quad bad.\overline{badreq}.TSD(M)) \quad + \\
& repinreq_p(u).TSDinreq(M, u, p) \quad +
\end{aligned}
$$

$rdreq_p(u).\overline{checkrd_p}(u).$
$\quad (good.\overline{goodreq}.TSDrdreq(M, u, p) +$
$\quad bad.\overline{badreq}.TSD(M))$ +
$reprdreq_p(u).TSDrdreq(M, u, p)$ +
$inpreq_p(u).\overline{checkinp_p}(u).$
$\quad (good.\overline{goodreq}..TSDinpreq(M, u, p) +$
$\quad bad.\overline{badreq}.TSD(M))$ +
$rdpreq_p(u).\overline{checkrdp_p}(u).$
$\quad (good.\overline{goodreq}.TSDrdpreq(M, u, p) +$
$\quad bad.\overline{badreq}.TSD(M))$ +
$eval_p(u).\overline{checkeval_p}(u).$
$\quad (good.\overline{goodreq}.TSD(M) +$
$\quad bad.\overline{badreq}.TSD(M))$

$TSDinreq(M, u, p) \stackrel{def}{=}$ *if* $match(M, u) = \emptyset$
then $\overline{fail}.TSD(M)$
else $\overline{in_p}(u').TSD(M - \{u'\})$

$TSDrdreq(M, u, p) \stackrel{def}{=}$ *if* $match(M, u) = \emptyset$
then $\overline{fail}.TSD(M)$
else $\overline{rd_p}(u').TSD(M)$

$TSDinpreq(M, u, p) \stackrel{def}{=}$ *if* $match(M, u) = \emptyset$
then $\overline{fail}.TSD(M)$
else $\overline{inp_p}(u').TSD(M - \{u'\})$

$TSDrdpreq(M, u, p) \stackrel{def}{=}$ *if* $match(M, u) = \emptyset$
then $\overline{fail}.TSD(M)$
else $\overline{rdp_p}(u').TSD(M)$

$Debugger \stackrel{def}{=}$ $checkout_p(u).$
$\quad (\overline{good}.Debugger +$
$\quad \overline{failout_p}(u).\overline{bad}.Debugger)$ +
$checkin_p(u).$
$\quad (\overline{good}.Debugger +$
$\quad \overline{failin_p}(u).\overline{bad}.Debugger)$ +
$checkrd_p(u).$
$\quad (\overline{good}.Debugger +$
$\quad \overline{failrd_p}(u).\overline{bad}.Debugger)$ +

$checkinp_p(u).$
$(\overline{good.Debugger} +$
$\overline{failinp_p}(u).\overline{bad.Debugger})$ +
$checkrdp_p(u).$
$(\overline{good.Debugger} +$
$\overline{failrdp_p}(u).\overline{bad.Debugger})$ +
$checkeval_p(u).$
$(\overline{good.Debugger} +$
$\overline{faileval_p}(u).\overline{bad.Debugger})$

$Process_1 \stackrel{def}{=} \overline{out_1}(u).(goodreq.ProcessOut_1 +$
$badreq.\overline{term_1}(false).\mathbf{0})$ +
$\overline{inreq_1}(u).(goodreq.ProcessIn_1(u) +$
$badreq.\overline{term_1}(false).\mathbf{0})$ +
$\overline{rdreq_1}(u).(goodreq.ProcessRd_1(u) +$
$badreq.\overline{term_1}(false).\mathbf{0})$ +
$\overline{inpreq_1}(u).(goodreq.ProcessInp_1 +$
$badreq.\overline{term_1}(false).\mathbf{0})$ +
$\overline{rdpreq_1}(u).(goodreq.ProcessRdp_1 +$
$badreq.\overline{term_1}(false).\mathbf{0})$ +
$\overline{eval_1}(u).(goodreq.\overline{start_u}.ProcessEval_1$
$badreq.\overline{term_1}(false).\mathbf{0})$ +
$\overline{term_1}(true).\mathbf{0}$

$ProcessOut_1 \stackrel{def}{=} \overline{done_1}.Process_1$

$ProcessIn_1(u) \stackrel{def}{=} in_1(u').\overline{done_1}.Process_1 +$
$fail.\overline{repinreq_1}(u).ProcessIn_1(u)$

$ProcessRd_1(u) \stackrel{def}{=} rd_1(u').\overline{done_1}.Process_1 +$
$fail.\overline{reprdreq_p}(u).ProcessRd_1(u)$

$ProcessInp_1 \stackrel{def}{=} fail.\overline{res_1}(false).Process_1 +$
$inp_1(u').\overline{res_1}(true).Process_1$

$ProcessRdp_1 \stackrel{def}{=} fail.\overline{res_1}(false).Process_1 +$
$rdp_1(u').\overline{res_1}(true).Process_1$

$ProcessEval_1 \stackrel{def}{=} \overline{done_1}.Process_1$

$Process_{sp} \stackrel{def}{=} start_{sp}.ProcessSt_{sp}$

$ProcessSt_{sp} \stackrel{def}{=} \overline{out_{sp}}(u).(goodreq.ProcessStOut_{sp} +$
$\quad badreq.\overline{term_{sp}}(false).\mathbf{0}) +$
$\overline{inreq_{sp}}(u).(goodreq.ProcessStIn_{sp}(u) +$
$\quad badreq.\overline{term_{sp}}(false).\mathbf{0}) +$
$\overline{rdreq_{sp}}(u).(goodreq.ProcessStRd_{sp}(u) +$
$\quad badreq.\overline{term_{sp}}(false).\mathbf{0}) +$
$\overline{inpreq_{sp}}(u).(goodreq.ProcessStInp_{sp} +$
$\quad badreq.\overline{term_{sp}}(false).\mathbf{0}) +$
$\overline{rdpreq_{sp}}(u).(goodreq.ProcessStRdp_{sp} +$
$\quad badreq.\overline{term_{sp}}(false).\mathbf{0}) +$
$\overline{eval_{sp}}(u).(goodreq.\overline{start_u}.ProcessStEval_{sp}$
$\quad badreq.\overline{term_{sp}}(false).\mathbf{0}) +$
$\overline{term_{sp}}(true).\mathbf{0}$

$ProcessStOut_{sp} \stackrel{def}{=} \overline{done_{sp}}.ProcessSt_{sp}$

$ProcessStIn_{sp}(u) \stackrel{def}{=} in_{sp}(u').\overline{done_{sp}}.ProcessSt_{sp} +$
$fail.\overline{repinreq_{sp}}(u).ProcessStIn_{sp}(u)$

$ProcessStRd_{sp}(u) \stackrel{def}{=} rd_{sp}(u').\overline{done_{sp}}.ProcessSt_{sp} +$
$fail.\overline{reprdreq_{p}}(u).ProcessStRd_{p}(u)$

$ProcessStInp_{sp} \stackrel{def}{=} fail.\overline{res_{sp}}(false).ProcessSt_{sp} +$
$inp_{sp}(u').\overline{res_{sp}}(true).ProcessSt_{sp}$

$ProcessStRdp_{sp} \stackrel{def}{=} fail.\overline{res_{sp}}(false).ProcessSt_{sp} +$
$rdp_{sp}(u').\overline{res_{sp}}(true).ProcessSt_{sp}$

$ProcessStEval_{sp} \stackrel{def}{=} \overline{done_{sp}}.ProcessSt_{sp}$

where: 1. $u \in$ set of all tuples
2. $u' \in match(M, u)$
3. P = set of all process identifiers
4. $p \in P$
5. $sp \in (P - \{1\})$

Note that in *Linda* and *LindaD*, the definitions for the distinguished process ($Process_1$), and spawned processes ($Process_{sp}$) are the same.

The Linda system with debugger is then defined as:

$$LindaD \stackrel{def}{=} ((TSD(M)|Debugger)\backslash Ld| Process_1|Process_2| \ldots |Process_n)\backslash L$$

where: 1. $Ld = \{\bigcup_{p \in P} (checkout_p, checkin_p, checkrd_p, checkinp_p, checkrdp_p, checkeval_p) \cup good \cup bad\}$

2. $L = \{\bigcup_{p \in P} (out_p, inreq_p, repinreq_p, in_p, rdreq_p, reprdreq_p, rd_p, inpreq_p, inp_p, rdpreq_p, rdp_p, eval_p) \cup fail \cup goodreq \cup badreq\}$

Appendix D

Syntax of the Linda Program Specification Language

The following is a top down description of the syntax of the Linda program specification language. The customary BNF extensions are used in the defintion:

{}	-	zero or more
[]	-	zero or one
"boldface"	-	terminal symbol

The specification language is case-sensitive. The specification may be entered in free-format.

1.	CompoundSpecification	::=	<CompoundBlock> {<CompoundBlock>}
2.	CompoundBlock	::=	**"spec"** <SpecName> ";" <Specification> **"endspec"**
3.	Specification	::=	<Declarations> <Processes>
4.	Declarations	::=	{**"var"** <CompoundDeclaration>}
5.	CompoundDeclaration	::=	<SimpleDeclaration> {<SimpleDeclaration>}
6.	SimpleDeclaration	::=	<IdentList> ":" <TypeIdentifier>";"

7. IdentList ::= <Identifier> {"," <Identifier>}

8. Processes ::= <ProcessSpec> {";" <ProcessSpec>}

9. ProcessSpec ::= "**process**" <ProcessName> "="
<CompoundStatement>

10. CompoundStatement ::= {<Statement> "."} <FinalStatement>

11. Statement ::= <Random>
| <LindaPrimitive>

12. FinalStatement ::= "**if**" <IfStatement>
| "**choice**" <ChoiceStatement>
| <ProcessName>
| "NIL"

13. Random ::= "**random**" "(" <LindaPrimitive>
{ "." <LindaPrimitive> } ")"

14. LindaPrimitive ::= <LindaOp>
| <WildOp>

15. LindaOp ::= <LindaNonPredOp>
| <LindaPredOp>

16. LindaNonPredOp ::= <NonPredicateOp> <Tuple>

17. LindaPredOp ::= <PredicateOp> <Tuple>

18. NonPredicateOp ::= "**out**"
| "**in**"
| "**read**"
| "**eval**"

19. PredicateOp ::= "**inp**"
| "**readp**"

20. Tuple ::= "(" <TElement> ")"

21. TElement ::= <WildSymbol>
| <Element> {"," <Element>}

22. Element ::= <TypeIdentifier>
| <Expression>
| "?" <FormalIdentifier>

23. WildOp ::= <WildSymbol>

24. IfStatement ::= "(" <Condition>
"then" <CompoundStatement>
"else" <CompoundStatement> ")"

25. Condition ::= <LindaCondition>
| <ExprCondition>

26. LindaCondition ::= <LindaPredOp>

27. ExprCondition ::= <Expression>

28. Expression ::= <SimpleExpression>
[<RelationalOp> <SimpleExpression>]

29. SimpleExpression ::= ["+" | "-"] <Term>
{ "+" | "-" | **"OR"** <Term>}

30. Term ::= <Factor>
{ "*" | **"DIV"** | **"MOD"** | **"AND"**
<Factor>}

31. Factor ::= <Identifier>
| <Integer>
| <String>
| "(" <Expression> ")"
| **"NOT"** <Factor>

32. RelationalOp ::= "<" | ">" | "<=" | ">=" | "#" | "="

33. ChoiceStatement ::= "(" <CompoundStatement>
{ "|" <CompoundStatement>} ")"

34. SpecName ::= <Identifier>

35. ProcessName ::= <Identifier>

36. FormalIdentifier ::= <Identifier>
| <TypeIdentifier>

37. Identifier ::= <Letter>
{<Letter> | <Digit> | <Other>}

38. TypeIdentifier ::= **"int"**
| **"str"**

39. WildSymbol ::= **"*"**

40. Integer ::= INTEGER

41. String ::= "'" {<PrintableChar>} "'"

42. Letter ::= **"a"** | **"b"** | ... | **"z"** |
"A" | **"B"** | ... | **"Z"**

43. Digit ::= **"0"** ... **"9"**

44. Other ::= **"["** | **"]"** | **"."** | **"_"**

45. PrintableChar ::= Implementation defined printable character

Appendix E

Modula-2 Linda Programs and Associated Program Specifications

The following is a collection of example Modula-2 Linda programs and the associated specifications.

E.1 Dining Philosophers

The dining philosophers problem [Dij68], describes the activities of dining philosophers who share common resources (forks).

The philosophers are seated at a table, and alternately eat and think. A single fork is placed between each philosopher. To eat, philosophers must first grab the forks on both their immediate lefthand and righthand sides. Once a philosopher has eaten, the forks are returned to the table, after which a period of thought is entered. It is imperative that two forks are used to eat, only forks adjacent to a philosopher may be used by that philosopher, forks that have been used must be put down before they may be used again, and no deadlock occur (deadlock occurs when all philosophers pick up one fork and then wait for the other to become free - which, of course, it does not).

The solution centres on the use of room tickets ([Car90b] page 183) to solve the specific problem of deadlock. One less room ticket than there are philosophers is made available in the dining-room. Before a philosopher attempts to eat (and therefore grab any forks), a room ticket must be obtained. This ensures that, at any time, at least one philosopher is able to grab two forks, and eat.

The master process (`mastphil`) indicates the number of dining philosophers, spawns the requisite constant number of philosophers, sets the dining-room (philosopher tags, forks, room tickets), and then terminates. Individual philosopher processes (`phil`) repeatedly get room tickets and forks, eat, return the forks and room tickets, and then think.

The `mastphil` specification makes use of a single sub-process. Note that the same identifier is used as both the Linda program `MODULE` name and the sub-process name. The `phil` specification employs variables to store the number of dining philosophers, and an identifying philosopher tag. Two sub-pocesses are used, the second of which is recursive. The **`choice`**-statement provides for continued dining or termination.

```
MODULE mastphil;
(*------------*)
(*  Modula-2 Linda program:  Dining philosophers
    Process:                 master

    An implementation of the dining philosophers problem.
*)

CONST
  NumPhils = 3;
VAR
  I : INTEGER;

BEGIN
  out("num philosophers", NumPhils);

  (* start philosophers *)
  FOR I := 0 TO NumPhils - 1 DO
    eval("phil")
  END;

  (* add philosopher tags, forks and
     room tickets *)
  FOR I := 0 TO NumPhils - 1 DO
    out("philtag", I);
    out("fork", I);
    IF I < NumPhils - 1
      THEN out("room ticket");
    END
  END

  (* now that the dining-room is set for action, terminate *)

END mastphil.

spec Global;
/*--------*/
/*  Program specification:  Dining philosophers
    Process:                mastphil
    Specification level:    global
    Description:            Spawn philosophers, and set the dining-room ready
                            for action.
*/

  process mastphil
     = /* add the number of philosophers coming to dine */
       out('num philosophers', 3).
       /* start philosophers */
       eval('phil').
       eval('phil').
       eval('phil').
       /* add tags, forks, and room tickets */
       out('philtag', 0). out('fork', 0). out('room ticket').
       out('philtag', 1). out('fork', 1). out('room ticket').
       out('philtag', 2). out('fork', 2).
       /* and now you're done */
       NIL

endspec


MODULE phil;
(*--------*)
(*  Modula-2 Linda program:  Dining philosophers
    Process:                 philosopher

    An implementation of the dining philosophers problem.
*)

VAR
  Index, NumPhils, I: INTEGER;

BEGIN
  read("num philosophers", ?NumPhils);
```

```
  (* get a philosopher tag *)
  in("philtag", ?I);

  FOR Index := 1 TO 4 DO
    (* start by THINKING *)
    (* get a room ticket *)
      in("room ticket");
    (* get forks *)
      in("fork", I);
      in("fork", (I + 1) MOD NumPhils);
    (* now EAT *)
    (* return forks *)
      out("fork", I);
      out("fork", (I + 1) MOD NumPhils);
    (* return room ticket *)
      out("room ticket")
  END

END phil.

spec Global;
/*--------*/
/*  Program specification:  Dining philosophers
    Process:                phil
    Specification level:    global
    Description:            Get the number of dining philosophers, an identifying
                            tag, and then dine.
*/

  var
    I, NumPhils : int;

  process phil
     = /* how many philosophers are dining? */
       read('num philosophers', ?NumPhils).
       /* acquire a unique tag */
       in('philtag', ?I).
       /* now dine */
       dine;

  process dine
     = choice (  NIL
               | /* THINK */
                 in('room ticket').
                 in('fork', I).
                 in('fork', (I + 1) MOD NumPhils).
                 /* EAT */
                 random(out('fork', I).
                        out('fork', (I + 1) MOD NumPhils).
                        out('room ticket')).
                 dine)

endspec
```

E.2 Readers and Writers

The readers and writers problem describes the action of a storage device to which mutiple processes wish to write, and from which multiple processes wish to read ([Car90b] page 184). Essentially, many readers or a single writer may have access to the device, but not both.

The master process (`mastrw`) spawns a user-specified number of readers and writers, and initialises the name-accessed (to store the number of readers and writers that are currently active) and stream structures (to store access request orders). Individual reader and writer processes (`readprc` and `writeprc`) repeatedly wait for an appropriate time to act, and then read or write respectively.

In the `mastrw` specification, the **`choice`**-statement is used to control the spawning of unspecified numbers of readers and writers. Note how control is directed to common specifications after each **`eval`**-operation. The `readprc` and `writeprc` specifications are similar, and are able to handle unlimited numbers of read and write operations respectively.

```
MODULE mastrw;
(*----------*)
(*  Modula-2 Linda program:  Readers and Writers problem
    Process:                 master

    An implementation of the readers and writers problem.
*)

FROM EasyInOut      IMPORT WriteString, WriteLn, WriteInt;
FROM ConNum         IMPORT StrToUnsigned_16;
FROM EntryExit      IMPORT argv, argc;

VAR
  Index, NumReaders, NumWriters : INTEGER;
  Success : BOOLEAN;

BEGIN
  StrToUnsigned_16(argv^[1]^, 10, NumReaders, Success);
  IF NOT Success
    THEN WriteString("Problem with number of readers - try again");
         WriteLn;
         HALT
    ELSE WriteString("Number of readers: ");
         WriteInt(NumReaders, 4);
         WriteLn;
         StrToUnsigned_16(argv^[2]^, 10, NumWriters, Success);
         IF NOT Success
           THEN WriteString("Problem with number of writers - try again");
                WriteLn;
                HALT
           ELSE WriteString("Number of writers: ");
                WriteInt(NumWriters, 4);
                WriteLn
         END
  END;

  (* start a number of readers and writers *)
  FOR Index := 1 TO NumReaders DO
    eval("readprc")
  END;
  FOR Index := 1 TO NumWriters DO
    eval("writeprc")
  END;

  (* initialise counters *)
  out("writers", 0);
  out("active-readers", 0);
  out("rw-head", 1);
  out("rw-tail", 1)

END mastrw.
```

```
spec Global;
/*---------*/
/*  Program specification:  Readers and Writers
    Process:                mastrw
    Specification level:    global
    Description:            Spawn a number of readers and writers, and initialise
                            a number of name accessed and stream structures.
*/

  process mastrw
  /* start a number of reader and writer processes */
     = choice (  addcounters
               | eval('readprc').mastrw
               | eval('writeprc').mastrw);

  process addcounters
     = /* initialise counters */
       out('writers', 0).
       out('active-readers', 0).
       out('rw-head', 1).
       out('rw-tail', 1).
       NIL

endspec
```

```
MODULE readprc;
(*-----------*)
(*  Modula-2 Linda program:  Readers and writers problem
    Process:                 reader

    An implementation of a reader of the readers and writers problem.
*)

CONST
  NUMREADS = 2;
VAR
  I, Discard : INTEGER;

  PROCEDURE Increment (CounterName : ARRAY OF CHAR) : INTEGER;
  (* Increment CounterName-accessed structure by 1 *)
  VAR
    Value : INTEGER;
  BEGIN
    in(CounterName, ?Value);
    out(CounterName, Value + 1);
    RETURN Value
  END Increment;

  PROCEDURE Decrement (CounterName : ARRAY OF CHAR) : INTEGER;
  (* Decrement CounterName-accessed structure by 1 *)
  VAR
    Value : INTEGER;
  BEGIN
    in(CounterName, ?Value);
    out(CounterName, Value - 1);
    RETURN Value
  END Decrement;

BEGIN
  FOR I := 0 TO NUMREADS DO
    read("rw-head", Increment("rw-tail"));
    read("writers", 0);

    Discard := Increment("active-readers");
    Discard := Increment("rw-head");

    (* READ !! *)

    Discard := Decrement("active-readers")
  END
END readprc.
```

```
spec Global;
/*--------*/
/*  Program specification:  Readers and writers
    Process:                readprc
    Specification level:    global
    Description:            Repeatedly: add reader job to queue,
                                        wait for job to reach head of queue,
                                        increment number of readers,
                                        delete job from queue,
                                        READ
                                        decrement number of readers
*/

  var
    Value : int;

  process readprc
     = choice (  NIL
               | /* start read process */
                 /* add READER job to queue */
                 in('rw-tail', ?Value).
                 out('rw-tail', Value + 1).
                 /* wait for job to reach head of queue */
                 read('rw-head', Value).
                 /* wait for no writers */
                 read('writers', 0).
                 /* increment number of readers -
                    can have multiple readers */
                 in('active-readers', ?Value).
                 out('active-readers', Value + 1).
                 /* delete READER job from queue - this allows
                    the next job to fire */
                 in('rw-head', ?Value).
                 out('rw-head', Value + 1).
                 /* READ !! */
                 /* stop reading */
                 in('active-readers', ?Value).
                 out('active-readers', Value - 1).
                 /* repeat the process */
                 readprc)

endspec


MODULE writeprc;
(*------------*)
(*  Modula-2 Linda program:  Readers and writers problem
    Process:                 writer

    An implementation of a writer of the readers and writers problem.
*)

CONST
  NUMWRITES = 3;
VAR
  I, Discard : INTEGER;

  PROCEDURE Increment (CounterName : ARRAY OF CHAR) : INTEGER;
  (* Increment CounterName-accessed structure by 1 *)
  VAR
    Value : INTEGER;
  BEGIN
    in(CounterName, ?Value);
    out(CounterName, Value + 1);
    RETURN Value
  END Increment;
```

```
  PROCEDURE Decrement (CounterName : ARRAY OF CHAR) : INTEGER;
  (* Decrement CounterName-accessed structure by 1 *)
  VAR
    Value : INTEGER;
  BEGIN
    in(CounterName, ?Value);
    out(CounterName, Value - 1);
    RETURN Value
  END Decrement;

BEGIN
  FOR I := 0 TO NUMWRITES DO
    read("rw-head", Increment("rw-tail"));
    read("writers", 0);
    read("active-readers", 0);

    Discard := Increment("writers");
    Discard := Increment("rw-head");

    (* WRITE !! *)

    Discard := Decrement("writers")
  END
END writeprc.

spec Global;
/*--------*/
/*  Program specification:  Readers and Writers
    Process:                writeprc
    Specification level:    global
    Description:            Repeatedly: add writer job to queue,
                                        wait for job to reach head of queue,
                                        wait for no other writers,
                                        wait for no other readers,
                                        increment number of writers,
                                        delete job from queue,
                                        WRITE,
                                        decrement number of writers
*/

  var
    Value : int;

  process writeprc
     = choice (  NIL
               | /* start write process */
                 /* add WRITER job to queue */
                 in('rw-tail', ?Value).
                 out('rw-tail', Value + 1).
                 /* wait for job to reach head of queue */
                 read('rw-head', Value).
                 /* wait for no other writers */
                 read('writers', 0).
                 /* wait for no readers */
                 read('active-readers', 0).
                 /* increment number of writers */
                 in('writers', ?Value).
                 out('writers', Value + 1).
                 /* delete WRITER job from queue - this allows
                    the next job to fire */
                 in('rw-head', ?Value).
                 out('rw-head', Value + 1).
                 /* WRITE !! */
                 /* stop Writing */
                 in('writers', ?Value).
                 out('writers', Value - 1).
                 /* repeat the process */
                 writeprc)

endspec
```

E.3 Cross Product of Two Matrices

The following Modula-2 Linda program determines the cross product of two matrices. Details of the program can be found in section 3.2.2.

A master process (`mastcrossp`) spawns a user-specified number of workers, adds the respective rows and columns of the matrices to tuple space, adds a work seed, and then awaits the results. Results are collected in random order. Worker processes (`crossp`) scavenge for a work seed, and immediately replace it with the seed's successor. If the work seed is poisoned, the process terminates. Otherwise, the worker retrieves the relevant row and column from tuple space, computes the result, and adds it to tuple space.

In the `mastcrossp` specification, no ordering is required on the addition of row and column data (the current implementation of the random-construct does, however, impose a sequential ordering). Results are extracted from tuple space in random order. Since values for `I`, `J`, and `Value` are not used, the variables are superfluous - each tuple element could be replaced by `?int`. The following varying degrees of specificity could have been employed: `in(*)`, `in(?int, ?int, ?int)`, or `in(?I, ?J, ?Value)`. The `getresults` sub-process merely provides for a more readable specification. In the `crossp` specification, internally decidable alternation (`if`), based on the value of `Seed`, determines process termination.

```
MODULE mastcrossp;
(*--------------*)
(*  Modula-2 Linda program:  Cross product of two matrices
    Process:                 master

    Implementation of the cross product of two matrices.
*)

FROM EasyInOut      IMPORT WriteString, WriteLn, WriteInt;
FROM ConNum         IMPORT StrToUnsigned_16;
FROM EntryExit      IMPORT argv, argc;

TYPE
  MATRIX = ARRAY [1..3] OF ARRAY [1..3] OF INTEGER;
VAR
  M1,M2,M3 : MATRIX;
  NumWorkers, Index, I, J, Value : INTEGER;
  Success : BOOLEAN;

  PROCEDURE PrintMatrix (Matrix : MATRIX);
  (* Print Matrix *)
  VAR
    I, J : INTEGER;
  BEGIN
    FOR I := 1 TO 3 DO
      FOR J := 1 TO 3 DO
        WriteInt(M1[I,J], 4)
      END;
      WriteLn
    END
  END PrintMatrix;

BEGIN
  (* initialise the matrices with arbitrary values *)
  M1[1,1] := 1; M1[1,2] := 2; M1[1,3] := 3;
  M1[2,1] := 4; M1[2,2] := 5; M1[2,3] := 6;
  M1[3,1] := 7; M1[3,2] := 8; M1[3,3] := 9;

  M2[1,1] := 2; M2[1,2] := 3; M2[1,3] := 5;
  M2[2,1] := 7; M2[2,2] := 2; M2[2,3] := 3;
  M2[3,1] := 5; M2[3,2] := 7; M2[3,3] := 2;
```

```
  StrToUnsigned_16(argv^[1]^, 10, NumWorkers, Success);
  IF NOT Success
    THEN WriteString("Problem with number of workers - sorry try again");
         WriteLn;
         HALT
    ELSE WriteString("Number of workers: ");
         WriteInt(NumWorkers, 4);
         WriteLn
  END;

  (* start a number of workers *)
  FOR Index := 1 TO NumWorkers DO
    eval("crossp")
  END;

  (* add all rows to tuple space *)
  FOR I := 1 TO 3 DO
    out("r", I, M1[I,1], M1[I,2], M1[I,3])
  END;

  (* add all columns to tuple space *)
  FOR I := 1 TO 3 DO
    out("c", I, M2[1,I], M2[2,I], M2[3,I])
  END;

  (* add work seed to tuple space *)
  out("next", 0);

  (* get the answers back - in any order *)
  FOR Index := 1 TO 9 DO
    in(?I, ?J, ?Value);
    M3[I,J] := Value
  END;

  (* print results *)
  WriteString("First matrix:"); WriteLn;
  PrintMatrix(M1); WriteLn;
  WriteString("Second matrix:"); WriteLn;
  PrintMatrix(M2); WriteLn;
  WriteString("Cross Product:"); WriteLn;
  PrintMatrix(M3); WriteLn

END mastcrossp.

spec Global;
/*--------*/
/*  Program specification:  Cross product of two matrices
    Process:                mastcrossp
    Specification level:    global
    Description:            Spawn workers, add rows and columns of matrix to
                            tuple space, add work seed, and await results.
*/

  var
    I, J, Value : int;

  process mastcrossp
  /* start a number of worker processes */
     = choice (  addrowcolumndata
               | eval('crossp').mastcrossp);

  process addrowcolumndata
  /* place the rows of the matrix in tuplespace */
     = random(out('r',1,int,int,int).
              out('r',2,int,int,int).
              out('r',3,int,int,int)).
       /* place the columns of the matrix in tuplespace */
       random(out('c',1,int,int,int).
              out('c',2,int,int,int).
              out('c',3,int,int,int)).
       /* place a worker seed in tuplespace */
       out('next', 0).
       getresults;
```

```
  process getresults
  /* now get the results out of tuplespace */
     = in(?I, ?J, ?Value).
       in(?I, ?J, ?Value).
       in(?I, ?J, ?Value).
       in(?I, ?J, ?Value).
       in(?I, ?J, ?Value).
       in(?I, ?J, ?Value).
       in(?I, ?J, ?Value).
       in(?I, ?J, ?Value).
       in(?I, ?J, ?Value).
       /* now you're done */
       NIL

endspec


MODULE crossp;
(*----------*)
(*  Modula-2 Linda program:  Cross product of two matrices
    Process:                 worker

    Implementation of the cross product of two matrices.
*)

FROM EasyInOut     IMPORT WriteString, WriteInt, WriteLn;

TYPE
  VECTOR     = ARRAY[1..3] OF INTEGER;
VAR
  Seed, I, J : INTEGER;
  Row, Col   : VECTOR;

BEGIN
  LOOP
    (* get element number to compute *)
    in('next', ?Seed);

    (* set up next piece of work *)
    out('next', Seed + 1);

    IF Seed >= 9
      THEN (* no more work *)
           EXIT
    END;

    I := (Seed DIV 3) + 1;
    J := (Seed MOD 3) + 1;

    (* get the appropriate row *)
    read('r', I, ?Row[1], ?Row[2], ?Row[3]);

    (* get the appropriate column *)
    read('c', J, ?Col[1], ?Col[2], ?Col[3]);

    (* compute the result *)
    out(I, J, Row[1]*Col[1] + Row[2] * Col[2] + Row[3] * Col[3]);

  END (* LOOP *)
END crossp.
```

```
spec Global;
/*---------*/
/*  Program specification:  Cross product of two matrices
    Process:                crossp
    Specification level:    global
    Description:            Get a work seed, and return its successor. If seed
                            poisoned, terminate. Otherwise, get relevant row and
                            column, and return result.
*/

  var
    Seed,
    Row[1], Row[2], Row[3],
    Col[1], Col[2], Col[3] : int;

  process crossp
     = /* get a work seed */
       in('next',?Seed).
       /* replace it with the next work seed */
       out('next',Seed+1).
       if (Seed >= 9
           then NIL
           else /* get the respective row */
                read('r',(Seed DIV 3) + 1,?Row[1],?Row[2],?Row[3]).
                /* and column */
                read('c',(Seed MOD 3) + 1,?Col[1],?Col[2],?Col[3]).
                /* return the result to tuplespace */
                out((Seed DIV 3) + 1,
                    (Seed MOD 3) + 1,
                    (Row[1]*Col[1] + Row[2]*Col[2] + Row[3]*Col[3])).
                crossp
          )

endspec
```

E.4 Prime Numbers

The following Linda program determines the number of prime numbers that occur within a particular range. The solution centres on the fact that, if k is prime, the primality of all numbers from k+1 to k^2 can be determined ([Car90b] page 86). Individual workers operate on subranges of the total range.

The master process (`mastprime`) spawns a user-specified number of workers, seeds the first task (a subrange specification), and then awaits the results. For all prime numbers returned by workers, those that are required to determine the primality of other numbers in future ranges, are added to tuple space.

Worker processes (`prime`) scavenge for a task seed, and replace it with its successor or the poison seed. If the task seed is poisoned, the process terminates. Otherwise, for each odd number in the subrange, it retrieves all previously determined prime numbers that are required to determine the primality of the current subrange. All new prime numbers are added to tuple space (in a batch).

The `mastprime` and `prime` specifications are characterised by vigorous use of recursive sub-processes and a nested **`choice`**-statement.

```
MODULE mastprime;
(*-------------*)
(*  Modula-2 Linda program:  Prime numbers
    Process:                 master

    Determination of the number of prime numbers in some range.
*)

FROM EasyInOut      IMPORT WriteString, WriteLn, WriteInt;
FROM ConNum         IMPORT StrToUnsigned_16;
FROM EntryExit      IMPORT argv, argc;

CONST
  LIMIT = 200; (* determine primes up to 200 *)
  GRAIN = 6;   (* in worker portions of 6 *)
VAR
  Primes, Primes2 : ARRAY [0..40] OF INTEGER;
  NewPrimes : ARRAY [0..GRAIN-1] OF INTEGER;
  I, NumWorkers, FirstNum, Num, NumPrimes, NP2 : INTEGER;
  Success, EndOfTable : BOOLEAN;

BEGIN
  StrToUnsigned_16(argv^[1]^, 10, NumWorkers, Success);
  IF NOT Success
    THEN WriteString("Problem with number of workers - sorry try again");
         WriteLn;
         HALT
    ELSE WriteString("Number of workers: ");
         WriteInt(NumWorkers, 4);
         WriteLn
  END;

  (* initialise data structures with data for the first five prime numbers *)
  Primes[0] := 2; Primes2[0] := 4;
  Primes[1] := 3; Primes2[1] := 9;
  Primes[2] := 5; Primes2[2] := 25;
  Primes[3] := 7; Primes2[3] := 49;
  Primes[4] := 11; Primes2[4] := 121;
  NumPrimes := 5;

  (* clear out remaing structure *)
  FOR I := 5 TO 40 DO
    Primes[I] := 0; Primes2[I] := 0
  END;
```

```
  (* start a number of worker processes *)
  FOR I := 1 TO NumWorkers DO
    eval('prime')
  END;

  (* start searching for more prime numbers at the last
     prime number + 2 - the last prime must be odd, skip the next
     even, and start at the next odd *)
  FirstNum := Primes[NumPrimes-1] + 2;

  (* initiate the first task *)
  out('next task', FirstNum);

  EndOfTable := FALSE;
  FOR Num := FirstNum TO LIMIT BY GRAIN DO
    (* get the first batch of prime numbers from any worker *)
    in('results', Num, ?NewPrimes[0], ?NewPrimes[1], ?NewPrimes[2],
                       ?NewPrimes[3], ?NewPrimes[4], ?NewPrimes[5]);

    FOR I := 0 TO GRAIN-1 DO
      IF NewPrimes[I] # 0
        THEN Primes[NumPrimes] := NewPrimes[I];
             WriteInt(NewPrimes[I], 4);
             IF NOT EndOfTable
               THEN NP2 := NewPrimes[I] * NewPrimes[I];
                    (* check whether the new prime number will be
                       required in the future *)
                    IF NP2 > LIMIT
                      THEN EndOfTable := TRUE;
                           NP2 := -1
                    END;
                    out('primes', NumPrimes + 1, NewPrimes[I], NP2);
             END;
             NumPrimes := NumPrimes + 1
      END
    END
  END;

  WriteString("Number of primes: ");
  WriteInt(NumPrimes, 4);
  WriteLn
END mastprime.

spec Global;
/*--------*/
/*  Program specification:  Prime numbers
    Process:                mastprime
    Specification level:    global
    Description:            Spawn workers, initiate the first task, and then
                            await results. Add certain of the results to tuple
                            space.
*/

  var
    NewPrimes[0], NewPrimes[1], NewPrimes[2],
    NewPrimes[3], NewPrimes[4], NewPrimes[5] : int;

  process mastprime
     = choice (   firsttask
                | /* start worker process */
                  eval('prime').mastprime);

  process firsttask
     = /* initiate the first task */
       out('next task', 13).
       dealwithsubranges;
```

```
  process dealwithsubranges
     = /* deal with the subranges in which primes must be determined */
       choice (  /* full range completed */
                 NIL
               | /* get the next set of results from a subrange */
                 in('results', int, ?NewPrimes[0], ?NewPrimes[1],
                                    ?NewPrimes[2], ?NewPrimes[3],
                                    ?NewPrimes[4], ?NewPrimes[5]).
                 considerprimes);

  process considerprimes
     = /* deal with all primes generated */
       choice (  /* all primes considered, deal with next subrange */
                 dealwithsubranges
               | /* must the prime be placed in tuple space ? */
                 choice (  /* yes, it will be needed */
                           out('primes', int, int, int).
                           considerprimes
                         | /* not needed in the future */
                           considerprimes))

endspec
```

```
MODULE prime;
(*---------*)
(*  Modula-2 Linda program:  Prime numbers
    Process:                 worker

    Determination of the number of prime numbers in some range.
*)

FROM EasyInOut     IMPORT WriteString, WriteLn, WriteInt;

CONST
  LIMIT = 200; (* determine primes up to 20 *)
  GRAIN = 6;   (* in worker portions of 6 *)
VAR
  Primes, Primes2 : ARRAY [0..40] OF INTEGER;
  MyPrimes : ARRAY [0..GRAIN-1] OF INTEGER;
  I, Limit, Start, Count, FirstNum, Num, NumPrimes, NP2 : INTEGER;
  EndOfTable, OK : BOOLEAN;

BEGIN
  (* initialise data structures with data for the first five prime numbers *)
  Primes[0] := 2; Primes2[0] := 4;
  Primes[1] := 3; Primes2[1] := 9;
  Primes[2] := 5; Primes2[2] := 25;
  Primes[3] := 7; Primes2[3] := 49;
  Primes[4] := 11; Primes2[4] := 121;
  NumPrimes := 5;

  (* clear out remaining structure *)
  FOR I := 5 TO 40 DO
    Primes[I] := 0; Primes2[I] := 0
  END;
  FOR I := 0 TO GRAIN - 1 DO
    MyPrimes[I] := 0
  END;

  EndOfTable := FALSE;

  LOOP
    in('next task', ?Num);
    IF Num = -1
      THEN (* return poison seed *)
           out('next task', Num);
           EXIT
      ELSE Limit := Num + GRAIN;
           IF Limit > LIMIT
             THEN (* replace with poison seed *)
                  out('next task', -1);
                  Limit := LIMIT
```

```
            ELSE (* replace with successor seed *)
                 out('next task', Limit)
          END;
          Start := Num;
          Count := 0;
          FOR I := 0 TO GRAIN - 1 DO MyPrimes[I] := 0 END;
          WHILE Num < Limit DO
            WHILE ((NOT EndOfTable) AND (Num > Primes2[NumPrimes-1])) DO
              read('primes', NumPrimes + 1, ?Primes[NumPrimes],
                                            ?Primes2[NumPrimes]);
              IF Primes2[NumPrimes] < 0
                THEN EndOfTable := TRUE
                ELSE NumPrimes := NumPrimes + 1
              END
            END;
            OK := TRUE;
            (* search table of primes starting at the second
               position - the number is odd and will thus never
               be divisible by 2 - the first prime *)
            I := 1;
            LOOP
              IF I >= NumPrimes
                THEN EXIT
                ELSE IF Num MOD Primes[I] = 0
                       THEN OK := FALSE;
                            EXIT
                     END;
                     IF Num < Primes2[I]
                       THEN EXIT
                     END
              END;
              I := I + 1
            END;
            IF OK
              THEN MyPrimes[Count] := Num;
                   Count := Count + 1
            END;
            (* check if next number in section is prime; ignore
               even numbers - num always initially odd *)
            Num := Num + 2
          END;
          out('results', Start, MyPrimes[0], MyPrimes[1], MyPrimes[2],
                                MyPrimes[3], MyPrimes[4], MyPrimes[5]);
          Count := 0
    END
  END
END prime.

spec Global;
/*--------*/
/*  Program specification:  Prime numbers
    Process:                prime
    Specification level:    global
    Description:            Repeatedly: get a task seed
                                        if no more work, terminate
                                        otherwise, get data required to
                                        determine primality of numbers in subrange
                                        return new prime numbers
*/

  var
    Num, Primes[NumPrimes], Primes2[NumPrimes] : int;

  process prime
     = /* get next task descriptor */
       in('next task', ?Num).
       if (Num = -1
           then /* no more work, return descriptor */
                out('next task', Num).
                NIL
```

```
            else if ((Num + 6) > 200
                     then /* last task,
                             insert "no more tasks" descriptor */
                          out('next task', -1).
                          prime1a
                     else /* there are more tasks,
                             insert next task descriptor */
                          out('next task', Num + 6).
                          prime1a
                    )
           );

  process prime1a
     = /* deal with all numbers in the subrange allocated for this task */
       choice (  /* get data required for each number */
                 prime1b
               | /* check for primality of each number, and
                    out all primes found in the subrange */
                 out('results', Num, int, int, int, int, int, int).
                 /* go get next task */
                 prime);

  process prime1b
     = /* read all previously determined primes that are required to
          determine whether a number in subrange is prime */
       choice (  /* more needed */
                 read('primes', int, ?Primes[NumPrimes], ?Primes2[NumPrimes]).
                 prime1b
               | /* got all that are requied */
                 prime1a)

endspec
```

E.5 N-Queens Problem

The N-Queens problem determines, for a given chessboard size (say, nxn), the ways in which n queens can be arranged such that no queen is threatened by any of the other n-1 queens. The solution entails a master and a number of workers. For a possible board arrangement, the master solves the problem partially (dependent on some "depth" factor, `BossLimit`), and then leaves the completion of the solution to a worker. An n-element vector, `Board`, is used to store the row position in which a queen is placed for each n columns.

The master process (`mastqueen`) spawns a worker for each partially-evaluated solution, and then awaits the solutions. Worker processes (`queen`) extract partial solutions from tuple space, and return any full solutions that they might develop.

The specifications are characterised by their brevity.

```
MODULE mastqueen;
(*-------------*)
(*  Modula-2 Linda program:  N-Queens problem
    Process:                 master

    Determine the ways in which n queens can be arranged on a nxn chessboard
    such that no queen is threatened by any of the other n-1 queens.
*)

FROM EasyInOut     IMPORT WriteString, WriteLn, WriteInt;

CONST
  BoardSize     = 5;
  BossLimit     = 1;
  KnownSolns    = 10;
TYPE
  BOARD         = ARRAY [0..BoardSize-1] OF INTEGER;
VAR
  Board         : BOARD;
  NumSolutions  : INTEGER;

  PROCEDURE IsSafe (Q, Col, Distance : INTEGER) : BOOLEAN;
  VAR
    BC : INTEGER;
  BEGIN
    IF Col < 0
      THEN RETURN TRUE
    END;
    BC := Board[Col];
    IF ((Q = BC) OR
        (Q-Distance = BC) OR
        (Q+Distance = Board[Col]))
      THEN RETURN FALSE
      ELSE RETURN (IsSafe(Q, Col-1,Distance+1))
    END
  END IsSafe;

  PROCEDURE PlaceQueens (NumPlaced : INTEGER);
  (* Attempt to place the queens on the board *)
  VAR
    Q, Pos : INTEGER;
  BEGIN
    FOR Q := 0 TO BoardSize-1 DO
      IF IsSafe(Q, NumPlaced-1, 1)
        THEN Pos := NumPlaced;
             Board[Pos] := Q;
             Pos := Pos + 1;
             (* the master determines valid positions for the first
                BossLimit columns, and then leaves the remainder of the
                solution to the worker *)
             IF Pos < BossLimit
               THEN PlaceQueens(Pos)
```

```
                ELSE (* out a task *)
                     out("partials", Pos, Board[0], Board[1], Board[2],
                                          Board[3], Board[4]);
                     (* start a worker process *)
                     eval("queen");
                END
        END
      END
    END PlaceQueens;

    PROCEDURE PrintBoard (Board : BOARD);
    VAR
      I, J : INTEGER;
    BEGIN
      FOR I := 0 TO BoardSize-1 DO
        WriteInt(Board[I], 3)
      END;
      WriteLn;
      FOR I := 0 TO BoardSize-1 DO
        FOR J := 1 TO Board[I] DO
          WriteString("- ")
        END;
        WriteString("Q ");
        FOR J := 1 TO BoardSize - Board[I] - 1 DO
          WriteString("- ")
        END;
        WriteLn
      END;
      WriteLn
    END PrintBoard;

BEGIN
    PlaceQueens(0);

    (* now get all the solutions *)
    NumSolutions := 0;

    WHILE NumSolutions # KnownSolns DO
      InitialiseTuple(Tuple);
      in("completed", ?Board[0], ?Board[1], ?Board[2],
                      ?Board[3], ?Board[4]);
      PrintBoard(Board);
      NumSolutions := NumSolutions + 1
    END

END mastqueen.

spec Global;
/*--------*/
/*  Program specification:  N-Queens
    Process:                mastqueen
    Specification level:    global
    Description:            Spawn workers for each partial solution developed,
                            and then await the results.
*/

  var
    Board[0], Board[1], Board[2], Board[3], Board[4] : int;

  process mastqueen
     = choice (  getresults
               | out('partials', int, int, int, int, int, int).
                 eval('queen').
                 mastqueen);

  process getresults
     = choice (  NIL
               | in('completed', ?Board[0], ?Board[1], ?Board[2],
                                 ?Board[3], ?Board[4]).
                 getresults)

endspec
```

```
MODULE queen;
(*---------*)
(*  Modula-2 Linda program:  N-Queens
    Process:                 worker

    Determine the ways in which n queens can be arranged on a nxn chessboard
    such that no queen is threatened by any of the other n-1 queens.
*)

CONST
  BoardSize = 5;
VAR
  NumPlaced : INTEGER;
  Board : ARRAY [0..BoardSize-1] OF INTEGER;


  PROCEDURE IsSafe (Q, Col, Distance : INTEGER) : BOOLEAN;
  VAR
    BC : INTEGER;
  BEGIN
    IF Col < 0
      THEN RETURN TRUE
    END;
    BC := Board[Col];
    IF ((Q = BC) OR
        (Q-Distance = BC) OR
        (Q+Distance = Board[Col]))
      THEN RETURN FALSE
      ELSE RETURN (IsSafe(Q, Col-1,Distance+1))
    END
  END IsSafe;

  PROCEDURE PlaceQueensW (NumPlaced : INTEGER);
  VAR
    Q, Pos : INTEGER;
  BEGIN
    FOR Q := 0 TO BoardSize-1 DO
      IF IsSafe(Q, NumPlaced-1, 1)
        THEN Pos := NumPlaced;
             Board[Pos] := Q;
             Pos := Pos + 1;
             IF Pos < BoardSize
               THEN PlaceQueensW(Pos)
               ELSE (* out a solution *)
                    out("completed", Board[0], Board[1], Board[2],
                                     Board[3], Board[4])
             END
      END
    END
  END PlaceQueensW;

BEGIN
  (* get a task *)
  in("partials", ?NumPlaced, ?Board[0], ?Board[1], ?Board[2],
                             ?Board[3], ?Board[4]);
  PlaceQueensW(NumPlaced);
END queen.
```

```
spec Global;
/*---------*/
/*  Program specification:  N-Queens
    Process:                queen
    Specification level:    global
    Description:            Extract a partial solution from tuple space, and
                            return completed solutions, if any.
*/

  var
    NumPlaced,
    Board[0], Board[1], Board[2], Board[3], Board[4] : int;

  process queen
     = in('partials', ?NumPlaced, ?Board[0], ?Board[1], ?Board[2],
                                  ?Board[3], ?Board[4]).
       addresults;

  process addresults
     = choice (  NIL
               | out('completed', int, int, int, int, int).
                 addresults)

endspec
```

E.6 Arithmetic Expressions

The following Linda program implements a simple arithmetic expression handler. Details of the program can be found in section 5.5.

The master process (`mastarith`) spawns a user-specified number of workers, adds the tasks to tuple space, and retrieves the results. Once all results have been retrieved, a poison task is added to tuple space. Worker processes (`arith`) repeatedly scavenge for tasks. If the task descriptor is poisoned, the process terminates. Otherwise, task data is retrieved, and a result is computed, after which it is returned to tuple space.

The `arith` specifications are characterised by their multiple sub-specifications, use of the wild tuple indicator, wild Linda primitive statement, nested **`choice`**-statements, and alternation constructs in which alternates are succeeded by common specifications.

```
MODULE mastarith;
(*-------------*)
(*  Modula-2 Linda program:  Simple arithmetic expression handler
    Process:                 master

    Implementation of a simple arithmetic expression handler.
*)

FROM LindaHQ        IMPORT STRING;
FROM EasyInOut      IMPORT FileInput, ReadInt, ReadWord, ReadLn,
                           WriteString, WriteLn, WriteInt;
FROM Strings        IMPORT Compare;
FROM ConNum         IMPORT StrToUnsigned_16;
FROM EntryExit      IMPORT argv, argc;

VAR
  NumWorkers, Op1, Op2, Index, Tag, Answer : INTEGER;
  Action : STRING;
  Success : BOOLEAN;

BEGIN
  StrToUnsigned_16(argv^[1]^, 10, NumWorkers, Success);
  IF NOT Success
    THEN WriteString("Problem with number of workers - sorry try again");
         WriteLn;
         HALT
    ELSE WriteString("Number of workers: ");
         WriteInt(NumWorkers, 4);
         WriteLn
  END;

  (* start a number of worker processes *)
  FOR Index := 1 TO NumWorkers DO
    eval('arith')
  END;

  FileInput;
  Tag := 0;
  LOOP
    ReadWord(Action);
    IF Compare(Action, "end") = 0
      THEN EXIT
      ELSE Tag := Tag + 1;
           ReadInt(Op1);
           ReadInt(Op2);
           WriteInt(Tag, 3); WriteString(": ");
           WriteInt(Op1, 3); WriteString(Action); WriteInt(Op2, 3); WriteLn;
           out(Tag, Action);
           out(Tag, Op1, Op2);
    END
  END;
```

```
  (* now get back the answers *)
  FOR Index := 1 TO Tag DO
    in(Index, ?Answer);
    WriteString("Answer for "); WriteInt(Index, 3); WriteString("is: ");
    WriteInt(Answer, 4); WriteLn
  END;

  (* add poison task descriptor *)
  out(Tag, 'end')
END mastarith.

spec Global;
/*--------*/
/*  Program specification:  Simple arithmetic expression handler
    Process:                mastarith
    Specification level:    global
    Description:            Spawn a number of workers, add task to tuple space, and
                            retrieve results.  Finally, add a poison task descriptor
                            is added to tuple space.
*/

  var
    Answer : int;

  process mastarith
     = choice (  addtasks
               | eval('arith').mastarith);

  process addtasks
     = choice (  getresults
               | out(int, '+').addoperands
               | out(int, '-').addoperands
               | out(int, '*').addoperands
               | out(int, '/').addoperands);

  process addoperands
     = /* add operands */
       out(int, int, int).
       /* go get next expression */
       addtasks;

  process getresults
     = choice (  /* no more results,
                    so add end of tasks descriptor */
                 out(int, 'end').
                 /* you're done */
                 NIL
               | /* get answer */
                 in(int, ?Answer).
                 /* go next answer */
                 getresults)

endspec


MODULE arith;
(*---------*)
(*  Modula-2 Linda program:  Simple arithmetic expression handler
    Process:                 worker

    Implementation of a simple arithmetic expression handler.
*)

FROM LindaHQ       IMPORT STRING;
FROM Strings       IMPORT Compare;

VAR
  Op1, Op2, Index, Tag, Answer : INTEGER;
  Action : STRING;

BEGIN
  LOOP
    (* get task *)
    in(?Tag, ?Action);
```

```
    IF Compare(Action, "end") = 0
      THEN (* replace poison task *)
           out(Tag, Action);
           EXIT
      ELSE (* get operands *)
           in(Tag, ?Op1, ?Op2);
           (* add answer to tuple space *)
           IF Compare(Action, "+") = 0
             THEN out(Tag, Op1+Op2)
             ELSE IF Compare(Action, "-") = 0
                    THEN out(Tag, Op1-Op2)
                    ELSE IF Compare(Action, "/") = 0
                           THEN out(Tag, Op1 DIV Op2)
                           ELSE IF Compare(Action, "*") = 0
                                  THEN out(Tag, Op1*Op2)
                                  ELSE EXIT
                                END
                         END
                  END
           END
    END
  END
END arith.

spec Global;
/*--------*/
/*  Program specification:  Simple arithmetic expression handler
    Process:                arith
    Specification level:    global
    Description:            Repeatedly: Extract a task descriptor from
                                        tuple space.
                                        If descriptor poisoned, terminate
                                        otherwise, get task data, compute result,
                                        and return result to tuple space.
*/

  var
    Tag, Op1, Op2 : int;
    Action : str;

  process arith
     = in(?Tag, ?Action).
       if (Action = 'end'
           then out(Tag, Action).
                NIL
           else in(Tag, ?Op1, ?Op2).
                if (Action = '+'
                    then out(Tag, Op1 + Op2).
                         arith
                    else if (Action = '-'
                             then out(Tag, Op1 - Op2).
                                  arith
                             else if (Action = '*'
                                      then out(Tag, Op1 * Op2).
                                           arith
                                      else if (Action = '/'
                                               then out(Tag, Op1 DIV Op2).
                                                    arith
                                               else NIL)))))

endspec
```

```
spec AtLeastOneAddition;
/*--------------------*/
/*  Program specification:  Simple arithmetic expression handler
    Process:                arith
    Specification level:    specific
    Description:            Check that the action of the worker includes,
                            at least, one addition operation.
*/

  var
    Action : str;

  process arith
     = in(?int, ?Action).
       in(*).
       out(*).
       if (Action = '+'
           then aritha
           else arith);

  process aritha
     = choice (  NIL
               | *.aritha)

endspec


spec OneAnswerofminus11;
/*--------------------*/
/*  Program specification:  Simple arithmetic expression handler
    Process:                arith
    Specification level:    specific
    Description:            Check that the action of the worker includes the
                            addition to tuple space of, at least, one answer
                            of -11.
*/

 process arith
    = choice (  *.arith
              | out(int, -11).aritha);

 process aritha
    = choice (  NIL
              | *.aritha)

endspec


spec NoTwoConsecutiveOuts;
/*----------------------*/
/*  Program specification:  Simple arithmetic expression handler
    Process:                arith
    Specification level:    specific
    Description:            Check that the action of the worker does not include
                            two consecutive out operations.
*/

process arith
   = choice (  NIL
             | *.arith
             | out(*).choice (  NIL
                              | in(*).arith
                              | read(*).arith
                              | inp(*).arith
                              | readp(*).arith
                              | eval(*).arith))

endspec
```

Appendix F

Modula-2 Linda System

The Modula-2 Linda system is an experimental system in which the ideas that were developed in this thesis could be tested.

F.1 Distinguishing Characteristic

Modula-2 Linda is a streamlined, functional Linda dialect that supports all the basic Linda primitives: **out, in, read, inp, readp,** and **eval,** for a limited set of tuple element types, namely, integers and strings.

F.2 Details of the Implementation

F.2.1 System Overview

The system is implemented as a series of modules that provide tuple manipulation and tuple space interaction facilities, and a tuple space server. Tuples are implemented (in concrete Linda) as abstract data types (Modula-2 does not provide for variable numbers of parameters), and Linda primitives as procedures and functions. A standard fork-mechanism is used to spawn processes, and a file-based transport layer is used to implement communication between processes and the tuple space server in particular, and between system components in general. A system diagram can be found in Figure F.1.

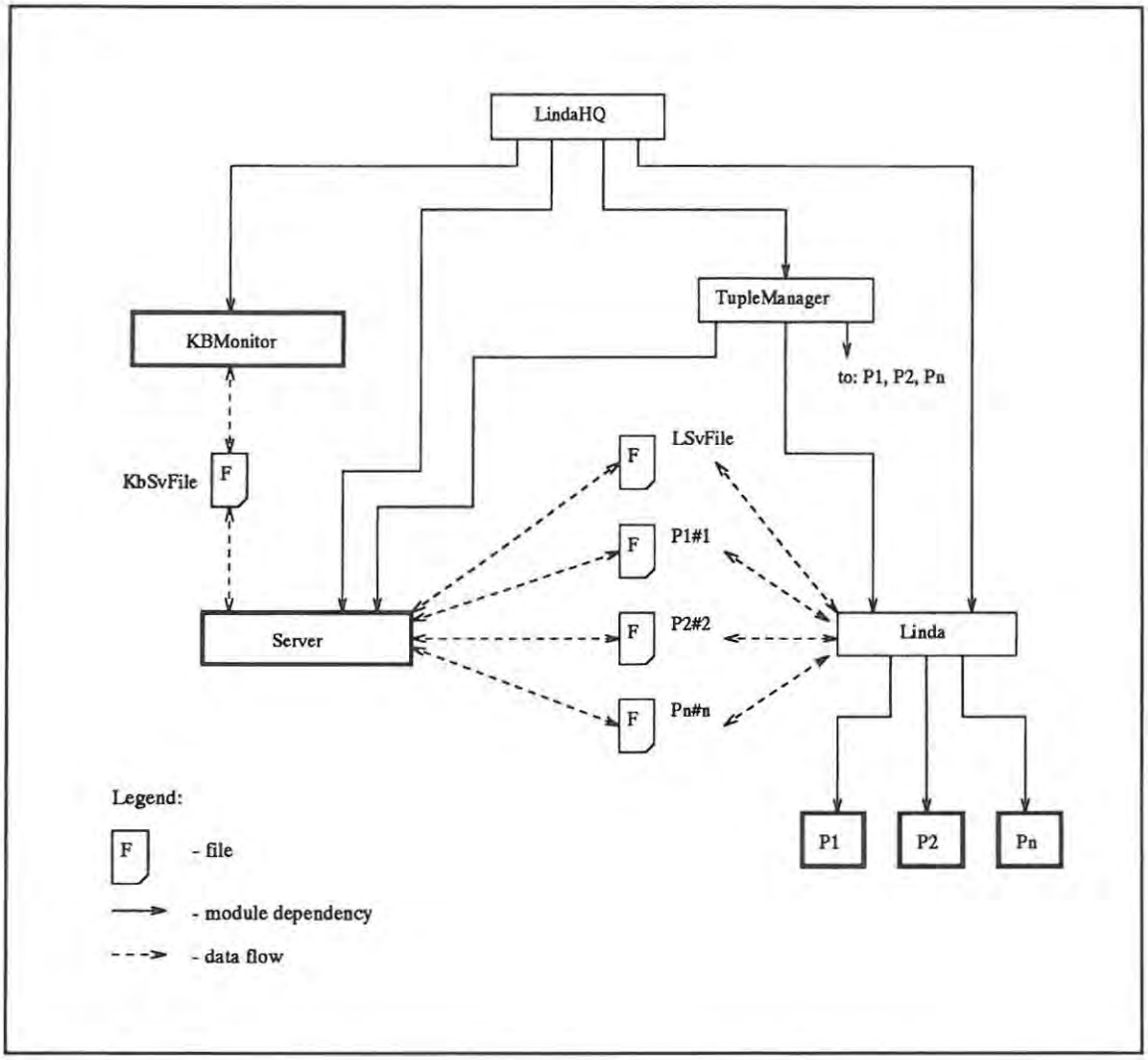

Figure F.1 Modula-2 Linda System

Index to Figure F.1:

1.	LindaHQ	-	the master module that contains base definitions and global communication file names.
2.	TupleManager	-	tuple manipulation facilities (construction, comparison and communication).
3.	Linda	-	Tuple space connect/disconnect facilities, and Linda primitives.
4.	KBMonitor	-	keyboard monitor.
5.	Server	-	tuple space server.
6.	P1, P2, Pn	-	Linda processes.
7.	KBSvFile	-	keyboard-server communication file.
8.	LSvFile	-	Linda-server communication file.

9. P1#1,
   P2#2,
   Pn#n - private (process) Linda-server communication files.

The Modula-2 Cross System (MCS) v.4.4 of Modula-2 implemented on a Sun 4 workstation running SunOS 4.1.1 is used to implement the system.

F.2.2 General Action of the System

A Modula-2 Linda program is a collection of standalone Modula-2 programs, each of which implements a Linda process (multiple copies of the same program may be executed simultaneously to create the effect of multiple, identical processes).

Prior to Linda program execution, the Modula-2 Linda run-time system is started by launching the Linda server (it also launches the keyboard monitor). Thereafter, processes are started either by using the **eval**-primitive or by manual instantiation at the operating system level (usually a main or master process is manually instantiated after which it spawns further processes using the **eval**-primitive).

After instantiation, a typical sequence of process actions is as follows:

connect to tuple space;
interact with tuple space;
disconnect from tuple space.

(Abstract Modula-2 Linda would not include connect/disconnect activities.)

Requests to connect to or disconnect from tuple space are channelled from the Linda module to the server via a predefined Linda-server file (LSvFile). On connection, a private communication file is established between the process and the server through which general tuple space requests and replies are routed. On disconnection, the private file is erased, and communication is severed.

The Linda server maintains:

1. a list of process descriptors that represent processes that are currently connected to the server, and
2. a list of free tuples.

The server polls the private files for interaction requests, as well as the Linda-server file for connect/disconnect requests. Processes fight for temporary control of the Linda-server file but make leisurely use of their private file. The list of free tuples is modified as tuples are added to and

removed from tuple space.

Two implementation strategies were tested for unsuccessful **in** and **read** requests:

1. processes that submit unsuccessful **in** and **read** requests are blocked, pending the arrival of appropriate tuples. The server maintains lists of blocked **in** and **read** requests. On arrival of an appropriate tuple, all matching **read** requests are serviced, and one matching **in** request, selected non-deterministically, is serviced.
2. processes that submit unsuccessful **in** and **read** requests are required to re-submit the request together with all other process requests (in practice, the Linda module controls the re-submission - the process is not aware of any internal policy or haggling that occurs between the Linda module and the server).

Both are acceptable interpretations of the Linda paradigm.

The server also maintains a graphical display of the state of tuple space, and the activities of the server (current request, request servicibility, result of request). It is anticipated that the user, having instantiated a master process, will concentrate attention on this display to monitor system activity.

F.2.3 System Specifics

F.2.3.1 Establishing/Severing Process-Server Communication

Processes inform the server of their desire to communicate by calling on the connect service offered by the Linda module. This places an appropriate request in the Linda-server file that includes the name of the requesting process.

The server deals with requests to connect in the following manner:

1. a private server-process communication file is created (the name of the file is a combination of the process name and an internal descriptor),
2. a descriptor is added to the process descriptor list that includes all relevant process data (process name, communication file name, file handle, and process status),
3. and, finally, a "successful connect" reply is posted in the Linda-server file that also includes the name of the private communication file. The reply is collected by the Linda module. The private communication file is then used for all future communications (except the final disconnect request).

Processes sever communication with the server by calling on the disconnect service offered by the Linda module. The sequence of events that ensues is much the same as the events that ensue when

communication is initially established with the server. The Linda-server file conveys the request to the server that then deletes the private communication file, and informs the process, again via the Linda-server file, that communication has been severed.

On connection, the request includes the name of the process but on disconnection, the request includes the name of the private communication file, so that processes are identified correctly.

F.2.3.2 Executing Linda Primitives

Linda primitives are implemented in the Linda module as procedures and functions. All primitives take a single tuple parameter, and, in the case of **inp** and **readp**, return a boolean result. Tuples are implemented as abstract data types (ADT).

Prior to and just after the call to the appropriate Linda primitive procedure/function, the facilities of the TupleManager are used to initialise, construct, and finally clear the tuple structure, that is:

initialise the tuple ADT;
construct the tuple ADT;
call the appropriate Linda primitive procedure/function;
clear the tuple ADT (deallocate memory).

Tuples are constructed by successive addition of tuple elements to the tuple ADT.

Within the Linda module, further TupleManager facilities are used to write/read tuple information to/from the private communication file.

F.2.3.3 The eval-primitive

Section 3.2.1 provided a brief description of the **eval**-primitive. It takes a single string tuple element that names an executable Modula-2 code file. The server forks a new process that executes the code contained in the file. It does not add a process descriptor to the process descriptor list but awaits a formal request from the process just instantiated to connect to the server. No partially-evaluated active data tuple is maintained nor does any passive data tuple result from the execution of the process (other than those that are added to tuple space in the normal course of events).

Appendix G

Modula-2 Linda System with Debugger

The Modula-2 Linda system discussed in Appendix E is modified to include a behavioural model debugger.

G.1 Implementation Strategy

The Modula-2 Linda system code is modified to generate notice of the occurrence of events (process-server interaction based on Linda primitives). Such notice is communicated to a separately executing behavioural model debugger that checks the behaviour, and posts replies back to the server for its consideration.

G.2 Details of Implementation

G.2.1 System Overview

The debugger is implemented as a series of modules that provide facilities for the construction of internal models of expected behaviour (based on behavioural specifications), and mechanisms (a recognition engine) to compare actual and expected process behaviour. The debugger is connected to the Linda server via a single communication file through which process-server connect/disconnect requests, notice of tuple space interactions, and results of behavioural comparisons are channelled. A system diagram can be found in Figure G.1.

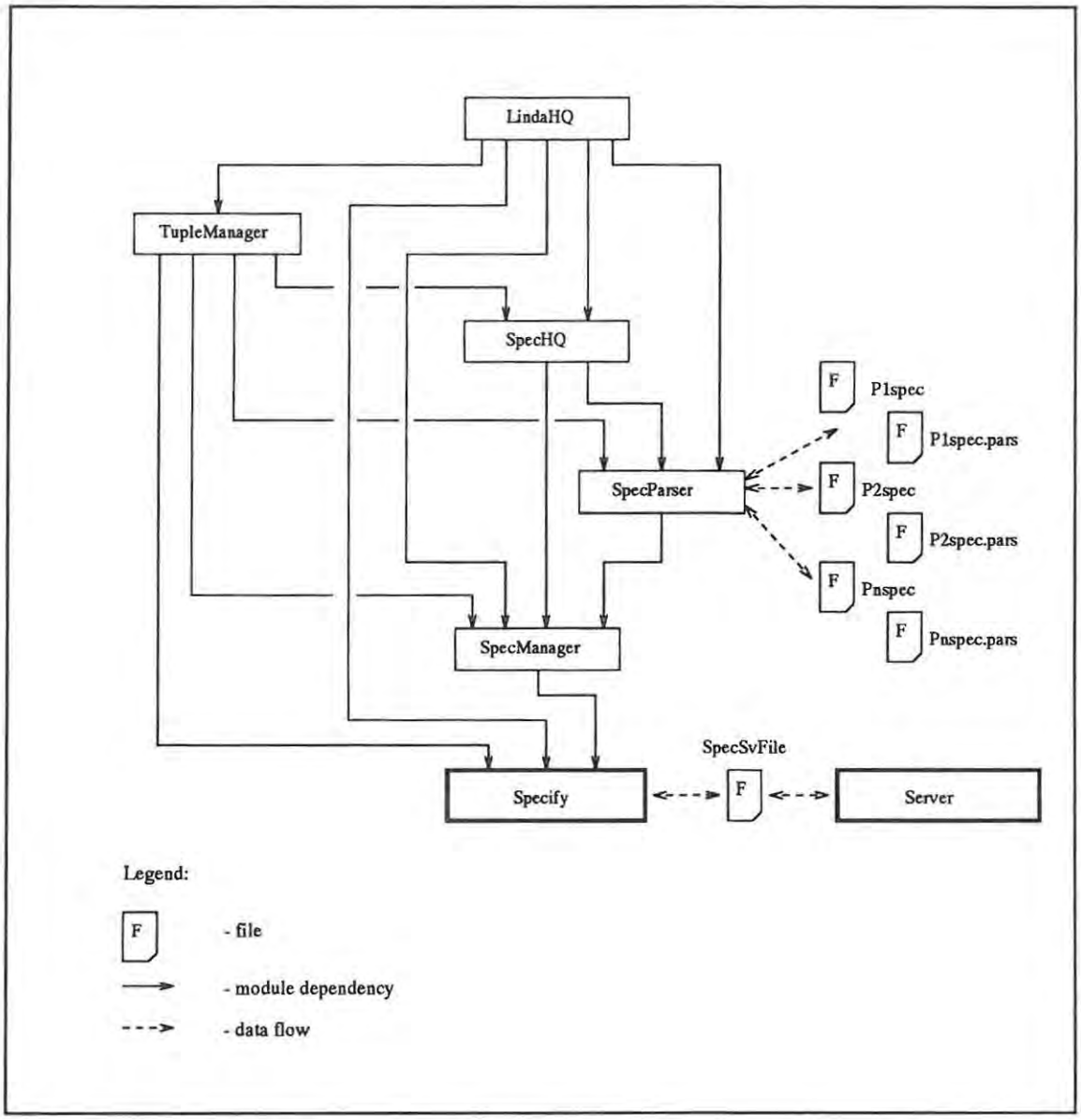

Figure G.1 Modula-2 Linda System with Debugger

Index to Figure G.1:

1.	LindaHQ	-	the master module that contains base definitions and global communication file names.
2.	TupleManager	-	tuple manipulation facilities (construction, comparison, communication).
3.	SpecHQ	-	the master specification module that contains the internal behavioural model data structures.
4.	SpecParser	-	specification language parser and internal model construction and facilities.

5.	SpecManager	-	behaviour recognition engine.
6.	Specify	-	process specification handler and server-debugger interface
7.	Server	-	tuple space server.
8.	SpecSvFile	-	specification handler-server communication file.
9.	P1spec, P2spec, Pnspec	-	process specification files.
10.	P1spec.pars, P2spec.pars, Pnspec.pars	-	specification parser list files.

The Modula-2 Cross System (MCS) v.4.4 of Modula-2 implemented on a Sun 4 workstation running SunOS 4.1.1 is used to implement the debugger.

G.2.2 General Action of the System

Each Linda process is supported by a behavioural specification. The specification is normally defined prior to process code development, and is stored in a file separate to that of the process.

As part of the Linda server initialisation code, the specification handler is launched using a standard fork-mechanism. A specification handler-server file (SpecSvFile) is also created for communication between the server and the specification handler.

After instantiation (either manually for the distinguished process or by **eval**), whenever a process attempts to connect to tuple space, the specification handler is immediately informed. The specification file that is associated with the process is read, parsed and an internal model[1] of the expected behaviour of the process is constructed. On completion, the server, and then the process, are informed of successful tuple space connection. (A variety of conditions may preclude tuple space connection, for example, lack of a specification file, parse errors, in which case connection is not permitted and the process is terminated.) At this stage, the recognition engine, based on the behavioural specifications supplied to the specification handler, is ready to check behaviour. Each time the process interacts with tuple space, the request and the requesting process name are referred to the specification handler for checking. It compares the request with that which is expected[2], and returns the result of the comparison to the server. If the behaviour matches, the tuple space server

[1] If the process has been specified at more than one level of abstraction, more than one model is constructed.

[2] If multiple models exist, multiple comparisons are made, each of which must succeed for a match to be declared.

continues to execute as normal. For mismatched behaviour, two approaches were tested:

1. the offending process is informed, and then terminated.
2. the tuple space server disregards the mismatch result, and continues to execute as normal.

Whenever a process disconnects from tuple space, the specification handler is again informed. It checks the corresponding internal models to determine whether no further activity was expected (it is in a final state), and replies to the server accordingly.

Notice of behavioural comparisons that result in a mismatch are also posted to the graphics display maintained by the tuple space server.

G.2.3 System Specifics

G.2.3.1 Specification Parser and Internal Model Constructor

The Linda program specification language is used to specify process behaviour (see chapter 5). Each specification is composed of one or more sub-specifications that specify the behaviour of the process at different levels of abstraction. The goal is to construct internal models, one for each sub-specification, to which is coupled an environment within which it exists.

A recursive descent, LL(1) parser is used to process the behavioural specifications.

For each sub-specification, the parser generates a labelled transition graph and a private environment (a collection of named memory locations) in which tuple information is maintained. Graph transitions are based on tuple space interactions. Since tuples may be based on run-time tuple information contained in the associated environment, the initial graph is incomplete. (See section G.2.3.3 for a further discussion on how and when the graph is complete.) Such incompleteness precludes conversion from a non-deterministic to deterministic graph.

G.2.3.2 Multiple, Identical Processes

It is frequently the case that the same Linda process, represented by a single Modula-2 program, is instantiated repeatedly. For example, given the Modula-2 program "crossp", multiple instances may

be invoked using the **eval**-primitive:

```
.
.
.
eval('crossp');
eval('crossp');
.
.
.
```

Since each 'crossp' process is associated with the same, single specification file, each instance of 'crossp' has the same expected behaviour. In the unlikely event that identical processes must exhibit differing expected behaviour (it may be necessary to monitor the particular behaviour of an instance of a process), different named files must be used that house identical code, for example:

```
.
.
.
eval('crossp1');
eval('crossp2');
.
.
.
```

Separate specifications are then required for 'crossp1' and 'crossp2'.

G.2.3.3 Model Control and Behaviour Comparison

Actual process behaviour is recognised by following multiple search paths in each graph that implements a sub-specification. A current state (actually a set of states) reflects the position in the multiple paths that has been reached by the recognition process.

Initially the current state is set to the start state in each graph. As successive behaviour is recognised, transition/s are taken to new state/s, until a final state is reached. Since the internal model is composed of a number of graphs that represent the many sub-specifications, the recognition process takes place a number of times, one for each graph.

Transitions from a particular state are only fully-defined when that state is reached. At that stage, relevant values are extracted from the environment that are used to complete the available transitions.

In Figure G.2, when state *B* is reached, values for *I* are extracted from the environment to complete *out(I+1)* and *out(I)*.

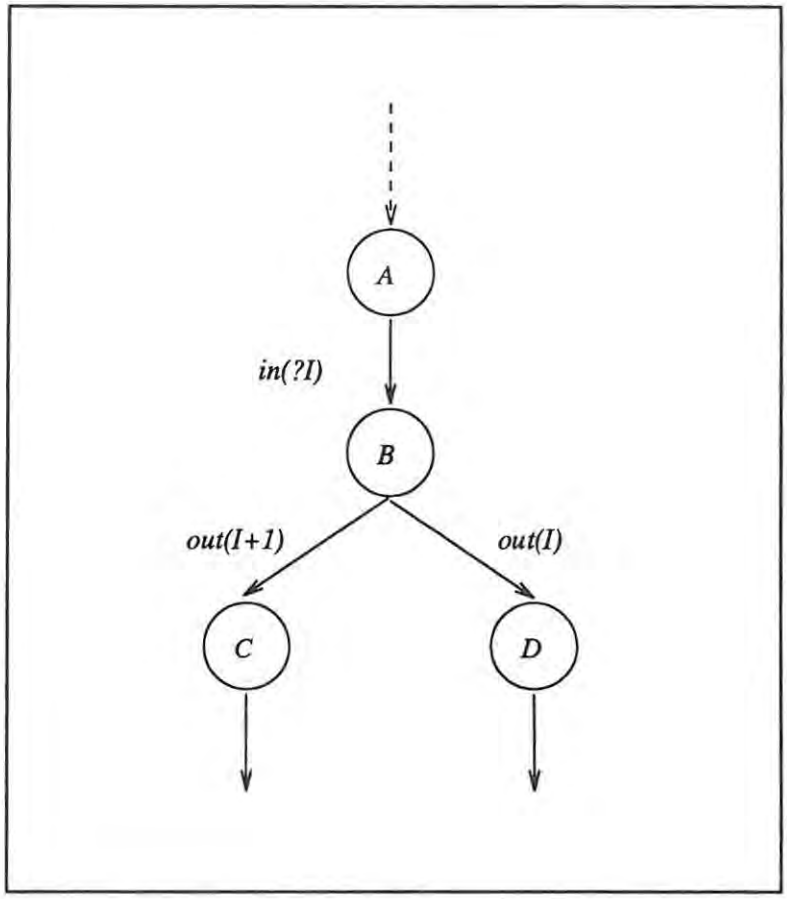

Figure G.2 Specification graph with partially-defined transitions

After a particular transition has been made, the environment is modified to reflect any changes brought about by the transition. The TupleManager module provides facilities for comparing actual behaviour (input symbols) with graph transitions.