NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

ISBN 0-315-53255-6

VERIFICATION OF CONCURRENT SYSTEM SPECIFICATIONS USING TEMPORAL LOGIC

by

Randall A. Harvey B.Eng., P.Eng.

a thesis submitted to the
School of Graduate Studies and Research
in partial fulfillment of the requirements
for the degree of
Master of Applied Science

Ottawa-Carleton Institute for Electrical Engineering

Department of Electrical Engineering
Faculty of Engineering
University of Ottawa

August 1989

Abstract

This thesis considers the problem of verifying the correctness of a set of Process Activity Language (PAL) expressions describing the behaviour of the processes of a special purpose concurrent computer system. A two step verification process is proposed and implemented to verify that the behaviour described by the process expressions will result in a system that will satisfy the needs of the user. First, the PAL expressions are transformed using a method based on Linear Time Temporal Logic into an equivalent global state transition graph. This state graph is regarded as a model of the system's behaviour. A user then specifies a set of correctness properties describing the required behaviour of the system. The correctness properties are expressed as Branching Time Temporal Logic formulae on the state variables of the system. Then, as the second step of the verification process, an efficient *model checking* algorithm is employed to check if the correctness properties are satisfied on the state graph model of the system. If the correctness properties are satisfied on the model then the PAL expressions from which the model was derived must also be correct.

Acknowledgements

I would first of all like to thank my advisor, Dr Moshe Krieger, for his support and guidance over the past three years. Thank for encouraging me to pursue this endeavour in the first place.

From Carleton University I would like to thank Dr Gerald Karam who was always available to help—and did. I would have asked many more questions if not for his gift of being able to deliver a clear understandable explanation of everything.

To my fellow graduate students at the University of Ottawa, thank you for your discussion, constructive criticism and friendship. For Robert Joannis, with whom I shared a common research interest, thank you for your insightful observations and comments. I could always count on Robert for two things: for his honest and frank appraisal of my work, and for a good game of squash—well, most of the time.

I must of course thank my employer, the Department of National Defence, who so generously allowed me to pursue my studies at their expense. I for one know they made a wise investment.

And finally, to my wife and best friend Alix who provided moral support during the more difficult times of my studies and put up with me coming to bed at odd hours of the night (morning), thanks!!

Symbols

LTL	–	Discrete Linear Time Temporal Logic
BTL	–	Discrete Branching Time Temporal Logic
CTL	–	Computational Tree Logic
CTL*	–	Extended Computational Tree Logic
$\mathcal{UB}$	–	Unified System of Branching Time Logic
DX	–	Pnueli's system of LTL
PAL	–	Process Activity Language
PTR	–	PAL Temporal Representation
COL	–	CAEDE Operational Langauge
PSA	–	Program State Assertion
SSA	–	System State Assertion
$\forall$	–	universal quantification (forall)
$\exists$	–	existential quantification (there exists)
$\models$	–	a universally valid formula
$\models\!\!\equiv$	–	a valid formula within a specific world
$\equiv$	–	logical equivalence
$\Rightarrow$	–	logical implication

The following symbols are given in two forms; the type set characters are used in the theory part of this thesis while the type-writer characters are used in the implementation part. If the type-writer form is not given then the symbol is not used in the implementation.

$\wedge$	and	-	logical and
$\vee$	or	-	logical or
$\neg$	~	-	logical negation
$\Box$		-	henceforth (LTL)
$\Diamond$	<>	-	eventually (LTL)
$\bigcirc$	*	-	next (LTL)
$\mathcal{U}$		-	until (LTL)
$\forall\Box$	ag	-	universal henceforth (BTL)
$\exists\Box$	eg	-	existential henceforth (BTL)
$\forall\Diamond$	af	-	universal eventually (BTL)
$\exists\Diamond$	ef	-	existential eventually (BTL)
$\forall\bigcirc$	ax	-	universal next-time (BTL)
$\exists\bigcirc$	ex	-	existential next-time (BTL)

PAL Constructs

$\mathcal{S}_i$	-	process starting event
$\mathcal{E}$	-	process expression delimiter
RDS	-	read system condition variable
STS	-	set system condition variable
STP	-	set process condition variable

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Background and Motivation

The fundamental goal of any design process is to translate a need into a product which satisfies this need. In the design of computer systems needs are usually first expressed as a specification of requirements and the product is referred to as an implementation of this specification. A major aspect of a system developer's task is to ensure that the implementation does indeed satisfy the stated requirements. As systems become more complex and our reliance on their proper operation increases, this task becomes ever more difficult.

To *verify* is to show via a formal proof method that a specification is satisfied by a given implementation. If both the specification and the implementation can be expressed as mathematical objects then we can apply formal methods to demonstrate a logical consistency between the two objects. If the statement of consistency can be constructed in such a way that it is independent of all allowable initial system conditions then we have provided a *proof of correctness.*

The overall behaviour of a system can be decomposed into two parts, *functional behaviour* and *temporal behaviour*[Kar87].

functional behaviour — refers specifically to those functions or activities that the

system is designed to perform that are not part of the control structure of the system.

temporal behaviour — refers to the control structure of the system; those system elements that decide the execution ordering of the system's functions. Included in a system's temporal behaviour are the process interactions (synchronizations) among the processes of a concurrent system.

The problem of producing a correct system is essentially two-fold: first, that the overall behaviour of a system is free of errors, and second that this error free behaviour results in the user's needs being satisfied. If both conditions are met then the system is correct with respect to the user's intentions. If we consider both the functional and temporal aspects of a system's overall behaviour then we can further refine the above two conditions into four conditions that a system must meet if it is to satisfy the user's intentions;

1. *The functional behaviour must be composed of functions, (units of function), that will allow the system to achieve its intended purpose.*

2. *The functional behaviour must be free of logical errors* — asserting an incorrect control line, attempting a divide by zero, using improper variables, etc.

3. *The temporal behaviour must sequence the functions in such a way that the system can achieve its intended purpose* — the control structure must coordinate the various system functions such that they act in concert to achieve the goals of the system.

4. *The temporal behaviour must be free of temporal errors* — temporal errors include *deadlock*, *starvation*, *violation of mutual exclusion*, etc.

For both the functional and temporal aspects of system behaviour, the behaviour must be error free and it must do something such that the user's needs are satisfied.

If any one of the conditions is not met then the system will not operate as intended, although it may still operate and do something. For example, if the wrong units of function are implemented but all other conditions are met then the system will do something correctly, it just will not do what the user wanted. Alternatively, even if each functional unit correctly reflects the user's intentions, the units must still be correctly coordinated for the system to possess the correct overall behaviour.

By providing a clear separation between the two aspects of a system's behaviour the problem of producing correct systems can be divided into more manageable parts. If it can be shown that a system possesses the correct temporal behaviour then the problem of producing a system that will meet the user's intentions will be reduced to one of ensuing that each individual function is correct.

The PAL[1] concurrent system specification language attempts to provide this distinction between functional and temporal behaviour[KHL88, KJH88]. PAL is an executable specification language [Zav82] which can be used to specify the processes of concurrent special purpose computer systems. A process in PAL is expressed as a collection of separate units of functional behaviour, called *activities*, whose execution ordering is specified by a surrounding control structure. An activity represents a unit of functional behaviour which has the property that it can be executed from beginning to end without further data or synchronization events. The control structure specifying the execution ordering of the activities is described using familiar high-level-language constructs. Thus, the overall behaviour of the system depends both on the functionality implemented by the activities and on the execution ordering of those activities. The functional behaviour of the system, represented by the activities, is distinct from the temporal behaviour as expressed by the surrounding control structure.

[1]for Process Activity Language

1.2 Thesis Objective

This thesis is concerned with detecting errors in a system specification that are due to incorrect temporal behaviour; specifically, temporal errors such as deadlock, starvation and mutual exclusion violation. It is the special problems caused by the non-deterministic interactions among the processes of a concurrent system that we are interested in solving. Once a framework for correct interaction of the system processes has been established then any errors with functional behaviour can be resolved using normal debugging techniques.

The PAL process expressions, when executed by a suitable interpreter, become a dynamic model of the proposed system. By exploring this model the user can check that the specified execution ordering of the activities will result in an overall system behaviour that will satisfy his intentions.

What cannot be shown just by executing the specifications is that the temporal behaviour expressed by the PAL expressions is free of temporal errors such as *deadlock* and *starvation*. For example, for a complex control system consisting of a number of cooperating processes, it would be impractical to check the PAL expressions for all possible input conditions. At most we could test and inspect the specification to see that the process expressions sequence the activities in such a way that the intentions of the user are satisfied. What is required is a means of verifying the PAL expressions against a formal specification describing correct temporal behaviour. This would provide a guarantee that the system is free of temporal errors.

By executing the PAL expressions, it can be shown that the activities will be sequenced in such a way that the user's intentions will be met. If we can prove that the PAL expressions are free of temporal errors then we will have reduced the problem of producing a correct system to that of ensuring that each individual activity is correctly implemented.

The goal of this thesis is to develop a method of verifying the PAL expressions against a formal specification expressing the absence of temporal errors.

The research direction we will follow in meeting our goal will be guided by the following considerations:

In meeting this goal we will assume that the PAL expressions to be verified will have already been executed to check that they provide the correct execution ordering for the activities. We also assume that each activity is free of logical errors and that it correctly implements the desired function. These assumptions will allow us to concentrate on the problem of proving that the expressions are free of temporal errors.

The overall aim of any verification method is to allow the user and the developer to gain confidence that a system or design will perform the intended task and that it will do this task correctly. In general, the more complex the verification method appears to the verifier the less confidence he will likely have in the results obtained. Obviously, the more confidence the verifier has in the verification results, the more confident he can be that the system will perform correctly. Therefore, we propose the following criteria that an appropriate verification method should possess:

- the method **must provide a proof** of correctness. This implies that the method must be formally based.

- the method should be **applicable to practical problems.** Some methods have been shown to work on classic example problems but would be unwieldy for real problems;

- the method should be **usable by knowledgeable system developers.** Some methods require considerable theoretical expertise to make them work. If the proof formalism can be hidden behind a suitable user interface then there is more chance that the method would be used to aid in the design of correct systems. This assumption is also made in consideration of the fact that PAL itself was designed such that the application specialist, who may not be a computer specialist, can design his own system.

1.3 Organization of this Thesis

This thesis is organized in the following way.

We begin in Chapter 2 with a presentation on the Process Activity Language. Although our main aim here is to present the PAL syntax that will be used in later chapters, we also wish to provide at least a partial understanding of the development philosophy behind PAL.

In Chapter 3 we present a more detailed discussion of verification concepts and issues leading to a survey of a select number of existing verification methods. Our aim here is to formulate a suitable verification approach by reviewing and analysing past and current verification research.

Chapters 4 and 5 describe the development of the verification method selected in the previous chapter. The method we have chosen is a two part process: first, transform the PAL expressions into a global state graph representation; and second, verify this global state graph against a formal specification of correct temporal behaviour. Chapter 4 describes the development of the state graph generation method which is based on the work of G.M. Karam in his Ph.D. thesis [Kar87]. In Chapter 5 we develop a verifier based on a model checking algorithm, as reported in [Bro86],[CES83],[CBES85] and [CES86], which uses the global state graph as input. Also in Chapter 5 we develop an extension to the basic model checking algorithm to allow *fairness* constraints to be considered when verifying the specification against the global state graph.

Chapter 6 discusses the implementation of the complete PAL verification system. Chapter 7 provides an analysis of the verification system's effectiveness as a practical development tool. A summary, conclusions and proposals for further research are given in the Chapter 8.

1.4 Research Contributions

In this thesis we developed a verification tool to verify that a set of PAL expressions is free of temporal errors. The verifier was developed by combining in a unique way certain elements of two existing verification methods.

We adapted a state graph generation method, developed by G.M. Karam[Kar87] for the Ada environment, to work with PAL expressions. The resulting state graph is then verified by an implementation of the model checking algorithm[Bro86] [CES83] [CBES85] [CES86] against a specification expressing absence of certain temporal errors. Thus, we have combined the power of the model checking approach with an efficient state graph generation method. The model checking approach which we employ allows a system to be checked for any temporal error that can be expressed in Branching Time Temporal Logic (BTL) while Karam's system checks only for certain important temporal errors. However, Karam's state graph generation method provides an efficient means of generating a state graph for a system of concurrent tasks while the model checking algorithm assumes the existence of a state graph model for the system to be verified.

In verifying the PAL expressions for certain temporal properties we also had to deal with the issue of *fairness* for a system of concurrent processes. We developed a set of fairness criteria tailored to the peculiarities of the PAL environment which enable the verifier to establish the fairness of any computation produced by the PAL system. The model checking algorithm was also modified to consider fairness when verifying certain temporal properties.

We developed and implemented the above verification system in MProlog[2] on a VAXStation 2000 workstation running VAX/VMS[3]. The verifier allows a user to specify a temporal property in BTL and have this property verified against the PAL expressions. As well as indicating whether or not the property is satisfied by the

[2]MProlog is a product of LogicWare Inc.
[3]VAXStation 2000 and VAX/VMS are products of Digital Equipment Corporation

expressions, the verifier also provides information to help locate the source of temporal errors, including if possible an explanation of how the property is or is not satisfied and a facility to trace through any execution sequence to the point of error. Also provided is a means to explore the state graph derived from the PAL expressions with an ability to examine the values of particular variables at any state.

Our concern in implementing the verifier was to show the possibilities of the approach rather than its limits. Thus, we have attempted to show how an appropriate user interface can hide the theory behind the method and how the verifier could be used as an effective development tool, rather than attempting to make the verifier as efficient as possible. Consequently, our implementation can only handle small example systems. However, for the most part this is not a limitation of the method but only of our implementation.

Chapter 2

Process Activity Language Overview

PAL is an executable specification language suitable for modelling concurrent system processes. A detailed report on PAL can be found in [KHL88]. For the purposes of this thesis we will outline only the significant features of the language. We begin with a brief presentation on the development philosophy behind PAL.

2.1 PAL Development Philosophy

The development of PAL represents a continuation of previous research into the problem of designing special purpose controllers for embedded systems[KHL88] [JK86] [Joa86]. Both PAL and PAD[1], the predecessor of PAL, provide a notation to express the control structure, i.e., the temporal behaviour, of an embedded process control application. In the PAD notation, temporal behaviour is described using a set of graphical operators similar in some aspects to both Petri nets [Pet77] and data flow diagrams. In PAL, temporal behaviour is described by a set of concurrent cooperating processes using a structured high-level-language. As well as moving to a structured

[1]PAD - for Process Activity Diagrams

language format, PAL also added message passing and global variable handling primitives to facilitate interprocess communication. The temporal behaviour expressed by the PAD/PAL notation describes the sequence of functions required of the control system in response to stimuli from the controlled environment.

The intended domain of application for PAD/PAL is in the area of *embedded* control systems. Embedded computer systems are characterized by the following attributes:

- they are usually only one element of a larger system whose overall task may or may not be computer related. For example, computers embedded in aircraft 'fly-by-wire' systems would be one element of a system which includes the aircraft's control surface acutators, sensors and human input devices. The role of the embedded computer is to coordinate the actions of the various subsystems in order that the intended purpose of the complete system can be fulfilled;

- they act in an event-response mode, that is, they perform functions in response to events coming from the controlled environment;

- they are usually configured for a single purpose, unlike a general purpose computer which may carry out a number of functions; and

- in the overall environment in which embedded systems are inserted there is usually a requirement for concurrent control actions. There may be a need for simultaneous control of multiple system resources or the need to detect and respond to multiple concurrent events.

Embedded systems applications areas that PAD/PAL seems particularily suited to are those which show a need for central coordination among a number of specialized microprocessor elements such as is found in Flexible Manufacturing Systems (FMS) or in other industrial process control applications[PKGP86]. PAD's was also applied in the design of a multiple user hospital system in [CK83].

In PAD/PAL, the various actions that an embedded system must perform in response to an event are termed *activities*. Each activity represents a unit of functional behaviour that can be started and completed without further information. For example, a routine in the control logic of an industrial robot to move a specific joint a certain number of degrees is an activity. The command routines to open or close a valve are also activities. In fact, each separate action needed to control an industrial process is represented as an activity. The PAD/PAL notation defines the precedence relation of the activities required to meet the needs of the application. The introduction of interprocess communication primitives in PAL allows the event-responses of a system to be represented as cooperating concurrent processes. This allows concurrency requirements in the controlled system to be matched by concurrent processes in the system controller.

The general aim of PAD/PAL research was to realize a system specification method that would eventually allow users to develop their own control systems in certain application areas. By bringing the development process closer to the user his specialized knowledge of his application area could be used to develop more suitable systems. The PAD/PAL notation and analysis tools were designed such that it would be relatively easy for a possibly non-computer expert to specify his own system.

Using various development tools the user can determine, before implementing the system, if the given temporal behaviour will sequence the activities in such a way that his intentions will be achieved. By separating the functional behaviour, the activities, from the temporal behaviour, the PAL process expressions, the user is able to concentrate first on correct coordination of activities and then on function. In [Lac89] a simulation tool was developed to model the effects of various activity scheduling algorithms on the meeting of real-time constraints. In the simulator, the activities were assumed to be correctly implemented and therefore could be represented as finite fixed time delays. The simulator gathers statistical data which allows the developer

to check that the system will meet real-time constraints. A similar statistical analysis tool was developed in [Joa86] for PAD. Once it has been determined that the control structure of the proposed system meets the needs of the user then the various activities can be implemented separately.

2.2 PAL Syntax

A system, $\mathcal{SYS}$, is represented as a set of processes $\mathcal{P}_i$:

$$\mathcal{SYS} = \{\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_n\}$$

Each $\mathcal{P}_i$ is of the form $\mathcal{P}_i ::= \mathcal{S}_i; \mathcal{X}_i; \mathcal{E}$ where $\mathcal{S}_i$ is the enabling condition for the process, $\mathcal{X}_i$ is the body and $\mathcal{E}$ is the delimiter of the expression.

PAL semantics are defined such that when a process reaches $\mathcal{E}$ it loops back to the beginning of the expression and waits for an enabling event represented by $\mathcal{S}_i$. When the enabling event occurs or a condition satisfying the enabling event becomes true, the process body $\mathcal{X}_i$, is executed.

Each $\mathcal{X}_i$, the body of the process, is composed of activities whose execution ordering is defined by PAL control constructs and primitives. In a high-level programming language context an activity can be thought of as a non-reentrant type of procedure call with the exception that an activity once initiated does not need any additional resources to reach completion. In general, a PAL process can be thought of as a collection of activities whose execution ordering is determined by a surrounding control structure.

2.2.1 Control Constructs

The execution ordering of the constituent activities of a process is governed through the use of the following constructs. PAL supports the notion of hierarchical development in that all of the following constructs allow a process expression to be

constructed from named sub-expressions, i.e., in the following examples each of the sub-expressions $\mathcal{X}_i$ may represent single language primitives, composites of activities, or even other sub-expressions.

2.2.1.1 sequential ordering

$$\mathcal{X}_1 ; \mathcal{X}_2 ; \ldots$$

On completion of the expression $\mathcal{X}_1$ the expression $\mathcal{X}_2$ is executed.

2.2.1.2 concurrent execution

conn

$\mathcal{X}_1 \|$

$\mathcal{X}_2 \|$

$\vdots$

$\mathcal{X}_n$

connend;

The concurrent construct indicates an opportunity for the expressions $\mathcal{X}_1$ to $\mathcal{X}_n$ to be executed concurrently. Whether or not the expressions are actually executed concurrently depends on the availability of system resources. The construct is considered complete only when all expressions have been fully executed. Thus in:

...

$\mathcal{A}$;

conn

$\mathcal{C} \|$

$\mathcal{D}$

connend;

$\mathcal{B}$;

...

the activity $\mathcal{B}$ will not be scheduled for execution until both $\mathcal{C}$ and $\mathcal{D}$ have completed.

2.2.1.3 assignment

In PAL, variables associated with the control structure of the system are termed *condition variables* or simply *conditions* due to the simple fact that system behaviour is conditional on the values of the control variables. PAL condition variables may be of any of the standard types such as *boolean*, *integer*, *real*, etc. The scope of a condition variable is either local to the process in which it is declared, in which case it is known as a *process condition*, or globally accessible by any process in the system, in which case it is known as a *system condition*. Process condition and system condition variables correspond to what are commonly known as local and global variables respectively in other high-level-languages.

$$\textbf{STP}(pc, expression)$$

The **STP**, *Set Process Condition*, primitive is the only means of modifying process condition variables. The process condition *pc* is assigned the value resulting from evaluation of the *expression*.

2.2.1.4 branching of control – case

case *pc*

$exp_1(pc) : \mathcal{X}_1 \mid$

$exp_2(pc) : \mathcal{X}_2 \mid$

$\vdots$

$exp_n(pc) : \mathcal{X}_n$

endcase;

Each $exp_i(pc)$ is a logical expression whose truth value is a function of a process condition variable *pc*. Only one of the case alternatives will be executed based on the value of *pc*.

2.2.1.5 iteration – repeat

repeat

$\mathcal{X}$

until *exit condition*;

The body of code $\mathcal{X}$ is repeatedly executed until the *exit condition* becomes true. The exit condition is a logical expression whose truth value is a function of one or more process condition variables.

2.2.2 Interprocess Communication

Interprocess communication in PAL is achieved via two mechanisms: message passing and system condition variables.

Message Passing

Messages are passed between PAL processes via named, one-way, finite capacity channels operated on by two messaging primitives:

- **send**(*channel,parameters*)
- **receive**(*channel,parameters*)

A channel is defined by a single **send** primitive in one process and a single **receive** primitive in another process. There can only be one **send** and one **receive** associated with each channel. Each channel has a fixed, finite message capacity.

If a channel is empty then the process executing the **receive** primitive will wait until a message is available. If the channel is full then the process executing the **send** primitive will wait until after a message has been received. If the channel is not full the **send** is non-blocking.

System Condition Variables

In PAL, *system conditions* are of a special type known as *billboard* variables which are defined such that any process may read the value of the variable but only one of the processes in the system has write privileges. Billboard variables may be any of the standard types such as *boolean*, *integer*, etc. Language primitives to access the system conditions are:

- **STS**(*sc*,*expression*)
- **RDS**(*sc*,*pc*)

The *Set System Condition*, **STS**, primitive sets the system condition, *sc*, to the value resulting from evaluation of the *expression*. The *Read System Condition*, **RDS**, primitive assigns to the process condition variable *pc* the current value of the system condition. To use a system condition variable within a process the value of the system condition must first be assigned to a process condition via the **RDS** instruction.

2.2.3 Select Construct

The **select** construct has been modified from [KHL88] to include the use of guarded commands[Dij75]. The use of guarded commands gives the designer more control over the non-deterministic outcome of the statement.

select

$\mathcal{G}_1 \rightarrow \mathcal{X}_1$ |

$\mathcal{G}_2 \rightarrow \mathcal{X}_2$ |

$\vdots$

$\mathcal{G}_n \rightarrow \mathcal{X}_n$

endselect;

Each of the guards $\mathcal{G}_i$ is a logical expression whose truth value is a function of one or more process conditions. This guarded command structure is similar to the Ada

select statement but is more general in scope in that it allows the command sequence following the guard, $\mathcal{X}_i$, to begin with either a **send** or a **receive** statement. In Ada the select statement is only used to build selective accepts with each guard being followed by one **accept** (receive) statement.

The select construct works in the following manner. The guards are first evaluated to determine the set of open execution alternatives. Next, for each of the open alternatives, the first statement in the command list $\mathcal{X}_i$ is evaluated for its eligibility to be executed. Eligibility is determined as follows:

- **receive** — eligible if there is a message in the specified channel
- **send** — eligible if the specified channel is not full

Of the alternatives whose first statement is eligible one is chosen non-deterministically to be executed. If none of the command lists in the open set begin with an eligible statement then the first to become eligible is selected. The construct does not end until one execution path has been selected and executed. An example follows:

select

$x = 3 \rightarrow \textbf{send}(chan1, req)$ |

$switch1 \rightarrow \textbf{send}(chan3, v1)$

endselect;

First both guards, $x = 3$ and $switch1$ are evaluated. If only one of the guards evaluates to **true** and the first primitive in the command list following this guard is eligible to be executed then the command list following this guard is executed. If for both alternatives the guards evaluate to **true** then the eligibility of the first primitive in each of the command lists, $\textbf{send}(chan1, req)$, and $\textbf{send}(chan3, v1)$ is determined. If both are eligible then one of the paths is chosen non-deterministically. If neither are currently eligible then the first to become eligible will be the path that is chosen.

2.2.4 Controller-Environment Interaction

An embedded computer system is usually only one element of a larger system. From the point of view of the embedded computer system we can distinguish between the embedded computer system itself and the rest of the system. We can refer to those elements of the overall system that the embedded system has an influence on as the environment or controlled element and we can refer to the embedded system as the controller element. In PAL, interaction between the controller and the environment is accomplished via *activities* and *events*. Relevant information about the state of the environment is communicated to the controller via events. Each event is caused by some action within the environment such as a temperature activated relay or a liquid level sensing switch, etc. The event is communicated to the controller who may initiate a response, a certain sequence of activities, which will act on the environment to deal with the event. For example an activity may close a valve in response to a level sensor event. The effects of an activity in response to one event may invoke further events from the environment.

If we can view the environment as one or more concurrent processes then we can say that the controller-environment interaction via the activities and events is a form of interprocess communication. The events communicate to the controller the state of the environment and the controller communicates an action to be taken in response to the event. When designing an embedded control system it is useful to explicitly model the environment as one or more processes. This will ensure that controller-environment interactions will be such that the user's intentions will be met. This is the desired approach to take in specifying concurrent systems using an executable specification langauge. The explicit modelling of the environment allows a greater understanding of complex environment-controller interactions. This in turn aids in developing a correct system. There are times however when the behaviour of the environment cannot be explicitly modelled. In these cases a certain behaviour for the environment is assumed and the controller designed accordingly. Validation of the

controller could then be performed based on the assumed environmental behaviour.

2.3 Assumptions

The above description and that found in [KHL88] and [KJH88] represent the complete PAL system. For the purposes of this thesis and our goal of verifying PAL expressions we will make the following assumptions to better define the scope of our work. An explanation as to why each assumption was made is also given.

1. All the process expression enabling events, S_i, are bound to a condition which represents the fact that the system is turned on, thus all expressions model continuously looping processes.

   This assumption will allow us to consider verification methods which assume infinite sequences of execution states.

2. All PAL expressions are syntactically correct as given. Syntax checking for errors such as unmatched parentheses, missing delimiters and misspellings has already been done at this point.

   The correction of syntax errors belongs in the domain of language editors and compilers. We wish to concentrate only on verification.

3. Any concurrent composition of activities within a process can be flattened out into a single sequential execution stream. This assumption can be made for the following reasons:

   - all the separate execution paths within a concurrent composition must complete before the construct itself is complete;
   - there can be no assumption about ordering of activity execution in a concurrent composition, i.e. the paths are independent which implies that they do not interact either directly or through a shared resource, therefore any ordering is suitable.

This assumption does not impose any unrealistic restrictions on our ability to specify process behaviour because the concurrent construct's main purpose is to indicate opportunities for concurrent execution of activities where separate resources are available. This assumption will simplify the verification problem.

4. In the PAL design environment, controller-environment interaction is accomplished via *activities* acting on the environment and environmental *events* affecting the controller. We will consider the environment as being modelled by one or more processes. Interaction between these environment processes and controller processes will be via message passing or global variables.

   This assumption allows us to consider only one type of process and only two types of interprocess interaction without losing any of the modelling power of the language.

5. All activities are functionally correct as given, that is, each activity does something and it does it without error.

   This assumption allows us to break the verification problem into two parts; correctness of the system's functional behaviour as represented by the activities, and correctness of the system's temporal behaviour as represented by the control structure. If we assume that all activities are correct then to show the correctness of the system as a whole, we need only to prove that the activities will eventually be executed and that the proper activity will be the one that is executed. Under this assumption each activity execution represents an indeterminate but finite time delay allowing us to ignore activities when constructing a proof of correctness of the system's temporal behaviour.

Chapter 3

Selection of Verification Method

In this chapter we select and outline an approach that we will follow in meeting our goals as stated in Chapter 1. We begin with a discussion of verification within the overall context of validation and system development. Our aim here is to provide an understanding of the complexities involved in trying to prove the correctness of a system. We then present a critical survey of verification methods, on the basis of which we define our approach to verifying the correctness of a PAL system.

3.1 Validation

In this thesis we will use the term validation to describe the activity of ensuring that a specification or implementation at every level of the development process is consistent with the user's needs. There are essentially three classes of validation methods [Hau84]:

1. **Inspection** involves the manual application of some informal method to judge the consistency between needs and requirements and between informally stated requirements and a specification or implementation. While this method is capable of detecting many types of errors its effectiveness is limited in discovering

logical inconsistencies or other complex process relationships. Overall, the effectiveness of validation by inspection is limited by the skill and experience of the inspector.

2. **Testing** is the application of test cases to an implementation of a system to see that it does perform as required. A specific set of input conditions and a corresponding set of output conditions constitute each test case. Each set of input conditions is mapped by the system to a specific set of output conditions. The system is judged to be consistent with requirements if on applying a certain set of input conditions we get the expected set of output conditions. Unfortunately, the correct performance of the system for the test cases is only a necessary but not sufficient condition for proof of correctness of the system. Thus, test strategies are inherently limited because the number of test cases required to thoroughly exercise even a simple system is prohibitively large[BBFM82]. In almost all cases testing for all possible input combinations is impractical, therefore we cannot prove correctness via testing. Testing can be considered as a semi-formal validation method as there are methods of formally deriving a set of test cases according to some criteria. i.e., the set of test cases that will exercise all modules of a system, or the set that represents all normally expected input conditions, etc.

3. **Verification** is a formal method of validation in which a proof is constructed to show the correctness of an implementation with respect to a specification. The object of a verification method is to develop an argument to prove that a formal specification is satisfied by an implementation. If the argument is valid for all allowable input conditions then this argument takes the place of an exhaustive set of test cases. Verification provides a proof of correctness. Prerequisites for verification are a formally stated specification and a formally based implementation.

In [BBFM82, JT79] (and others) validation and verification are defined as separate activities but with essentially the same definitions. We use the terms as they are defined in [Hau84] and [Gal87]. In all cases verification implies a formal proof of correctness but definitions for validation are wide ranging. In [BBFM82] and [JT79] validation is defined as a means to ensure that a specification is complete and accurate with respect to a user's needs, while verification is a separate activity which proves that this specification is satisfied by a certain implementation. Thus, if the user's needs are properly described in the specification then any implementation that is verified as being correct against this specification must also meet the user's needs. Our definition of validation, which follows that of [Hau84] and [Gal87], is extended to include the activity of verification. We believe that this is the more pragmatic definition because some form of acceptance testing will always be performed even after an implementation has been verified. Even though verification can prove that a system is correct with respect to a specification, it is doubtful that a user would accept the system without some form of testing. Thus, verification is just one means of ensuring that the final product is accurate and complete with respect to the user's needs.

3.2 The Validation Problem

Validation and hence verification are not autonomous concepts. That is, a specification or implementation cannot be validated in isolation. Validation shows a relationship between two entities: the user's needs and a specification, or a specification and an implementation. Verification provides a proof that two formally stated entities are logically consistent with each other. It follows then, that the entity that is validated is only as good as the standard against which it was validated. This is important, especially for verification, where one would expect that if we can prove that an implementation is correct with respect to a specification then the system should achieve its

intended purpose. However, this is only the case if the specification itself was an accurate and complete statement of the user's needs. Thus, the problem still exists that although formal verification methods are available to show that a formal specification is satisfied by an implementation, very little exists in the way of formal methods to show that the specification accurately reflects the user's requirements[BBFM82].

We can also note that the cost of revealing and fixing errors becomes greater with each step in the development process[JT79, page 339]. Hence it is most cost effective to apply validation methods as early as possible in the development process. The earlier that a user can see that his intentions will be satisfied by the proposed system the better (and cheaper). Validation by inspection does not always reveal complex errors, and testing, being a dynamic means of validation, can only be applied once an implementation is available. Therefore, obtaining meaningful validation early in the development process is a problem.

One solution is the operational approach to system design as proposed by Zave in [Zav82]. The main feature of this approach is that the formal specification is expressed in an executable language. This specification when executed by a suitable interpreter actually becomes a dynamic model of the proposed system. By exploring this model through the use of various inspection and testing tools the developer and the system user can validate the specification against their requirements. Thus the specification becomes an operational part of the development process rather than being just a reference document. The advantage of this method is that with a suitably expressive specification language the full functions and timing relationships of the proposed system can be modelled and validated without the expense of developing an implementation. As well, since the language must be formally based in order to be executable, the validated specification can be used as the standard against which to verify any implementation.

PAL provides such an approach to system design. By adopting the operational approach PAL allows the user to validate the functional and temporal behaviour of

the system against his intentions. Execution of the PAL process specifications allows an informal validation of the functional and temporal behaviour of the system in so far as the sequencing of the system's functional components is concerned. That is, it can be validated that the functional units of the system are sequenced in such a way that the overall functions required of the system are performed.

The above discussion serves to reiterate our approach to the problem of developing correct systems. The use of an executable specification language allows the developer to informally validate the temporal behaviour of his proposed system. We can then take this informally validated specification and formally validate, i.e., verify, that it is correct with respect to a specification expressing absence of temporal errors. In so doing the developer can be confident that the system will provide the correct sequencing of functions and that the system will be free from temporal errors.

3.3 Correctness

To verify is to check or test the truth or correctness of something. If we can somehow prove that at each step of the development process the specification for a system is satisfied by an implementation, then we can say that we have a *correct* implementation of the specification.

Differing notions of correctness have evolved from basic research in verification theory. Original verification efforts were aimed at proving the correctness of existing programs[Flo67, Hoa69, BBFM82]. The specifications against which the programs were verified were expressed as a set of *correctness properties* grouped into two main classes:

- **safety properties** — those properties which state that nothing bad will ever happen to the system; and
- **liveness properties** — those properties which state that desirable actions will eventually be performed.

These properties state in terms of the variables of a given system that the system possesses certain behavioural attributes such as freedom from deadlock, absence of starvation, freedom from violation of mutually exclusive access to a shared resource, etc. If an implementation is correct with respect to a set of correctness properties then it can be stated that the system possesses the specified properties. The set of correctness properties is also referred to as a *partial specification*[CP88] since only limited aspects of system behaviour are specified. However, if the properties are carefully chosen to cover fundamental aspects of the system's behaviour then we can be confident that the system will do something correctly.

Early methods of verifying existing programs have led to a better understanding of the nature of programming languages. The fundamental understanding that has been gained is a recognition that a program can be treated as a mathematical object and therefore it and its specification can be reasoned with in the same manner as other mathematical entities[Lon79]. This understanding has in turn has led to programming languages whose semantics aid verification efforts. A continuing parallel development in verification theory, language semantics and development methods is aimed at providing a guarantee that a system is correct by virtue of the development methodology used. Thus, another notion of correctness is a proof of equivalence between a formally derived specification and an implementation. If a formal development methodology can guarantee that a specification is correct (with respect to a set of correctness properties) then any implementation which we can prove is equivalent must also be correct. However, a system which is guaranteed correct by virtue of the development methodology used is not yet an attainable goal. While we can use mathematical transformations to show that one formally expressed program object is equivalent to another we still have the problem of, first, showing that the object describes a system that will meet the user's intentions and second, that the object is free of temporal errors.

In this thesis we have chosen to verify correctness with respect to a set of correctness properties. We have been given as a starting point a set of PAL process expressions describing the temporal behaviour of the proposed system. To show that the PAL expressions are free of temporal errors we will verify them against a partial specification expressing certain correctness properties for the system.

3.3.1 Definition of Terms

Before carrying on with a more detailed description of the various correctness properties we should define some terms that we will be using.

Both safety and liveness properties are usually expressed in terms of *system states*, s_i, which are defined as being a conjunct of the values of all variables v_i of the system at a particular instant in time, i.e.,

$$s_i = \{v_1, v_2, \ldots, v_n\}$$

Associated with each system state s_i is an *accessibility relation*, r, which defines allowable transformations from one state to another. The accessibility relation is usually defined in terms of the control program for the system, thus a particular system state represents the effects of system execution up to a certain point for a given set of initial variable values. Given the above definition of a system state we can think of the execution of a system over time as being composed of a number of discrete system states with transformations from one state to the next being defined by the control logic of the system and the values of system variables at that state.

The *state space* of a system is defined as the Cartesian product of the allowable range of values for each variable in the system. In general, only a subset of the state space of a system represents allowable system states since certain combinations of system variables are not valid for the system.

A *computation* is an infinite sequence of system states. For a system with a finite state space a computation is composed of a finite sequence of system states arranged in a loop.

For a concurrent system consisting of a m processes executing in parallel the state of the system is given as a conjunct of the states of the individual processes. The state space of a concurrent system is a Cartesian product of the state spaces of each of the individual processes and of the range of values of any global variables in the system. Any process or set of processes whose state space is independent of the rest of the system, i.e., the process does not interact with the other system processes, can be considered as a separate system. Again, the execution of a concurrent system can be viewed as being composed of a number of discrete system states. Execution state sequences in the concurrent context imply an interleaved model of execution where transitions from one system state to the next represent a change in state of one of the concurrent processes [MP81b].

3.3.2 Safety Properties

Safety properties assert that bad things will never occur, i.e. the system will never enter an unacceptable state. These are a class of properties that hold continuously throughout execution and therefore are also referred to as *invariance* properties. Some examples of safety properties include:

partial correctness — a program is partially correct if at the start of execution the set of initial variable values for the system satisfies some pre-condition $\varphi(x_i)$ and then if the system terminates the output variable values satisfy some post-condition $\psi(x_i, y_i)$; where $\psi(x_i, y_i)$ describes a relation between the initial variable values x_i and the output variable values y_i, i.e., $\psi(x_i, y_i)$ is an input/output specification for the system. Informally, partial correctness states that given the proper input values then if the system terminates it will produce the expected result. Partial correctness does not, however, guarantee that the system will ever terminate.

mutual exclusion — Suppose that two processes of a concurrent system each contain a critical section representing access to a common resource. The mutual exclusion property asserts that the system will never a reach a state in which both processes are in their respective critical sections.

absence of deadlock — a concurrent system consisting of m processes is *deadlocked* if the system reaches a state at which no process is enabled [MP81b]. A process is not enabled, i.e., it is disabled, when the accessibility relation defining process state transformations for the process is not satisfied for the particular set of variable values at that process state. For the system to progress from one system state to another at least one of the processes must be enabled.

This property is also referred to as *system-wide deadlock* in [Kar87] and as *absolute deadlock* in [Pnu81]. A *local deadlock* may also occur in which only a subset of the processes of a system are simultaneously unable to progress while the rest of the processes of the system are enabled.

global invariance — the invariance of some properties may be independent of any particular execution state of the system, i.e., they must hold no matter what the system does [MP81b]. For example, for an elevator control system, a global invariance property could assert that the controller will never allow the elevator to exceed the maximum number of physical floors.

3.3.3 Liveness Properties

Liveness properties state that something good will eventually happen, i.e. the system will eventually enter a desirable state. Examples of liveness properties are:

total correctness — a system is *totally correct* if it is partially correct and termination is guaranteed [Pnu81]. Like partial correctness, this property is only relevant for systems which are expected to terminate. Informally, total correctness states that if the system input conditions x_i satisfy the system input specification φ

then eventually the system is guaranteed to reach a terminal location where the output conditions y_i satisfy the output specification ψ.

absence from starvation — starvation is defined as a situation in which some process(es) cannot proceed even though the system may still progress by having other processes execute [MP81b]. In general, this property can be interpreted as asserting that a process will not always be in the same state.

Absence from starvation is also referred to as *liveness* or *freedom from individual starvation* or *absence of livelock* in [MP81b].

intermittent assertion — this property expresses a causality relationship between any two events [MP81b]. For example, if a process ever reaches a state where ϕ is true then an intermittent assertion expresses the fact that it will eventually reach a state where ϕ' is true.

accessibility — if a process has a critical section then this property asserts that if the process wishes to enter its critical section it will eventually be allowed access [MP81b]. The correct construction of critical sections should allow both the safety property, *mutual exclusion* and the liveness property, *accessibility*, to be satisfied on all system computations.

responsiveness — this property is concerned with client-server interactions in continuously operating systems. Many continuously executing systems such as embedded systems and operating systems provide services on request to other elements of the system. A desirable property of client-server interactions is that for each request for service, the service is eventually provided.

Responsiveness properties can also be combined with other liveness properties specifying that the resource granted is eventually given back to system by the client. The mutual exclusion property can also be stated for each resource to ensure its continued correct behaviour.

3.3.4 Correctness for Concurrent Systems

The concepts of partial and total correctness were originally developed to prove the correctness of sequential programs; the works of Floyd [Flo67] and Hoare [Hoa69] provide the earliest theories of program correctness from which the concepts of partial and total correctness were derived. A sequential program is said to be totally correct if it can be proven that it is partially correct and that it terminates. However, for concurrent systems which are often meant to operate continuously, termination is not important. For such systems other types of properties are more important, such as:

- will a request for service eventually be answered;
- will mutual exclusive access to a shared resource be preserved;
- will a message eventually be received; or
- will a certain system variable always have an acceptable value.

Thus, for concurrent systems we are more interested in continuous properties of the system rather than, as for a sequential program, the results it achieves on terminating.

At this point we should pause and summarize the role that we want our verification method to play in the development of a correct system. We are given a set of PAL expressions describing the temporal behaviour required of a system. The expressions have been executed on a suitable interpreter to validate that the described temporal behaviour will sequence the activities is such a way that the resulting overall system behaviour will satisfy the user's intentions. Our goal now is to provide a verification method that will prove the correctness of the process expressions with respect to a partial specification expressing absence of temporal errors. The correctness properties that we wish to be able to prove about the process expressions include:

- absence of deadlock;
- absence of starvation;

- correct access to shared resources (mutual exclusion).

In addition we would also like to be able to prove that certain properties expressing global invariants and intermittent assertions peculiar to the given system are satisfied by the process expressions. Carefully chosen properties of this type will aid in showing that the correct sequencing of activities will occur under all possible conditions.

3.4 A Survey of Verification Methods

We have two main aims in presenting this survey. First, we wish to acquaint ourselves with some of the more influential research contributions in the field of verification methods. For the most part the methods surveyed form a theoretical basis for much of the current research in the field. Second, in presenting the survey we will also comment on the relative strengths and drawbacks of each method in terms of the criteria outlined in our statement of research objectives (Section 1.2). Our purpose in doing this is to define the verification approach we will pursue in the remainder of this thesis. The following survey is tailored to meet the above aims. We do not attempt to cover all of the various verification methods but only those which have provided significant direction to verification research in general or to our research in particular. Some of the methods, in particular the use of temporal logic for verification, are covered in greater depth in order to provide the necessary theoretical basis for later discussions.

3.4.1 Floyd's Inductive Assertion Method

In *Assigning Meaning to Programs* [Flo67] Floyd introduced the inductive assertion method of program verification. In his paper program behaviour was represented in flow chart form and the specification was stated in terms of an input/output relationship for the program. The method involved assigning to each flow chart element a propositional assertion about the program variables. The assertions are

chosen such that they are satisfied each time program control reaches the statement with which they are associated. Because each assertion must be satisfied each time it is reached they are also referred to as *invariants* of the program. The assertion attached to the *start* of the program is chosen such that it is true if control is at the start of the program and the input specification of the program is met. Similarly the assertion at the *end* of the program is chosen such that it becomes true if the program terminates and the program's output specification is met.

The program is correct with respect to the input/output specification if it can be shown that when the program starts up that the starting assertion is true and that if the program terminates then that the end assertion is also true. To show this Floyd used an inductive argument proceeding from the starting assertion to the end assertion. What the proof shows is that if a program is started with correct inputs then *if* it terminates it will give the intended result.

What is not proven however is that the program *will* terminate thus the inductive assertion method only provides a proof that the program satisfies the safety property of *partial correctness*. To prove termination and thus provide a proof of *total correctness* Floyd used a method involving well-ordered sets in which a function of the program's variables is associated with each program location. The functions are devised such that the value of the function decreases with respect to the well-ordered set from which the function variables are taken. A commonly used well-ordered set is the set of non-negative integers. One important characteristic of well-ordered sets is that they must have a *least* member. If it can be shown that the value produced by the function of the program variables is eventually equal to the least member, then this is proof that the program terminates.

Floyd's paper provided assertions for three key programming constructs: conditional branching (*if-then-else*), join of control and assignment. Floyd's inductive assertion method is important in that it introduced a set of rules for the construction of invariants, i.e., it was a first attempt to define an underlying formalism for

programming constructs on which to base proofs of correctness.

3.4.2 Hoare's Axiomatic Basis for Computer Programming

Continuing on in the verification research direction that Floyd had established was the work of C.A.R. Hoare as reported in [Hoa69]. Hoare took Floyd's work one step further by formulating Floyd's rules into a formal axiomatic system. By formalizing the rules of assertion construction as an axiomatic system Hoare was able to bring the methods of formal logic to bear on the problems of correctness.

Hoare defined the notation $\mathcal{P}\ \{\ \mathcal{S}\}\ \mathcal{Q}$ which is interpreted to mean that if the assertion $\mathcal{P}$ is true before the program statement $\mathcal{S}$ is executed then the assertion $\mathcal{Q}$ will be true after $\mathcal{S}$ is executed. The assertions, as in Floyd's method, are propositions on the program variables. Hoare defined a single axiom of assignment and a set of four inference rules which he used to axiomatically represent structured program statements such as *case* and *while*. Each statement of a program can be represented in the form of an axiom.

Given the above axiom system and a starting assertion Hoare was able to employ the methods of formal logic to construct proofs of correctness for simple sequential programs. Again, however, the proof is made under the assumption of termination. Thus the notation $\mathcal{P}\ \{\ \mathcal{S}\}\ \mathcal{Q}$ must be interpreted as *given the input conditions* $\mathcal{P}$ *then if the statements* $\mathcal{S}$ *terminate the results will satisfy* $\mathcal{Q}$.

One of the main ideas in Hoare's paper is that axioms and rules of inference should be accepted *as the ultimately definitive specification of the meaning of the language*. Thus the axiomatic system serves as both a specification for a programming language and as a basis for the proof of correctness of a program expressed in that language. By defining the complete semantics of a language axiomatically as was done for Pascal [HW73] the correctness of any implementation of the language can be verified independently of machine specific features of the language. Thus, a program verified as being correct in one implementation of its language will still be correct for

another implementation of its language.

While the Floyd-Hoare approach has been applied with much success to sequential programs they are not well suited for reasoning with concurrent programs [Gal87, page 33]. What their work has done however, is to bring about an increased emphasis on the formalisms of program construction and verification. The result of this emphasis has been the development of a number of alternative verification approaches more suited to the notions of concurrent computation. The temporal logic approach discussed in the next section is one such alternative.

3.4.3 Temporal Logic Verification

Temporal logic was first introduced as a means of reasoning about concurrent programs by Pnueli [Pnu81] who based his work on that of Burstall and Manna and Waldinger [Gal87, page 35]. Temporal logic was extended from classical logic by adding temporal operators which allow the validity of a formula to be based not only on its normal logical assertion but also on a time element. For example, the validity of the statement "It rains today." depends on the date that the assertion is made as well as on the validity of the assertion that it is raining. Whereas the validity of the non-temporal statement, "It rains," only depends on the current weather conditions. Another way to consider temporal logic is as a *dynamic* logic able to express conditions changing over time while non-temporal logic is a *static* logic able to make assertions only about conditions at one particular instant.

We consider a sequence of system states $\sigma = s_0, s_1, \ldots$, with each state providing a complete description of the system at each instant $i = 0, 1, \ldots$. Temporal logic can be used to make assertions about changing system conditions over the execution sequence, while classical logic can only make assertions about system conditions at one particular instant or state. The temporal operators and their semantics described below provide the notation for expressing dynamic properties of a system.

In defining temporal logic operators there are two main views regarding the underlying nature of time. One such view is that time is deterministic or linear and the other is that time is non-deterministic or branching. These two views lead to the two basic temporal logic systems described below.

3.4.3.1 Linear Time Temporal Logic

Under the assumption of linear time the situation at each present or instant in time has only one possible future or next instant. Translated on to a computer model this means that each possible system state has only one possible next system state. This results in a linear sequence of system states. Non-determinism in system execution is accommodated by defining a linear state sequence or computation for each possible non-deterministic alternative. Collectively the set of deterministic computations describes all possible behaviours of the system.

On the basis of these linear state sequences we can reason about and describe system behaviour using Linear Time Temporal Logic (LTL). Each state of the state sequences can be thought of as a snapshot of the system execution state at a particular instant in time. At each state are the actual values of all control and data variables in the system. Correctness properties required of the system are expressed in LTL in terms of the state variables of the system and proven for each possible computation of the system. If the LTL formulae hold for all computations then the system is correct with respect to those LTL formulae. Thus the set of LTL formulae which are checked on the computations constitute a partial specification for the system.

Formulation rules for well-formed formulae (wff) for LTL are as follows:

1. Every wff of the classical first-order logic $\mathcal{L}$ (with equality) belongs to LTL;

2. If $p, q \in$ LTL then:

   - $\bigcirc p$ (*next-time*)
   - $\Diamond p$ (*eventually*)

- $\Box p$ (*always*)
- $p \mathcal{U} q$ (*until*)

are $\in$ LTL as well.

Before defining the semantics of the LTL temporal operators it would be best to pause briefly and review the meaning of some of the symbols we will be using. We use the symbol $\models$, as in $\models p$, to state that p is valid in *all* domains, whereas $\models\!\!\!\models p$ denotes that p is valid within some *particular* domain. We can write $\mathcal{M} \models\!\!\!\models p$ to say that the formula p is valid for the domain $\mathcal{M}$. Typically, the domain of reference for a $\models\!\!\!\models$ valid formula is the set of computations for a particular system. By the above, it follows that a formula valid under $\models$ is also valid under $\models\!\!\!\models$ [MP81b, page 252].

The classical atomic elements from which the well-formed formulae are constructed are termed *immediate assertions*[OL82]. An immediate assertion is a boolean valued function of the control or data variables of a system state. For example, the function $(x < 3)$ is an immediate assertion expressed in terms of some system variable x. We can write $s_i \models\!\!\!\models (x < 3)$ to denote that the function is *true* or is *satisfied* at the system state s_i. Also, each atomic statement of a program is given a label, i.e., $\mathbf{l_i}$, and each state contains a *location variable*, lv_i, for each process. The value of the location variable indicates which program statement is currently being executed. We can then express in an LTL formula the fact that program control is at a certain statement by the notation $at\mathbf{l_i}$ which would be valid in the present state if the value of lv_i is $\mathbf{l_i}$.

The semantics of the temporal operators $\bigcirc$, $\Diamond$, $\Box$ and $\mathcal{U}$ are described below. Any formula without temporal operators is termed a *classical* formula since its validity is determined according to the semantics defined for the normal first-order classical logic, $\mathcal{L}$.

LTL Semantics

The semantics of the temporal operators are defined below for an arbitrary computation $\sigma = s_0, s_1, s_2, \ldots$. Each s_i is a state giving a full description of the system at

time instant t_i where $i = 0, 1, \ldots$. The state s_0 is considered the present or current state corresponding to time t_0. A formula, $\mathcal{F} \in$ LTL, is satisfied, denoted by $\sigma \models \mathcal{F}$, on the computation, when the following definitions hold:

- $\mathcal{F} \in \mathcal{L}$ iff $s_0 \models p$

  If $\mathcal{F}$ is a classical formula then the validity of $\mathcal{F}$ is determined in the current state according to the rules of first-order logic, $\mathcal{L}$.

- $\mathcal{F} \equiv \Box p$ iff $\forall i,\ i \geq 0\ \ s_i \models p$.

  The *always* operator asserts that p is true in the present state and will be true for all future states on the computation.

- $\mathcal{F} \equiv \Diamond p$ iff $\exists i,\ i \geq 0\ \ s_i \models p$.

  The *eventually* operator asserts that p is true either now or at some future state of the computation.

- $\mathcal{F} \equiv \bigcirc p$ iff $s_1 \models p$

  The *next-time* operator asserts that p is true at the next state on the computation.

- $\mathcal{F} \equiv p \mathcal{U} q$ iff $\exists k$ such that $s_k \models q$ and $\forall i,\ 0 \leq i < k,\ s_i \models p$

  The binary *until* operator asserts that p will be true until q becomes true. Note that by this definition $p \mathcal{U} q \Rightarrow \Diamond q$.

Some examples of well-formed LTL formulae are:

1. $\models p \Rightarrow \Diamond q$

   If p is true in the present state then q will eventually be true.

2. $\models \bigcirc \Diamond p$

   In the next state from the present state p will eventually be true.

3. $\models \Diamond (p \wedge \bigcirc q)$

   Eventually, at a state s_i, p will be true and at s_{i+1} q will be true.

In terms of LTL the general form of a formula expressing a safety property is:

$$\models \Box p$$

which states that p holds for every computation of the system.

For example to express a *global invariance* property we could write

$$\models \Box(count < 4)$$

to state that the variable *count* must always be less than four for all allowable computations of the system.

Liveness properties are expressed in LTL with formulae of the form:

$$\models p \Rightarrow \Diamond q$$

which states that for all allowable computations, if p is presently true then q must eventually be true.

To express an *intermittent assertion* we could write formulae of the form

$$\models (at\mathbf{l}_1 \wedge \phi_1) \Rightarrow (at\mathbf{l}_2 \wedge \phi_2)$$

which states that whenever the program location $at\mathbf{l}_1$ is reached and the immediate assertion ϕ_1 is true then eventually the program will reach a program location $at\mathbf{l}_2$ at which ϕ_2 will be true.

Many systems of temporal logic based on linear time have been defined. Some have defined past operators [Pri67] and others have been defined with either additional operators [Hal85],[DGU87],[Wol83] or a restricted set of operators [SZ87]. We have described above the LTL system known as $\mathcal{DX}$ as defined in [Pnu81]. Good introductions to temporal logic can be found in [RU71] and [Gal87]. Appendix A contains a list of axioms and deductive rules defined for the $\mathcal{DX}$ system of LTL.

3.4.3.2 Branching-Time Temporal Logic

In the branching-time model of time the situation at each present or instant of time has many possible futures; any of which could be true. In computer system terms

the system state at each instant of time has many possible next instant states. Thus, in the branching-time model non-determinism in system execution is naturally represented as a tree structure of possible state sequences. From each system state there may exist a number of alternative execution paths. Thus all possible computations of a system are expressed as a single tree structure of execution states.

Temporal operators defined for Branching-Time Temporal Logic (BTL) are the same as those defined for LTL except that they are quantified over the different possible execution paths. We can reason about system behaviour based on the existence of at least one (existential quantification $\exists$) state sequence or for all (universal quantification $\forall$) possible state sequences.

Formulation rules for well-formed BTL formulae are as follows:

1. Every wff of the classical first-order logic $\mathcal{L}$ belongs to BTL;

2. If $p, q \in$ BTL then:

   - $\forall \Box p$
   - $\exists \Box p$
   - $\forall \Diamond p$
   - $\exists \Diamond p$
   - $\forall \bigcirc p$
   - $\exists \bigcirc p$
   - $\forall p \mathcal{U} q$
   - $\exists p \mathcal{U} q$

   are $\in$ BTL as well.

BTL Semantics

The semantics of the BTL temporal operators are defined below for an arbitrary computation tree Σ. A unique computation on Σ is denoted $\sigma_i = s_0, s_1, s_2, \ldots$. Each

s_i is a state giving a full description of the system at time instant t_i where $i = 0, 1, \ldots$. The state s_0 is considered the present or current state corresponding to time t_0. A formula, $\mathcal{F} \in$ BTL, is satisfied, denoted by $\Sigma \models \mathcal{F}$, (or just $\models \mathcal{F}$ when Σ is understood), on the computation tree when the following definitions hold:

- $\mathcal{F} \in \mathcal{L}$ iff $s_0 \models p$

  If $\mathcal{F}$ is a classical formula then the validity of $\mathcal{F}$ is determined in the current state according to the rules of first-order logic, $\mathcal{L}$.

- $\mathcal{F} \equiv \forall\Box p$ iff $\forall\sigma \forall i,\ i \geq 0,\ s_i \models p$

  The *universal always* operator asserts that p will be true at s_0 and will be true for all future states along all computations starting at s_0.

- $\mathcal{F} \equiv \exists\Box p$ iff $\exists\sigma \forall i,\ i \geq 0,\ s_i \models p$

  The *existential always* operator asserts that there exists a computation on Σ starting at s_0 on which p will always be true.

- $\mathcal{F} \equiv \forall\Diamond p$ iff $\forall\sigma \exists i,\ i \geq 0,\ s_i \models p$

  The *universal eventuality* operator asserts that for all computations starting at s_0 there exists a state at which p is true. Alternatively stated, the $\forall\Diamond$ operator asserts that p is either true at s_0 or is true at some future state on all computations starting at s_0.

- $\mathcal{F} \equiv \exists\Diamond p$ iff $\exists\sigma \exists i,\ i \geq 0,\ s_i \models p$

  The *existential eventuality* operator asserts that there exists a computation starting at s_0 on which there exists a state at which p is true.

- $\mathcal{F} \equiv \forall \bigcirc p$ iff $\forall\sigma\ s_1 \models p$

  The *universal next-time* operator asserts that for all next states from the current state p is true.

- $\mathcal{F} \equiv \exists \bigcirc p$ iff $\exists \sigma\ s_1 \models p$

  The *existential next-time* operator asserts that there exists a next state from the current state at which p is true.

- $\mathcal{F} \equiv \forall p \mathcal{U} q$ iff $\forall \sigma\ \exists k$, such that $s_k \models q$ and $\forall i,\ 0 \leq i < k,\ s_i \models p$

  The *universal until* operator asserts that for all computations starting at s_0 there exists a future state at which q is true and for all states until then p is true.

- $\mathcal{F} \equiv \exists p \mathcal{U} q$ iff $\exists \sigma\ \exists k$ such that $s_k \models q$ and $\forall i, 0 \leq i < k,\ s_i \models p$

  The *existential until* operator asserts that there exists a computation starting at s_0 on which there exists a future state at which q is true and for all states until then p is true.

Some example of well-formed BTL formulae are:

1. $\forall \Box (p \Rightarrow \exists \Diamond q)$

   For all computations of the system starting from its present state, each time that p is found true there will exist a computation starting at that state along which q will eventually be true. This type of formula could be used to express the notion that the correct response to a given system event occurs along at least one of the possible state sequences starting from the state at which the event occurred.

2. $\forall \Box (p \Rightarrow \forall \Diamond q)$

   This formula is similar to the first except that when the event p occurs a correct response q is required for all possible computations.

3. $\forall \bigcirc (\exists \Diamond p)$

   For all possible next states of the current state there exists a computation along which p will eventually be true.

The BTL system described above is $\mathcal{UB}$, the *unified* system of branching time defined in [BAMP81]. As for LTL, many other systems of BTL exist, each with varying degrees of expressive power. In CTL [EC82][CES83] only the operators $\forall\bigcirc$, $\exists\bigcirc$, $\forall\mathcal{U}$ and $\exists\mathcal{U}$ are given as basic operators. The other temporal operators are defined in terms of the four basic operators as follows:

1. $\exists\Diamond p \equiv \exists(true\,\mathcal{U}\,p)$
2. $\forall\Diamond p \equiv \forall(true\,\mathcal{U}\,p)$
3. $\exists\Box p \equiv \neg\forall\Diamond\neg p$
4. $\forall\Box p \equiv \neg\exists\Diamond\neg p$

In $\mathcal{UB}$ only a single temporal operator ($\bigcirc, \Diamond, \Box, \mathcal{U}$) can be combined with a computation quantifier ($\forall, \exists$). However, in CTL*, an extension of CTL, as defined in [EH83], a computation quantifier can be prefixed to any combination of temporal operators thus greatly increasing the expressive power of the logic system.

Again, [Gal87] and [RU71] provide good introductions to many of the different logic systems. Appendix B gives a partial set of BTL axioms and inference rules for the system $\mathcal{UB}$.

3.4.3.3 Applying Temporal Logic for Verification

Programming languages (or formal system description languages) and temporal logic can both be thought of as being related to the execution sequences of a system [Gal87, page 42]. That is, the operational semantics of a language are the means by which the possible computations of a system can be derived from the program describing the system, or even more simply, a program can be thought of as a generator of sequences of system states. The semantics of temporal logic are defined on the computations of a system, i.e., the validity of temporal formulae are directly related to the state sequences of a system. Thus, the computations of a system are a common

element linking temporal formulae and language constructs. Verification researchers have exploited this common ground to directly relate temporal logic formulae to the language statements. That is, the semantics of the language can be stated directly as temporal logic axioms. These temporal axioms can be thought of as a specification for the program. Given that a direct relationship between a language and a set of temporal logic formulae can be established there result two possible ways to verify that a system is correct with respect to its temporal specification.

One way is to take a proof-theoretic approach in which certain temporal premises are first established for the system. These premises are derived by inspection from the semantics of the language describing the system, i.e., deriving temporal formulae for those semantical aspects of the system for which temporal formulae are well known and obviously satisfied. The derived premises are then used in conjunction with temporal logic axioms and deductive rules to deduce other formulae. The goal is to deduce formulae expressing correctness properties for the system. This is the approach taken by Manna and Pnueli in [MP81b, MP81a] and [Pnu81]. A major drawback with this approach is that the construction of the proofs of correctness requires considerable expertise in the formalisms of temporal logic and experience in the construction of logical proofs. As well, for all but the simplest cases proof construction is a tedious and detailed affair. Research into automatic theorem provers for temporal logic to automate the proof construction process is continuing [Aba87] but whether or not practical results are attainable is not yet clear [Gal87].

Another approach to verification using temporal logic is the model-theoretic approach in which a set of temporal formulae expressing desirable correctness properties for the system, i.e., a partial specification, is checked directly on the set of computations generated by the system. The set of computations of a system can be thought of as a model of system behaviour. If the temporal formulae are satisfied on the computations then the set of computations are considered to be a valid model of the system and the system which generated the computations is considered correct with respect

to the partial specification. This *model checking* approach is due to Clarke et.al., in a series of papers [CES83][CBES85][Bro86][CES86]. In the model checking approach the set of computations of a finite state system is expressed as a global state graph (tree of system states) and the specification for the system is expressed in CTL. The main advantage of model checking approach is that it is easily and efficiently automated. The requirement for the behaviour of the system to be completely modelled by a global state graph obviously limits the method to finite state systems. However, the class of finite state systems includes many useful systems.

In order to make the production of verifiably correct systems an attainable goal, the means of ensuring a correct system or of verifying that the system is correct must be accessible and useable by those who actually develop the systems. Obviously a method which requires considerable expertise in temporal logic formalisms and proof construction is not generally usable by system developers. What would be usable is a system in which the formalisms allowing verification could be hidden behind a friendly user interface. Until research into automatic theorem provers for temporal logic is more mature a model-theoretic approach seems to hold the most promise for the development of useable verifiers.

The last two verification methods that we will look at have emphasized the model-theoretic approach with good results. The first is a system for verifying certain correctness properties for Ada programs and the second is the model checking system briefly outlined above.

3.4.4 COL for Verifying Ada Programs

COL[1] is the basis of a system developed by G.M. Karam in [Kar87] to check Ada programs for certain temporal errors such as deadlock and starvation. In COL an Ada program is represented as a set of LTL implications describing the possible state

[1]COL for *CAEDE Operational Language*
CAEDE for *CArleton Embedded system Design Environment*

transitions of each task in the system. The set of implications, referred to as a COL specification, an assertion describing the initial conditions of the system and a special set of LTL implications describing the semantics of the Ada rendezvous provide a complete temporal logic description of the Ada program. Thus, Karam has exploited the proof-theoretic aspect of temporal logic to directly relate temporal logic formulae to Ada language constructs. However, instead of using the formulae to construct proofs of correctness he uses them to generate the minimal set of computations that will fully describe the program's behaviour. That is, instead of the Ada program being a generator of its own computations, he uses the temporal specification to generate the computations. The COL formulae describing an Ada program are manipulated in such a way as to produce a series of program state assertions with each successive assertion showing the change in state of one task in the system. Thus an operational representation, the set of system computations, is derived from the axiomatic representation. Karam developed an inference engine incorporating special LTL inference rules and inference algorithms which mechanically applied inferences to generate program computations from the COL representation. The complete behaviour of the system is shown as a tree structure of program states.

To verify this state graph representation of an Ada program, Karam developed three different checking algorithms; one each for: deadlock detection, starvation detection and assisting in detecting critical race situations. Absence of system-wide deadlock is actually detected during the generation of the state graph. If the inference engine deduces a state assertion from which no succeeding state can be inferred then it has discovered a system-wide deadlock state. Each of the other correctness checking elements of Karam's system use the deadlock free state graph as input.

While this method is essentially a state space exploration method it does have significant advantages over standard state based analysis methods. State based analysis methods such as state machines and Petri-nets employ a method known as *reachability analysis* based on a *reachability tree* to detect deadlock and starvation [Pet77]. A

common problem with state based analysis is *state space explosion* where the number of states or nodes of the reachability tree grows exponentially with each added data element in the system. In [Kar87, pp59-62] Karam showed that while the problem of state explosion cannot be eliminated its effects can be minimized. He showed how a temporal logic representation of a system requires fewer system states to completely represent system behaviour than an equivalent state machine representation because the temporal logic representation automatically removes equivalent system state sequences from the global state graph. State explosion is still a major drawback of state space exploration methods although the temporal logic representation does provide some relief.

3.4.5 The Model Checking Approach

The model checking approach to system verification was developed by E.M. Clarke, E.A. Emerson, M.C. Browne and A.P. Sistla and presented in a series of papers outlining the basic algorithm and some improvements [CBES85][CES83][Bro86][CES86]. Their approach involves first representing the finite state system as a global state transition graph and then applying an efficient algorithm called a *model checker* to check the state graph against a specification expressed in CTL. As was noted in Section 3.4.3.3 (page 43) this method exploits the model-theoretic aspect of temporal logic in that the validity of the temporal logic formulae (the system specification) is determined directly on the system state sequences in the state graph. If the formulae are satisfied on the state graph then the state graph is a valid model or implementation of the specification.

The complexity of the model checking algorithm was shown to be linear in both the size of the specification and the size of the global state graph. The cost of generating a global state graph for the system is not included however, nor is it specified how the global state graph is generated.

The main benefits of this approach are that it provides an efficient algorithm which

is easily automated for the verification of finite state systems. With this algorithm the model can be checked against any correctness property that can be expressed in CTL. On the other hand the model checker requires a global state transition graph for a system. Somehow this graph must be generated. As well the method is inherently limited to the verification of finite state systems.

3.5 Selection of Verification Method

In the traditional approach to the verification of concurrent systems a proof of correctness is established by hand by following one or combinations of the first three methods outlined above. In general the construction of the proof is tedious and requires considerable ingenuity on the part of the verifier.

However, the underlying formalism of these manual methods can also provide a basis for a more operational approach to verification as shown in COL and the model checking approach. In both these systems proof construction has been mechanized and applied to reasonable sized problems. The underlying formalism of the these operational methods is not apparent to the user, i.e., they can be made user friendly, increasing the likelihood that they could and would be used by system designers.

We can combine the key advantages of both COL and the model checking approach into one system. The model checking approach provides an efficient means of verifying, in general, any correctness property with a single algorithm, however it requires a global state graph representation for the concurrent system under analysis. In the COL system, each temporal error that an Ada program was to be analysed for required a separate algorithm, each of which used a global state graph representation of the program as input. However, Karam also developed an efficient generator to produce a minimal global state graph from the tasks in an Ada program.

Since, in general, PAL and Ada have similar semantics, and we can consider the processes or tasks of both languages to be based on a similar execution model, we

can employ Karam's approach to generate a global state graph representation for the PAL expressions. We can then use a model checking algorithm to verify this global state graph against a specification expressed in BTL.

Therefore, our approach to verifying the correctness of PAL process expressions will be as follows:

1. Transform the PAL process expressions into an equivalent global state graph representation. To transform the PAL expressions into a state graph model we will employ a version of COL modified to handle the semantics of PAL.

2. Verify that this global state graph model of the system is correct with respect to a set of correctness properties. We will develop a verification system based on the model checking algorithm to check that a set of correctness properties expressed in BTL are satisfied on the model.

Chapter 4

State Graph Generation

In this chapter we develop a method to generate a global state graph from a set of PAL expressions. The method described here closely follows a method developed by G.M. Karam to check for deadlock in an Ada program [Kar87]. We adapt his method to handle the unique features of PAL with the result that we can produce a global state graph from a set of PAL expressions and at the same time check for system-wide deadlock. We begin with a detailed overview of Karam's method.

4.1 CAEDE Operational Language (COL)

In this section we provide a detailed overview of COL, Karam's temporal logic based operational language, which we have already briefly looked at in Section 3.4. Our aim in examining the method in more detail here is to gain a better idea of what must be done to adapt COL for our own needs. This section is based on material from [Kar87].

COL provides a complete LTL representation for an Ada program which because of its basis in a logical formalism provides both an axiomatic and an operational view of the system. The LTL formulae representing the system can be manipulated in such a way as to infer a sequence of program states. These series of program states can

be viewed as the different possible behaviours of the system over time. We begin by taking a more detailed look at the elements of COL's LTL representation system.

4.1.1 Connective Implications

In COL the behaviour of an Ada program is described by a set of LTL clauses of the form:

$$t_0 \models predicate_1(\ldots) \wedge ts(t_1, \pi_1(v_1), q) \Rightarrow \bigcirc \Diamond predicate_2(\ldots) \wedge ts(t_1, \pi_2(v_2), q) \quad (4.1)$$

Each of the clauses, known as a *connective implication*, describes allowable state transitions of the task. From each task state there is a connective implication which defines a transition to a next task state.

The notation $predicate_i(\ldots)$ represents the control state of the task and $ts(\ldots)$ the state of the task's variables. $\pi_i(v_i)$ is an array of task variables and q is a queue containing the states of callers making a rendezvous with that task. $Predicate_1$ is the present state of task t_1 and $predicate_2$ is a possible next state. The connective implication says that if in the present state, s_i, the conditions of $predicate_1$ and the task state ts are valid then in the next state, s_{i+1}, eventually $predicate_2$ will be true and the task state will be such that the variable array $\pi_2(v_2)$ is true and the caller queue remains unchanged.

The domain of validity, t_0, for the above formula indicates that each connective implication is valid only in the present state. That is, a connective implication allows system transitions from the present state of the system. For the remainder of this thesis the domain of validity for formulae will be t_0 unless otherwise indicated.

The elements of the task state $\pi_1(v_1)$ and $\pi_2(v_2)$ represent respectively *pre-* and *post-conditions* of the variables of the task. For the implication to be valid the state of the task must be such that the pre-conditions are satisfied in the current task state, s_i. If the pre-conditions are satisfied then in the next state, s_{i+1}, eventually the task

state must be such that the post-conditions are satisfied. For example:

$$\{var_1(x), var_2(y)\}$$

as an array of variables in the antecedent of the implication, i.e., as pre-conditions, would be interpreted as:

"if in the present state the value of var_1 is x and the value of var_2 is y,"

and if used as an array of post-conditions:

"in the next state eventually the value of var_1 must be x and the value of var_2 must be y."

Relationships among task variables can be defined by including special *supplementary* predicates, prefixed by a "@", in the task's array of variables. For example:

$$\{var_1(x), var_2(y), @equal(x, y)\}$$

as a pre-condition would be interpreted as:

"if in the present state the value of var_1 is x and the value of var_2 is y and x is equal to y."

The use of supplementary predicates in post-conditions allows setting of variable values as functions of other variables. For example:

$$\{var_1(x), var_2(y), @plus(x, 1, y)\}$$

as a post-condition means:

"in the next state eventually the value of var_1 must be x and the value of var_2 must be y such that y is x plus 1."

The use of pre- and post-conditions allow the description of alternative transition paths from one state to another, i.e., we can represent possible *if-then-else*, *case* or *looping* constructs through the use of the pre- and post-conditions. For example, the behaviour of a program branching point such as:

IF $var_1 < 3$ THEN

$var_1 := var_1 + 1;$

$\vdots$

ELSE

$var_1 := var_1 - 2;$

$\vdots$

END IF;

could be described by the following connective implications:

$$(1) \models predicate_1(\ldots) \wedge ts(t_1, \{var_1(x), @less(x,3)\}, q) \Rightarrow \bigcirc\Diamond predicate_2(\ldots) \wedge ts(t_1, \{@plus(x,1,newx), var_1(newx)\}, q)$$

$$(2) \models predicate_1(\ldots) \wedge ts(t_1, \{var_1(x), @geq(x,3)\}, q) \Rightarrow \bigcirc\Diamond predicate_3(\ldots) \wedge ts(t_1, \{@minus(x,2,newx), var_1(newx)\}, q)$$

The meaning of these implications can be interpreted as:

> "if in the present state the value of var_1 is some x and x is less than 3 then the task will eventually progress to a new state $predicate_2$ and the value of x will be incremented by 1 (Connective implication (1)). If, however, x is not less than 3 the task will eventually progress to a different new state $predicate_3$ and the value of x will be decremented by 2 (Connective implication (2))."

Thus the pre-conditions in effect turn on or off the connective implications so that only one is valid from each state. This allows the behaviour of various language control structures to be described. An important point in understanding the relationship of these implications to an Ada program is that these implications only describe all possible state transitions or behaviours of a task—they do not of themselves define

an execution ordering for the task in the same manner that Ada code would. While the implications can be arranged as above in some order suggesting a sequencing of operations they are really just a set of independent LTL formulae.

4.1.2 Ada Language Implications

In addition to the connective implications which describe allowable state transitions Karam also developed a set of *Ada language implications*. These implications describe the behaviour of a task in certain states involving the rendezvous and procedure calls. The four language implications describe exactly the semantics of the rendezvous call, rendezvous exit, procedure calls and procedure returns. The language implications are shown in Figure 4.1.

The rendezvous entry implication (ren) can be interpreted as:

> "if from a state sequence beginning with the present state eventually a task t_1 makes an entry call at entry y and eventually another task t_2 accepts a call at y then in the next state eventually the rendezvous will be started by t_2 indicating that the call was accepted for t_1."

The accepting task's caller queue contains the calling task's state $rdz(\ldots)$ which describes how a caller is held by the acceptor until the end of the rendezvous. In a similar manner (rex) describes how a rendezvous is completed. The implications (pen) and (pex) describe respectively the behaviour of a procedure call and return.

Thus, for intertask communication the connective implications describe how a task may reach a state where it will become involved in a rendezvous and then the language implications describe the behaviour of the partners within the rendezvous. The language implications describe the behaviour of the task and its rendezvous partner from the beginning of the rendezvous through to its eventual completion.

$$\left\{ \begin{array}{l} \Diamond(task\_call(t_1, x, y) \wedge ts(t_1, \{prm(p),,\}, q_1)) \wedge \\ \Diamond(start\_accept(t_2, y) \wedge ts(t_2, \{prm(q),,\}, q_2)) \end{array} \right\} \Rightarrow$$

(ren) $\models$

$$\left\{ \begin{array}{l} \bigcirc\Diamond\left(\begin{array}{l} start\_rendez(t_2, x, y) \wedge \\ ts(t_2, \{prm(p),,\}, [rdz(t_1, \{prm(p),,\}, q_1) \mid q_2]) \end{array} \right) \\ \wedge \bigcirc\Diamond(call\_accepted(t_1, x, y) \wedge ts(t_1, \{prm(p),,\}, q_1)) \end{array} \right\}$$

(rex) $\models$

$$\left\{ \begin{array}{l} \Diamond\left(\begin{array}{l} end\_rendezvous(t_2, y) \wedge \\ ts(t_2, \{prm(q),,\}, [rdz(t_1, x, y, \{prm(p),,\}, q_1) \mid q_2]) \end{array} \right) \\ \\ \wedge\Diamond(call\_accepted(t_1, x, y) \wedge ts(t_1, \{prm(p),,\}, q_1)) \end{array} \right\}$$

$$\Rightarrow \left\{ \begin{array}{l} \bigcirc\Diamond(end\_accept(t_2, y) \wedge ts(t_2, \{prm(q),,\}, q_2)) \wedge \\ \bigcirc\Diamond(return\_task\_call(t_1, x, y) \wedge ts(t_1, \{prm(q),,\}, q_1)) \end{array} \right\}$$

(pen) $\models$ $proc\_call(t, c, p) \wedge ts(t, \{,,\}, q) \Rightarrow$
$\bigcirc\Diamond(start\_proc(t, p) \wedge ts(t, \{,,\}, [prc(c) \mid q]))$

(pex) $\models$ $end\_proc(t, p) \wedge ts(t, \{,,\}, [prc(c) \mid q]) \Rightarrow$
$\bigcirc\Diamond(return\_proc\_call(t, c, p)\, wedge\, ts(t, \{,,\}, q))$

Reproduced from [Kar87] page 81.

Figure 4.1: Ada Language Implications for COL

4.1.3 Initial State Assertion

The complete behaviour of an Ada program can be represented by the set of connective implications for each task, the Ada language implications and an assertion of the program state. A program state assertion (PSA) is of the form:

$$t_0 \models \left\{ \begin{array}{l} predicate(t_1, \ldots) \wedge ts(t_1, \pi_1(v_1), q_1)) \wedge \\ predicate(t_2, \ldots) \wedge ts(t_2, \pi_2(v_2), q_2)) \wedge \\ \vdots \\ predicate(t_n) \wedge ts(t_n, \pi_n(v_n), q_n)) \wedge \end{array} \right\}$$

A PSA is an assertion that each task is in a particular state with certain values for each of the variables in the array $\pi_i(v_i)$ and the state of the caller queue is q_i. Again, t_0 indicates that each PSA describes the state of the system in the present.

A special PSA, the initial-PSA whose general form is given below is an assertion that each task t_i eventually starts at the top of its endless loop structure with some initial variables values, $\pi_i(v_i)$. The rendezvous caller queue is empty as shown by the empty list [].

$$t_0 \models \left\{ \begin{array}{l} \Diamond(init(t_1) \wedge ts(t_1, \pi_1(v_1), [])) \wedge \\ \Diamond(init(t_2) \wedge ts(t_2, \pi_2(v_2), [])) \wedge \\ \vdots \\ \Diamond(init(t_n) \wedge ts(t_n, \pi_n(v_n), [])) \wedge \end{array} \right\}$$

4.1.4 Deducing a Tree of Program States

Using the above LTL representation system an Ada program can be analysed axiomatically by forming a proof from the formulae in the same manner as is done for classical logic formulae. This would allow a system developer to prove certain properties about the system. However such proofs are not easy to construct requiring specialized knowledge of temporal logic and experience in applying it in the construction of proofs. For this reason Karam oriented the development of the above representation system towards an operational interpretation.

In an operational sense the LTL formulae can be manipulated in such a way as to produce a sequence of PSA starting from the initial-PSA. Since each PSA is in effect a snapshot of the system at a particular point in its execution, we can view the sequence of PSA as being representative of the behaviour of the system over time.

To automatically produce this sequence of PSA Karam developed an inference engine which uses unification-based pattern matching[Kow74] to apply the LTL formulae to deduce PSA. To make inferences the engine uses the set of LTL inference rules below:

$$if \models \mathcal{P} \wedge \Diamond \mathcal{Q} \; and \models \mathcal{Q} \Rightarrow \bigcirc\Diamond \mathcal{R} \; then \models \mathcal{P} \wedge \bigcirc\Diamond \mathcal{R} \tag{4.2}$$

$$if \models \mathcal{P} \wedge \bigcirc\Diamond \mathcal{Q} \; and \models \mathcal{Q} \Rightarrow \bigcirc\Diamond \mathcal{R} \; then \models \mathcal{P} \wedge \bigcirc\Diamond \mathcal{R} \tag{4.3}$$

$$if \models \mathcal{P} \wedge \bigcirc \mathcal{Q} \; and \models \mathcal{Q} \Rightarrow \bigcirc\Diamond \mathcal{R} \; then \models \mathcal{P} \wedge \bigcirc\Diamond \mathcal{R} \tag{4.4}$$

These rules are similar to the standard *modus ponens* inference rule from first order predicate logic, i.e., $if \models \mathcal{P} \; and \models \mathcal{P} \Rightarrow \mathcal{Q} \; then \models \mathcal{Q}$, which allows the deduction of a conclusion $\mathcal{Q}$ given a premise $\mathcal{P}$. Given one PSA which represents the present state and either a language or connective implication the above inference rules allow the deduction of a new PSA. This new PSA represents the change in state of one task in the system. By inferring other PSA from this new PSA and repeating the process the inference engine can produce a sequence of PSA starting from the initial-PSA.

The inference process terminates when all possible PSA have been deduced. If a PSA is deduced from which no succeeding PSA ca be deduced then this PSA represents a system-wide deadlock state.

Since LTL is able to reason only with deterministic state sequences, each time a state is encountered that represents a non-deterministic branching point, i.e., the next state is non-deterministic, the inference engine enumerates all possible next states as deterministic alternatives. Karam has termed this process of enumerating all possible next states as *expansion*. Thus the sources of non-determinism in Ada, the *select* statement and the arrival order of callers at a single task entry, are represented

as a set of deterministic execution paths. This gives rise to the tree structure of PSA with the initial PSA as the tree root and each branching node a non-deterministic expansion point.

Since each PSA provides an instantaneous image of the system, a path on the state graph represents the execution of the system over time. Any system state where the next system state is indeterminate is represented as a branching state. A transition from one state to another represents the execution by one system process of one indivisible instruction. The next system state is the state after which the instruction has been executed.

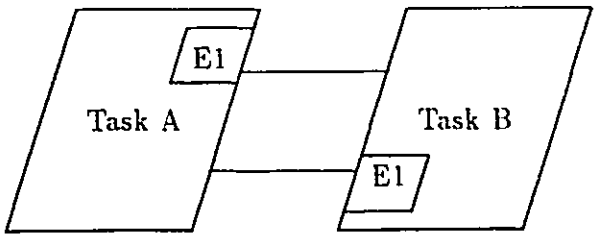
What must be understood about the branching nodes is that they represent deterministic execution alternatives, not conditional branches. That is, each alternative is equally possible, the branching node represents the fact that the choice as to which alternative is actually taken is non-deterministic.

4.1.5 Equivalent State Pruning

One of the benefits of using a temporal logic representation for system behaviour is that it has greater expressive power than normal finite state representations. That is, LTL formulae can express state sequences in fewer terms than an equivalent finite state representation. The following example has been adapted from [Kar87] with additional comments to present it in the context of this thesis.

Figure 4.2 shows a simple Ada program represented in structure diagram notation [Buh84] and Ada code segments for the bodies of the two tasks. Both tasks loop forever in the following manner: Task A calls entry `B.E1` and then accepts a call at `A.E1`; Task B accepts a call at `B.E1` and then calls `A.E1`. Both tasks loop forever alternately calling and accepting each other.

An equivalent communicating finite state machine (FSM) representation is shown in Figure 4.2(c). The behaviour of the Ada rendezvous has been modelled in the FSM by message passing where:



(a) Two Task Structure Diagram

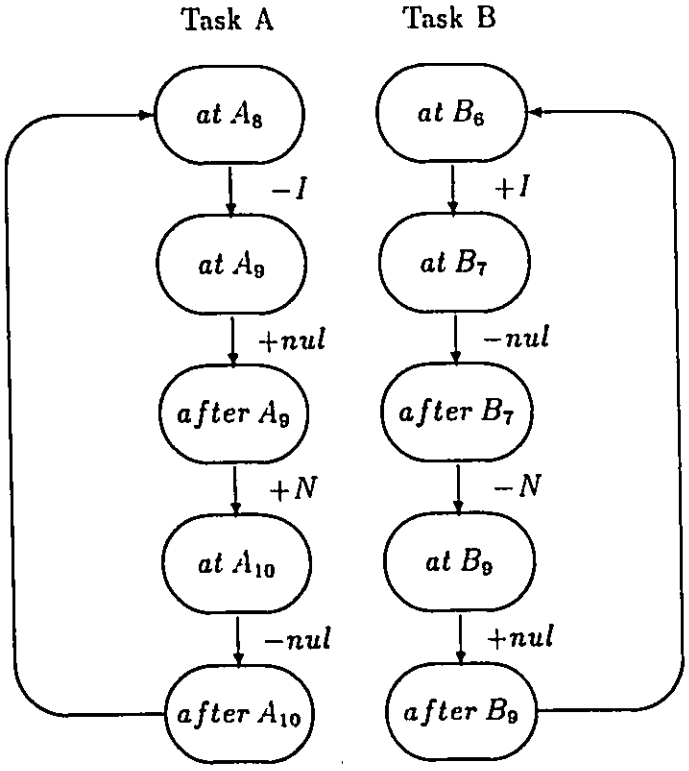
Task A

```
...
 8  loop
 9      B.E1(I);
10      accept E1(N : in INTEGER) do
11      end E1;
12      I := (I + 1) mod 10;
13  end loop;
...
```

Task B

```
...
 6  loop
 7      accept E1(N : in INTEGER) do
 8      end E1;
 9      A.E1;
10  end loop;
...
```

(b) Ada Code Fragment



(c) State Machine Representation

Figure 4.2: Ada Code Fragment and Structure Diagram and FSM

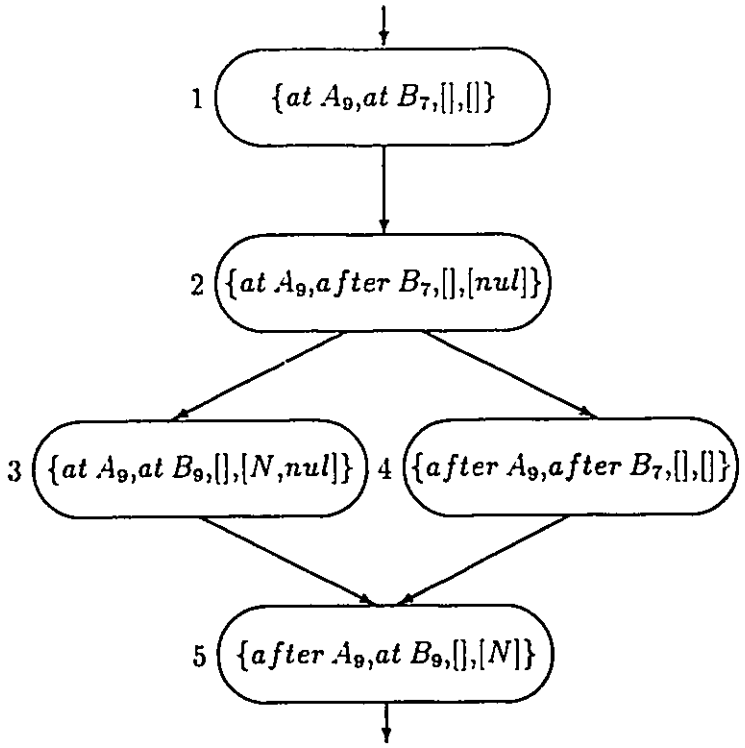


Figure 4.3: Program State Machine Segment

- a task call is defined as a message transmission followed by the reception of a 'nul' message, which represents the release of the caller from the rendezvous;
- an accept is a message reception followed by the transmission of a 'nul' message, which signifies the release of the caller.

In the FSM representation a message transmission is shown as '$-m$' (where m is the message) and a message reception as '$+m$.' For the FSM message transmissions are non-blocking and reception is blocking.

The notation $at\ X_i$ indicates that task X is presently ready to execute the Ada program statement at line i. Similarly $after\ X_i$ indicates that task X has just completed the execution of the Ada program statement at line i and is ready to execute the next line.

Figure 4.3 shows a segment of a global or program state machine derived from the two FSMs. Each state of the global state machine is a tuple composed of the individual states of the FSM's and the state of the two message channels between them. A transition in program states represents the change in state of one FSM. The program segment in Figure 4.3 shows a portion of the system behaviour starting from the beginning of the rendezvous on `B.E1` to the release of the calling task. In this program state machine segment there are two possible execution sequences: (1,2,3,5) and (1,2,4,5) resulting from the non-deterministic execution rates of the two FSM's. Both sequences give the same result since each task performs a deterministic sequence of operations. Both sequences can be represented by the same LTL formula:

$$\models at\, A_9 \wedge at\, B_7 \Rightarrow \bigcirc\Diamond after\, A_9 \wedge \bigcirc\Diamond after\, B_7 \tag{4.5}$$

This formula is interpreted as:

> "if in the present state it is true that task A's state is $at\, A_9$ and B's state is $at\, B_7$ then in the next state eventually task A must be $after\, A_9$ and in the next state eventually task B must be $after\, B_7$."

As is shown below this LTL formula is satisfied on both execution sequences:

(1,2,3,5) — $at\, A_9 \wedge at\, B_7$ is satisfied in state 1, so in the sequence starting from the next state, state 2, $after\, A_9$ must be true eventually and $after\, B_7$ must be true eventually; these predicates are satisfied in states 2 and 5 respectively;

(1,2,4,5) — the same initial conditions are satisfied in state 1 and starting from the next state eventually $after\, A_9$ and $after\, B_7$ are satisfied at states 2 and 4 respectively.

What this simple example demonstrates is the greater expressive power of the LTL representation versus a program state representation. In this example a single LTL formula expressed two computationally equivalent program state sequences.

In an operational sense this expressive power can be translated into an improved inference algorithm which can be used by the inference engine to generate the PSA tree. Since both program state sequences are equivalent only one is required to completely describe the program's behaviour. Therefore, the inference engine needs only to deduce PSA representing one of the paths. The actual execution path deduced by the COL inference engine represents the relative rate of execution of the tasks in the system. For example if the inference engine deduced a sequence of PSA which followed the (1,2,4,5) program sequence, then this would imply that task A is ready for another rendezvous before task B. The actual deduction algorithm employed by the inference engine is one in which all tasks are given an equal opportunity to progress.

4.2 Adapting COL for PAL

In adapting COL to a PAL based representation we will focus on the operational rather than the axiomatic aspects of the method. However, we will develop the necessary axiomatic representation system in so far as it will serve as a formal basis for the automated generation of a tree of system states.

To adapt this method for our use we must be able to develop PAL versions of the four elements of the COL axiomatic system: connective implications, language implications, initial state assertion and LTL inference axioms. As well we must develop an inference engine to apply the rules to generate the sequences of system states. To lessen the amount of development work we can see that:

1. The LTL inference rules, Formulae 4.2,4.3 and 4.4, are valid in general for LTL, therefore they will still be valid for any LTL representation system developed for the PAL language.

2. PAL control constructs are similar enough to Ada that the format of the connective implications and the form of the state assertions should require only minor changes.

The main part of the work will be to develop and show the validity of the PAL language implications describing PAL messaging semantics and to show that we can employ the same equivalent state pruning techniques as in [Kar87]. In particular we must:

1. Develop language implications for the PAL messaging semantics; we must show that a non-blocking send to a finite capacity channel can be described in axiomatic form;
2. Express system behaviour with respect to global variables;
3. Express PAL control constructs in connective implication form; and
4. We must show that the equivalent state sequence pruning method employed by the COL inference engine can also be applied for PAL.

In modifying COL to serve our needs we will follow a similar development approach to that taken in [Kar87]. We will begin by working with a subset of the PAL language consisting of only those PAL constructs which result in *process-deterministic* processes. Once we have developed our method for this subset of PAL we will extend it to include all PAL constructs.

4.3 Process Deterministic Development

Task-determinism is a term introduced in [Kar87] to describe a task that perceives the system to be deterministic, in that the task experiences a total ordering of its rendezvous. The system as a whole may experience many different rendezvous orderings due to the non-deterministic execution rates of the system tasks. However, the different system orderings do not affect the rendezvous ordering perceived by the individual task. Recall from Section 3.4.3.1 (page 36) that LTL can reason only with deterministic state sequences. We therefore require that a system be deterministic

in order to apply LTL formalisms. To be able to consider an Ada system as being deterministic Karam defined two conditions that it must meet:

1. Each task follows a deterministic rendezvous ordering, i.e., a call to `B.E1` is always followed by an accept at E3 followed by ..., etc;

2. The different total rendezvous orderings perceived by the system as a whole result in functionally equivalent system behaviour.

If it can be assumed that tasks use structures that result only in task-deterministic tasks then the first condition is satisfied. The second was shown to be true in Section 4.1.5 for the Ada rendezvous.

Since all of the different total system rendezvous orderings result in equivalent system behaviour then one particular rendezvous ordering is as representative of system behaviour as another. Thus if we chose one of the functionally equivalent rendezvous orderings with which to reason then the system can then be viewed as being deterministic.

We apply the term *process-determinism* here to describe the same concept in regards to processes[1] for the PAL environment. To enable us to provide an LTL representation for PAL expressions we must first ensure that the set of PAL expressions making up a system can be considered as being deterministic. Thus both of the above conditions must be satisfied in the PAL environment.

To ensure that that we will be working exclusively with process-deterministic processes we will assume for now that no process expression will use the following constructs:

- *select* — since by the definition of the construct the next state is indeterminate;

- operations involving *global variables* — the value of the global variable is unknown at the time it is read, e.g., we cannot determine whether process A read the value before or after process B modified the value.

[1] For our purposes the terms *task* and *process* are essentially interchangeable.

	Process A		Process B
1.	**A** ::=	1.	**B** ::=
2.	$\mathcal{S}_{on}$	2.	$\mathcal{S}_{on}$;
3.	**send**($chan1, req$);	3.	**receive**($chan1, req$);
4.	**receive**($chan2, ack$);	4.	**send**($chan2, ack$);
5.	**do_activity_a**;	5.	**do_activity_b**;
6.	**receive**($chan3, req$);	6.	**send**($chan3, req$);
7.	**send**($chan4, ack$);	7.	**receive**($chan4, ack$);
8.	$\mathcal{E}$	8.	**do activity c**;
		9.	$\mathcal{E}$

This example depicts two processes, **A** and **B**, who alternately send and receive messages over four channels. Both processes will loop forever due to assumption 1, Section 2.3, concerning the continuously enabled starting events $\mathcal{S}_i$.

Figure 4.4: PAL expressions for the two process example

The above restrictions will be lifted in Section 4.5. For now these restrictions will ensure all PAL expressions are process-deterministic.

If the different total orderings of channel operations between processes result in functionally equivalent system behaviour then we can chose one of the total orderings to describe the behaviour of the system. For now we will assume that all system orderings of channel operations will result in functionally equivalent behaviour. We will show that this assumption is valid in Section 4.4.1. Given the above assumption and restricting PAL expressions to process-deterministic constructs, we can state that any given set of PAL expressions will model a completely deterministic system.

4.3.1 PAL Language Implications

State Predicates

To enable us to describe PAL expressions in implication form we must first define some basic predicates with which to work. The following predicates are defined for PAL. Usage examples are taken from the two process system example of Figure 4.4.

receive(p,ch) — This predicate is true if process p is in a state such that it is receiving a message from channel *ch*. The receiving location in the process is uniquely

identified by the channel name *ch*. For example to express the state *at* A_4 we can write *receive(A,chan2)*. We can tell that *receive(A,chan2)* uniquely corresponds to *at* A_4 rather than *at* A_6 because of the channel name.

send(p,ch) — This predicate is true if process *p* is in a state such that it is sending a message to channel *ch*. The sending point in the process is identified by the unique channel name *ch*. For example *send(B,chan2)* describes the state *at* B_4.

end_recv(p,ch) — The predicate *end_recv* is true if a receiving process has completed a receive from channel *ch*. The receiving point in the process is identified by the unique channel name *ch*. For example *end_recv* can be used to express the state $after\ B_3$ as *end_recv(B,chan1)*.

end_send(p,ch) — The predicate *end_send* is true if a sending process has completed a send to channel *ch*. The sending point in the process is identified by the unique channel name *ch*. For example *end_send* can be used to express the state $after\ A_7$ as *end_send(A,chan4)*.

ps(p,prms(ϕ),π(v)) — The predicate *ps(...)* represents the variable state of process *p*. *prms(ϕ)* is a standard variable whose value is equal to the message to be sent for a **send** instruction or to the message received for a **receive** instruction. $\pi(v)$ is an array of process condition variables. The *ps(...)* predicate is true if the array of process variable values *v* satisfies the predicate $\pi(v)$ and if the process is in either the **send** or **receive** state then ϕ represents the value of the message. For example for the send *at* A_3 the value of ϕ will be *req* and at the state $after\ B_7$ the value of ϕ will be *ack*.

chan(ch,q,n,m) — The *chan* predicate is true if there exists a channel *ch* with a queue of messages *q*, and the number of messages in the queue is *n*. The capacity of the channel is *m*. For example if the state of the channel *chan2* at state *at* B_4 was *chan(chan2,[],0,1)* then at the state $after\ B_4$ the channel state would be

chan(chan2,[ack],1,1).

init(p) — The *init()* predicate is true if the process p is at the top of its endless loop structure. For PAL the top of a process expression's endless loop structure is the S_{on} statement. For example *init(A)* is true if process A is *at* A_2.

PAL Messaging Semantics

The semantics of the PAL **send** and **receive** primitives defining operations on message channels are much simpler than the Ada rendezvous mechanism. For example:

1. Only one process may send on a specific channel and only one (different) process may receive on that channel. This means that we do not have to deal with a queue of callers to a channel. In Ada many callers may queue on a single task accept entry. The unknown arrival order of the callers to the accept entry is a source of non-determinism in the Ada messaging semantics.

2. A send or receive operation is simply the dropping-off or picking-up of a message. In the Ada rendezvous a caller task is held by the acceptor task until the acceptor decides to release the caller. The acceptor may in fact accept and release other callers in a nested fashion before finally releasing the original caller. Since this concept does not exist in PAL the semantics of the **send** or **receive** instructions are less complex than their Ada counterparts.

Using the basic predicates defined above we can construct compound predicates to partially express the behaviour of the PAL **send** and **receive** operations.

$$send(p,ch)\,\mathcal{U}\left\{\begin{array}{l} send(p,ch)\wedge \\ chan(ch,q,n,m)\ \wedge\ n<m \end{array}\right\} \vee\ \Box send(p,ch) \qquad (4.6)$$
$$\Leftrightarrow\ send\_msg(p,ch)$$

Formula 4.6 describes the behaviour of the **send** primitive. It is interpreted as:

"if we enter the send state then we will remain there until the channel ch is not full, $(n < m)$, otherwise we will henceforth remain in the send state."

$$receive(p, ch)\,\mathcal{U}\left\{\begin{array}{l} receive(p, ch) \wedge \\ chan(ch, q, n, m) \wedge n > 0 \end{array}\right\} \vee \Box receive(p, ch) \qquad (4.7)$$
$$\Leftrightarrow recv\ msg(p, ch)$$

Similarly Formula 4.7 describes the behaviour of the **receive** primitive. It is interpreted as:

"if we enter the receive state then we will remain there until there is at least one message in the channel queue $(n > 0)$, otherwise we will henceforth remain in the receive state."

Informally we can see that both the above formulae capture the behaviour of the PAL messaging primitives. The **send** primitive expressed in LTL form in Formula 4.6 exactly describes the desired operation of the primitive. If the channel is full then the process will remain in the send state until it is not full. i.e., the non-blocking send becomes a blocking send when the capacity of the channel is reached. Similarly a process is blocked on a **receive** primitive until a message is available from the channel.

For ease of notation we have defined the behaviour for the **receive** primitive to be equivalent to *recv_msg(p,ch)* and the **send** primitive to be equivalent to *send_msg(p,ch)*.

Operations on Channels

Given the above compound formulae we can now describe how the **send** and **receive** primitives affect the state of the channel and the state of the sending and receiving processes.

For a **send** instruction to reach completion the channel must not be full and upon completion of the **send** the channel must contain the message that is being sent and the channel's message count incremented. If the channel is full then the process issuing the **send** must be blocked until the channel is not full This behaviour is expressed in Formula 4.8.

$$t_0 \models \begin{array}{l} \Diamond(send\_msg(p,ch) \wedge ps(p,prms([msg]),\{\ldots\})) \wedge \\ \Diamond(chan(ch,q,n,m) \wedge n < m) \\ \Rightarrow \left\{ \begin{array}{l} \bigcirc\Diamond(end\_send(p,ch) \wedge ps(p,prms([]),\{\ldots\})) \\ \wedge \bigcirc \Diamond chan(ch,[q\|msg],n+1,m) \end{array} \right\} \end{array} \tag{4.8}$$

Formula 4.8 says that if eventually a process p begins a send to channel ch and the state of channel ch is eventually such that the number of messages in its queue is less than the channel's capacity, then in the next state eventually the sending process will exit the send state and the channel will now have the message in its message queue. The notation $[q\|msg]$ represents the channel as a FIFO queue with the sent message msg being the end, or last message in the queue, and q being the rest of the queue.

A **receive** from a channel will complete if the channel contains at least one message. Upon completion of the **receive** the message at the head of the channel should be transferred to the receiving process, the channel message count decremented and the message removed from the channel. If the channel is empty then the receiving process must wait until a message is available. This behaviour is expressed in Formula 4.9.

$$t_0 \models \begin{array}{l} \Diamond(recv\_msg(p,ch) \wedge ps(p,prms([]),\{\ldots\})) \wedge \\ \Diamond(chan(ch,[msg|q],n,m) \wedge n > 0) \\ \Rightarrow \left\{ \begin{array}{l} \bigcirc\Diamond(end\_recv(p,ch) \wedge ps(p,prms([msg]),\{\ldots\})) \\ \wedge \bigcirc \Diamond chan(ch,q,n-1,m) \end{array} \right\} \end{array} \tag{4.9}$$

Formula 4.9 says that if eventually a process p begins to receive a message on

channel *ch* and eventually the state of the channel is such that there is a message available, then in the next state eventually the process will exit the receive state with a copy of the message *msg* and eventually there will be one less message in the channel. The notation $[msg|q]$ represents the FIFO channel as a queue; *msg* being the head, or the first message in the channel, and q being the rest of the queue.

4.3.2 Connective Implications

In this section we define the second part of the PAL temporal logic representation system—the connective implication. Connective implications describe all allowable state transitions of a process as a set of temporal logic formulae in implication form. A suitable representation must:

- be able to model the various control constructs available in the language; and
- show the effects on control variables resulting from process behaviour.

We will use the following general form of connective implication:

$$t_0 \models predicate_1(\ldots) \wedge ps(p_1, prms(\phi), \pi_1(v)) \Rightarrow \bigcirc \Diamond(predicate_2(\ldots) \wedge ps(p_1, prms(\phi), \pi_2(v))) \tag{4.10}$$

which is based on the form used in COL (Formula 4.1) with minor changes to better represent PAL semantics. The caller queue q has been removed since PAL messaging semantics have no such concept. As well we have made the message parameter, $prms(\phi)$, an explicit part of the task state whereas in COL the message was part of the variable array. Each $predicate_i$ is a term asserting the control state of the process where $predicate_i$ can assume any of the basic state predicates defined earlier.

Connective implications for the processes of Figure 4.4 are shown in Figure 4.5. We can see how the connective implications describe the allowable state transitions of the two PAL expressions. The PAL expression for process A begins with the starting event S_{on} then sends a message to *chan*1. This behaviour is represented in connective

$$(a1)\quad init(A) \wedge ps(A, prms([]), \{\}) \Rightarrow \bigcirc\Diamond(send\ msg(A, chan1) \wedge ps(A, prms([req]), \{\}))$$

$$(a2)\quad end\_send(A, chan1) \wedge ps(A, prms([]), \{\}) \Rightarrow \bigcirc\Diamond(recv\_msg(A, chan2) \wedge ps(A, prms([]), \{\}))$$

$$(a3)\quad end\_recv(A, chan2) \wedge ps(A, prms([ack]), \{\}) \Rightarrow \bigcirc\Diamond(recv\_msg(A, chan3) \wedge ps(A, prms([]), \{\}))$$

$$(a4)\quad end\_recv(A, chan3) \wedge ps(A, prms([req]), \{\}) \Rightarrow \bigcirc\Diamond(send\ msg(A, chan4) \wedge ps(A, prms([ack]), \{\}))$$

$$(a5)\quad end\_send(A, chan4) \wedge ps(A, prms([]), \{\}) \Rightarrow \bigcirc\Diamond(init(A) \wedge ps(A, prms([]), \{\}))$$

Connective implications for process A

$$(b1)\quad init(B) \wedge ps(B, prms([]), \{\}) \Rightarrow \bigcirc\Diamond(recv\ msg(B, chan1) \wedge ps(B, prms([]), \{\}))$$

$$(b2)\quad end\_recv(B, chan1) \wedge ps(B, prms([req]), \{\}) \Rightarrow \bigcirc\Diamond(send\_msg(B, chan2) \wedge ps(B, prms([ack]), \{\}))$$

$$(b3)\quad end\_send(B, chan2) \wedge ps(B, prms([]), \{\}) \Rightarrow \bigcirc\Diamond(send\_msg(B, chan3) \wedge ps(B, prms([req]), \{\}))$$

$$(b4)\quad end\_send(B, chan3) \wedge ps(B, prms([]), \{\}) \Rightarrow \bigcirc\Diamond(recv\_msg(B, chan4) \wedge ps(B, prms([]), \{\}))$$

$$(b5)\quad end\_recv(B, chan4) \wedge ps(B, prms([ack]), \{\}) \Rightarrow \bigcirc\Diamond(init(B) \wedge ps(B, prms([]), \{\}))$$

Connective implications for process B

Figure 4.5: Connective Implications for two process example

implication (a1). Beginning from *init*, the initial state, connective implication (a1) describes a possible transition of process A to a *send_msg* state. On completion of a send to *chan*1 connective implication (a2) describes the transition of process A to a *recv msg* state. The remaining implications describe each possible transition of the process with the last implication describing how the process may return to the initial state.

Once again it is important to note that although the ordering of the connective implications in Figure 4.5 is highly suggestive of the operational behaviour of the process they are in fact just a set of LTL formulae describing all possible behaviours of the processes. That is, each implication stands on its own as an LTL formulae describing one particular aspect of the process behaviour. However, through the application of the inference rules 4.2, 4.3 and 4.4 from Section 4.1 we will (in Section 4.4) be able to use these implications to obtain a sequence of program states representing the operational behaviour of the process.

The notation $\pi_i(v_i)$ in Formula 4.10 represents the array of process variable values that are affected by the state transitions. As in COL the array of variable values represent *pre-conditions* when placed in the antecedent of the implication and *post-conditions* in the consequent of the implication.

To show the effects of relationships among variables we will use, as in COL, supplementary predicates prefixed with a "@" character. The supplementary predicates can be added to the array of process variables to show relationships amongst the variable values. For example

$$\{count(p), limit(q), @leq(p, q)\}$$

when used as a pre-condition means:

> "that the value of the variable *count* is some p and the value of *limit* is q and p must be less than or equal to q."

When used in a post-condition variable array the supplementary predicate allows us to specify new variable values as a function of the other variables. For example

$$\{count(p), @plus(p, 1, p1), newcount(p1)\}$$

can be interpreted as:

> "the value of *count* is p and the value of *newcount* is $p1$ such that $p1$ is equal to $p + 1$."

To show how these pre- and post-conditions allow us to describe the behaviour of the various PAL control constructs we present the following two examples. For each example we first give a PAL code fragment then the fragment represented in connective implication form and finally a brief explanatory paragraph.

Example 1

```
6   ...
7   send(chan1, ack);
8   case
9      count < 2 :
10        do_activity_2;
11        STP(count, count + 1);
12        receive(chan2, ack) |
13     count >= 2 :
14        do_activity_a;
15        STP(count, count − 1);
16        send(chan4, done)
17  endcase;
18  ...
```

$$(\mathrm{CI1}) \quad end\_send(A, chan1) \wedge ps(A, prms([]), \{count(x), @lss(x, 2)\}) \Rightarrow \bigcirc\Diamond \left(\begin{array}{l} recv\ msg(A, chan2) \wedge \\ ps(A, prms([]), \{@plus(x, 1, y), count(y)\}) \end{array} \right)$$

$$(\mathrm{CI2}) \quad end\_send(A, chan1) \wedge ps(A, prms([]), \{count(x), @geq(x, 2)\}) \Rightarrow \bigcirc\Diamond \left(\begin{array}{l} send\ msg(A, chan4) \wedge \\ ps(A, prms([done]), \{@minus(x, 1, y), count(y)\}) \end{array} \right)$$

This first example shows how the *case* construct is represented in connective implication form. The *case* construct in this example has two alternatives, $count < 2$ and $count >= 2$ with one connective implication clause describing the behaviour for each. The post-conditions in both alternatives describe the behaviour of the **STP** instruction.

We can also see from this example that the activities, **do_activity_2** and **do_activity_a**, are not represented in the connective implications. We assume (from Section 2.3) that the activities are called and eventually complete correctly.

Example 2

```
4   ...
5   receive(chan1, ack);
6   repeat
7         do_activity_3;
8         STP(count, count + 1)
9         send(chan2, req);
10  until count >= 10;
11  receive(chan5, done);
12  ...
```

$$\text{(CI3)}\quad end\_recv(A, chan1) \wedge ps(A, prms([ack]), \{count(x)\}) \Rightarrow \bigcirc\Diamond \left(\begin{array}{l} send\_msg(A, chan2) \wedge \\ ps(A, prms([req]), \{@plus(x, 1, y), count(y)\}) \end{array} \right)$$

$$\text{(CI4)}\quad end\_send(A, chan2) \wedge ps(A, prms([]), \{count(x), @geq(x, 10)\}) \Rightarrow \bigcirc\Diamond \left(\begin{array}{l} recv\_msg(A, chan5) \wedge \\ ps(A, prms([]), \{\}) \end{array} \right)$$

$$\text{(CI5)}\quad end\_send(A, chan2) \wedge ps(A, prms([]), \{count(x), @lss(x, 10)\}) \Rightarrow \bigcirc\Diamond \left(\begin{array}{l} send\_msg(A, chan2) \wedge \\ ps(A, prms([req]), \{@plus(x, 1, y), count(y)\}) \end{array} \right)$$

This example shows a repeat loop entered following the completion of a receive from *chan*1. Connective implication (CI3) describes the transition of process A from the end of the receive state to the *send_msg* on *chan*2. The pre-conditions of implications (CI4) and (CI5) describe the behaviour of the process within the repeat loop. Connective implication (CI4) describes the case where the process may exit the loop and (CI5) the case where the process returns to the top of the loop. For both (CI4) and (CI5) the pre-conditions represent the exit condition of the loop. Again we can see that the activities are not represented in the connective implications.

4.3.3 System State Assertions

Having developed the connective implications to describe allowable state transitions of a process and the PAL language implications to describe the behaviour of the PAL messaging semantics we have only to describe an appropriate system state representation to be able to axiomatically prove various system properties.

A system state assertion (SSA)[2] is a conjunct of the states of the individual processes and channels of the system. In operational terms the SSA represents the state of the system at a particular instant in time. The general form of an SSA in

[2]We refer to the PAL expressions as *system* specifications rather than as *program* statements therefore our SSA is essentially equivalent to Karam's PSA.

PAL is:

$$t_0 \models \left\{ \begin{array}{l} \left(\begin{array}{l} (predicate_1 \wedge ps(p_1, \pi_1(v_1))) \\ \wedge (predicate_2 \wedge ps(p_2, \pi_2(v_2))) \\ \vdots \\ \wedge (predicate_n \wedge ps(p_n, \pi_n(v_n))) \end{array} \right) \\ \\ \wedge \left(\begin{array}{l} chan(ch_1, q_1, n_1, m_1) \\ \wedge chan(ch_2, q_2, n_2, m_2) \\ \vdots \\ \wedge chan(ch_n, q_n, n_n, m_n) \end{array} \right) \end{array} \right\} \tag{4.11}$$

Where $predicate_i$ is the execution state of the process p and $\pi_i(v_i)$ is an array of process variable values. The $predicate_i$ can assume any one of the basic predicates defined in Section 4.3.1. Each process state and channel state term may be prefixed by either a $\Diamond$ or $\bigcirc\Diamond$ temporal operator.

The initial state of a system is asserted as a special SSA—the initial-SSA. For example to describe the initial state of the two process example of Figure 4.4 we can assert:

$$t_0 \models \left\{ \begin{array}{l} \Diamond(init(A) \wedge ps(A, \{\})) \wedge \\ \Diamond(init(B) \wedge ps(B, \{\})) \wedge \\ \Diamond chan(chan1, [], 0, 1) \wedge \Diamond chan(chan2, [], 0, 1) \wedge \\ \Diamond chan(chan3, [], 0, 1) \wedge \Diamond chan(chan4, [], 0, 1) \end{array} \right\} \tag{4.12}$$

If we wished to express the fact that at a certain instant process A was $after\ A_3$, process B was $at\ B_6$ and the appropriate channels contained messages then we could assert:

$$\models \left\{ \begin{array}{l} \bigcirc\Diamond(end\_send(A, chan1) \wedge ps(A, prms([]), \{\})) \\ \wedge \bigcirc\Diamond(recv\_msg(B, chan4) \wedge ps(B, prms([]), \{\})) \\ \wedge \bigcirc\Diamond chan(chan1, [], 0, 1) \wedge \bigcirc\Diamond chan(chan2, [ack], 1, 1) \\ \wedge \bigcirc\Diamond chan(chan3, [req], 1, 1) \wedge \bigcirc\Diamond chan(chan4, [], 0, 1) \end{array} \right\} \tag{4.13}$$

For the remainder of the thesis we will refer to the above temporal logic representation system consisting of the connective implications, PAL language implications and the initial-SSA as the PAL Temporal Representation or PTR.

4.4 Deducing a Tree of Program States

PTR gives us an axiomatic representation for PAL which with some experience can be manipulated to prove various things about a specific system. However our objective in pursuing the above development was to realize a means of producing a global state graph from a set of PAL expressions. We want to manipulate the PTR elements in such a way that we can produce a tree of system states representing the behaviour of the system. In this section we outline an inference engine similar to that developed in [Kar87] that will automatically apply inferences to produce a tree of program states.

Since an SSA is in effect a snapshot of the system at a particular instant in time we can view a sequence of successive SSA as representing the operational behaviour of the system over time. A transition from one system state to another represents the execution by the system of one indivisible operation. We show by way of example how an inference engine can apply the elements of PTR to generate a sequence of SSA. We restate the inference rules 4.2, 4.3 and 4.4 here for convenience.

$$if \models \mathcal{P} \wedge \Diamond \mathcal{Q} \; and \models \mathcal{Q} \Rightarrow \bigcirc\Diamond \mathcal{R} \; then \models \mathcal{P} \wedge \bigcirc\Diamond \mathcal{R} \quad (4.2)$$

$$if \models \mathcal{P} \wedge \bigcirc\Diamond \mathcal{Q} \; and \models \mathcal{Q} \Rightarrow \bigcirc\Diamond \mathcal{R} \; then \models \mathcal{P} \wedge \bigcirc\Diamond \mathcal{R} \quad (4.3)$$

$$if \models \mathcal{P} \wedge \bigcirc \mathcal{Q} \; and \models \mathcal{Q} \Rightarrow \bigcirc\Diamond \mathcal{R} \; then \models \mathcal{P} \wedge \bigcirc\Diamond \mathcal{R} \quad (4.4)$$

To make inferences the inference engine unifies the state of one process in the SSA with the antecedent of either a connective implication or a language implication. The consequent of the implication becomes the next state of the process in the new SSA. The new process may assume new variable values in the new state as a result of the application of the post-conditions of the implication. Thus each successive SSA

represents the change in state of only one of the processes. This inference process is illustrated for the two PAL expression example of Figure 4.4 (page 65). We show the change in state of process A from the initial state, Formula 4.14 below, to a next state, Formula 4.15.

We begin with the initial-SSA below:

$$\models \left\{ \begin{array}{l} \Diamond(init(A) \wedge ps(A, prms([]), \{\})) \\ \wedge \Diamond(init(B) \wedge ps(B, prms([]), \{\})) \\ \wedge \Diamond chan(chan1, [], 0, 1) \wedge \Diamond chan(chan2, [], 0, 1) \\ \wedge \Diamond chan(chan3, [], 0, 1) \wedge \Diamond chan(chan4, [], 0, 1) \end{array} \right\} \qquad (4.14)$$

and the connective implications of Figure 4.5 (page 71). From the initial-SSA we deduce a new SSA (Formula 4.15) by applying inference rule 4.2 and connective implication (a1). We show the deduction process in proof format below.

For ease of presentation we make the following substitutions:

(s1) $\mathcal{P}$ for $\left\{ \begin{array}{l} \wedge \Diamond(init(B) \wedge ps(B, prms([]), \{\ldots\})) \\ \wedge \Diamond chan(chan1, [], 0, 1) \wedge \Diamond chan(chan2, [], 0, 1) \\ \wedge \Diamond chan(chan3, [], 0, 1) \wedge \Diamond chan(chan4, [], 0, 1) \end{array} \right\}$ in (4.14)

(s2) $\mathcal{Q}$ for $init(A) \wedge ps(A, prms([]), \{\ldots\})$ in (4.14) & (a1)

(s3) $\mathcal{R}$ for $send\_msg(A, chan1) \wedge ps(A, prms([req]), \{\ldots\})$ in (a1)

A proof showing the transition of process A from the *init* state to the *send_msg* state

(1) $\Diamond\mathcal{Q} \wedge \mathcal{P}$ (4.14) [s1,s2]

(2) $\mathcal{Q} \Rightarrow \bigcirc\Diamond\mathcal{R}$ (a1) [s2,s3]

(3) $\mathcal{P} \wedge \bigcirc\Diamond\mathcal{R}$ 1,2,(4.2)

The format of the deduction is as follows: each line of the proof is given a numerical label as in (1); the second column is a formula or assertion and the third column gives

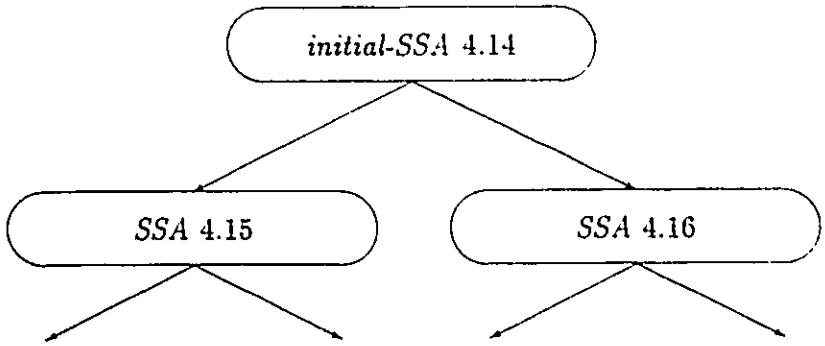


Figure 4.6: Beginnings of a State Graph

a basis for the validity of the formula or assertion from the second column. Applicable substitutions are given by name in brackets in the third column.

Line (3) of the proof with the substitutions removed is the new deduced SSA 4.15 below:

$$\models \left\{ \begin{array}{l} \bigcirc\Diamond(send\_msg(A, chan1) \wedge ps(A, prms([req]), \{\})) \\ \wedge\Diamond(init(B) \wedge ps(B, prms([]), \{\})) \\ \wedge\Diamond chan(chan1, [], 0, 1) \wedge \Diamond chan(chan2, [], 0, 1) \\ \wedge\Diamond chan(chan3, [], 0, 1) \wedge \Diamond chan(chan4, [], 0, 1) \end{array} \right\} \tag{4.15}$$

In a similar manner we can also deduce SSA 4.16 from the initial-SSA by making a different set of substitutions and applying connective implication (b1) instead of (a1).

$$\models \left\{ \begin{array}{l} \Diamond(init(A) \wedge ps(A, prms([]),)) \\ \wedge \bigcirc\Diamond(recv\_msg(B, chan1) \wedge ps(B, prms([]),)) \\ \wedge\Diamond chan(chan1, [], 0, 1) \wedge \Diamond chan(chan2, [], 0, 1) \\ \wedge\Diamond chan(chan3, [], 0, 1) \wedge \Diamond chan(chan4, [], 0, 1) \end{array} \right\} \tag{4.16}$$

From both of these two new SSA we can by applying appropriate substitutions derive other next state SSA from which we can deduce more SSA, etc. If each deduced SSA and a record of which current SSA it was deduced from is somehow recorded

```
1.  present state SSA queue Q = initial-SSA
2.  list of generated next states L = empty list
3.  start inference engine
4.  remove first SSA from Q
5.  infer all possible next states, next-SSA, from SSA
                and add to next-SSA set
6.  if the set next-SSA set is empty then
7.        goto 20
8.  else
9.        for each next-SSA in next-SSA set do
10.           begin
11.           if next-SSA is not a member of L then
12.                add next-SSA to L and to Q
13.           else
14.                if next-SSA is a member of L then
15.                     discard next-SSA
16.           end
17. if Q is empty then
18.       goto 21
19. else goto 4
20. system-wide deadlock detected
21. terminate inference engine
```

Figure 4.7: Basic Inference Algorithm

then this information will constitute a global state graph for the system. The SSA deduced from the initial-SSA and subsequent deductions to derive other SSA can be shown graphically as in Figure 4.6.

The algorithm for generating all possible next states of a system from an initial SSA is given in Figure 4.7

The list of generated next states $\mathcal{L}$ accumulates each new SSA that is generated by the inference process. Also recorded with each SSA is a list of all next state SSA deduced from that SSA. Thus, when the inference process terminates $\mathcal{L}$ will represent a global state graph for the system describing all possible system execution sequences.

In Appendix C we give a sequence of derivations using all of the elements of our temporal logic representation system that show a message being sent from one

process to another. In Chapter 6 we will discuss the implementation of an inference engine that will automatically deduce a tree of SSA from an initial-SSA and a set of implications.

4.4.0.1 Termination of the Inference Process

Given that we can now infer new SSA from existing SSA how do we know that eventually we will have inferred all the different possible states of the system? How do we know that in fact the number of possible SSA is finite?

Assuming that the data space of the system is finite then there exist only a finite number of possible states for the system. In a finite state system each variable of the system can only take on a finite set of values. Therefore, when generating SSA we must eventually reach a point where each new SSA has previously been deduced. The inference engine can be terminated at this point.

This action is represented in the algorithm of Figure 4.7. Lines 11–15 of the inference algorithm decide what action to take after deducing an SSA. If the SSA is not a duplicate it is recorded (line 12) otherwise it is discarded (line 15). Each unique SSA is also added to the present state queue Q from which next-SSA are deduced. If Q becomes empty then this implies that only duplicate SSA are being deduced therefore the inference process can be terminated.

4.4.1 Pruning Equivalent States

As was stated in Section 3.4 our motivation for employing Karam's method to generate a global state graph was that it would allow us to automatically generate the minimum number of SSA that would completely represent all possible system behaviours. The algorithm outlined in Figure 4.7 generates all possible next states from each current SSA, however many of the resulting state sequences, as was demonstrated for Ada in the example of Section 4.1.5, represent duplicate execution paths. What we will show in this section is that we can improve our inference algorithm by applying the

equivalent state pruning principles of Section 4.1.5 to the PAL environment. To be able to apply the equivalent state pruning method we must show that the different system execution orderings result in functionally equivalent behaviour.

If we re-examine the example of Section 4.1.5 we note that there are two conditions which allow the one temporal logic formula to express two computationally equivalent program state sequences:

1. The tasks contributing to the system state sequence were task-deterministic, therefore calls made to either task must eventually be accepted; and

2. Because of the semantics of the rendezvous we know that once a rendezvous is started, both partners will progress to a point where the rendezvous is finished.

Thus, the semantics of the Ada rendezvous guarantee that Formula 4.5 holds. That is, if a call is made by one task and it is eventually accepted by another task then both tasks must eventually progress to a point where the rendezvous is complete.

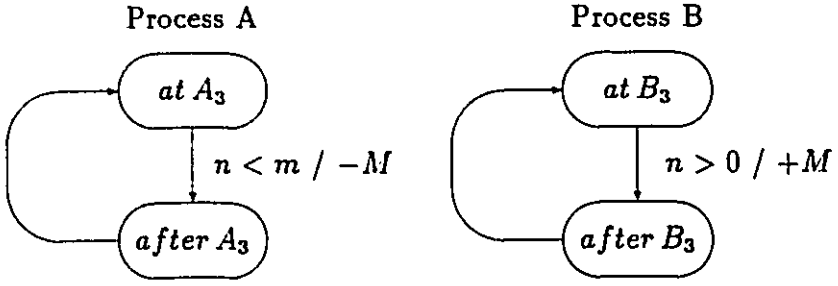
However, in the PAL environment where the messaging semantics place no conditions on the calling process to participate in the message reception there exists a possible state sequence along which messages are sent but never received. Consider the example processes of Figure 4.8. Given an infinite capacity channel, i.e., $m = \infty$, there exists a state sequence along which process A could repeatedly send messages while process B would remain in the receive state forever. Thus, a temporal formula similar to Formula 4.5 (page 61) asserting the eventual completion of a message transaction would not hold for all possible state sequences.

Fortunately, in PAL, message channels have a finite capacity. Therefore along any possible state sequence it must be the case that if a message is sent then it will eventually be received. We demonstrate this to be the case in the following example.

Referring again to Figure 4.8 we show two PAL expressions and their communicating FSM representation. Process A continually sends a message M on $chan1$ and process B continually receives messages from the same channel. We will assume that

Process A		Process B	
1.	$\mathcal{P}_A ::=$	1.	$\mathcal{P}_B ::=$
2.	$\mathcal{S}_{on}$	2.	$\mathcal{S}_{on}$
3.	**send**$(chan1, M)$;	3.	**receive**$(chan1, M)$;
4.	$\mathcal{E}$	4.	$\mathcal{E}$

PAL Expressions



FSM Representation

In the above state machine representation of the PAL expressions m is the capacity of the channel between the two processes and n is the number of messages in the channel.

Figure 4.8: PAL Equivalent State Pruning Example

the channel *chan*1 has been declared to have a finite capacity of two messages. In the FSM representation the transition condition for process A is $n < m$, where n is the number of messages in the channel queue and m is the capacity of the channel. Thus process A can only make a transition if the channel is not full. Similarly process B cannot make a transition unless a message is available, i.e., $n > 0$. A global state graph generated by following the inference algorithm of Figure 4.7 is shown in Figure 4.9.

All of the different execution paths in the state graph of Figure 4.9 can be represented by the single LTL formula below:

$$\Diamond at\, A_3 \wedge \Diamond at\, B_3 \Rightarrow \bigcirc\Diamond after\, A_3 \wedge \bigcirc\Diamond after\, B_3 \qquad (4.17)$$

which is valid at any of the 12 global system states. Thus we can chose any one of the states to represent the present state at time t_0 and Formula 4.17 will describe the behaviour of the system for all possible state sequences from t_0.

This example shows that in PAL, process interaction via message passing through finite capacity channels results in a number of different total system orderings. The different system orderings can be interpreted as describing the different relative rates of execution of the processes in the system. Each of these is, however, equivalent meaning that we can select one and reason with it using LTL. For example we could pick the state sequence $(3, 5, 8, 10, 3)$ to describe the behaviour of the system. In this state sequence process A executes faster relative to process B but the sequence is still functionally representative of system behaviour.

We can use the above knowledge to improve the inference algorithm of Figure 4.7. Instead of generating all possible new SSA from each current SSA we need only generate one new SSA. This indicates that we have somehow chosen one particular state sequence. The improved algorithm is shown in Figure 4.10. The particular state sequence that we will pick will be one such that all processes are given a equal opportunity to progress. The selection of which process to progress from a current SSA will be done by some selection algorithm θ as indicated in line 5 of the new

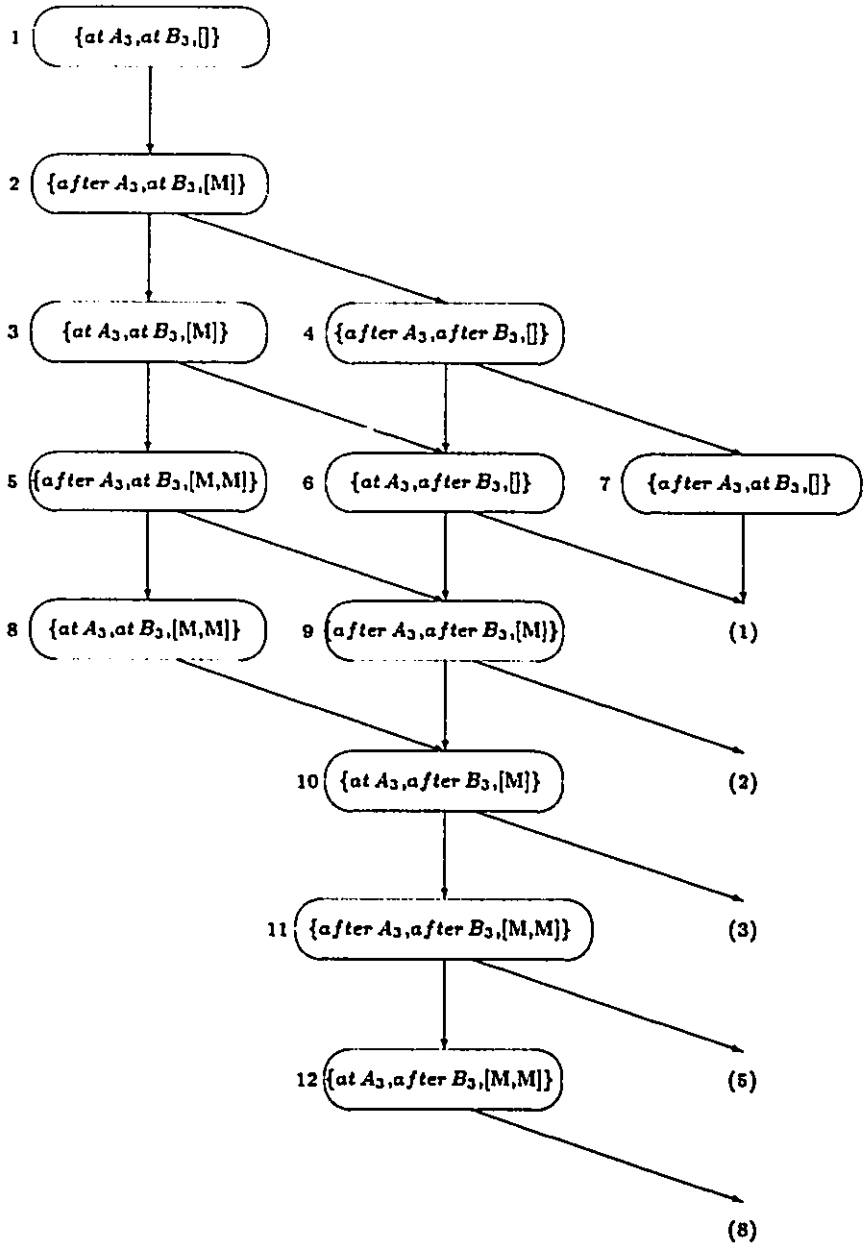


Figure 4.9: Global State Graph

```
1.  present state SSA queue Q = initial-SSA
2.  list of generated next states L = empty list
3.  start inference engine
4.  remove first SSA from Q
5.  select a process P to progress according to a
                selection algorithm θ
6.  infer next state, next-SSA, from P
7.  if cannot infer next-SSA then
8.        goto 18
9.  else
10.       if next-SSA is not a member of L then
11.             add next-SSA to L and to Q
12.       else
13.             if next-SSA is a member of L then
14.                   discard next-SSA
15. if Q is empty then
16.       goto 19
17. else goto 4
18. system-wide deadlock detected
19. terminate inference engine
```

Figure 4.10: Inference Algorithm with Pruning

inference algorithm.

By applying our improved inference algorithm to the PAL expressions of Figure 4.8 we can reduce the number of SSA that are needed to describe the complete behaviour of the system. Thus the global state graph of Figure 4.9 could be reduced to the global state graph of Figure 4.11. Of course the actual state sequence chosen to represent the system will depend on the selection algorithm θ.

4.5 Extension to Non-deterministic Processes

In this section, we extend the PAL Temporal Representation (PTR) to handle the non-deterministic aspects of PAL processes. The restrictions introduced in Section 4.3 on the use of the **select** construct and global variables are lifted to allow the building of any type of PAL expression.

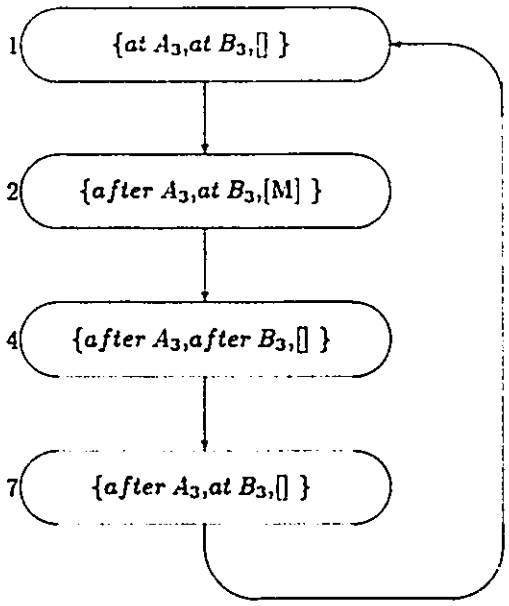


Figure 4.11: Pruned Global State Graph

We handle non-determinism by converting it into a set of deterministic alternatives. That is, at each state where the next state is non-deterministic we will consider each next state alternative as a separate deterministic continuation of the execution sequence. Thus each alternative in the set represents a separate process-deterministic process preserving our ability to reason with the program sequences using LTL. Figure 4.12 demonstrates this concept graphically.

Figure 4.12(a) shows a sequence of program states ending with one process having reached a non-deterministic construct. Figure 4.12(b) shows how each non-deterministic alternative is converted into a deterministic execution path. Thus the system can be viewed as being composed of n different deterministic execution sequences. We can represent the n sequences as one tree structure by combining the common state sequences as in Figure 4.12(c). We will refer to this process of enumerating deterministic alternatives at a non-deterministic branching point as *expansion*. This is the same term used in [Kar87].

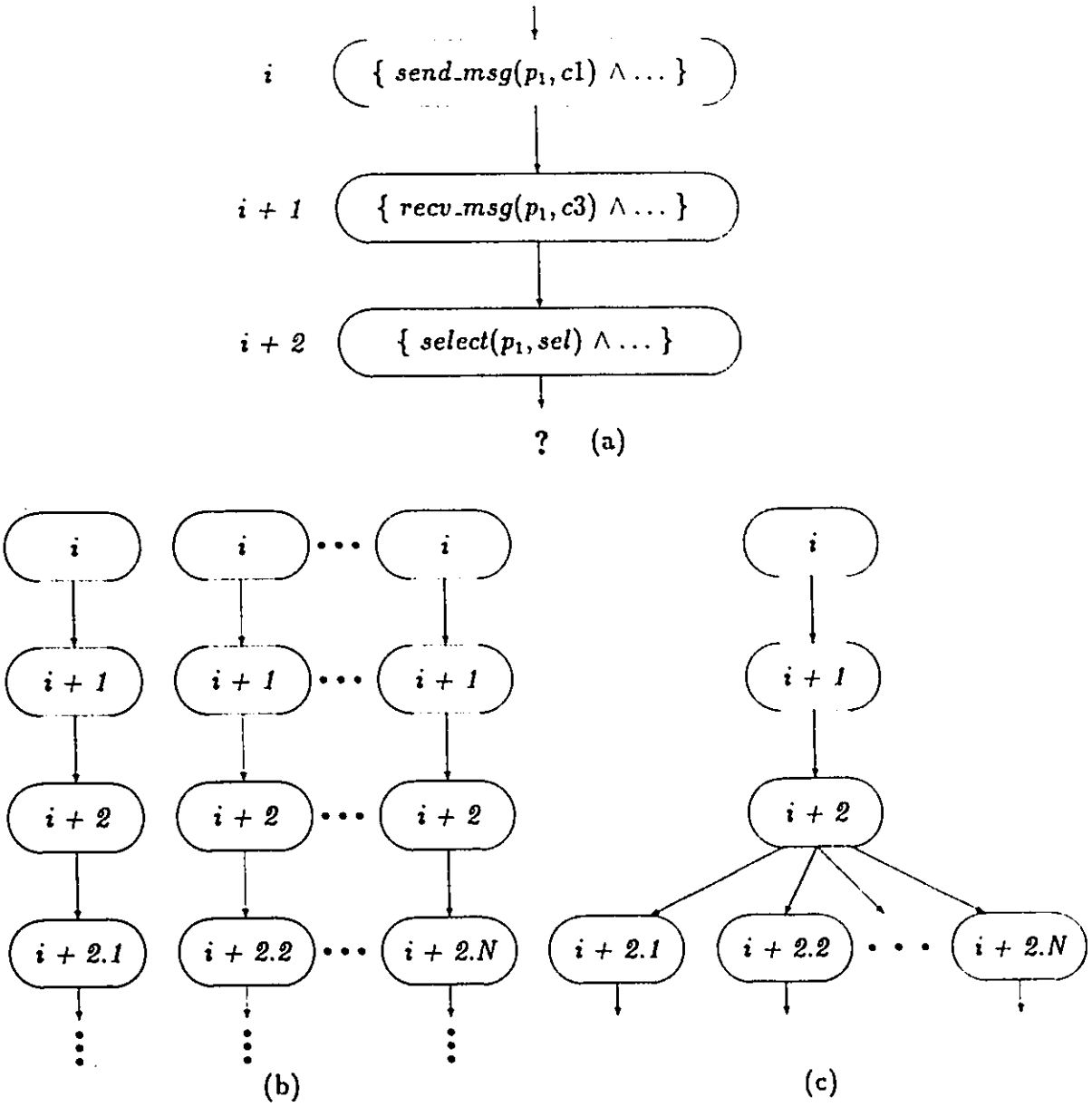


Figure 4.12: Enumerating Deterministic Alternatives

There are two sources of non-determinism in PAL semantics:

- the *select* construct — the execution sequence following a select construct is non-deterministic by the construct's own definition. Depending on the values of the process variables of which the construct's guards are composed and the execution state of the other processes there may be one or more possible execution alternatives. The open alternatives of each select construct must be expanded into a new SSA. Any alternative that is closed in the present state, i.e., the alternative is disabled, could not have been picked as a possible execution path and therefore does not need to be considered during expansion.

- the use of *global variables* — global variables can be read into a process or modified by a process at any time. We cannot assume a particular ordering of the writer and the reader processes which access the global variable because of their non-deterministic rates of execution, therefore we must account for all possible values that the global variable could have assumed at the time that it is read. Thus when a process executes either an **RDS** primitive or an **STS** primitive we must perform expansion to generate new SSA enumerating all the possible values that the global variable could have assumed at that point in the system execution.

To modify our generator to handle non-deterministic statements we must:

1. Modify the connective implications to allow the expression of the select construct and global variables;

2. Modify the inference algorithm to allow expansion.

Non-deterministic State Predicates

To begin the modifications we define the following predicates:

select(p,sel) — The *select* predicate is true if process *p* is ready to execute the **select** construct at the program location *sel*.

end select(p,sel) — *end select* is true if process *p* is at the end of the select construct at *sel*.

rds(p,at) — The *rds* predicate with two arguments is true if process *p* is ready to execute the **RDS** instruction at the program location *at*.

rds(p,at,sv,pv) — The *rds* predicate with four arguments is true if process *p* is reading the system variable *sv* at the process location *at*. The value of the process condition variable *pv* assumes the value of the system condition variable *sv*.

end_rds(p,at) — The *end_rds* predicate is true if process *p* has finished reading the system variable *sv* and the process variable *pv* has assumed the value of *sv*. The location of the **RDS** primitive in the process is indicated by the unique place name *at*.

sts(p,at) — The *sts* predicate with two arguments is true if process *p* is ready to execute the **STS** instruction at program location *at*.

sts(p,at,π(sv)) — The *sts* predicate with three arguments is true if in the present state process *p* is modifying the value of the system variable *sv*. The argument *π(sv)* is a structure similar to a post-condition array which allows the value to which *sv* will be modified to be determined as a function of the process variables. For example, *{pv(x),@plus(x,2,y),sv(y)}* would cause the system condition variable *sv* to be modified to a new value that is a function of the current value of process variable *pv*. The location of the **STS** primitive in the process is indicated by the unique place name *at*.

end_sts(p,at) — The predicate *end_sts* is true if process *p* has finished setting the system variable *sv*.

These predicates will be added to our vocabulary for constructing connective implications. As well we define one new form of connective implication below:

$$select(p, sel) ::= \left\{ \begin{array}{l} \psi_1(gv_1) \rightarrow \bigcirc\Diamond(predicate_1 \wedge ps(p_1, prms(\phi_1), \pi_1(v_1)) \\ \psi_2(gv_2) \rightarrow \bigcirc\Diamond(predicate_2 \wedge ps(p_2, prms(\phi_2), \pi_2(v_2)) \\ \vdots \\ \psi_n(gv_n) \rightarrow \bigcirc\Diamond(predicate_n \wedge ps(p_n, prms(\phi_n), \pi_n(v_n)) \end{array} \right\} \tag{4.18}$$

Each alternative of the **select** construct is defined as a possible consequent of the select connective implication. The guards $\psi_i(gv_i)$ are composed of process variables in the same manner as a pre-condition array of variable values. Supplementary predicates are allowed as part of a guard.

To handle global variables we modify the general form of the initial state assertion and SSA to add an array of system variables $\Pi(sv)$ as below:

$$\models \left\{ \begin{array}{l} \left(\begin{array}{l} \Diamond(init(p_1) \wedge ps(p_1, prms(\ldots), \pi_1(v_1))) \\ \wedge \Diamond(init(p_2) \wedge ps(p_2, prms([]), \pi_2(v_2))) \\ \wedge \ldots \end{array} \right) \wedge \\ \wedge \Pi(sv) \\ \wedge \left(\begin{array}{l} \Diamond chan(chan_1, q_1, n_1, m_1) \\ \wedge \Diamond chan(chan_2, q_2, n_2, m_2) \\ \wedge \ldots \end{array} \right) \end{array} \right\} \tag{4.19}$$

Initial SSA

The following example illustrates how the new predicates and connective implications work.

$$\models \left\{ \begin{array}{l} \bigcirc\Diamond(select(A, s1) \wedge ps(A, prms([]), \{pv_1(true), pv_2(5)\})) \wedge \\ \bigcirc\Diamond(recv\_msg(B, chan1) \wedge ps(B, prms([]), \{var_1(3)\})) \wedge \\ \bigcirc\Diamond(send\_msg(C, chan2) \wedge ps(C, prms([M]), \{\})) \wedge \\ \{sv_1(2), sv_2(false)\} \wedge \\ \Diamond chan(chan1, [], 0, 1) \wedge \\ \bigcirc\Diamond chan(chan2, [M, M], 2, 2) \wedge \\ \Diamond chan(chan3, [], 0, 4) \end{array} \right\} \quad (4.20)$$

Current SSA

$$select(A, s1) ::= \left\{ \begin{array}{l} \{pv_1(true)\} \rightarrow \left(\begin{array}{l} \bigcirc\Diamond(send\ msg(A, chan3) \\ \quad \wedge ps(A, prms([ack]), \{\})) \end{array} \right) \\ \\ \{pv_2(z), geq(z, 3)\} \rightarrow \left(\begin{array}{l} \bigcirc\Diamond(send\_msg(A, chan1) \\ \quad \wedge ps(A, prms([req]), \{\})) \end{array} \right) \end{array} \right\} \quad (4.21)$$

Connective Implication

Given the current SSA and connective implication above we can invoke expansion on the select construct in process A to infer the two new SSA (4.22) and (4.23) below. Two SSA are generated from the expansion since the guards of both select alternatives in the connective implication are open in the current SSA. i.e., in the current SSA 4.20 the process variable, pv_1, is *true* and the process variable, pv_2, has a value of 5, thus both guards are open. Because the alternative that would be chosen in an actual implementation is non-deterministic both alternatives must be enumerated.

$$\models \left\{ \begin{array}{l} \bigcirc\Diamond(send\ msg(A, chan3) \wedge ps(A, prms[ack], \{pv_1(true), pv_2(5)\})) \wedge \\ \bigcirc\Diamond(recv\_msg(B, chan1) \wedge ps(B, prms[], \{var1(3)\})) \wedge \\ \bigcirc\Diamond(send\_msg(C, chan2) \wedge ps(C, prms[M], \{\})) \wedge \\ \{sv_1(2), sv_2(false)\} \wedge \\ \Diamond chan(chan1, [], 0, 1) \wedge \\ \bigcirc\Diamond chan(chan2, [M, M], 2, 2) \wedge \\ \Diamond chan(chan3, [], 0, 4) \end{array} \right\} \tag{4.22}$$

$$\models \left\{ \begin{array}{l} \bigcirc\Diamond(send\_msg(A, chan1) \wedge ps(A, prms[req], \{pv_1(true), pv_2(5)\})) \wedge \\ \bigcirc\Diamond(recv\_msg(B, chan1) \wedge ps(B, prms[], \{var_1(3)\})) \wedge \\ \bigcirc\Diamond(send\_msg(C, chan2) \wedge ps(C, prms[M], \{\})) \wedge \\ \{sv_1(2), sv_2(false)\} \wedge \\ \Diamond chan(chan1, [], 0, 1) \wedge \\ \bigcirc\Diamond chan(chan2, [M, M], 2, 2) \wedge \\ \Diamond chan(chan3, [], 0, 4) \end{array} \right\} \tag{4.23}$$

To ensure that all possible alternatives are open at a select construct the selection algorithm θ must be set such that all processes are progressed as far as possible before invoking expansion. This will ensure that every alternative of the select construct has a chance to be open. In the above example, expansion of the select construct was the only option since both of the other processes were blocked on a message channel. However, what if the current SSA had been:

$$\models \left\{ \begin{array}{l} \bigcirc\Diamond(select(A, s1) \wedge ps(A, prms[], \{pv_1(true), pv_2(5)\})) \wedge \\ \bigcirc\Diamond(recv\_msg(B, chan1) \wedge ps(B, prms[], \{var_1(3)\})) \wedge \\ \bigcirc\Diamond(send\_msg(C, chan2) \wedge ps(C, prms[M], \{\})) \wedge \\ \{sv_1(2), sv_2(false)\} \wedge \\ \bigcirc\Diamond chan(chan1, [req], 1, 1) \wedge \\ \bigcirc\Diamond chan(chan2, [M, M], 2, 2) \wedge \\ \Diamond chan(chan3, [], 0, 4) \end{array} \right\} \quad (4.24)$$

Now channel *chan*1, rather than being empty contains one message and is therefore full. In this case the inference engine would have had a choice as to whether to expand the select construct or to infer a new SSA by allowing process B to progress. If expansion was performed first then only one new execution path would be enumerated instead of two. The second select alternative (a *send_msg* to *chan*1) would be blocked because *chan*1 is full to capacity. If instead we inferred a new SSA first, i.e., allowed process B to receive a message from *chan*1, then both select alternatives would again be open during expansion.

When enumerating possible alternative execution paths we must consider the fact that the SSA from which expansion will occur was reached by following one of possibly many equivalent total system orderings. For example, process X of a certain system could have reached a **select** construct expansion point by following one of possibly many different total system orderings. Suppose that we were able to examine the system states at which process X was at a **select** construct for each of the different total orderings. For each of these system states process X would be at the **select** construct but each of the other processes in the system might be in any other control state, e.g., *send_msg*, *recv_msg*, etc. The differing process states result from the fact that each different total system ordering assumes different relative rates of execution for the system processes. Recall that for the **select** construct, whether or not a select alternative is open is determined indirectly by the execution state of the other

processes. Therefore it is possible that for each different system ordering where the expansion point is reached we would enumerate a different set of alternative execution paths. This is the case with SSA 4.24 above. By invoking expansion directly from SSA 4.24 only one alternative execution path would be enumerated. If a different total system ordering was followed, i.e., the ordering where process B receives a message from *chan*1 first and then process A arrives at the **select** construct, then two alternative execution paths would be enumerated during expansion.

To ensure that all possible alternative execution paths are enumerated during expansion we must ensure that all of the possible alternatives that would be open for each of the different system orderings are included. One system ordering that ensures that all possible alternatives are open is the ordering where all processes, except for the process at the non-deterministic construct, have progressed a far as possible, i.e., to a point where they must wait for some event. At the SSA where the one process is at an expansion point and all other processes have progressed as far as possible we can apply expansion. Similarly for global variables we must apply expansion at a point where we can enumerate all the possible values that the global variable could have assumed at that point in its execution.

Thus, a modified expansion algorithm to ensure that all possible alternatives are enumerated is as follows:

1. Infer new system states by following the normal inference algorithm until a state is inferred in which a process is at a non-deterministic construct (**select**,**RDS** or **STS**), then

2. Deduce new system states representing the progress of all processes until all processes have reached either a non-deterministic construct, (**select**, **RDS** or **STS**) or a message wait state (a **receive** on an empty channel or a **send** to a full channel).

3. Apply expansion for each non-deterministic alternative represented by this state.

$$(c1)\quad end\_recv(B, chan1) \wedge ps(B, prms([req]), \{\}) \Rightarrow \bigcirc\Diamond(sts(B, s1) \wedge ps(B, prms([]), \{\}))$$

$$(c2)\quad sts(B, s1) \wedge ps(B, prms([]), \{\}) \Rightarrow \bigcirc\Diamond(sts(B, s1, \{var_1(x), sv_1(x)\}) \wedge ps(B, prms([]), \{\}))$$

$$(c3)\quad sts(B, s1, \{var_1(x), sv_1(x)\}) \wedge ps(B, prms([]), \{var_1(x)\}) \Rightarrow \bigcirc\Diamond(end\_sts(B, s1) \wedge ps(B, prms([]), \{@(x1 = x + 1), var_1(x1)\}))$$

$$(c4)\quad end\_sts(B, s1) \wedge ps(B, prms([]), \{\}) \Rightarrow \bigcirc\Diamond(recv\_msg(B, chan2) \wedge ps(B, prms([]), \{\}))$$

$$(c5)\quad rds(A, r1) \wedge ps(A, prms([]), \{\}) \Rightarrow \bigcirc\Diamond(rds(A, r1, \{sv_1(y), pv_1(y)\}) \wedge ps(A, prms([]), \{\}))$$

$$(c6)\quad rds(A, r1, sv_1(y), pv_1(y)) \wedge ps(A, prms([]), \{\}) \Rightarrow \bigcirc\Diamond(end\_rds(A, r1) \wedge ps(A, prms([]), \{\}))$$

$$(c7)\quad end\_rds(A, r1) \wedge ps(A, prms([]), \{\})) \Rightarrow \bigcirc\Diamond(send\_msg(A, chan3) \wedge ps(A, prms([ack]), \{\}))$$

Figure 4.13: Connective Implications for **RDS/STS** Example

The following two examples show how the modified inference algorithm performs expansion for the global variable constructs **RDS** and **STS**. We will refer to the seven connective implications of Figure 4.13.

In the first example we will use SSA 4.25 below as the current SSA.

$$\models \left\{ \begin{array}{l} \bigcirc\Diamond(rds(A, r1) \wedge ps(A, prms[ack], \{pv_1(true), pv_2(5)\})) \wedge \\ \bigcirc\Diamond(recv\_msg(B, chan1) \wedge ps(B, prms[], \{var1(3)\})) \wedge \\ \bigcirc\Diamond(send\_msg(C, chan2) \wedge ps(C, prms[M], \{\})) \wedge \\ \{sv_1(2), sv_2(false)\} \wedge \\ \Diamond chan(chan1, [], 0, 1) \wedge \\ \bigcirc\Diamond chan(chan2, [M, M], 2, 2) \wedge \\ \Diamond chan(chan3, [], 0, 4) \end{array} \right\} \qquad (4.25)$$

In SSA 4.25 process A is at an *rds* primitive, process B is waiting to receive a message from *chan*1 and C is waiting to send a message to *chan*2. In this current SSA process A is at a non-deterministic construct so the next step in the expansion algorithm is to advance all other processes to either a waiting point or to a non-deterministic construct. Since both processes B and C are already blocked at waiting points the conditions of step 2 are already met therefore we can go to step 3 and apply expansion to the current SSA. To expand the current SSA the inference engine applies connective implications (c5), (c6) and (c7) to yield the new SSA below.

$$\models \left\{ \begin{array}{l} \bigcirc\Diamond(send\_msg(A, chan3) \wedge ps(A, prms[ack], \{pv_1(true), pv_2(2)\})) \wedge \\ \bigcirc\Diamond(recv\_msg(B, chan2) \wedge ps(B, prms[], \{var_1(3)\})) \wedge \\ \bigcirc\Diamond(send\_msg(C, chan2) \wedge ps(C, prms[M], \{\})) \wedge \\ \{sv_1(2), sv_2(false)\} \wedge \\ \Diamond chan(chan1, [], 0, 1) \wedge \\ \bigcirc\Diamond chan(chan2, [M, M], 2, 2) \wedge \\ \Diamond chan(chan3, [], 0, 4) \end{array} \right\} \quad (4.26)$$

The first thing to note about the above SSA is that the inference engine considers the transition of a process from the end of one instruction to the beginning of the next instruction to be one indivisible operation. Thus, the SSA of equation 4.26 represents the transition of process A through one indivisible operation. The inference engine used connective implication (c5) and (c6) to advance process A to the *end_rds* state and then immediately applied connective implication (c7) to advance process A to the *send msg* state. Thus, the three inferences using connective implications (c5), (c6) and (c7) are represented as one indivisible operation.

Only the one SSA resulting from the expansion on the current SSA implies that the system variable sv_1 could have only one possible value at this system state. Any other process that could have modified the value of sv_1 and thus given rise to an alternative SSA was blocked at a message waiting point.

In the second example we use the following SSA as the current SSA:

$$\models \left\{ \begin{array}{l} \bigcirc\Diamond(rds(A,r1) \wedge ps(A,prms[],\{pv_1(true),pv_2(5)\})) \wedge \\ \bigcirc\Diamond(recv\_msg(B,chan1) \wedge ps(B,prms[],\{var_1(3)\})) \wedge \\ \bigcirc\Diamond(send\_msg(C,chan2) \wedge ps(C,prms[M],\{\})) \wedge \\ \{sv_1(2),sv_2(false)\} \wedge \\ \Diamond chan(chan1,[req],1,1) \wedge \\ \bigcirc\Diamond chan(chan2,[M,M],2,2) \wedge \\ \Diamond chan(chan3,[],0,4) \end{array} \right\} \quad (4.27)$$

In this current SSA *chan*1 now contains one message whereas in the current SSA 4.25 from the last example *chan*1 was empty. Since process A is again at a non-deterministic construct we must apply expansion. We begin the expansion process by progressing all other processes (B and C) as far as possible. SSA 4.28 below represents the system state where process A is at a non-deterministic construct and all other processes have progressed as far as possible.

$$\models \left\{ \begin{array}{l} \bigcirc\Diamond(rds(A,r1) \wedge ps(A,prms[],\{pv_1(true),pv_2(5)\})) \wedge \\ \bigcirc\Diamond(sts(B,s1) \wedge ps(B,prms[],\{var_1(3)\})) \wedge \\ \bigcirc\Diamond(send\_msg(C,chan2) \wedge ps(C,prms[M],\{\})) \wedge \\ \{sv_1(2),sv_2(false)\} \wedge \\ \Diamond chan(chan1,[],0,1) \wedge \\ \bigcirc\Diamond chan(chan2,[M,M],2,2) \wedge \\ \Diamond chan(chan3,[],0,4) \end{array} \right\} \quad (4.28)$$

To reach the above SSA 4.28 the inference engine combined connective implication (c1) with process B in the current SSA 4.27. Process C is still blocked waiting to send a message to channel *chan*3. In the above SSA each process is either at a non-deterministic construct or is at a message waiting point. The inference engine can

now apply expansion to the above SSA to yield the two new SSA below:

$$\bullet \models \left\{ \begin{array}{l} \bigcirc\Diamond(send\_msg(A, chan3) \wedge ps(A, prms[ack], \{pv_1(true), pv_2(2)\})) \wedge \\ \bigcirc\Diamond(sts(B, s1) \wedge ps(B, prms[], \{var_1(3)\})) \wedge \\ \bigcirc\Diamond(send\ msg(C, chan2) \wedge ps(C, prms[M], \{\})) \wedge \\ \{sv_1(2), sv_2(false)\} \wedge \\ \Diamond chan(chan1, [], 0, 1) \wedge \\ \bigcirc\Diamond chan(chan2, [M, M], 2, 2) \wedge \\ \Diamond chan(chan3, [], 0, 4) \end{array} \right\} \tag{4.29}$$

$$\models \left\{ \begin{array}{l} \bigcirc\Diamond(rds(A, r1) \wedge ps(A, prms[], \{pv_1(true), pv_2(5)\})) \wedge \\ \bigcirc\Diamond(recv\_msg(B, chan2) \wedge ps(B, prms[], \{var_1(3)\})) \wedge \\ \bigcirc\Diamond(send\ msg(C, chan2) \wedge ps(C, prms[M], \{\})) \wedge \\ \{sv_1(3), sv_2(false)\} \wedge \\ \Diamond chan(chan1, [], 0, 1) \wedge \\ \bigcirc\Diamond chan(chan2, [M, M], 2, 2) \wedge \\ \Diamond chan(chan3, [], 0, 4) \end{array} \right\} \tag{4.30}$$

In applying expansion to the SSA of equation 4.28 the inference engine enumerates two new SSA representing two deterministic execution paths. SSA 4.29 was reached by combining process A in the current SSA with connective implications (c5), (c6) and (c7). SSA 4.30 was reached by combining process B in the current SSA with connective implications (c2), (c3) and (c4). The two new paths represent the non-deterministic outcome of the access to the global variable sv_1.

A new inference algorithm which includes the expansion algorithm for handling non-deterministic constructs is given in Figure 4.14.

SSA 4.29 represents the path where process A executes faster relative to process B. In this SSA the **RDS** instruction executed by process A will retrieve a value of 2 for the system variable pv_1.

The second execution path, SSA 4.30, represents the case where process B is faster relative to process A. In this SSA the **RDS** instruction that will eventually be

```
1.  present state SSA queue Q = initial state
2.  list of generated next states L = empty list
3.  start inference engine
4.  remove first current SSA from Q
5.  select a process Pr from the current SSA to progress according
                    to a selection algorithm θ
6.  if no process can progress then
7.        goto 32
8.  if Pr is at a non-deterministic construct (RDSSTS,select) then
9.        begin
10.       progress all other processes until either they are
              blocked waiting for a message or they reach
              a non-deterministic construct (RDS,STS,select)
11.       infer all possible next states from Pr
12.       for each next-state
13.             if next-state is not a member of L then
14.                   add next-state to L and to Q
15.             else
16.                   discard next-state
17.       end for
18        end
19. else
20.       begin
21.       infer next-state from Pr
22.       if next-state is not a member of L then
23.             add next-state to L and to Q
24.       else
25.             if next-state is a member of L then
26.                   discard next-state
27.       end
28. if Q is empty then
29.       goto 33
30. else
31.       goto 4
32. system-wide deadlock detected
33. terminate inference engine
```

Figure 4.14: Inference Algorithm with Expansion

executed by process A will retrieve a value of 3 for the system variable pv_1.

Thus, the non-determinism present in the access to a global variable is represented by the two new SSA. Since we cannot assume that the reader process is faster relative to the writer process, or vice versa, then we must account for both possibilities.

4.6 Chapter Summary

In this chapter we developed a temporal logic representation system, PTR, for PAL Temporal Representation, to describe the behaviour of PAL expressions. PTR is based on work done by G.M. Karam in [Kar87]. The temporal formulae of which PTR is composed, when manipulated by a suitable inference engine, will produce a tree of system state assertions or global state graph which can be interpreted as showing the behaviour of the system over time. In the next chapter we discuss a means of verifying this global state graph against a specification expressing correct temporal behaviour for the PAL system described by the PTR formulae. If the specification of temporal behaviour is satisfied on the global state graph then we can say that the graph, and hence the PAL expressions from which the graph was derived, is a valid representation of the user's requirements.

Chapter 5

Model Checking

In this chapter we discuss the development of the model checker part of our verification system. Our model checker is based on the model checking approach outlined in Chapter 3 (Section 3.4.5) and first proposed in [CES83]. We begin this chapter with an overview of the method and then move on to the details of applying it in the PAL environment.

5.1 The Model Checking Approach

Figure 5.1 shows a global state graph for a simple finite state concurrent system. Values for each of four state variables are shown for each of the six states in the system. In the model checking approach this global state graph is considered to be an implementation or model of the system under analysis. If it can be shown that the model is correct with respect to a specification then the system description from which the model was derived must also be correct with respect to the specification.

The specification against which the model is checked is expressed as a BTL Formula **F** which describes certain correctness properties (Section 3.3) relevant to the system under analysis. In general, multiple correctness properties can be expressed in **F** as a conjunct of BTL terms where each of the terms f_i is a BTL formula describing

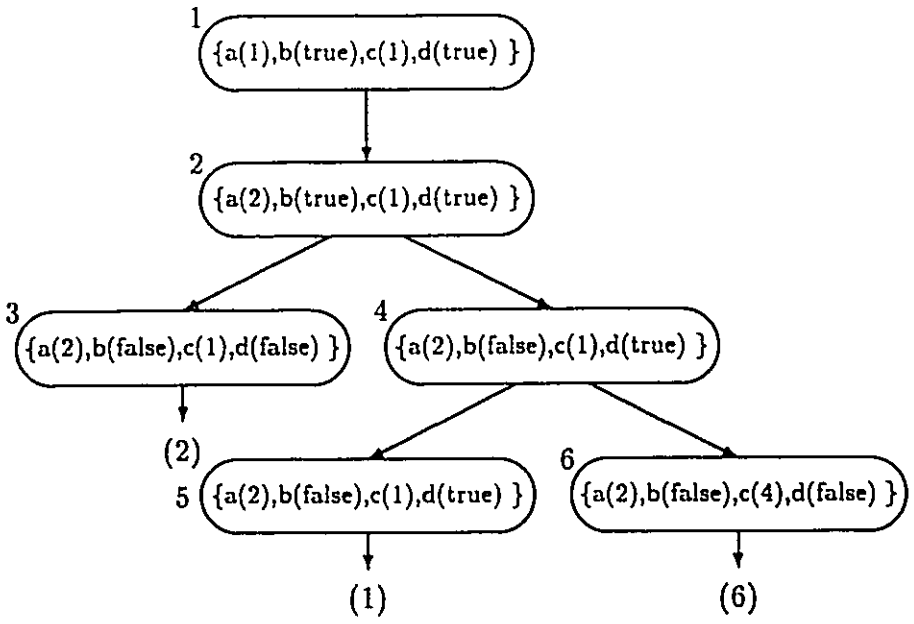


Figure 5.1: Model Checking: State Graph Example

some correctness property that any valid implementation of the specification must meet. The correctness formulae are composed according to the normal rules for BTL as given in Chapter 3.5. Each formula is composed of immediate assertions on the state variables of the system under analysis.

For our example system, of which the state graph of Figure 5.1 is a model, a specification describing correct temporal behaviour must be composed using the state variables a,b,c and d. We will assume that the BTL formula below describes the correct temporal behaviour for the system represented by Figure 5.1.

$$\{\forall\Box(a < 4) \ \wedge \ \forall\Diamond\neg(b = true) \ \wedge \ \exists\Diamond(c > 3) \ \wedge \ \exists\Box(d = true)\} \qquad (5.1)$$

To prove that the model of Figure 5.1 is a correct implementation of this specification it must be shown that the BTL formula is satisfied on the model. For Formula 5.1 to be satisfied on the state graph the formula must be valid at each of the six system states in the graph. In the following sections we discuss an efficient algorithm (the

model checking algorithm) to determine whether or not a BTL formula is satisfied on a state graph.

5.1.1 The Model Checking Algorithm

The model checking algorithm is designed such that when it completes each state of the state graph will be marked with the terms f_i and formulae **F** that are satisfied at that state. If for all states s_i in the model, **F** is marked at s_i then the model is correct with respect to the specification,F.

The algorithm for marking the model must be able to handle the ten different allowable forms of a BTL formula. We restate in the table below the ten allowable forms of BTL formulae:

f	$\neg$f	$f_1 \vee f_2$	$f_1 \wedge f_2$		
$\forall\Box$f	$\exists\Box$f	$\forall\Diamond$f	$\exists\Diamond$f	$\forall\bigcirc$f	$\exists\bigcirc$f

Table 5.1: Allowable forms for BTL formulae

Each f in the table is an immediate assertion on the state variables of the system, e.g., $(a < 3)$ is a valid immediate assertion for the system represented by the graph of Figure 5.1. Well formed BTL formulae are composed by treating formulae in one of the above forms as a subformula term and substituting it back as f in one of the forms above. For ease of explaining the model checking algorithm we will assume that a specification **F** is composed only of simple immediate assertions in one of the ten basic forms above.

As is evident from the above table we have chosen to use the simple BTL system $\mathcal{UB}$[1] proposed by [BAMP81] rather than the more expressive CTL* [2] used by Clarke et.al., in their development of the model checking method[CES83]. For our purposes we can express all basic correctness properties in BTL without having to deal with

[1] $\mathcal{UB}$ for Unified system of Branching time logic
[2] Computational Tree Logic

the additional complexity of CTL*. In any case once a model checker for PAL has been developed using BTL we could then modify it to handle CTL* expressions.

To be able to mark the validity of a BTL formula **F** at each state in the the model we begin by assuming that the truth value for each immediate assertion of which **F** is composed can be determined at each state of the state graph. Our state graph model in fact contains the actual value for all valid state variables of the system at each node of the state graph. The truth value of the immediate assertion at any node in the graph is determined by substituting the actual value of the state variable in the immediate assertion and evaluating the expression. For example if the immediate assertion was $(a < 3)$ then substituting the actual value for a at each node would yield the truth value for the assertion at that node. For example, the truth value of the immediate assertion $(a < 3)$ at state 1 of the state graph of Figure 5.1 is *true*.

Given that we can determine the truth value of each immediate assertion in **F** at each node of the state graph then the validity of **F** at state s_i can be determined according to the following model checking algorithms. The state s_i at which we wish to determine the validity of **F** is considered to be the present state s_0 in each of the following cases. We first present the four non-temporal forms of **F**.

5.1.1.1 Non-temporal Forms

For the four non-temporal forms of a BTL specification the validity of **F** at a state s_0 depends simply on the truth value of the immediate assertion(s) in the current state. The current state, s_0, is marked as satisfying **F** if the formula is true at s_0. For example, if $\mathbf{F} \equiv (a < 3) \wedge (d = false)$ then for the state graph of Figure 5.1 f would only be satisfied at states 3 and 6.

5.1.1.2 Temporal Forms

As should be expected for the non-temporal forms, the validity of the specification depends only on the system variable values in the current state, s_0. However, the

validity of the temporal forms of a BTL formula at s_0 depends not only on the variable values at s_0 but also on the variable values along each of the state sequences starting at s_0, i.e., the validity of a temporal formula in the present depends on the behaviour of the system in the future.

The model checking algorithm is actually a set of algorithms, one for each of the valid BTL operators. Each of the algorithms is similar in that a depth-first search is made starting at the current state s_0 and proceeding through the branches of the state graph. The search proceeds according to the definition of the temporal operator and terminates when the validity of the formula at the present state has been established.

The checking algorithms for each of the temporal operators is based on a *fixpoint characterization* of their definitions[CES83]. That is, each of the operators can be defined in terms of a condition in the current state and a condition along all state sequences starting in the current state. The conditions in the current state and the state sequences starting in the current state are termed, respectively, the least and greatest fixpoints. The fixpoint characterizations for each of the temporal operators is given below:

$$\forall\Box \mathbf{f} \equiv \mathbf{f} \wedge \forall\bigcirc(\forall\Box \mathbf{f}) \tag{5.2}$$

$$\exists\Box \mathbf{f} \equiv \mathbf{f} \wedge \exists\bigcirc(\exists\Box \mathbf{f}) \tag{5.3}$$

$$\forall\Diamond \mathbf{f} \equiv \mathbf{f} \vee \forall\bigcirc(\forall\Diamond \mathbf{f}) \tag{5.4}$$

$$\exists\Diamond \mathbf{f} \equiv \mathbf{f} \vee \exists\bigcirc(\exists\Diamond \mathbf{f}) \tag{5.5}$$

The $\forall\bigcirc$ and $\exists\bigcirc$ operators do not have fixpoint representations since their validity depends only on the next state to the current state.

As is shown by the above fixpoint characterizations of the temporal operators, the model checking algorithms consist basically of a recursive test for the truth value of f in the current state. For example, for the $\forall\Box$ operator, $\forall\Box$f is true at s_0 if and only if f is true at s_0 and for all immediate successor states s_i of s_0, $\forall\Box$f is true at s_i. Thus the truth value of $\forall\Box$f at s_0 depends on the truth value of f at s_0 and on the truth value of $\forall\Box$f at each s_i.

To explain the checking algorithm for the temporal forms we will begin by explaining in detail the algorithm for the $\forall\Box$ operator. Detailed descriptions, in the form of self-documented Prolog code, for each of the model checking algorithms can be found in Appendix D.

The algorithm is explained in pseudo code format in Figure 5.2. The boolean variable `visited(`s`)` is used to indicate that the algorithm has previously visited the node s_0. The function `true_at` determines the truth value of the immediate assertion f at the current node of the graph s_0.

The first part of the algorithm (lines 1 to 10) serve to initialize the model checker by setting the variable `visited` to false for each state. The `visited` flag is set true as each state is visited during the algorithms depth-first search of the graph. Lines 4 to 9 call the function check for each state in the graph. check determines the validity of $\forall\Box$ F at a given state s_i on the graph.

The function check(F,s_i) is a direct implementation of the fixpoint characterization of the $\forall\Box$ operator. The recursive calls to check(F,s_i) in line 24 cause a depth-first search of the state graph. The function `true_at` at line 18 determines the true value of the immediate assertion f at the current node. If at any node it is found that f is not true then the function returns false. This false value is propagated back to the first part of the algorithm causing it to halt. Only if $\forall\Box$ f is valid at all states in the graph is the graph considered to be a valid implementation of the specification.

Since the state graph has only a finite number of states a depth-first search along one branch of the graph must eventually end with a state that has already been visited. Having reached a state that has already been visited and knowing that no states at which f is false have yet been found (otherwise the algorithm would have halted) the algorithm can safely conclude that f is true for the branch of the graph that is currently being investigated. Hence the `return` statement at line 14 when a state is reached that has already been visited.

The checking algorithms for the temporal forms $\exists\Box$f, $\forall\Diamond$f and $\exists\Diamond$f are similar

```
1.  for all states s_i in the Model do
2.       visited(s_i) := false
3.  end for
4.  for all states s_i in the Model do
5.       if check(F,s_i) then
6.           marked(F,s_i) := true
7.       else
8.           halt
9.  end for
10. success.

11. function check(F,s_i)
12. if visited(s_i) or marked(f,s_i) then
13.       check(F,s_i) := true
14.       return
15. else
16.       begin
17.       visited(s_i) := true
18.       if not true_at(f,s_i) then
19.           check(F,s_i) := false
20.           return
21.       else
22.           begin
23.           for all immediate successor states s_j of s_i do
24.               if not check(F,s_j) then
25.                   return
26.           end for
27.           check(F,s_i) := true
28.           return
29.           end
30.       end
31. end function
```

Figure 5.2: Model Checking Algorithm for $\forall\Box$

to the above algorithm with the main differences being the conditions for which the function check returns the value false. For example, for the $\exists\Box$ operator line 23 of the above algorithm would be:

23. for some immediate successor state s_j of s_i do

i.e., check will return false only if there is not at least one successor state to s_0 at which $\exists\Box$f is true.

Again, the algorithms for each of the temporal operators follow directly from their fixpoint characterizations.

For the two *next-time* temporal operators, $\forall\bigcirc$ and $\exists\bigcirc$, the checking algorithms are fairly straightforward. For a formula $\forall\bigcirc$ f the function check is given in Figure 5.3.

```
1.  function check(F,s_i)
2.  check(F,s_i) := true
3.  for all immediate successor nodes s_j of s_i do
4.       if not true_at(f,s_j) then
5.            check(F,s_i) := false
6.            return
7.  end for
8.  return
9.  end function
```

Figure 5.3: Model Checking Algorithm for $\forall\bigcirc$

For a formula $\exists\bigcirc$ f the function check is given in Figure 5.4.

```
1.  function check(F,s_i)
2.  check(F,s_i) := false
3.  for some immediate successor nodes s_j of s_i do
4.       if true_at(f,s_j) then
5.            check(F,s_i) := true
6.            return
7.  end for
8.  return
9.  end function
```

Figure 5.4: Model Checking Algorithm for $\exists\bigcirc$

To handle a specification of arbitrary complexity the above checking algorithms are applied successively to the subformulae of the specification. The specification is first decomposed into simple BTL assertions whose validity is checked and marked on the graph. The simple assertions are then combined to form the next higher level subformula of the specification. The graph is then checked and marked for this subformula. The successive building up and checking is continued until the original formula is checked and marked on the graph. Note that by marking the graph for each formula and term of a specification we can eliminate the need to check the graph for other formula containing the same terms. For example, if the BTL assertion $\forall\Box(a < 3)$ was a term in a number of higher level formula within the specification it would only have to be checked and marked once on the state graph.

Clarke, et.al., in their series of papers [CES83][CBES85][CES86][Bro86] on the model checking approach only presented and discussed the complete model checking algorithm for the CTL operator $\forall\mathcal{U}$. Using the same approach as was given for the $\forall\mathcal{U}$ operator we derived the model checking algorithms for each of the six temporal operators that we are using.

5.1.2 An Example

The BTL formula given as Formula 5.1 specifies the correct temporal behaviour required of some simple system. The state graph of Figure 5.1 describes, in terms of system state sequences, the actual temporal behaviour of a proposed system which we believe will satisfy the requirements as given by the specification. We wish to know if the state graph is in fact a valid model of the specification, i.e., is the specification satisfied on the state graph?

To check the validity of the specification for the state graph of Figure 5.1 we begin by decomposing the specification into its four simple BTL assertions $\forall\Box(x < 4)$, $\forall\Diamond(b = true)$, $\exists\Diamond(c > 3)$ and $\exists\Box(d = true)$. We then use the appropriate checking algorithm to mark the state graph for each of the temporal terms. Once each state in

the graph is marked with the terms that are valid at that state we can visit each state to determine the validity of the complete specification at that state. i.e., if $\forall\Box(x < 4)$ and $\forall\Diamond(b = true)$ and $\exists\Diamond(c > 3)$ and $\exists\Box(d = true)$ are each marked (true) at s_i then the complete specification is valid for s_i. If the specification is valid for all states in the graph then the graph must be a valid implementation of the specification.

The marking algorithm for the first term $\forall\Box(a < 4)$ begins at state 1 of the graph. State 1 has not yet been visited, therefore `visited(1)` is set true and the truth value of the immediate assertion $(a < 4)$ is determined at state 1. We find that $(a < 4)$ is true at state 1 therefore check is called recursively for each immediate successor state of state 1. In this case state 2 is the only successor state of state 1.

The function check is called for state 2. State 2 is marked as `visited`. The immediate assertion $(a < 4)$ is true at state 2 therefore check is called for each successor state to state 2. States 3 and 4 are the only immediate successor states of state 4.

At state 3 `visited(3)` is set true. Since $(a < 4)$ is true at state 3 the function check is called for each successor state of state 3, in this case only state 2.

At state 2 we find that `visited(2)` is true therefore we return to the last call on check (state 3). Since all successor states from state 3 have been found to satisfy $(a < 4)$ we can return to the last call on check (state 2).

Back again at state 2 we have found that $(a < 4)$ is satisfied for the state sequence starting at state 3. Now check is called for the second of state 2's successor states, i.e., state 4.

At state 4 $(a < 4)$ is true therefore check is called for each of state 4's successor states, states 5 and 6. etc...

This recursive calling of check is continued until either $\forall\Box(a < 4)$ is marked at each state or $(a < 4)$ is found not to be true at a certain state in which case the algorithm is halted. If the checking algorithm was allowed to continue it would find that $\forall\Box(a < 4)$ is valid at all the states in the graph. Model checking for the other

three BTL terms would continue in a similar fashion using the appropriate checking algorithms.

For our example the model checker would eventually find that the specification is not satisfied on the state graph because the term $\exists\Box(d = true)$ is false at state 3 and at state 6. The value of d at both states 3 and 6 is $false$ therefore according to the definition of the $\exists\Box$ operator $\exists\Box(d = true)$, is not valid at either state.

One of the conjunctive terms, $\exists\Box(d = true)$, of the specification is not satisfied on the graph therefore the complete specification is not satisfied on the graph. This implies that the state graph does not represent a correct implementation of the specification. Armed with the knowledge of which sequences of states or individual states caused the specification to fail we could modify the system and again check it against the specification. This can be repeated until the specification is satisfied on the state graph.

5.2 PAL Correctness Specifications

In this section we define a means of expressing immediate assertions on the state variables of a PAL system and show how they can be combined with the BTL operators to construct correctness specifications. To express immediate assertions on the state variables of a system we introduce the following predicates:

pv(p,π(v)) — this predicate is true if the array of variable values $\pi(v)$ for the process p is true in the current state. Supplementary predicates can be used to express relationships among the variable values. For example, *pv(A,{limit(x),@lss(x,3)})* could be used to assert the fact that the value of the process condition variable *limit* in process A should be less than *3*.

sv($\Pi(v)$*)* — the *sv* predicate is true if the value of the system variable Π is v. Again, supplementary predicates can be used to express relationships among the system variable values.

st(l) — this predicate is true if in the current state the process is at the program label *l*. Where *l* can be any of the valid state predicates from Section 4.3.1 or the additional predicates for describing non-deterministic constructs from Section 4.5. For example, *st(recv_msg(A,chan1)* is true if in the current state, process *A* is at a point in its execution where it is waiting to receive a message from *chan1*.

ch(ch,ψ) — this predicate is true if in the current state the number of messages in the channel queue for channel *ch* satisfies some relation ψ. The use of supplementary predicates is allowed in order to express some relation on the number of messages in the queue. For example, *ch(chan2, {n,@lss(n,3)})* specifies that the number of messages in *chan2*'s message queue should not exceed *3*.

Using these predicates we can build meaningful BTL terms to express desirable correctness properties for the proposed system. Each BTL formula **F** is composed of simple state predicates f_i in one or more of the allowable BTL forms from Table 5.1. For example to express the liveness property that a process does not experience starvation we could specify:

$$\left\{ \begin{array}{l} st(init(p_1)) \Rightarrow \forall\Diamond\neg st(init(p_1)) \\ \wedge\, \neg st((init(p_1)) \Rightarrow \forall\Diamond st(init(p_1)) \end{array} \right\}$$

which can be interpreted as follows:

> "for all system states if process p_1 is in the *init* state then for all execution paths starting with that state p_1 must eventually not be in the *init* state and for all states where p_1 is not in the *init* state then for all execution paths starting with that state p_1 must eventually reach the *init* state."

This partial specification expresses the notion of non-starvation for a single process. Similarly we can specify non-starvation for all other processes in the system and add them to the specification. There are other ways of expressing non-starvation

such as:

$$st(init(p_1)) \Rightarrow \forall \bigcirc (\forall \Diamond st(init(p_1)))$$

To show that two processes never violate certain mutual exclusion conditions we can express as part of a specification:

$$\neg \exists \Diamond (st(end\_recv(t_1, chan1)) \wedge st(end\ recv(t_2, chan2)))$$

where in this case both processes are waiting to receive an acknowledgement message from some resource management process allowing access to a certain resource.

As an example of a global invariance property we can specify:

$$\forall \Box (pv(lift_1, \{floor(x), @leq(x, 3)\})$$

This property expresses the fact that the value of the variable *floor*—the floor that the elevator controller process $lift_1$ thinks the elevator is at—is never greater than the maximum number of floors in the building; 3 in this case.

A complete specification consists of a set of BTL formulae as above in conjunctive form with each conjunctive term expressing a desirable correctness property of the system. All general system properties such as starvation and mutual exclusion can be included as well as any global invariance properties that are specific to the particular system. The specification can contain any property that is expressible in BTL form but it must be stated in terms of the state variables of the particular system under investigation.

5.2.1 Complexity

The complexity of the model checking algorithm depends on the number of states in the state graph **S**, the number of paths between states **R** and the complexity of the formula being checked. The complexity of a BTL formula **F** is given as the *length* of **F**[CES83]. The actual complexity given in [CES83] is

$$O(length(\mathbf{F}) \times (CARD(\mathbf{S}) + CARD(\mathbf{R})))$$

where CARD(**X**) gives an integer value for the number of possible values of the variable **X**, i.e., the *cardinal* value of **X**. The length of a formula **F** is equal to the number of simple BTL terms and logical connectives ($\vee$,$\wedge$,$\Rightarrow$) of which it is composed. A simple BTL term is any unary BTL operator combined with an immediate assertion on the variables of the system. e.g., *length*($\forall\Box$**a**) is equal to 1, and *length*(($\forall\Box$**a** $\vee$ $\neg$ **b**) is equal to 3.

To see how the above order of complexity was established we can note that to check the validity of a formula $\forall\Box$**a** at a single node s_0 requires ($CARD$(**S**) + $CARD$(**R**)) calls to the function **check**. That is, the number of times that a recursive call is made to **check** depends on the number of states in the graph and the number of paths between states. To determine the validity of $\forall\Box$**a** at s_0 requires determining the validity of $\forall\Box$**a** at all successor states s_i of s_0 and of each successor state of s_i, etc. Therefore, to determine the validity of $\forall\Box$**a** for the complete graph requires each state s_i to be visited at most twice. The first time to determine the validity of $\forall\Box$**a** at s_0, at which time `visited`(s_i) is set true. The second time to determine the validity of $\forall\Box$**a** at itself, s_i. If the state s_i has previously been `visited` then we know that $\forall\Box$**a** must be valid at s_i (the checking algorithm would already have halted if $\forall\Box$**a** was not valid at a previously `visited` state).

Thus, to check the complete state graph for $\forall\Box$**a** requires at most

$$2 \times (CARD(\mathbf{S}) + CARD(\mathbf{R}))$$

calls to the function **check**. A formula of arbitrary complexity would require calling the appropriate checking algorithm once for each simple BTL term of which the formula was composed plus once more for each logical connective used. Thus, the total complexity also depends on the length of the formula being checked.

In the above we used the $\forall\Box$ operator to show how the order of complexity for the checking algorithms was determined, however the complexity of the other operators is of comparable order.

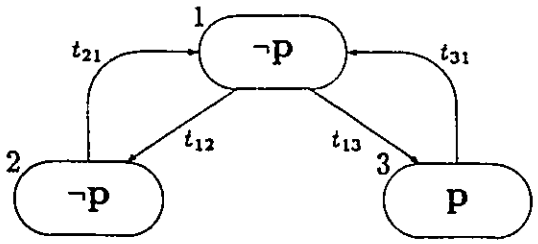


Figure 5.5: Simple Fairness Example

5.3 Model Checking Assuming Fairness

In this section we discuss modifications to the model checking algorithm to allow temporal formulae to be checked over only the *fair* computations of the system. *Fairness* is a property which must be considered when non-deterministic models, such as our state graph, are used to represent system behaviour[QS83]. If from a certain system state there exists a number of actions that the system may execute and the choice of which of these actions to execute next is unspecified then this state represents an element of non-determinism in the system. Problems related to fairness arise due to these elements of non-determinism.

Consider the simple state graph of Figure 5.5. We wish to prove that this model satisfies the temporal formula $\forall\Diamond\mathbf{p}$. State 1 of the model, at which **p** is false, represents a non-deterministic branching state in that either transition t_{12} or transition t_{13} can be taken from this state. In attempting to verify $\forall\Diamond\mathbf{p}$ at state 1 we find a computation, $\phi_1 = s_1, s_2, s_1 \ldots$, i.e., an infinite loop through transitions, t_{12} and t_{21}, along which **p** is never true. Therefore, the property $\forall\Diamond\mathbf{p}$ is not satisfied on the model.

However, intuitively, we know that for the system to always chose to follow transition t_{12} such that the system never reaches state 3, where **p** is true, should not happen; unless, somehow the system is biased towards transition t_{12}. This example

illustrates the concept of *fairness*. If we encounter a computation, (an infinite sequence of system states), on which some event becomes possible infinitely often but is realized only a finite number of times, then this event is being unfairly treated by the system. Recall the definition of a computation from Chapter 3.5; a computation is an infinite sequence of system states. For a system with a finite number of states a computation must be the result of the repeated execution of a finite number of system states,i.e., a loop. The *events* that *becomes possible* at state 1 for the infinite computation $\phi_1 = s_1, s_2, s_1 \ldots$ from our example, are the transitions t_{12} and t_{13}. Each time the system is in state 1 it is possible for the system to take either transition t_{12} or t_{13}. On the computation ϕ_1 only transition t_{12} is taken. The event, transition t_{13}, which becomes possible infinitely often occurs only a finite number of times (zero times) on ϕ_1. If we ignore the *unfair* computations, i.e., ϕ_1, on the model of Figure 5.5 we will find that the temporal formula $\forall\Diamond\mathbf{p}$ is now satisfied on the state graph model of the system.

We need to be concerned with fairness when checking the validity of a formula on our state graph because, as is illustrated above, the validity of the formula may depend on the fairness of the computations. Having presented an intuitively based discussion of fairness. We will now proceed to give a more general definition of fairness and show how it is handled by our verifier for the PAL environment.

5.3.1 A Definition of Fairness

Informally, fairness can be expressed as follows:

> "A computation is fair if during its execution every event which becomes possible infinitely often is realized infinitely often." [QS83]

Any notion of fairness is meaningful only for an infinite sequence of system states, i.e., on a computation of the system. A finite sequence of system states is always considered to be fair since there can be no notion of an event becoming possible infinitely often or of being satisfied infinitely often on a finite state sequence.

When discussing fairness it is necessary to qualify the above informal definition with respect to the system being investigated. For example, Quielle and Sifakis [QS83] formally defined three different types of fairness for non-deterministic state transition systems. Each of the formal definitions could only account for fairness with respect to certain aspects of the state transition model. The differing definitions of fairness resulted from the differing interpretations as to what constitutes an event becoming possible for the transition system. They concluded that any one formal definition is inadequate to characterize fairness in general and went on to state that any formal characterization of fairness must be relativized to the system model and the system events of interest. For this reason we will use the informal definition as a yardstick against which to develop our own definition of what constitutes a fair computation in the PAL environment.

Problems related to fairness arise due to non-determinism in the models used to describe system behaviour. If there is more than one possible action that a system may take from a certain system state and the action to be taken is not specified then this state represents an element of non-determinism in the system. To be able to detect an unfair computation we must consider all sources of non-determinism in the system model.

5.3.2 Fairness in PAL

There are four possible sources of non-determinism in the PAL execution model:

1. An interleaved representation of concurrency;
2. The non-deterministic execution rates of the individual processes;
3. The select construct; and
4. Access to global variables.

We will consider each of these in turn below.

Interleaved Model of Concurrency

In a system of m processes executing on m processors each process that is enabled is guaranteed to be executed since it has the exclusive use of a processor. In this purely concurrent model of execution no process that is enabled can be unfairly treated. However, in building our global state graph model of system behaviour we have assumed an *interleaved* execution model to represent concurrency . That is, all processes execute on a single processor. Transitions between system states represent the progress of one of the processes through one indivisible operation. This interleaved model of execution introduces an element of non-determinism not found in the concurrent model—which process will be the one that is picked to progress at each interval? To resolve the non-determinism we assume the existence of a *scheduler* to select which process will execute the one indivisible operation that will cause a transition to the next state. In generating the interleaved representation of concurrency we require that the scheduler be fair[MP81b]. That is, if a process is enabled then it will eventually be selected by the scheduler to progress.

Non-deterministic Rate of Execution

A major source of non-determinism in any concurrent system is the unknown rate of execution of the individual processes of the system. In the interleaved model of execution non-deterministic rates of execution are represented by including all possible process interleavings in the model. That is, from each system state the interleaved model gives a next state representing the progress of each process that is enabled in the current state. By following the transitions in the model along which process x always advances we are assuming that x executes at a faster rate than the other processes. If we followed the transitions along which process y progresses then we would be assuming that y executes faster relative to the other processes. Thus, the unknown execution rates of the individual processes are directly represented in the interleaved model by giving all possible interleavings of the processes.

In fact, non-determinism resulting from both the interleaved model of concurrency and the non-deterministic execution rates are represented in the state graph model by considering all possible process interleavings. By showing each process advancing from each state we represent both the fact that a fair scheduler schedules all enabled processes for execution and the fact that each process executes at an indeterminate rate.

In Chapter 4 (Section 4.4.1) we showed that the different possible interleavings resulting from the non-deterministic execution rates of the individual processes resulted in equivalent system behaviour. Therefore, in building our state graph model of the system we chose to include only one possible interleaving and thus reduced the number of system states in the model. That is, when building the state graph, as long as we know that the processes' relative rates of execution have no bearing on overall system behaviour then we are able to force each process to execute at a known rate relative to the other processes. By doing this we have removed from the state graph the element of non-determinism resulting from the unknown execution rates of the processes.

By using only one of the equivalent process execution interleavings to represent system behaviour we have removed the element of non-determinism resulting from the unknown execution rates of the processes. However, the different interleavings also represented the effects of a fair scheduler. That is, each process is given equal opportunity to progress and this is shown in the model by the different next state transitions from each current state for each enabled process. In order to preserve the effects of a fair scheduler, the particular interleaving that we have chosen for our state graph is one along which all processes are given equal opportunity to progress.

Select Construct

The semantics of the PAL select construct are defined such that the choice of which open select alternative to execute is non-deterministic. In generating the state graph

model for the system the inference engine represents this non-determinism by enumerating a new deterministic execution path for each alternative. Thus, the non-deterministic choice as to which alternative to follow is represented directly in the state graph.

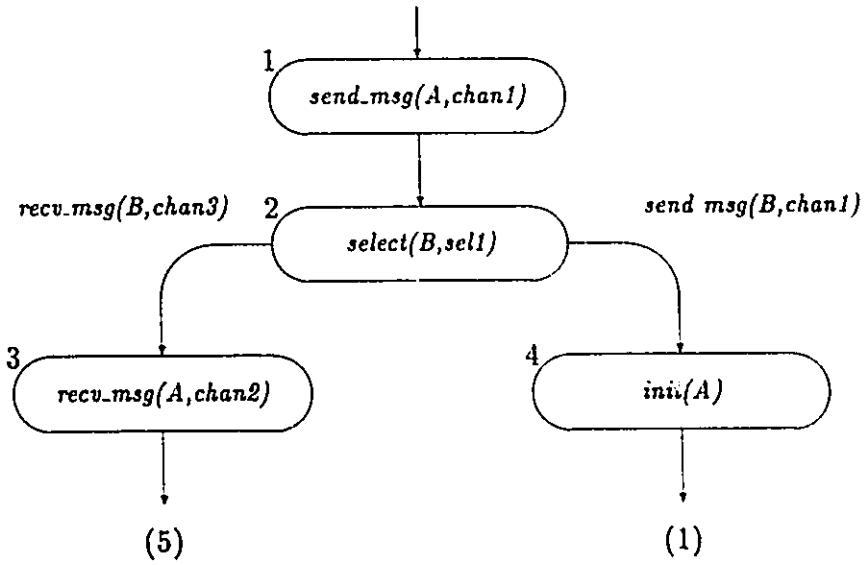
Global Variables

A fourth source of non-determinism in the execution model is caused by the scheduling of access to global variables. In PAL, global variable reading and writing is unprotected by semaphores or other synchronization mechanisms. This means that any process that wants to read or write a global variable must only be scheduled by the process scheduler to do so. We have already concluded that the inference engine assumes a fair process scheduler when generating the state graph model therefore any process wanting access to a global variable will eventually be granted access. However, due to the non-deterministic execution rates of the processes who may access the global variables we do not know whether a process will read a particular variable value before or after it has been modified. In this case the different possible interleavings do not result in equivalent system behaviour. Therefore, when a process accesses a global variable we represent the indeterminate value of the variable by enumerating all possible process interleavings in the state graph model. (refer to Section 4.5, page 86).

5.4 Detecting an Unfair Computation

Of the four sources of non-determinism that we have identified, only two, the select construct and access to global variables, are represented in the state graph. We can now proceed to show how we can detect any unfairness in the model resulting from these sources of non-determinism.

Recall the informal definition for fairness given in Section 5.3.1. In order to



This figure shows a computation taken from some arbitrary state graph. Only the states that are part of the computation and the next states at branching points are shown. For the *select* branching point the two alternative operations are also shown.

Figure 5.6: A Computation

relativize this definition for the PAL execution model we must first define what constitutes an event *becoming possible* in terms of our state graph model. For our state graph, an event becomes possible if the system state in which it could be satisfied is the current system state. For example, in the computation shown in Figure 5.6 the operation *send msg(A,chan1)* becomes possible each time the system enters state 1. An operation is satisfied, i.e., it has been executed, if a transition is made from the state in which the event becomes possible to a system state where the event is no longer possible. Thus, the *send msg(A,chan1)* event is satisfied if the system makes transitions from state 1 to either state 3 or state 4. If process A was trying to send a message to *chan1* in state 1 and at a later state, i.e., state 3 or 4, process A is no longer trying to send a message to *chan1* then the message must have been sent, i.e., the *send msg(A,chan1)* was satisfied.

At any branching state (a state with more than one outgoing transition) all of the operations associated with each of the branch alternatives become possible at that state. For example, at state 2 both the operations *send_msg(B,chan3)* and *recv_msg(B,chan1)* become possible. However, only the operation associated with the branch alternative that is actually followed is satisfied. We can see therefore, that the *events* that become possible are actually the *atomic operations* that may be executed along the transition from one state to a next state.

We can distinguish among the various operations that are available for the PAL system to define two different types of events that can become possible.

message events — operations involving the sending of a message to a channel and the receiving of a message from a channel;

access to global variables — operations for reading a global variable into a local variable and for writing a new value to a global variable;

The operations with which message events are associated are the *send_msg* and *recv msg* operations and for global variables, the operations are *rds* and *sts*. For

messaging events, the events are satisfied if a message is sent to a particular message channel by a *send_msg* operation or if a message is received from a particular channel by a *recv_msg* operation. For global variable events, the events are satisfied if a value for a particular global variable is either read by an *rds* operation or is modified by an *sts* operation.

Distinguishing between the two event types allows us to more efficiently detect an unfair computation. We can split the task of detecting an unfair computation into two parts corresponding to the two types of events. By grouping the events into similar types we can develop a detection method best suited to the type of event. If any one type of event is being unfairly treated then the computation is unfair.

A general purpose algorithm for detecting unfairness on a computation can be stated as follows:

1. Identify a computation on the state graph;
2. Compile a list of all events that become possible along the computation;
3. Compile a list of all events that are satisfied on the computation;
4. Compare the two lists, if there exists an event that becomes possible but is never satisfied then the computation is unfair, otherwise it is a fair computation.

We realize that this algorithm will not judge fairness for certain *arbitrarily branching* computations, i.e., those where a finite loop of states is traversed for n times and then another loop for m times and then another, etc. Fairness checking for such computations is not possible. The type of fairness we are concerned with is if a given finite sequence of states is followed an infinite number of times does this represent fair or unfair behaviour. This type of fairness can be determined for a finite sequence of states and does provide useful information about the behaviour of the system.

We now examine each type of event and develop an appropriate means of detecting unfairness based on that type of event.

5.4.1 Fairness With Respect to Message Events

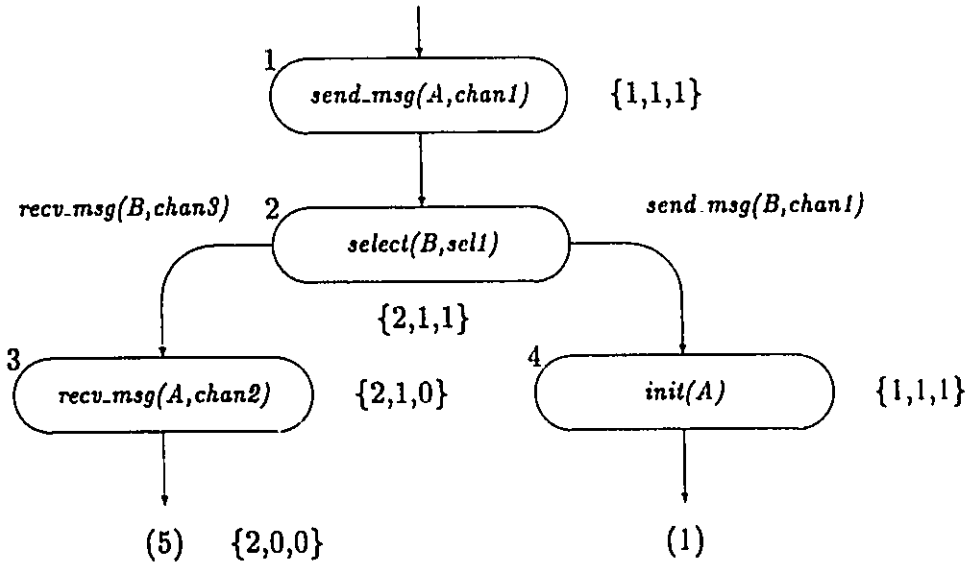
To determine the fairness of a computation with respect to message events we observe the state of the system message channels along the computation. Any channel whose state changes along the computation must be infinitely often experiencing both a *send_msg* and a *recv_msg* message event. This is so because:

- any computation on our global state graph must be made up of a finite loop structure of program state assertions;
- if a channel experiences change over this computation then the number of sending events along the computation must be matched by the same number of receive events (because message channels have a finite capacity); and
- for a channel to change state, at least one send and one receive event must become possible and be realized over the computation.

If a channel experiences change over a computation then the events associated with this channel meet the informal fairness criteria of an event becoming possible infinitely often and realized infinitely often. That is, the events are being fairly treated. If all channels in the system experience change over a computation then all message events along the computation are being fairly treated. Therefore, the computation must be fair with respect to message events.

If a channel does not change state along a computation then either, a message event is being ignored along the computation or, the computation is such that no events become possible for that particular channel along the computation. In the first case the computation would be considered unfair and in the second fair. To distinguish between the two cases we employ a closed world assumption. That is, we try to prove that a message event is being ignored along the computation. If we cannot find a message event being ignored then we assume that the unchanging state of the channel does not represent an unfairness in the computation.

To determine if a message event is being ignored on a computation we can note that the only point where message events can become possible and not be satisfied infinitely often is where there is a select construct branching point. If an open select alternative is never chosen then any events associated with that alternative are becoming possible infinitely often and not being satisfied infinitely often. If we examine each possible branching point along the computation we can determine whether or not possible events for a channel are being unfairly treated. If at a select branching point along the computation a non-progressing channel could have experienced a message event by taking an alternate path then the computation is unfair. It is unfair because an event was becoming possible infinitely often but was not being realized infinitely often along the computation. i.e., the event was being continuously ignored.



This figure shows a computation taken from an arbitrary state graph. Only the SSA that are part of the computation and SSA at branching points are shown. The system from which the computation was derived has three channels. The number of messages in each channel is shown at each system state.

Figure 5.7: A Computation with Channel Information

To give an example of the above discussion we have reproduced our example computation from Figure 5.6 as Figure 5.7. To the computation we have added information about the number of messages in each of the system's three channels. The notation {1,2,2} indicates that *chan1* contains one message, *chan2*—two messages and *chan3*—two messages.

If we look at the computation (1,2,4) of Figure 5.7 we can see that along the computation, *chan1* changes state from having one message in its queue to having two messages back to having one message in its queue. *chan1* changes state along the computation and therefore the message events associated with *chan1* are being fairly treated since as they *become possible infinitely often* they are *realized infinitely often.*

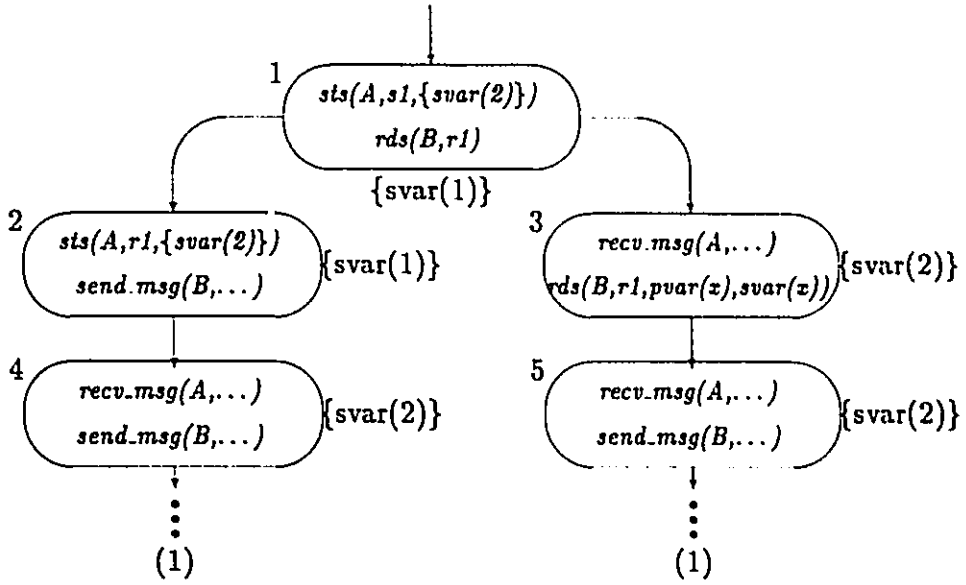
The other two channels, however, do not experience change along the computation. Both *chan2* and *chan3* each contain one message for the entire computation. Either of these non-progressing channels could indicate that a message event is being ignored on the computation. To determine if an event is being ignored we examine the non-deterministic branching points on the computation. We can see that at the branching point at state 2 the events that become possible are the *recv_msg(B,chan3)* and *recv_msg(B,chan1)* events. No message events for *chan2* become possible on the computation. Since no message events for *chan2* become possible infinitely often then obviously they cannot be realized infinitely often. Therefore the non-progressing state of *chan2* does not indicate unfairness on the computation.

For *chan3* we can see that a *recv_msg* event becomes possible infinitely often at the non-deterministic branching point, however, the message event is never satisfied because the system always follows the other branch alternative. Because an event is becoming possible infinitely often but is being satisfied only a finite number of times (exactly zero times) the computation (1,2,4) is unfair.

Again, to emphasize the notion of an event *becoming possible*, Figure 5.7 shows a *recv_msg(A,chan2)* event at system state 3. Because this event follows after state 2

at the branching point it does not become possible until after the event associated with state 2 has been satisfied. That is, along the computation the event *recv_msg(B,chan3)* becomes possible infinitely often but the event *recv_msg(A,chan2)* does not. Therefore, while the event *recv_msg(B,chan3)* is a source of unfairness for the computation, the event *recv_msg(A,chan2)* is not.

5.4.2 Fairness With Respect to Global Variables



Process Data States for process B showing the value of the process variable *pvar*:

1. ps(B,prms([]),{pvar(0)}) *2. ps(B,prms([]),{pvar(1)})* *3. ps(B,prms([]),{pvar(0)})*
4. ps(B,prms([]),{pvar(1)}) *5. ps(B,prms([]),{pvar(2)})*

Figure 5.8: A Computation with Global Variable Operations

To determine the fairness of a computation with respect to the reading and writing of global variables again we look for events that become possible but are never satisfied. Intuitively, an unfair computation with respect to the *rds* and *sts* operations is one along which the access to a global variable always assumes a certain ordering. That is, if we assume that an *rds* operation always precedes an *sts* operation on a

computation then the process executing the *rds* operation always assumes a certain value for the global variable. In assuming one ordering we have not allowed the possibility that a different value for the global variable could have been read if the other ordering was allowed, i.e., *sts* to modify the value then the *rds* to read the new value.

Figure 5.8 shows a part of some arbitrary state graph. On the graph we have noted at each state the process states for two of the processes in the system, *A* and *B*, and the value for a system condition variable *svar*. In this example both processes attempt to access the system condition (global) variable *svar*. Process A modifies the value of *svar* via an *sts* operation and process B reads the value of *svar* via an *rds* operation. We also show below the state graph the value of *pvar* the process condition variable to which process B assigns the value of *svar* via the *rds* operation.

The branching point (state 1) in the graph represents the non-determinism in the system behaviour resulting from the access to the system condition variable *svar*. One alternative branch (1,2,...) represents the behaviour where the *rds* operation precedes the *sts* operation. The other branch (1,3,...) represents the behaviour where the *rds* operation follows the *sts* operation. As we can see from the data state for process B the different orderings result in different values for the process variable *pvar*.

We note that on the state graph there are at least three separate computations, namely: (1,2,...,4), (1,3,...,5) and (1,2,...,4,1,3,...,5). Computation (1,2,...,4) represents the behaviour where the *rds* operation always precedes the *sts* operation while computation (1,3,...,5) represents the behaviour where the *rds* operation always follows the *sts* operation. Both of these computations are unfair since they allow only one of the two possible orderings of the *sts* and *rds* operations. The events that become possible at the branching points on the computations are the possible orderings of the operations. Along the computations (1,2,...,4) and (1,3,...,5) both orderings become possible but only one is satisfied. The computation (1,2,...,4,1,3,...,5) is fair since both possible orderings become possible and are satisfied.

To detect unfairness resulting from the access to system condition variables we do

the following:

1. Search for those states on the computation at which both an *sts* and an *rds* operation on the same system condition variable become possible. These system states represent non-deterministic branching points similar to state 1 of Figure 5.8;

2. For each of these branching states identify their next states resulting from both orderings of the execution of the *sts* and *rds* operations. In Figure 5.8 these would be system states 2 and 3.

3. Check the computation to see that both possible next states are members of the computation.

4. If both next states are members of the computation then both orderings must be allowed on the computation, therefore the computation is fair. If only one of the next states of a non-deterministic branching state is included in the computation then the computation must be unfair since the other next state representing the other possible ordering is not being satisfied.

Recall that the possibility for unfairness with respect to the access to global variables resulted from the non-deterministic execution rates of the processes that access the variables. That is, we do not know whether a process reading a system variable accesses that variable before or after a writing process has modified the variable value. With respect to this then, an unfair computation would be one which **always** assumed that the reader read before the writer modified or **always** assumed the opposite. Therefore, a fair computation must be one that includes both possible process interaction orderings.

5.5 Fairness in Practice

Now that we have a means of determining the fairness of a given computation how does this affect the model checking algorithm? We have two ways to proceed[QS83]: first, if ϕ is the set of all computations on the state graph model then we can derive a subset ϕ^f of ϕ such that ϕ^f contains only the fair computations of ϕ. We can then check the validity of temporal formulae on ϕ^f. A second way to proceed would be to modify the definitions of the temporal operators such that if the temporal formulae with the redefined operators are valid over ϕ then the same formulae using the normal operators are valid over ϕ^f. That is, we redefine the temporal operators so that their validity depends not only on the usual criteria but also on the fairness of the computation.

If we decide to add fairness constraints to the model checker by first removing all unfair computations then we must derive from the state graph the set of all possible computations ϕ and then test each for fairness in order to derive the subset ϕ^f. The task of searching the state graph to find all possible computations in itself represents a considerable amount of work considering the number of possible distinct computations on a fully connected graph with n nodes is $O((n-1)!)$. Since our state graph is only sparsely connected and is a directed graph it should have considerably less than $(n-1)!$ computations. However, the number of computations on a graph and the computational effort required to find them all would still be considerable.

A more efficient way to approach the problem is to redefine the temporal operators such that over all computations they give equivalent results to the normal operators over only fair computations. The main advantage of this approach is that not every computation needs to be checked for fairness. Only if during the model checking algorithm a computation is established does its fairness need to be determined. Complete computations are not always encountered during model checking because the truth value of formula under certain temporal operators can be established on a finite sequence of system states. For example, to prove the validity of $\exists\Diamond$f we need find only

one state reachable from the current state at which f is true, i.e., for this operator we do not need to show that f is true for all states along a computation.

Another benefit of this approach is that only those operators whose validity may depend on the fairness of a computation need be modified. For example, in the case of the formula $\forall\Diamond$f, if a computation is found along which f is never true, then this computation forms an argument that denies the validity of the formula on the state graph. i.e., it will cause the model checker to conclude that the formula is not satisfied on the state graph. However, if the computation is unfair then this computation should not form a basis for the failure of the formula. In other words, the validity of formulae under the $\forall\Diamond$ operator depends on the fairness of a computation. The validity of formulae under certain other temporal operators are independent of the fairness of a computation. In the next section we define six new temporal operators that automatically take fairness into account when determining the validity of temporal formulae on the state graph.

5.5.1 Temporal Operators That Consider Fairness

Our goal is to redefine the temporal operators such that the truth value of a formula over all computations is identical to the truth value of a formula using the normal operators over only fair computations. For each of the temporal operators the truth value of a formula at a particular state is defined on the computations starting at that state, that is, the computations form arguments to either support or deny the validity of the formula at that state. Based on this we must consider two possible outcomes that may result when unfair computations are encountered. First, that an unfair computation may be used as an argument to support the validity of a formula, and second, that an unfair computation may be used to deny the validity of a formula.

In the first case above, the validity of a formula should not be supported by an unfair computation and neither in the second case should the denial of a formula be supported by an unfair computation. We examine the semantics for each of the six

temporal operators below to determine whether their validity depends on the fairness of a computation.

In the following discussion we will consider the validity of a temporal formula at a state s_0 on the state graph. We define the following symbols to clarify the presentation:

ϕ — the set of all computations of which the state s_0 is a member;

ϕ^f — the set of fair computations of which s_0 is a member;

ϕ^u — the set of all unfair computations of which s_0 is a member; and

σ — a particular computation of states $s_0, s_1, \ldots$.

$\exists\Box$ — considered over the set ϕ of all computations of which the state s_0 is a member, a formula $\exists\Box$ f is valid at s_0 if there exists at least one computation $\sigma \in \phi$ such that f is always true on σ. However, if we wish to consider $\exists\Box$ formulae under conditions of fairness we need be concerned with the fairness of the computation σ. If $\sigma \in \phi^u$ then the computation cannot be used as an argument to support the validity of the formula. Therefore, we can conclude that under conditions of fairness the validity of a $\exists\Box$ formula depends on the fairness of the computation.

We can define a modified $\exists\Box$ operator such that over all computations it provides results identical to the normal operator over only fair computations. We define a new operator below:

$$\exists\Box^f \equiv \exists\Box \wedge \sigma \textit{ such that } \sigma \in \phi^f$$

This definition states that the validity of $\exists\Box^f$ depends on the validity of $\exists\Box$ and on the fairness of the computation. That is, for a $\exists\Box^f$ formula to be valid its equivalent $\exists\Box$ formula must be valid on a computation σ and σ must be a fair computation. Therefore, the checking algorithm for the $\exists\Box^f$ operator must add the test that the computation supporting the validity of $\exists\Box$ f must be fair.

For this operator, the fairness checking algorithm must be invoked once for each candidate computation.

$\forall\Diamond$ — a formula $\forall\Diamond$ f is valid at s_0 if for all computations in ϕ there exists a state s_{0+n} at which f is true. If a computation is found along which f is never true then this computation may form an argument to deny the validity of $\forall\Diamond$ f at s_0. If however, the computation is unfair then it cannot not be used to deny the validity of the formula. Therefore, under conditions of fairness the validity of a $\forall\Diamond$ formula depends on the fairness of the set of computations ϕ.

To make the operation of the $\forall\Diamond$ operator valid under the assumption of fairness we define a new operator below:

$$\forall\Diamond^f \equiv \forall\Diamond \vee \sigma \textit{ such that } \sigma \in \phi^u$$

This definition states that a formula $\forall\Diamond^f$ f is valid if either it is valid for the normal $\forall\Diamond$ operator or for the computations along which the formula under the normal operator was not valid those computations are unfair.

$\forall\Box$ — a formula $\forall\Box$ f is valid at s_0 if for all computations f is always true.

Suppose there exists a computation along which f is not always true then this computation forms an argument denying the validity of $\forall\Box$ f at s_0. However, what if the computation is unfair? We may wish to argue that since the computation is unfair it should not form an argument to deny the validity of the formula. We argue below that the fairness of a computation does not affect the validity of a $\forall\Box$ f formula.

1. An unfair computation is found containing a state s_{0+n} at which f is not true;
2. Because the state graph was generated by a fair scheduler (Section 5.3.2) every state on the graph must be a member of at least one fair computation;

3. Therefore, the state s_{0+n} where f was found to be false must also be a member of at least one fair computation;

4. Therefore, since the failure state belongs to both a fair and an unfair computation, fairness does not affect the validity of the formula.

In other words, the failure of $\forall\Box$ f on an unfair computation would be a redundant argument denying the validity of $\forall\Box$ f at the given state. i.e., the state where f was not true would also be found on a fair computation. Conversely, neither is the success of $\forall\Box$ f on an unfair computation a necessary condition supporting the validity of the formula. In order for the formula to be valid at a given state it must still be valid for all fair computations.

Since the validity of a $\forall\Box$ formula on an unfair computation can neither support nor deny the validity of the formula at the given state, the operation of the $\forall\Box$ operator does not change under the assumption of fairness. Therefore we can define a new operator:

$$\forall\Box^f \equiv \forall\Box$$

which is identical with the normal operator.

$\exists\Diamond$ — considered over the set ϕ of all computations of which the state s_0 is a member, a formula $\exists\Diamond$ f is valid at s_0 if there exists at least one computation σ such that f is true at s_{0+n}.

Suppose there exists an unfair computation containing a state s_{0+n} at which f is found to be true. Since the computation is unfair we may wish to argue that this computation should not support the validity of $\exists\Diamond$ f at s_0. However, we can form an argument similar to that presented for the $\forall\Box$ operator that shows that the validity of $\exists\Diamond$ f at s_0 is independent of whether σ is fair or not.

1. There exists an unfair computation containing a state s_{0+n} at which f is found to be true;

2. Every state on the graph must be a member of at least one fair computation;

3. Therefore, the state s_{0+n} must also be a member of at least one fair computation;

4. Since the state s_{0+n} is a member of both a fair and an unfair computation the validity of the formula $\exists\Diamond$ f is not affected by the fairness of the computation.

Since the operation of the $\exists\Diamond$ operator does not change under the assumption of fairness we can define a new operator:

$$\exists\Diamond^f \equiv \exists\Diamond$$

which is identical with the normal operator.

$\forall\bigcirc,\exists\bigcirc$ — the validity of formulae $\forall\bigcirc$ f and $\exists\bigcirc$ f at s_0 depends on the truth value of the immediate assertion f at the next state s_1 of each computation. Since s_1 must belong to at least one fair computation we know that the validity of $\forall\bigcirc$ and $\exists\bigcirc$ formulae is unaffected by the fairness of the computation. Therefore, we can define fair next-state operators below:

$$\forall\bigcirc^f \equiv \forall\bigcirc$$

$$\exists\bigcirc^f \equiv \exists\bigcirc$$

which are identical to the normal next-state operators

An interesting point to note here is that the $\forall\Box$ and $\exists\Diamond$ operators and the $\exists\Box$ and $\forall\Diamond$ operators are duals. That is,

$$\forall\Box\mathbf{f} \equiv \neg\exists\Diamond\neg\mathbf{f}$$

and

$$\exists\Box\mathbf{f} \equiv \neg\forall\Diamond\neg\mathbf{f}$$

Therefore, we should expect that if the fairness of a computation affects the validity of a certain temporal operator it should affect the validity of the operator's dual as well. If we review the definitions for the fair temporal operators we find that this is the case.

5.5.2 Complexity Assuming Fairness

The complexity of the algorithm to check the fairness of a computation is $O(n^2)$, where n is the number of states in the computation. To show how this order of complexity was determined we can see that the algorithm to generate the two lists, the list of *events that become possible* and the list of *events that are satisfied*, has a complexity that is linear with the number of states in the computation. However, the algorithm to compare the two lists has a worst case complexity of n^2. If at each state an event becomes possible then the *becomes possible list* will contain n elements. If at each state an event not in the *becomes possible* list is satisfied then the *satisfied* list will contain n different elements. To determine that the two lists are unequal will require n searches through n items, giving a complexity on the order of n^2.

The only operators whose complexity is affected by adding fairness constraints are the $\forall\Diamond$ and $\exists\Box$ operators. In the worst case either operator would have to determine the fairness of all computations on the state graph. A non-directed graph with n fully connected nodes has $O((n-1)!)$ computations. Therefore, the worst case complexity for the $\exists\Box$ and $\forall\Diamond$ operators under fairness is

$$O((CARD(\mathbf{S}) + CARD(\mathbf{R})) + (\mathbf{S}-1)!)$$

where $\mathbf{S}$ is the number of states in the graph and $\mathbf{R}$ is the number of paths between states.

However, since our state graph is only sparsely connected and it is a directed graph the actual number of computations in the graph should be considerably less than $(\mathbf{S}-1)!$. As well, only those computations which could affect the validity of a formula need be tested for fairness.

5.6 Chapter Summary

The model checking approach developed by Clarke, et.al., as reported in [CES83] [CBES85] [Bro86] [CES86] provides an efficient means of verifying the correctness of a global state graph model of a system's behaviour against a specification expressed in BTL. Based on the method outlined in [CES83] [CBES85] [Bro86] [CES86] we developed model checking algorithms for each of our six BTL operators. We also defined predicates for PAL which allow immediate assertions to be made on the state variables of a PAL system. Thus we have a means of specifying correctness properties in BTL for a given PAL system and we have a means of verifying that these properties are satisfied on a state graph model of the system's behaviour.

In Section 5.3 we developed a means, based on the non-deterministic elements of PAL semantics, of detecting an unfair computation on the state graph. We also showed how the model checking algorithms for each of the six BTL operators would have to be modified in order for fairness to be considered when verifying a formula on the state graph.

Chapter 6

Implementation

In this chapter we discuss the practical issues of implementing and using the verification system developed in the previous two chapters. Our aim here is show how the two basic elements of our verification approach, the state graph generator and the model checker, are implemented and integrated into a complete tool to aid in system development with PAL. Our objective in providing an implementation is first, to demonstrate that our verification method is viable, and second to better understand how such a verification system can be used to aid in system development. We want to show that the method could form the basis of a practical tool for use in the development of actual PAL systems.

We will not attempt to describe the complete implementation in detail. Only those aspects that will show how certain implementation problems are handled and those elements that will aid in an understanding of how the verifier can be used to aid development are discussed. A listing of the Prolog source code and an introduction to the commands and features of the complete implementation can be found in [Har89], a brief user's guide.

6.1 Overall Structure

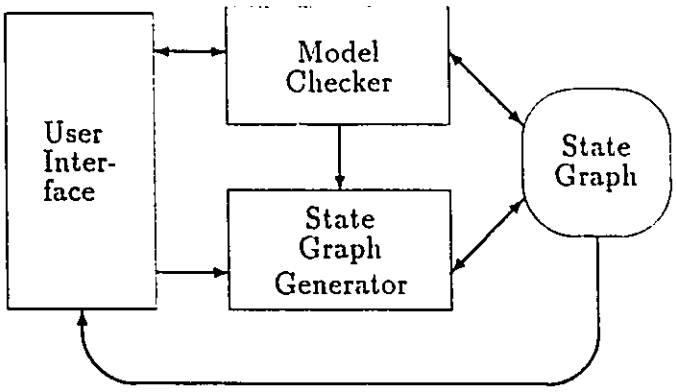


Figure 6.1: Verifier Block Diagram

The overall structure of the verification tool is shown in block diagram form in Figure 6.1. The arrows show the direction of information flow among the various elements. The state graph generated by the generator module is accessible to both the model checking module and the user interface module as will be explained in Section 6.5.

To show how the three modules of the system interact we present the following scenario:

1. The user loads into the verifier a file containing a set of connective implications and an initial state assertion describing a PAL system;

2. The user specifies a correctness property against which he wants to verify the PAL system;

3. The model checking algorithm is invoked to check the PAL system against the correctness property;

4. The model checker calls on the state graph generator to build the state graph model of the system from the connective implications. In building the state graph the generator also checks for system wide deadlock states;

5. When the model checker has determined that the property is or is not satisfied on the state graph model it reports its results to the user;

6. Based on the information produced by the model checker the user interface attempts to explain to the user why or why not the property was or was not satisfied on the state graph model. To aid in assessing the quality of the PAL system the user interface provides facilities to trace execution paths and examine variables at each state. In this way the user can follow the execution history up to the point where a property fails or succeeds thus allowing him to better understand the dynamics of the problem.

The role of the user interface module, as its name implies, is to provide a means for the user to interact with the model checker in a way that is comfortable for the user. The interface provides a user friendly environment for the verification of PAL specifications. The model checking module assists in this role by saving key information during the model checking process that will allow the interface to explain the verification results and assist in finding errors.

We choose to implement the verifier totally in Prolog; specifically MProlog[1] running on a VAXStation on the VMS[2] operating system. Our choice of Prolog as an implementation language is based on the following factors:

- Both Prolog and temporal logic have a common basis in the theory of classical logic. Prolog is a practical realization of the theory of logic programming while temporal logic is an extension of the normal classical logic. To make deductions from a database of logical terms Prolog employs practical implementations of the theories of *resolution* and *unification*. Similarly, in Chapter 4 we applied the same theories to deduce new system states from a database of temporal implications and facts. By using Prolog as an implementation language we can

[1] MProlog is a product of LogicWare Inc.
[2] VAXStation and VMS are products of Digital Equipment Corporation

take advantage of Prolog's built in unification and resolution algorithms as a base on which to build our LTL based state graph generator.

As well, we note a similarity between the model checking algorithm and the resolution algorithm employed by Prolog. Both algorithms perform an ordered, depth-first search of a data structure; the data structure being a state graph in the case of the model checking algorithm and the facts and rules of a Prolog program in the case of Prolog's resolution algorithm. By implementing our system in Prolog we can use Prolog's built in resolution mechanisms. Unfortunately, the search order and the realization of the success or failure of the search differs considerable between the two algorithms. The search order used by the Prolog's resolution algorithm is based on the ordering of clauses in the program and on the ordering of goals within the clauses, while the model checker's search order is defined by the structure of the graph itself. Thus, we must somehow make Prolog search the state graph according to the requirements of the model checking algorithm. Fortunately Prolog's meta-programming features will allow us to easily redefine the search ordering to suit our purposes while still retaining the underlying resolution mechanism.

- The nature of the Prolog language makes it especially suitable for rapid development of experimental systems. Prolog requires only statements of fact and relationship describing a particular problem. Prolog's built in inference mechanism itself determines how to solve the problem. This aspect of Prolog will allow us to concentrate on problems with the verification method itself rather than on the details of implementation.

As was stated in the opening paragraphs to this chapter, our aim in developing an implementation is to demonstrate the viability and general usefulness of our verification approach. In concert with this our verification system implementation concentrates on function rather than efficiency. Therefore we can accept the limitations imposed by a Prolog implementation. Being an interpreted language the

implementation may be relatively slow, and as well, the Prolog data structure and inference mechanisms are not very memory efficient; a fact which will limit the size of the problems that the verifier will be able to handle. However, these are limitations of the implementation method, not necessarily of the verification method. If necessary the same verification methods could be re-implemented with a greater emphasis on execution speed and efficient memory usage.

In the following sections we discuss implementation aspects of each of the major elements of the verification system. As was stated in the introductory paragraph to this chapter, we will not attempt to provide complete details of the implementation. Complete Prolog source code for the verification system can be found in [Har89].

6.2 Data Structures

We begin describing the Prolog implementation by showing how the logic terms used in our PAL Temporal Logic Representation (PTR) system can be converted into their Prolog equivalents. Because of the inherent similarity between the logic terms used in PTR and Prolog data structures most of the conversion can be achieved by following a simple set of rules given below.

rewrite all terms using only ASCII characters — The main effort in converting our logical terms into Prolog data structures consists simply of replacing the symbols used in our representation with symbols from the ASCII character set. Table 6.1 below shows how certain of the symbols we have been using are represented in Prolog. The symbols given for the BTL operators are those used in [Pnu81]. As well, subscripts, superscripts, boldface, italics characters, etc, must all be replaced by characters from the normal ASCII set.

represent arrays of items as Prolog list structures — In the development of our method we have used arrays to represent related sets of data items. The equivalent data

logical operators	$\Rightarrow$	—	==>	$\wedge$	—	&
BTL operators	$\forall\Box$	—	ag	$\exists\Box$	—	eg
	$\forall\Diamond$	—	af	$\exists\Diamond$	—	ef
	$\forall\bigcirc$	—	ax	$\exists\bigcirc$	—	ex
LTL operators	$\bigcirc$	—	*	$\Diamond$	—	<>

Table 6.1: Prolog Form of Operators

structure in Prolog is the *list*, [...]. An empty list, equivalent to an array of size zero, is denoted by [].

For example, the array of variable values for a process state becomes a Prolog list, i.e.,

$$\{var_1(3), var_2(false)\}$$

becomes

```
[var1(3),var2(false)]
```

all facts and rules are delimited by a period "." character

variable names are capitalized but arguments with known values are not — Variables in Prolog are represented by capitalized names while specific values are not capitalized. For example, the array of variable values representing pre- or post-conditions for a connective implication

$$\{var_1(2), var_2(x), @less(x, 2)\}$$

will be represented in Prolog as

```
[var1(3),var2(X),@(X < 2)]
```

In the following sections we apply the above rules to form Prolog implementations for the major data structures of the verification system.

6.2.1 State Graph Representation

A system state graph consists of a set of system state assertions (SSA) and a set of relation assertions. The system state assertions give the state of the system at a particular point in its execution history while the relation assertions define the structure of the graph by giving for each SSA a list of next SSA. To implement the graph in Prolog we need only define the two terms using Prolog syntax.

System State Assertions

The general form of an SSA in Prolog is given below.

```
state(state number,
      [list of process states],
      [list of global variable values],
      [list of channel states]).
```

Each term in the `list of process states` is of the form:

```
state_predicate & ps(P,[[],Vars])
```

Where `state predicate` is any of the process state predicates defined in Chapter 4, Sections 4.3.1 and 4.5. The variable `P` is the name of the process and `Vars` is a list of process variables and their values at this state. For example:

```
init(a) & ps(a,[[],[level(3),quit(false)]])
```

is the state for process a. Process a has two variables, `level` and `quit` whose values in this state are 3 and `false` respectively.

The `list of global variable values` is a Prolog list containing the values of each global variable in the system. For example, if `temp`, `name` and `age` are global variables then their values in the state are given in the following list:

```
[temp(37),name(fred),age(27)]
```

Each element in the `list of channel states` is of the form:

```
chan(Name,[list of messages],N,M)
```

Where the `list of messages` is a list representing the channel queue, `N` is the number of messages in the queue and `M` is the capacity of the queue. An example of a channel state is:

```
chan(chan23,[[ack,1],[reply],[false]],3,15)
```

The initial system state has a form similar to the other system states. For example, the initial state for the two process example system of Figure 4.4 would be:

```
init_state([init(a) & ps(a,[[],[]]),init(b) & ps(b,[[],[]])],
           [],
           [chan(chan1,[],0,1),chan(chan2,[],0,1),
            chan(chan3,[],0,1),chan(chan3,[],0,1)]).
```

Relation Assertions

Relation assertions define the structure of the system state graph by giving for each SSA a list of next SSA. We will use the predicate `tree/2`[3] to provide the relation information for each SSA. For example:

```
tree(3,[6,5,4]).
tree(6,[7]).
tree(5,[8]).
```

asserts that the next states of state 3 are states 6, 5 and 4 and that the next state of state 6 is state 7, etc.

Thus, in our implementation a state graph will consist of a database of Prolog facts; a set of `state/4` facts will give the state information for each SSA on the graph,

[3]We will use the notation predicate_name/n to refer to specific Prolog predicates. The /n indicates that the named predicate has n arguments.

and a set of `tree/2` facts will define the structure of the graph by giving a relation between each SSA and its next SSA.

6.2.2 Connective Implications

Connective implications are represented as Prolog facts of the form `A ==> B.`, where `A` and `B` represent respectively, the antecedent and the consequent of the connective implication. The general form of a connective implication term in Prolog is as follows:

```
state_predicate1 & ps(P,[Prms1,Pre-conditions]) ==>
                *<> (state_predicate2 & ps(P,[Prms2,Post-conditions])).
```

The `state predicate1` and `state predicate2` terms represent the control state of the process and can take on any of the predicates defined in Chapter 4, Sections 4.3.1 and 4.5. The `Prms1` and `Prms2` terms are Prolog lists containing the message parameters that are passed via the message passing constructs. The `pre-conditions` and `post-conditions` terms are lists of variable values forming the pre- and post-conditions of the implications. Connective implications for the simple two process example problem from Chapter 4 (Figure 4.4) are given in Figures 6.2 and 6.3.

Note that the $\Rightarrow$, $\bigcirc$ and $\Diamond$ symbols have been replaced by their Prolog equivalents as given in Table 6.1.

6.3 Generator Implementation

A block diagram of an inference engine to implement the state graph generator module is shown in Figure 6.4. The generator consists of three main parts as described below:

PTR Interpreter — This part contains the actual inference engine whose behaviour is governed by the PAL language implications, LTL inference rules and the inference algorithm with which to apply them to deduce new system states.

```
(a1) init(a) & ps(a,[[],[]])
     ==> *<>(send_msg(a,chan1) & ps(a,[[req],[]])).

(a2) end_send(a,chan1) & ps(a,[[],[]])
     ==> *<>(recv_msg(a,chan2) & ps(a,[[],[]])).

(a3) end_recv(a,chan2) & ps(a,[[ack],[]])
     ==> *<>(recv_msg(a,chan3) & ps(a,[[],[]])).

(a4) end_recv(a,chan3) & ps(a,[[req],[]])
     ==> *<>(send_msg(a,chan4) & ps(a,[[ack],[]])).

(a5) end_send(a,chan4) & ps(a,[[],[]])
     ==> *<>(init(a) & ps(a,[[],[]])).
```

Figure 6.2: Connective implications for process A

```
(b1) init(b) & ps(b,[[],[]])
     ==> *<>(recv_msg(b,chan1) & ps(b,[[],[]])).

(b2) end_recv(b,chan1) & ps(b,[[req],[]])
     ==> *<>(send_msg(b,chan2) & ps(b,[[ack],[]])).

(b3) end_send(b,chan2) & ps(b,[[],[]])
     ==> *<>(send_msg(b,chan3) & ps(b,[[req],[]])).

(b4) end_send(b,chan3) & ps(b,[[],[]])
     ==> *<>(recv_msg(b,chan4) & ps(b,[[],[]])).

(b5) end_recv(b,chan4) & ps(b,[[ack],[]])
     ==> *<>(init(b) & ps(b,[[],[]])).
```

Figure 6.3: Connective implications for process B

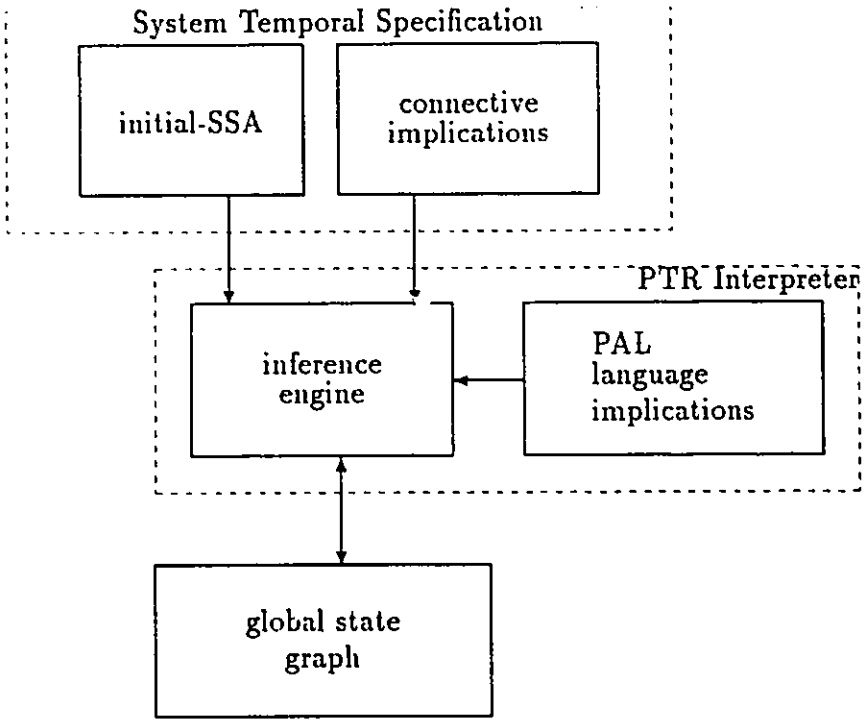


Figure 6.4: Inference Engine Block Diagram

System Temporal Specification — This part consists of the connective implications and the initial-SSA; the temporal representation of the PAL system. This part can be considered as the input to the PTR interpreter.

Global State Graph — This part is composed of a database of system states, `state/4`, and `tree/2` facts defining the structure of the state graph. The two way arrow indicates that the state graph is produced as output by the PTR Interpreter but is also used by the interpreter during the inference process.

6.3.1 State Graph Generation

The role of the state graph generator is to transform the connective implications representing a PAL system into a state graph model of the system. In implementing the generator to carry out this role we must consider how best to integrate the building of the state graph with the checking of the state graph. Basically, we have two options; either generate the complete state graph first and then pass the graph to the model checker or, generate the state graph only as it is required by the model checker. In both cases, to prove that a property is completely satisfied on the state graph will require that the complete state graph be generated. However, in the early stages of development it would be safe to assume a greater frequency of errors in the PAL expressions. It would, therefore, be inefficient to generate the complete state graph only to find an error within the first 100 states. A major benefit offered by the incremental approach is that only the states required to prove or contradict a correctness property need be generated. As soon as it has been shown that a property is not satisfied on the existing states of a graph then there is no further need to generate the rest of the graph. For these reasons our approach is to construct the state graph on a state by state basis as required by the model checker.

The model checker initiates the building of the state graph by calls to a `next states/2` predicate whose clauses are shown in Figure 6.5. The line numbers given to the left of each line are included for explanation purposes only, they are not part of the Prolog

```
1  next_states(S,B) :-
2          tree(S,B),!,
3          (B=[],
4           add_statement(stop),
5           add_statement(return_status(deadlock,S)),-;
6          succeed).
7  next_states(S,B) :-
8          generate_new_states(S,B),
9          (B=[],
10         add_statement(stop),
11         add_statement(return_status(deadlock,S)),
12         !,fail;
13         succeed).
```

Figure 6.5: The next states/2 predicate

syntax. The flow pattern for this predicate is next_states(i,o). The flow pattern indicates which arguments are known when the predicate is called, i.e., the *inputs*, i, and which arguments are returned by the predicate, i.e., the *outputs*, o. Thus, the next states/2 predicate is given an existing state S for which the model checker wants to know the names of the next states B of S. If the next states of S have already been generated then the tree/2 predicate in the first clause will find the required information in the database, otherwise, the second clause at line 7 invokes the state graph generator via generate_new_states/2 to generate the next states of S.

Lines 3-6 and 9-13 handle the case where there are no next states of the current state, i.e., S is a deadlock state. If S is a deadlock state then the model checker is terminated and the reason for the termination is saved to allow the user interface to explain the failure.

The generate_new_states/2 predicate called in line 8 causes the state graph generator module to deduce the next states of the current state and to add them to the state graph. The next states are deduced by a set of transformation rules incorporating both the LTL inference rules and the PAL language implications. The transformation rules operate by taking a current state and transforming it into a next

state by performing a substitution on one of the process states. The process state substitution represents the progress of that process through one indivisible operation. There is one transformation rule for each of the possible control states that a process could be in in the current state.

If one of the processes in the current state is at a non-deterministic control construct (`select,rds,sts`) then a second set of transformation rules are used which perform expansion on the current state. The expansion rules generate next states by performing as many different substitutions on the current state as there are non-deterministic execution paths possible from that state.

6.3.2 Recording Density

In generating the state graph for a PAL system we need to strike a compromise between providing the information needed by the model checker to check the validity of a formula and the limits on how much information can be stored in memory at one time.

The state graph generated by the state graph generator describes the temporal behaviour of the PAL system under analysis. We would like to be able to represent the system's temporal behaviour with as small a state graph as possible as this would allow a higher upper limit on the size of the problem that we could handle.

On the other hand, to be able to define meaningful correctness properties for the system requires a certain amount of information in the state graph. For each state that is removed from the state graph we lose the ability to make assertions about that state.

In [Kar87] Karam used the term *recording density* to describe the amount of information in a state graph model of a system. A high recording density means that all possible system states are included in the state graph. Conversely, a state graph with a low recording density would contain the minimum number of states required to completely describe the system. For the model checker, a high recording

density would allow more complex correctness properties to be stated and checked, however, a lower recording density would allow larger systems to be analysed. In our implementation we will attempt to achieve the lowest recording density possible at which we can still form meaningful correctness properties for the system.

We have already taken steps to minimize the size of the state graph by automatically pruning equivalent state sequences from the graph (Section 4.4.1). To further limit the size of the graph the inference engine generating the system states will consider the transition of a process from the end of one instruction to the beginning of the next instruction to be one indivisible operation. For example, for the two connective implications below:

```
init(a) & ps(a,[[],[]]) ==> *<>(recv_msg(a,chan1) & ps(a,[[],[]])).
end_recv(a,chan1) & ps(a,[[ack],[]]) ==>
    *<>(send_msg(a,chan2) & ps(a,[[reply],[]])).
```

If process a is currently in the recv_msg state then the transition from the recv_msg state to the end recv state to the send msg state is considered to be one indivisible operation. Thus the inference engine would record the system state at which process a is in the recv_msg state and the next state recorded would be the state where process a was in the send msg state. Thus, the temporary transition states *end recv*, *end send*, etc, will not be recorded.

By not recording the temporary transition states we will lose the ability to express such properties as:

```
st(recv_msg(a,chan1)) ==> af st(end_recv(a,chan1)).
```

which specifies that if process a is in the recv_msg state then eventually it must receive a message and exit the recv_msg state. However, we can still express such liveness properties by using other available state information. For example,

```
st(recv_msg(a,chan1)) ==> af st(send_msg(a,chan2)).
```

expresses the same liveness property as the previous formula. If process a must eventually reach the send_msg state then this implies that it must eventually receive a message and exit the recv_msg state.

6.4 Model Checker Implementation

Recalling the general form of the model checking algorithms given in Section 5.1.1.2 we note that each algorithm consisted basically of two parts: a part to check the validity of a BTL term at a particular state and a part to ensure that the validity of the BTL term is checked at every state in the graph. For a property to be valid for the system in general it must be valid at all states in the graph. Our implementation of the algorithms for each of the operators will follow the same pattern. In the following sections we will present and explain six sets of Prolog clauses with each set implementing the model checking algorithm for one of the BTL operators. Each set of clauses is designed to perform a search of the state graph to determine the validity of a BTL term at a particular state, i.e., state S. A set of clauses common to all the model checking clauses will ensure that the appropriate model checking clauses are invoked for each of the states in the graph thus proving the validity of the BTL term for the complete system.

The ∀□ operator

The Prolog clauses implementing the model checking algorithm for the ∀□ onerator is given in Figure 6.6. These clauses are a direct implementation of the algorithm for the function **check** given in Chapter 5 (Figure 5.2).

The checking algorithm is invoked by a call to the check/2 clause at line 1 which passes the BTL term, ag(F), and the state at which the validity of the this term must be checked, S. The check/2 clause then calls the main part of the checking algorithm implemented by the three check/3 clauses at lines 4, 6, and 9.

```
 1  check(ag(F),S) :-
 2        check(ag(F),[],S),
 3        label_state(ag(F),S).
 4  check(ag(F),_,S) :-
 5        visited1(S,ag(F)).
 6  check(ag(F),_,S) :-
 7        label(S,L),
 8        member(ag(F),L).
 9  check(ag(F),Path,S) :-
10        (check(F,S);
11           add_statement(result(ag(F),Path,S)),-),
12        add_statement(visited1(S,ag(F))),
13        next_states(S,B),!,
14        all_paths(ag(F),B,[S|Path]).
```

Figure 6.6: Prolog Clauses for the $\forall\Box$ Operator

The second argument in the check/3 clauses, Path, is a list of system states representing a trail or search history of the states visited along the current execution path starting at S. This path history is not actually used in the model checking algorithm for the $\forall\Box$ operator but is updated for use by the user interface. The call invoking the check/3 clauses at line 2 initializes this argument as an empty list.

The three check/3 clauses implement the main part of the checking algorithm for the $\forall\Box$ operator. The first two check/3 clauses ensure the termination of the algorithm (i.e., the search through the state graph) by identifying states that have already been visited or labelled with the validity of the formula ag(F). The check/3 clause at line 4 checks to see if a state has previously been visited. If a state has already been visited then the search along that execution path is terminated. The check/3 clause at line 6 checks to see if the validity of ag(F) is already known at the current state. If so, then the search along that execution path can be terminated as well.

The check/3 clause starting at line 9 is a direct implementation of the fixpoint characterization of the $\forall\Box$ operator, that is:

$$\forall\Box f \equiv f \wedge \forall \bigcirc \forall\Box f$$

For a $\forall\Box$f term to be valid at state S, f must be valid at S and $\forall\Box$f must be valid for all successor states of S. The third check/3 clause is a direct implementation of this characterization of the operator. The first predicate of the third check/3 clause (line 10) checks that the immediate assertion part, F, of the temporal formula is satisfied in the current state. This corresponds to the f term in the conjunction in the above formula. The next_states/2 predicate at line 13 finds all the next states of state S and line 14 recursively calls the checking algorithm for each next state in the list. This corresponds to the second conjunctive term in the above formula.

To ensure that each state only needs to be visited once line 12 asserts the fact that the current state has been visited. If the immediate assertion F is not satisfied in the current state then line 11 asserts a diagnostic fact which is used by the user interface to help the user locate where in the state graph the correctness property failed.

Remember that the above clauses and the clauses implementing the other temporal operators are designed to determine the validity of a formula at any one given state, i.e., state S. To check the validity of the formula for all states requires calling the check/2 clauses for each state in the graph. This is accomplished by a set of driver clauses common to all the model checking clauses.

The $\exists\Diamond$ operator

The Prolog implementation of the $\exists\Diamond$ operator is given in Figure 6.7. Again, the main part of the algorithm is derived directly from the following fixpoint characterization of the $\exists\Diamond$ operator:

$$\exists\Diamond f \equiv f \vee \exists \bigcirc \exists\Diamond f$$

```
 1  check(ef(F),S) :-
 2        check(ef(F),[],S).

 3  check(ef(F),_,S) :-
 4        visited2(S,ef(F)),-.
 5  check(ef(F),_,S) :-
 6        label(S,L),
 7        member(ef(F),L).
 8  check(ef(F),Path,S) :-
 9        check(F,S),!,
10        label_state(ef(F),S),
11        add_statement(result(ef(F),Path,S)).
12  check(ef(F),Path,S) :-
13        add_statement(visited2(S,ef(F))),
14        next_states(S,B),!,
15        some_path(ef(F),B,[S|Path]),
16        label_state(ef(F),S).
```

Figure 6.7: Prolog Clauses for the $\exists\Diamond$ Operator

The single check/2 clause at line 1 initializes the search history argument, Path to the empty list and calls the check/3 clauses to perform the search to determine the validity of ef(F) at S. The first two check/3 clauses ensure that the search through the state graph will terminate by identifying states that have already been visited or those states at which the validity of ef(F) is already known.

The second two check/3 clauses directly implement the fixpoint characterization of the $\exists\Diamond$ operator. These two clauses correspond respectively to the first and second disjunctive terms from the fixpoint formula above. The first clause checks to see if the immediate assertion part of the formula F is satisfied in the current state. If F satisfied in the current state then the graph is labelled and some diagnostic information is saved for use by the user interface. If F is not satisfied in the current state then the check/3 clause at line 12 attempts to find a next state at which it is satisfied.

The Prolog implementations of the checking algorithms for the $\exists\Box$ and $\forall\diamond$ operators are discussed in Section 6.4.4 in the context of fairness. Prolog clauses implementing the $\forall\bigcirc$ and $\exists\bigcirc$ operators are given in Appendix D.

The model checking algorithms for each of the temporal operators depend on an ability to perform a depth-first search of the state graph model of the system under analysis. The following two predicates, `some_path/3` and `all_paths/3`, together with the clauses for each temporal operator, control the search through the state graph.

6.4.1 all_paths/3

The `all paths/3` predicate is called by the universally quantified BTL operators $\forall\Box$, $\forall\diamond$ and $\forall\bigcirc$ to ensure that a property holds at all of the next states of a current state. The clauses for the `all_paths/3` predicate are given below.

```
1  all_paths(F,[],_).
2  all_paths(F,[S|R],Path) :-
3        check(F,Path,S),!,
4        all_paths(F,R,Path).
```

The flow pattern for this predicate is `all paths(i,i,i)`. The first argument passes the temporal formula that must be checked. The second argument is a list of all the next states at which the formula must be checked and the third argument is a list of each state along the current path that has previously been visited in checking the formula.

Line 3 calls the appropriate checking algorithm to check the formula for each next state to the current state. When all next states to the current state have been checked the `all_paths/3` clause at line 1 succeeds and causes a return to the point where `all_paths/3` was last called.

6.4.2 some path/3

The some_path/3 predicate is called by the existentially quantified BTL operators $\exists\Box$, $\exists\Diamond$ and $\exists\bigcirc$ to ensure that a property holds at least one of the next states of a current state. The clauses for the some path predicate are given below.

```
1  some_path(F,[],_) :- !,fail.
2  some_path(F,[S|R],Path) :-
3          (check(F,Path,S);
4          !,some_path(F,R,Path)).
```

The same arguments are passed to the some_path predicate as are passed for the all_paths/3 predicate. The clauses for some_path attempt to find *some path* along which the formula F is satisfied. In line 3 check/3 is called to see if F is satisfied at one of the next states to the current state. If not, then some_path/3 is called recursively in line 4 to try another next state. If F is not satisfied at any of the next states of the current state then the some path/3 clause at line 1 causes a non-backtrackable failure of the predicate.

6.4.3 Model Checking Compound Formulae

The following clauses allow the model checker to handle formulae of arbitrary complexity by breaking a compound formula into its constituent terms:

The four check/2 clauses handle the four logical connectives ($\Rightarrow$,$\wedge$,$\vee$,$\neg$) of which logical formula are composed. These simple clauses serve to decompose a complex BTL formula into simple BTL terms. The validity of each of the terms is checked using the appropriate model checking algorithms. For example, for the BTL formula:

```
ag st(recv_msg(a,chan1)) and ef st(send_msg(b,chan2))
```

the check/2 clause at line 4 would decompose the formula into two BTL terms, F1 and F2, corresponding to the two conjunctive terms in the formula. The check/2

```
 1  check(F1 ==> F2,S) :-
 2        (not(check(F1,S));
 3        check(F2,S)).
 4  check(F1 and F2,S) :-
 5        check(F1,S),!,
 6        check(F2,S).
 7  check(F1 or F2,S) :-
 8        (check(F1,S);
 9        check(F2,S)).
10  check(~(F),S) :-
11        not(check(F,S)).
```

Figure 6.8: Clauses for Handling Logical Connectives

clause at line 5 would check the validity of F1 at state S. If F1 is valid at S then the validity of F2 would be checked. If both F1 and F2 are valid at S then the complete formula is valid at S.

6.4.4 Fairness

In Section 5.3 we defined six new temporal operators which allow us to accommodate fairness constraints into the model checking algorithm. We determined that the definitions of the $\forall\Box^f$, $\exists\Diamond^f$, $\forall\bigcirc^f$ and $\exists\bigcirc^f$ operators were equivalent to their unfair counterparts and therefore the same model checking algorithms can be used for both. However, we had to redefine the $\exists\Box$ and $\forall\Diamond$ operators such that they give valid results under the assumption of fairness. This requires that the model checking algorithm for the $\forall\Diamond^f$ and $\exists\Box^f$ operators incorporate additional checks to determine the fairness of the computations in the state graph.

Instead of implementing the model checking algorithms for both versions (fair and unfair) of the two operators separately we combined them into the same implementation. Thus, both the fair and unfair versions of the $\forall\Diamond$ operator are implemented with the same set of clauses and the fair and unfair versions of the $\exists\Box$ operator are both implemented with a second set of clauses. To accomplish this we allow the additional

checks for fairness to be turned on or off depending on which version of the operator is desired.

The $\exists\Box$ Operator

```
 1  check(eg(F),S) :-
 2        check(eg(F),[],S).
 3  check(eg(F),_,S) :-
 4        label(S,L),
 5        member(eg(F),L).
 6  check(eg(F),Path,S) :-
 7        member(S,Path),
 8        tree(S,[_]),
 9        (check_fair(S,Path),
10         add_statement(result(eg(F),Path,S));
11             !,fail).
12  check(eg(F),Path,S) :-
13        check(F,S),
14        next_states(S,B),!,
15        some_path(eg(F),B,[S|Path]),
16        label_state(eg(F),S).
```

Figure 6.9: Prolog Clauses for the $\exists\Box$ Operator

Prolog clauses implementing the model checking algorithm for both the $\exists\Box$ and $\exists\Box^f$ operators are given in Figure 6.9. Again, we notice that these clauses follow the same pattern as we have already seen for the $\forall\Box$ and $\exists\Diamond$ operators. The fixpoint characterization for the $\exists\Box$ operator is:

$$\exists\Box\mathbf{f} \equiv \mathbf{f} \wedge \exists\bigcirc\exists\Box\mathbf{f}$$

In the above code we can see that the check/3 clause at line 12 implements the fixpoint representation for the operator. Line 13 checks that the immediate assertion F is valid in the current state and line 15 checks that eg(F) is valid for at least one next state of the current state.

The check/3 clauses at lines 3 and 6 cause the search algorithm to terminate if either a state is encountered at which eg(F) is already known to be true (line 3) or if a computation is established (line 6). If a state is reached which is a member of the current search path, Path then a loop of states or computation has been established. The check_fair/2 predicate at line 9 passes the states comprising the computation to a fairness checking routine. If the computation is fair then it can be used as an argument to support the validity of eg(F) and the algorithm terminates. If the computation is unfair then the search continues for another computation along which F is always true.

As was mentioned above, the clauses in Figure 6.9 implement the model checking algorithms for both the $\exists\Box$ and the $\exists\Box^f$ operators. The fairness checking constraint, the check_fair/2 predicate at line 9, ensures that only fair computations are used to support the validity of eg(F) at S. To remove this constraint the check_fair/2 predicate has been designed such that it can be turned off. That is, if the unfair $\exists\Box$ operator is required then check_fair/2 will always report that the computation is fair. This effectively turns off the fairness constraints.

The $\forall\Diamond$ Operator

Prolog clauses implementing the model checking algorithm for both the $\forall\Diamond$ and $\forall\Diamond^f$ operators are given in Figure 6.9. The check/3 clauses at lines 14 and 17 are direct implementations of the fixpoint representation for the af operator. The immediate assertion must either be true in the current state (line 14) or must be true at some state along the state sequences starting at each next state of the current state (line 17).

The check/3 clauses at lines 4 and 7 handle the cases where the validity of af(F) is already known at the current state or the current state has already been visited. If the validity of af(F) is already known at the current state then there is no point in continuing the search. This case is handled by the clause at line 4.

```
 1  check(af(F),S) :-
 2       check(af(F),[],S),
 3       label_state(af(F),S).
 4  check(af(F),Path,S) :-
 5        label(S,L),
 6        member(af(F),L).
 7  check(af(F),Path,S) :-
 8        member(S,Path),
 9        tree(S,[_]),
10        (check_fair(S,Path),
11         add_statement(result(af(F),Path,S)),
12         !,fail;
13             succeed).
14  check(af(F),_,S) :-
15        check(F,S),!,
16        label_state(af(F),S).
17  check(af(F),Path,S) :-
18        next_states(S,B),!,
19        all_paths(af(F),B,[S|Path]),
20        all_paths_labelled(af(F),B,S).
```

Figure 6.10: Prolog Clauses for the $\forall\Diamond$ Operator

The check/3 clause at line 7 handles the case where we have encountered a state that is a member of the current search path, i.e., a loop of states or computation has been established. The search along a branch of the state graph will continue until it finds a state at which the immediate assertion F is true. Since the search has established a computation then this must imply that F is not true at any of the states on the computation. If the computation is fair then we have found an argument to deny the validity of af(F) at state S. A call to the check_fair/2 predicate, line 9, determines whether the computation is fair or not. If the computation is fair then the algorithm terminates and it is concluded that af(F) is not valid on the state graph. If the computation is unfair then it is ignored and the algorithm continues.

Again, to remove the fairness constraints the normal operation of the check_fair/2 predicate can be turned off such that it will report that all computations are fair.

A Prolog implementation that is in some respects similar to ours is given in [BGG84]. The similarity between aspects of our implementation and that given in [BGG84] result from a common basis in the fixpoint characterization of the temporal operators. Major differences include the temporal operators that are implementated and the way that fairness is handled.

6.5 User Interface

The user interface module provides a menu driven environment in which to assess the quality of a PAL specification. The user interface plays an important role in the verification system. Not only does it provide the normal user friendly features but it also interprets the results obtained by the model checker. On its own the model checker can only determine the validity of a temporal formula on the state graph, i.e., it can only determine whether or not a property is satisfied on the state graph. However, a simple pass/fail answer does little to give the user confidence in the result, nor does it provide much information that would help to find a possible error.

The minimal information needed to help diagnose a suspected error includes the state at which the formula was (not) satisfied, the state or states which caused the formula to succeed (fail), and a state history giving the execution sequence leading up to the error state. To ensure that the above information is available we included additional statements in the model checking clauses for each BTL operator which cause the following two Prolog facts to be saved whenever the model checker reaches a point where it can conclude that a property is either satisfied or is not satisfied on the state graph.

```
result(Formula,Path,FailState).
return_status(Condition,State).
```

The `result/3` predicate saves information about the state of the model checker at the time the model checker halted. `Formula` gives the actual BTL term that the model

checker was working on at the state where failure or success occurred. The BTL term saved in `Formula` may have been only one term in a complex formula but it was this term that finally decided the validity of the whole formula. `Path` is a list of states comprising the execution state sequence leading up to the state where the failure or success occurred. The actual failure or success state is given in `FailState`. Actually, the `Failstate` may not in itself have caused the success or failure of the property but rather it was the last state that the model checker was checking when it decided that the property should either succeed or fail.

The `return_status/2` predicate returns the final results of the model checker's effort to prove the correctness property on the state graph. `Condition` can return one of three values:

1. `ok` — the property was satisfied on the state graph;
2. `failure` — the property is not satisfied on the state graph; and
3. `deadlock` — a system-wide deadlock state was found during the checking algorithm.

`State` returns the name of the last state checked by the model checker or the name of the deadlock state if a deadlock state was found.

From the information provided by the `result/3` and `return_status/2` predicates the user interface attempts to explain the results obtained by the model checker. A typical explanation is given below:

```
***** failure *****

The property
ag pv(sch,[nr(X),@(X<2)])
is not satisfied at system state 1
```

```
Suspected cause of failure is system state 27

Do you want to enter Playback mode? (y/n)
```

6.5.1 Playback Mode

The execution sequence, Path, is used by the user interface to further aid the user in determining why the property was or was not satisfied on the state graph. After receiving the explanation of the model checker's results the user can enter a **playback** mode in which the user interface displays on a state by state basis complete information about each state in the execution sequence. The state information is presented one state at a time giving the states of the individual processes, process variables and system variables. The user can move either forward or backward one state at a time along the execution sequence to aid him in understanding the system events that led to the failure or success of the property. A typical **playback** state presentation is given below:

```
System State #27

Process States:
 Control State              Data State
 send_msg(sch,ch10)          [nr(2),write(f),read(t)]
 recv_msg(wr,ch2)            []
 recv_msg(rd1,ch8)           []
 recv_msg(rd2,ch10)          []

System Variables:
 []

Next System State Information:
```

```
Next State  Process  Next State of Process
28          sch      select(sch,sch1)

*playback* Menu - Next Command? Press first letter of command
forward  back  redisplay  look  jump  trace  main  quit
```

The user interface can also provide additional information about each state such as the state of message channels including the actual messages in the channel queues.

6.5.2 Trace Mode

The **trace** mode provides a similar facility to that provided by the **playback** mode except that the user can chose to follow any sequence of states in the graph. Using the **trace** mode the user can explore the state graph state by state moving backward or forward through the graph. At non-deterministic branching states the user can chose which alternative to explore.

To aid in exploring the state graph the user interface also provides a **search** utility by which the user can find a state or set of states satisfying any non-temporal formula specified by the user. If a search finds a state where the specified formula is satisfied the user can enter the **playback** mode to follow the execution sequence through the state graph to the state.

The **trace** facility is independent of the model checker, that is, a user can explore a state graph without having to specify any correctness properties. If the state graph for a set of connective implications is either unavailable or incomplete the **trace** utility will call upon the state graph generator to build the state graph as needed.

Other services provided by the user interface include:

- utilities to save and load the state graph to or from a file. This can reduce the computational cost of the verification effort by removing the need to regenerate the graph for each session.

- direct access to operating system utilities such as a text editor and file management commands.

6.6 Sample Verification Session

The following text is an actual transcript from a session on the verification system. The PAL system being verified is the simple two process example from Chapter 4 (Figure 4.4). The Prolog form of the connective implications for the example are given in Figures 6.2 and 6.3. The session transcript begins with the first menu presented to the user after the verification system has been initialized. Comments, *in italics*, have been added at certain points for explanation purposes. User input is indicated in boldface.

```
*main* Menu - Next Command? Press first letter of command
verify  files  trace  quit

at the top level menu the user has three command options
1. verify a property against the PAL system
2. file management (edit,delete,directory)
3. enter trace mode
verify   go to the verify menu

*verify* Menu - Next Command? Press first letter of command
go  parameters  build  clear  load  save  main  quit

load   load in a PTR file

*load* Menu - Next Command? Press first letter of command
pal_file  graph

pal_file
```

Enter PAL file name

: **twoproc**

Ok *the PTR file has been loaded into memory*

verify Menu - Next Command? Press first letter of command

go parameters build clear load save main quit

parameters *set up the model checking parameters*

Enter new formula:

: **af st(recv msg(b,chan4))**

Enter the correctness property against which the state graph will be checked. In this case we are checking that process b never experiences starvation

af st(recv_msg(b,chan4))

<CR> to accept, <e> to edit, <SPACE> to get previous

The user is given a chance to confirm that the property has been entered properly.

<CR>

Enter node number that you would like property verified for.

Enter 0 to verify for all nodes.

: **0**

We want to ensure that the property is satisfied at all system states. To aid in diagnosing correctness errors it is sometimes helpful to try to check a property at a single state.

FAIRNESS is currently OFF

<CR> to accept, <SPACE> to turn FAIRNESS ON

<CR>

Although we are checking for a liveness property we can leave the fairness constraints turned off since we know that the two process example is completely deterministic, i.e., it cannot be unfair.

```
*verify* Menu - Next Command? Press first letter of command
go  parameters  build  clear  load  save  main  quit

go   start the model checker using the entered parameters

 Beginning verification for formula:

af st(recv_msg(b,chan4))

for all system states,
for all computations, both fair and unfair.

 Press any key to start...
state 2
```

The state graph generator writes a "state #" *indication as each system state is generated.*

```
state 3
state 4
state 5
state 6
state 7
1  Ok
```

When the model checker determines that the given property is satisfied at a system state it writes a "# ok" *indication. Note that only those states required to determine the validity of the property at state 1 have so far been generated. In fact, no further states need to be generated to determine the validity of the property at state 2 through 7. The next state that needs to be generated is state 8.*

```
2  Ok
```

```
3  Ok
4  Ok
5  Ok
6  Ok
7  Ok
state 8
8  Ok
state 9
9  Ok
state 10
10  Ok

Success for all system states.
*verify* Menu - Next Command? Press first letter of command
go  parameters  build  clear  load  save  main  quit
```

We have determined that process b does not experience starvation on the state graph. To further assess the quality of the system we can enter the trace mode and explore the state graph.

```
main   go back to the main menu

*main* Menu - Next Command? Press first letter of command
verify  files  trace  quit

trace   enter the trace mode

 System State #1


 Process States:
 Control State              Data State
  init(a)                    []
```

```
 init(b)                    []

System Variables:
 []

Next System State Information:
 Next State  Process  Next State of Process
 2           b        recv_msg(b,chan1)
 2           a        send_msg(a,chan1)

*trace* Menu - Next Command? Press first letter of command
forward  back  redisplay  look  search  goto  main  quit
```

Upon entering the trace mode certain variables of the first system state are displayed. The user can move forward or backward through the state graph. In state 1, both processes are in the initial state. In the next state, state 2, process b will be in the recv_msg state and process a will be in the send_msg state.

forward *move forward one state*

```
 System State #2

 Process States:
 Control State              Data State
  send_msg(a,chan1)           []
  recv_msg(b,chan1)           []

 System Variables:
  []

 Next System State Information:
```

```
 Next State  Process  Next State of Process
 3           a        recv_msg(a,chan2)

*trace* Menu - Next Command? Press first letter of command
forward  back  redisplay  look  search  goto  main  quit
```

forward *move forward one state*

```
 System State #3

 Process States:
 Control State              Data State
  recv_msg(a,chan2)          []
  recv_msg(b,chan1)          []

 System Variables:
  []

 Next System State Information:
  Next State  Process  Next State of Process
  4           b        send_msg(b,chan2)

*trace* Menu - Next Command? Press first letter of command
forward  back  redisplay  look  search  goto  main  quit
```

At this state process a is waiting to receive a message from channel 2 and process b is waiting to receive a message from channel 1. We can examine the channels to check if any messages are available for reception.

look

```
*look* Menu - Next Command? Press first letter of command
```

```
channels  message_parameters
```

channels *Enter the look menu and ask to look at the channel states.*

```
  chan1        1/2     [[req]]
  chan2        0/1     []
  chan3        0/1     []
  chan4        0/1     []

*trace* Menu - Next Command? Press first letter of command
forward  back  redisplay  look  search  goto  main  quit
```

We can see that chan1 contains one message. We can also note that all of the channels have a capacity of 1 message except for chan1 which has a capacity of 2.

In the above manner the use could explore the state graph to gain an understanding of process behaviour.

quit *end of session*

6.7 Chapter Summary

In this chapter we have described the essential elements of an implementation of our verification method. Our purpose in developing the implementation was first to show that the method is viable and second to provide a source of practical experience in the area of verification. In the next chapter we will analyse a number of example system problems in order to evaluate the performance of our verification system.

Chapter 7

Evaluation

In this chapter we evaluate the capability of the verification system to meet our goals as stated in Chapter 1. The evaluation will be based on results obtained in specifying in PAL and analysing a number of example problems. In the first section of this chapter we discuss the analysis of four example problems; three different solutions to the classic *readers/writers* problem and one simple process control problem. We concentrate first on a detailed look at the results obtained in analysing a version of the *readers/writers* problem with starvation and then move on to present the results for the other example problems. In Section 7.2 we discuss in general the capabilities of the verification system as based on our experiences in using it to verify the example problems. We also extrapolate our evaluation results to comment on the verification system's ability to deal with practical sized concurrent system problems.

7.1 Example Problems

The *readers/writers* problem is a classic concurrent system problem usually used to demonstrate the concurrent system issues of starvation and mutual exclusion. Essentially the problem consists of a common data structure that is accessed by a number of processes. Some of the processes only need to read data from the data structure

Scheduler Process

```
sch::=
        S_nn
        select
                write_ok →
                        receive(ch1,req(wr));
                        send(ch2,ack);
                        STP(read_ok,false);
                        STP(write_ok,false) |
                read_ok →
                        receive(ch5,req(rd1));
                        send(ch6,ack);
                        STP(nr,nr + 1);
                        STP(write_ok,false) |
                read_ok →
                        receive(ch9,req(rd2));
                        send(ch10,ack);
                        STP(nr,nr + 1);
                        STP(write_ok,false) |
                true →
                        receive(ch3,rel(wr));
                        send(ch4,ack);
                        STP(write_ok,true);
                        STP(read_ok,true) |
                true →
                        receive(ch7,rel(rd1));
                        send(ch8,ack);
                        STP(nr,nr - 1);
                        case nr
                                nr = 0: STP(write_ok,true) |
                                nr <> 0: true
                        endcase |
                true →
                        receive(ch11,rel(rd2));
                        send(ch12,ack);
                        STP(nr,nr - 1);
                        case nr
                                nr = 0: STP(write_ok,true) |
                                nr <> 0: true
                        endcase
        endselect;
        ε
```

Writer Process

```
wr::=
        S_on
        send(ch1,req(wr));
        receive(ch2,ack);
        do_modify_activity;
        send(ch3,rel(wr));
        receive(ch4,ack);
        ε
```

First Reader Process

```
rd1::=
        S_nn
        send(ch5,req(rd1));
        receive(ch6,ack);
        do_reading_activity;
        send(ch7,rel(rd1));
        receive(ch8,ack);
        ε
```

Second Reader Process

```
rd2::=
        S_nn
        send(ch9,req(rd2));
        receive(ch10,ack);
        do_read_activity;
        send(ch11,rel(rd2));
        receive(ch12,ack);
        ε
```

Figure 7.1: PAL Processes for Readers-Writers

7.1.1 Analysis of Readers/Writers

We wish to analyse the given PAL expressions to verify that they describe system behaviour that will solve the given problem. To do so we will first transform the PAL expressions into a global state graph model representing the operational behaviour of the system. Second, we will translate the concerns enumerated above into a set of correctness properties, and third we will attempt to prove that these properties are satisfied on the state graph model. If the properties are satisfied on the state graph then they are also satisfied by the PAL expressions themselves.

To transform the PAL expressions into a global state graph we first must translate the expressions into their PAL Temporal Representation (PTR) a process which at this point must be done by hand. The PTR for this solution to the *readers/writers* problem is given in Appendix E, Section E.3.

We first loaded the PTR for the problem into the verifier and then generated the state graph representation of the PAL expressions. The state graph for the PAL *readers/writers* system consists of 37 system states which the inference engine generated in approximately 4 seconds. For the purposes of our evaluation we separated the generation of the global state graph from the model checking of that graph. In this way we could more clearly attribute evaluation results to either section of the verifier.

We then specified a set of correctness properties to express the concerns enumerated above. Specifically we wish to show that the integrity of the data structure will be preserved, a safety property, and that all processes will have access to the data, a liveness property.

To preserve the integrity of the data we must ensure that both a **reader** and a **writer** process cannot simultaneously be within their respective critical regions. This safety property *mutually exclusive access* can be expressed as follows:

```
ag ~(chan(ch3,[n(1)]) and chan(ch7,[n(1)])) and
ag ~(chan(ch3,[n(1)]) and chan(ch11,[n(1)]))
```

The first term of the property expresses that for all states it must be the case

that ch3 (the channel by which the **writer** processes sends a release message to the scheduler) and ch7 (the channel by which the first **reader** process sends a release message to the scheduler) never both contain a release message. If in a certain state both channels contain a message releasing the critical region then this implies that both processes must have been allowed to enter the critical region simultaneously. The second term of the property expresses a similar requirement between the **writer** and the second **reader** process.

We could also have specified mutual exclusion using the $\exists\Diamond$ operator which is the dual of the $\forall\Box$ operator.

```
ef (chan(ch3,[n(1)]) and chan(ch7,[n(1)])) or
ef (chan(ch3,[n(1)]) and chan(ch11,[n(1)]))
```

In this case, if the property is satisfied on the state graph then this will indicate that mutually exclusive access to the data structure has been violated.

Of importance to note here is the manner in which we have specified the safety property. A more straightforward method of expressing mutual exclusion would have been as follows:

```
ag ~(st(end_recv(wr,ch2)) and st(end_recv(rd1,ch6))) and
ag ~(st(end_recv(wr,ch2)) and st(end_recv(rd2,ch10)))
```

i.e., if both a **reader** and the **writer** processes have just received acknowledgements for their requests to access their critical regions then this is clearly a violation of mutual exclusion. However, if we recall Section 6.3.2 concerning the recording density of the state information in the state graph we realize that `end_recv` state information is not available in the graph. Thus we are limited in specifying correctness properties to the information that is available in the state graph model.

To verify that all processes have access to the data we can specify a liveness property expressing *accessibility* to the data structure for each process.

```
af st(recv_msg(wr,ch4)) and
af st(recv_msg(rd1,ch8)) and
af st(recv_msg(rd2,ch12))
```

This property asserts that for each process the system must eventually reach a state where the process is waiting to receive an acknowledgement message from the **scheduler** process. Since this message waiting point is past each of the processes' critical regions then this implies that the process must have been granted access to its critical region. Of course, this property must be satisfied under fairness constraints since we must assume that a request for access to the critical region cannot be ignored forever.

More specifically, the above property expresses *absence of starvation* or *liveness* for each process rather than *accessibility*. However, since each of the **reader** and **writer** processes consist only of a continuous loop through a critical region, *absence of starvation* implies *accessibility*. To express *accessibility* directly we could specify the following property:

```
st(send_msg(wr,ch1)) ==> af st(recv_msg(wr,ch2)) and
st(send_msg(rd1,ch5)) ==> af st(recv_msg(rd1,ch6)) and
st(send_msg(rd2,ch9)) ==> af st(recv_msg(rd2,ch10))
```

which asserts that if a process requests access to its critical region it will eventually receive access. However, again we are limited by the information available in the state graph. The state graph generator records only those process states where a process is blocked at a message waiting point, i.e., either a *send_msg* to a full channel or a *recv_msg* from an empty channel. Therefore, we cannot guarantee that a *send_msg* state for a process will be recorded in the state graph.

Results

To verify the safety property expressing mutually exclusive access to the data structure the the model checker required under 2 seconds for each of the three processes

or 6 seconds overall. As was expected the model checker verified that the property was satisfied on the state graph.

When the liveness property was checked on the state graph the model checker determined that the property was not satisfied on the graph. The model checker required approximately 8 seconds to find that the **writer** process was experiencing starvation along a state sequence in which the **reader** processes alternately gained access to their critical regions. This computation was determined to be fair since the **scheduler** process was prevented from considering the writer's request by the `read_ok` guard.

When the model checker found that the liveness property was not satisfied on the state graph model of the system the user interface reported this failure and gave the actual state at which the property failed. The user interface then allowed us to enter the **playback** mode to explore the computation along which the property had failed.

7.1.2 Other Examples

PAL expressions and connective implications for each of the example problems discussed in this section can be found in Appendix E. In this section we will just outline the basic structure of the problem and give the analysis results returned by the verifier.

7.1.2.1 Readers/Writers Without Starvation

Another solution to the *readers/writers* problem was specified and analysed. This solution, whose PAL expressions can be found in Appendix E, Section E.4, eliminates the possibility of the **writer** process experiencing starvation by adding a writer pending flag (`write_pend`) to the **scheduler** process. Thus, if the **writer** process requests access to its critical region and there are currently **reader** processes in their respective critical regions then the writer's request is still noted by the **scheduler** process. After the writer's request no further reader requests are processed until the

writer has exited its critical region.

The behaviour of this solution of the *readers/writers* problem was described by a state graph of 57 states which required 10 seconds to generate. We specified and checked the same correctness properties as for the solution with starvation. To verify that the state graph model was correct with respect to the mutually exclusive access to the data structure required approximately 9 seconds. To check that each process was free of starvation required 10 seconds per process or 30 seconds overall.

7.1.2.2 Readers/Writers Using System Variables

In specifying a solution to the *readers/writers* problem we do not need to model the actual data structure that the **reader** and **writer** processes access. We are interested only in the correct and eventual access by the processes to a common structure whatever that structure actually represents. This abstraction allows us to describe a solution and verify its correctness using a minimum of system states as in the two previous solutions.

To help evaluate how our verification system handles larger problem spaces we specified a third solution to the *readers/writers* problem which actually used a system condition variable as a common data structure. The PAL expressions and PTR for this solution can be found in Appendix E, Section E.5.

Because we actually used a data structure in our representation of the problem the number of states needed to represent system behaviour jumped from 37 to 360. To generate the 360 states the state graph generator required 130 seconds of CPU time. We specified the same correctness properties as for our two previous examples and checked them on this state graph. To verify that the state graph was correct with respect to mutually exclusive access to the common data structure the model checker required 33 seconds. Since we based this solution to the *readers/writers* problem on our first solution, i.e., the one in which the **writer** process was experiencing starvation, we expected the model checker to again find that our liveness property

was not satisfied on the state graph model. The model checker required approximately 10 seconds to determine that the **writer** process was experiencing starvation.

7.1.2.3 A Process Control Problem

The mixer problem is a simple process control example taken from a manual for the Grafcet specification language[BBFM77]. Grafcet is a graphical notation used to describe in graphical terms the behaviour of process control applications. The physical aspects of the mixer problem are shown in Figure 7.2. Essentially, the problem is to maintain the proper mixture level in the **hopper**. If the mixture in the **hopper** drops below an indicated lower level this causes new batches of mixture to be produced until the hopper is filled to the high level. Each batch of final mixture is made by mixing carefully controlled amounts of two liquids, **product 1** and **product 2**, with one cart load of dry material, **product 3**. The two liquids must first be mixed separately for a certain time in **mixer 1** and then mixed with the dry material in **mixer 2** for a certain time. The **buffer** between the two mixers can temporarily hold one mixed batch of products 1 and 2. The mixing of one batch of products 1 and 2 can be done concurrently with the mixing of the dry material with another batch. Once a mixture has been started it must be dumped within a certain period of time, i.e., neither the **buffer** nor either mixer can be used to hold material; it must all be dumped to the hopper once the mixing cycle is complete.

The main issues in this example are:

1. to ensure that **hopper** is never allowed to become empty, i.e., the low level sensor on **hopper** should be taken as a request for more batches of material to be prepared. In order that **hopper** never becomes empty this request must eventually be serviced.

2. to ensure that **hopper** does not become too full. Since the mixed material has a limited shelf life only a certain amount should be available at any one time. As well, **hopper** has a finite capacity so it must not be allowed to overflow.

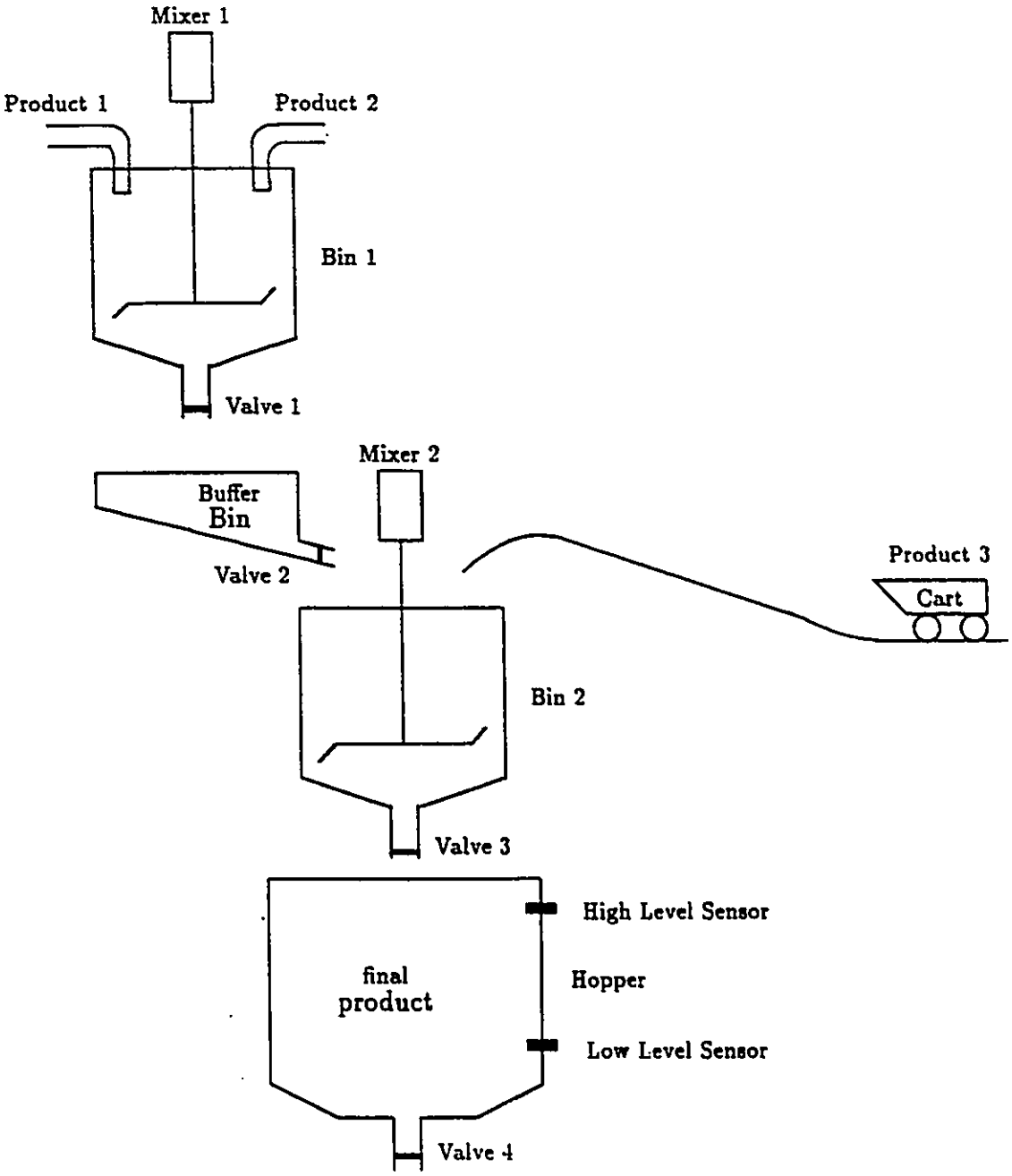


Figure 7.2: Mixer Problem

Therefore, the high level sensor should be taken as a request that no more batches are to be started. Only those batches that are currently being mixed should be allowed to continue to be processed.

The PAL process expressions and PTR for our solution to the *mixer* problem can be found in Appendix E, Section E.6. Our solution to the problem uses a system condition variable to indicate the state of the high and low level sensors. The sensor state is set by an environment modelling process which monitors the number of batches of mixture that are produced by **mixer 2**. We modelled the actions of the two mixers and the buffer with three separate processes. The hopper was modelled as part of the environment. We also included a fifth process to model the consumers of the mixture being produced.

The state graph model produced by the state generator consisted of 271 states which required 108 seconds to generate. To check that all requests for more mixture resulted in more mixture being produced we specified the following liveness property:

```
sv([level(low)]) ==> af sv([level(high)])
```

This property simply states that if the level in the hopper ever reaches the low mark then eventually it must be filled until the high mark is reached. The model checker required under 2 seconds to determine that this this liveness property is not satisfied on the state graph. It is possible that the **consumer** process could consume the batches as fast as they are produced, therefore the hopper level would never reach the high mark. This example shows that in verifying a system's behaviour a user can gain a better understanding of that behaviour.

Another way of expressing the liveness property that the low hopper level events are always responded to would be check the liveness of the **consumer** process. If the consumer always is able to consume batches of material then this implies that material is being produced. The liveness of the **consumer** process can be expressed as follows:

```
af st(recv_msg(consumer,ch7))
```

To check the second requirement, that the hopper should not overflow, we specified the following safety property:

```
ag pv(env,[batches(X),@(X < 4)])
```

The amount of mixture in the hopper is modelled by the process condition variable batches and is maintained by the environment modelling process env. When mixer2 dumps a new batch into the hopper batches is incremented and when a batch is consumed by a consumer process, consumer, batches is decremented. In the environment modelling process we decided to turn the high level sensor on when there were at least three batches in the hopper, i.e., batches(3). The above safety property specifies that the number of batches in the hopper should never exceed 3. The model checker was able to determine within 15 seconds that the above property was not satisfied on the state graph. We were able to determine that with the high level sensor set at 3 batches the actual number of batches in the hopper could get as high as 7. The reason for this is due to the pipelined batch mixing procedure. Even after the high mark has been reached there may still be up to four batches pending in the buffer and mixers.

This example shows how modelling of a problem can help system designers to understand complex system/environment interactions. In order to resolve the safety errors, (hopper overflow), detected by our analysis it is apparent that the environment might need to be modified, not necessarily the control system. The three level, independent buffer/mixer combination allows for at least three extra batches of mixture to be dumped into the hopper *after* the high level mark has been reached. (the fourth extra batch results from the non-deterministic execution rates of the two processes accessing the system condition variable level). There are several ways to overcome this difficulty but they involve changing the environment itself. Two possibilities are: move the high level sensor to a lower position on **hopper**; or do not allow **mixer 1**

to begin processing a new batch until after mixer 2 has dumped its mixture into the hopper.

7.2 Evaluation

In this section we present an informal evaluation of the suitability of our verification system for verifying the correctness of P/T expressions. The informal comments presented here are based on our experiences in using the verifier to analyse the above example problems. Our judgement of *suitability* will be based on the factors outlined in Section 1.2. Stated again briefly they are:

- the system must provide a proof of correctness;
- it should be applicable to practical problems; and
- it should be useable by people who actually do develop systems;

We do not need to address the first factor because our use of the model checking approach guarantees a proof of the correctness of any property that is checked. In the next two sections we will comment on the suitability of our approach based on the second two factors, that is, is our verification approach applicable to systems of a reasonable size and complexity and is it useable by the people who typically design these systems.

In the following sections we must keep in mind that in developing the implementation of our verification method we have concentrated on function rather efficiency. That is, the capabilities of our implementation of our verification approach may not reflect the capabilities of the verification approach itself. In general, the following comments will be derived from our experiences in using the verification system to analyse the example problems. However, we will also try to extrapolate the perceived merits and faults of our implementation to comment on the verification approach itself.

7.2.1 Useable by the Average Designer

The first thing to do is to define what is meant by *useable by the average designer*. What we are concerned with here is that the verification method should be useable by the people who are actually involved in developing the system that is being analysed. If the method is easy to use then there will be more likelihood of it actually being used to develop correct systems. As well, if the method was not easy to use, i.e., it appears technically complex and/or consists of a number of tedious manual steps, then the users may have end up having as little confidence in the correctness of the verification results as they do in the system being verified.

In analysing the simple example problems we found that the verification system developed in this thesis is fairly easy to use. The accessibility of our approach can be attributed to two factors: one, the model checking method of verification, and two, the features of the user interface.

The Model Checking Approach

The model checking method is based on a model theoretic rather than a proof theoretic application of temporal logic, therefore it naturally presents a more operationally oriented view of the verification process. The state graph model of a system on which the method is based is a structure that is already well understood by system developers. The introduction of the BTL operators and their well defined model checking algorithms are not difficult concepts to understand. The model checking algorithms are easily automated and therefore their logical foundations and function can be hidden from the user. However, understanding of the BTL operators on which they are based is essential for specifying meaningful correctness properties. On the other hand a method based on the proof theory of temporal logic does not usually include either automation or well defined algorithms for constructing proofs. Rather the successful construction of proofs depends on the expertise of the prover.

By adopting the model checking approach we have shielded the user from any

requirement to understand the details of the verification method itself. The user is only required to understand the BTL operators and how they are used to express correctness properties for a system.

The User Interface

The features of the user interface, while being fairly primitive in our implementation, greatly aided in the use of the model checker. By itself the model checker can only provide a pass/fail answer about the correctness of a temporal formula on the state graph. Obviously, it would be very useful to know why and where a formula failed. Answers to these questions aid a user in confirming that the error does indeed exist, thus increasing his confidence in the model checker's results. As well, these answers are essential in diagnosing the exact cause of the error to allow the system to be corrected. To aid the user, the user interface attempts to provide as much information as possible about the success or failure of a temporal formula on the state graph. This information is provided via the following three services:

1. The user interface provides as detailed an explanation as possible of the model checker's results. This serves to increase the confidence of the user in those results.

2. The **trace** and **playback** modes allow the user to review system events along a computation. This aids the user in diagnosing the exact reason for the failure of a correctness property.

3. The user can retrieve state information such as message queues and variable values at any state.

For the simple problems that we analysed we found that the above services as provided by the user interface were vital factors in making the model checking approach a useable verification method. Given that the user interface is a critical factor

in making the verifier a useable tool we should consider how its features would extend to a larger problem space. We will address this concern in the next section.

Accessibility for Larger Problems

While we did find that our verifier was suitable for analysing the correctness of the simple example problems, we were unable to judge its effectiveness in dealing with larger systems. For the simple example systems we were aided in our analysis by our complete familiarity with their process interactions. Because the problems were fairly simple and we were familiar with the temporal errors that could be expected we were easily able to diagnose the failure of a correctness property. For a much larger problem, say with 1000 states or more, we would not have this advantage. In this respect then, can the features that make this verification approach accessible to the average designer be extended to deal with larger problems and if so how?

To properly diagnose and understand the reasons for the failure of a correctness property requires an understanding of the process interaction events along the computation on which the error occurs. For example, to understand the chain of events causing the starvation of the writer process in the *readers/writers* problem required a review of the process interactions along the computation. In our implementation, the **playback** mode attempted to convey this understanding via a state by state presentation of the system events along the computation. However, even given the relatively simple problem space and our familiarity with it it still required considerable concentration to keep track of the events that had transpired and how they could lead to starvation. For a larger problem space it would be even more difficult to gain an understanding of the error. The problem in essence is that the format of the **playback** and **trace** modes do not allow an easy conceptualization of the process interactions and of the system as a whole.

A possible solution to this problem is an animated presentation of the processes and their interactions as proposed in [Kar87]. In an animated presentation each

system object, such as a process, a message channel or a message, is represented as a graphical icon. Using a graphical representation we can show a message being sent from a process to a message channel by moving a message icon from a process icon to the appropriate message queue icon. Thus a series of process interaction events can be animated to graphically illustrate a particular computation. Such a presentation would aid the user in understanding the complex process interactions leading to temporal errors.

Syntax Errors

In analysing the example problems we found that syntax errors in the connective implications were frequently to blame for the failure of a correctness property on the state graph. Usually these types of errors resulted in system-wide deadlock states but occasionally the error was less obvious. In our case we were analysing fairly simple and well understood systems and therefore we were able to distinguish between a syntax error and a real temporal error. However, for larger systems syntax errors in the connective implications would pose a significant and very frustrating problem. Part of the reason for this problem is that our use of the verification system is outside of its expected context of operation. The PAL process expression verifier is meant to be part of a system development environment which would include expression editors and syntax checkers. An essential element of the development environment would be a pre-processor to directly translate a set of PAL expressions in their equivalent connective implication form. This would eliminate the time consuming and error prone manual translation of the PAL expressions.

7.2.2 Applicable to Practical Problems

Due to the limitations imposed by our particular implementation we we were not able to judge directly the suitability of either the model checking approach or our overall verification approach for problems of a practical size and complexity. However, based

on our experiences in verifying the example problems we can make the following observations.

Computational Performance

The computational performance of our implementation was more than adequate for the simple example problems that were analysed, however, we would expect this performance to degrade as the size of the problem increases. An analysis of the computational complexities of the individual algorithms of which our system is composed will help to establish the limiting factor in the size of the problem that our approach can handle.

1. To generate the n^{th} state in the graph requires a search through the first $n-1$ states to see that the n^{th} state has not already been generated. Therefore, the complexity of generating the state graph itself is $O(n^2)$ where n is the number of states in the graph.

2. The number of states n in the graph is in the worst case proportional to the size of the state space of the whole system which is known to be a cross product of the individual process states. If each process has a state space equal to the largest process P_s then the state space of the system is $(P_s)^m$ where m is the number of processes in the system.

3. Based on the above two orders of complexity, the complexity of the state graph generation method is $O((P_s)^{2m})$.

4. There are $O((n-1)!)$ computations on a non-directed, fully connected graph with n nodes. Therefore the number of computations in any state graph generated from m processes is $O((P_S^m - 1)!)$. However, the actual number of computations on any state graph will be considerably less than $(n-1)!$ since the state graph is a directed graph with sparsely connected nodes.

5. The complexity of the basic model checking algorithm is linear in the number of states in the graph S and in the number of connections between states R, i.e., $O(\mathbf{S} + \mathbf{R})$ (Section 5.2.1).

6. The complexity of checking the fairness of a computation of m states is $O(m^2)$ (Section 5.5.2). The size of the computation m depends on the number of states in the graph n. If we assume that there are an equal distribution of short and long computations then we can say $m = n/2$. Therefore the complexity to determine the fairness of all computations on a state graph is $O(n^2(n-1)!)$ since there are $O((n-1)!)$ computations on a graph with n states.

   If we replace the number of states n with P_s^m, the number of states in a graph for a system of m processes then the complexity to determine the fairness of all the computations on a state graph is $O((P_S^m)^2(P_s^m - 1)!)$.

From the above estimates of the computational complexities of various facets of our verification system we can draw a number of conclusions about the problem solving capacity of our approach. The most important conclusion to be drawn from the above is that the size of the state graph is the limiting factor in determining the capacity of our verification system.

The worst case computational cost is for model checking under fairness constraints, item 6. However, recall that the actual number of computations on a state graph will be considerably less than $(n-1)!$ because the state graph is a directed graph with sparsely connected nodes. As well, recall that not every computation needs to be checked for fairness. Therefore, although the actual number of computations on a graph will depend on the number of nodes and on how they are connected, we can say that in practice the complexity of model checking under fairness constraints is proportional to P_s^m. In analysing the example problems we found that the number of computations that were checked for fairness was always less than the number of states in the graph. Therefore, the limiting factor in our approach is the number of states required to describe the system's behaviour. What this implies is that for as

large a state graph as can in practice be generated our approach should provide a viable means of verifying correctness properties on it.

State Explosion

The size of the state graph model for a system grows exponentially with the number of processes in the system. This state explosion problem is common to state based verification methods. While the problem of state explosion cannot be removed completely it effects can be minimized to allow larger problems to be dealt with. Since the size of the state graph model of a system depends on the number of processes in the system and on the size of the state space of each process, a reduction in either quantity will significantly reduce the size of the state graph. In [Kar87] Karam outlined a number of analysis heuristics which can significantly reduce the size of the state graph model of a system. We briefly outline these heuristics below:

minimum number of processes—The size of the state graph model of a system depends on the number of processes in the system. Therefore, when analysing a system include only those processes which are essential to faithfully describe the behaviour of the system.

divide and conquer—partition the system into smaller subsystems and analyse each individually. Interactions between the subsystem under analysis and the rest of the system can be simulated. The same principle can be applied to the simulation of the environment. Simulate only those environmental functions that affect the subsystem under analysis.

minimum buffer size—All buffers, queues, message channels, etc, should be specified at their minimum levels. Restricting the range of possible states of a data structure directly affects the size of the state space of a process. The same principle applies to the number and range of variable values. Use only those variables that are necessary and restrict their range to the minimum set of values

that will allow the essential elements of the system to be faithfully described.

Flexibility

A major benefit of the model checking approach is the flexibility it allows in the range of correctness properties which one can verify. We were easily able to specify various correctness properties for each of the examples and verify the correctness of these properties on a state graph model of the example system. For larger problems with more complex process interactions this flexibility would continue to be an asset as the complexity of the specified correctness properties could be made to match the complexity of the problem.

However, this aspect of the verifier also puts the onus on the person doing the analysis to be able to specify meaningful correctness properties. The user must not only understand the use of the BTL operators but he must also understand the system he is analysing enough that he can form meaningful correctness properties for it. Therefore, to some extent, the success of this verification method depends on the expertise of the user. As was discussed in Section 2.1 PAL is being targeted towards users who will design their own systems, therefore they should have an intimate knowledge of their application. In any case, the problem of understanding and developing meaningful correctness criteria for a system is common to most verification methods. The difference with the model checking approach is that the user does not need to be an expert in logic or mathematics.

Again, it is the user interface that becomes important here. The more complex a correctness property is the more difficult it will become to explain exactly why or why not it was or was not satisfied on the state graph. The user interface will accordingly have have an impact on the ability of the user to exploit the model checker's flexibility.

A factor limiting the model checker's flexibility is the information content of the state graph model of the system. In building the state graph we employed various techniques to reduce the number of states required to describe system behaviour.

However, in reducing the number of states we also removed information about the system's behaviour from the graph. This problem was illustrated in Section 7.1.1 dealing with the *readers/writers* problem. Although we were able to specify the required properties we were forced to do so in a rather convoluted fashion. What this implies is that in our implementation we may not have achieved the best compromise between the information content required to properly analyse the system and the need to deal with the state explosion problem. Indeed this may indicate a weakness in the model checking approach itself. As the problem space becomes larger any attempts to control it will reduce the effectiveness of the model checker.

7.3 Evaluation Summary

Overall the evaluation shows that our verification system does meet the requirements as stated in Section 1.2. For the example problems that were analysed the verifier was found to provide a very accessible verification environment.

Based on our experience in analysing the example problems we believe that the approach could be extended to a practical environment for analysing systems with a larger number of states. In general the factors which have limited our implementation to the analysis of small problems are not related to our verification approach. Ultimately, the size of problem that can be handled with our approach is limited by the size of state graph that can be accommodated by the implementation. However, for any problem within these limits the verification approach should meet our requirements as stated in Section 1.2.

To actually implement a verification system based on our approach capable of handling problems of a reasonable size and complexity, development would have to proceed along two fronts:

1. Development of those aspects of the verifier that will allow larger problems to be handled. i.e., dealing with the state explosion problem. Included in this area

are:

- development of an efficient method of storing of state information—to allow larger state graphs to be analysed. Our reliance on MProlog's data handling mechanisms contributed to the limits on the size of the graph that could be analysed. The main sources of inefficiencies in the current implementation stem from the use of recursive model checking algorithms. The model checking code would have to be implemented without this reliance on recursion in order to allow larger state graphs to be generated and analysed. [CES83] provides a good outline of the basic algorithms.
- analysis of methods to achieve the best compromise between information content and size of the graph—to allow the benefits of the model checking approach to be extended to as large a problem space as possible.
- increased computational performance—implementation on a faster processor and/or with a more efficient language to ensure that a state graph can be generated and checked within a reasonable time frame.
- development of an automatic PAL to connective implication translation utility—even for the small example problems, syntax errors in the connective implications were responsible for a large percentage of correctness errors.

2. Extension of user interface facilities to ensure that they remain effective for larger problems. The user interface with its explanation and diagnostic aids was considered to be a vital element in making the verifier a useable verification tool. Therefore certain features of user interface must be enhanced to ensure that the verifier remains as accessible for larger problem as it was for the example problems. Based on the above evaluation the following aspects of the user interface would require further development:

   - animated trace and playback modes—animation of the trace and playback

modes will aid the user in diagnosing the temporal errors of more complex systems;

- more intelligent explanation facility—to help pinpoint the exact cause of a temporal error;
- facilities to ease the application of analysis heuristics—use of the analysis heuristics will allow larger problems to be analysed. User oriented features to ease their application to a problem will increase the accessibility of the verifier for larger problems.

Chapter 8

Conclusions

8.1 Summary of Research

Our objective in this thesis was to develop a means of verifying a set of PAL process expressions against a specification describing correct temporal behaviour. Factors which influenced the nature of a solution to this goal were:

- the verification method must supply a proof of correctness
- the method should be applicable to practical problems
- the method should be capable of being used by knowledgeable system developers

The approach we took in meeting our goal was to first transform the PAL expressions into a global state graph and then to verify the correctness of this graph against a specification expressed in Branch Time Temporal Logic (BTL) by using a *model checking* algorithm. The global state graph is a directed graph representing the overall temporal behaviour of the concurrent system. In the model checking approach to verification this state graph represents a model of the proposed system and an efficient algorithm, called a model checker, is used to determine whether or not this graph is a valid model of a correctness specification expressed in BTL. If the graph is correct with respect to the specification then the PAL expressions are also correct.

In the following sections we summarize our research contributions made towards the realization of this approach to verification.

Generating the State Graph

To transform the PAL expressions into an equivalent state graph model we used a method developed by G.M. Karam in [Kar87] for the Ada language. In [Kar87] Karam developed COL, a linear time temporal logic (LTL) based language to represent the temporal behaviour of Ada programs, and an inference engine which derives state sequences from the temporal logic representation of the program. We developed our own version of COL, which we have called PTR, for PAL Temporal Representation, and an inference engine both of which have been modified to handle the unique features of PAL.

We were drawn to Karam's method of state graph generation because its basis in LTL allowed the automatic generation of the minimal state graph needed to completely represent the system's behaviour. That is, the COL representation of Ada allowed equivalent state sequences to automatically be pruned from the state graph. This feature allowed more complex problems to be analysed than would otherwise be the case. In developing PTR we showed that these equivalent state pruning methods which were developed based on the Ada rendezvous messaging semantics were also valid for PAL's asynchronous messaging mechanisms.

As for COL our state graph generation method was shown to have an exponential complexity in the number of processes in the system under analysis. The size of the state graph is dependent on the number of processes in the system and on the size of the state space of each process. A reduction in either quantities would greatly reduce the size of the state graph. In Section 7.2.2 we discussed a number of analysis heuristics whose application can reduce either the number of processes or the process state space while still allowing the essence of the problem to be expressed.

The Model Checking Approach

The heart of the model checking approach is a series of algorithms which can efficiently determine the validity of a BTL formula on a state graph model of a system. In this thesis we developed Prolog implementations of the model checking algorithms for each of six BTL operators ($\forall\Box$,$\exists\Box$,$\forall\Diamond$,$\exists\Diamond$,$\forall\bigcirc$,$\exists\bigcirc$). Using the model checking algorithms we were able to determine the validity of BTL formulae expressing various correctness properties on state graph representations of a number of example problems.

We also confirmed the results given in [CES83][CES86] that the computational complexity of the model checking algorithms for each of the BTL operators was linear in both the number of states in the graph and the number of branches between states. We could not verify the accuracy of this order of complexity with experimental results since our implementation only allowed systems of less than 500 states to be analysed, however, we did conclude that in theory the complexity of our implementation of the model checking algorithm was linear.

Fairness

A requirement when dealing with certain correctness properties (liveness properties) is to consider the fairness of the computations on which the validity of the properties is determined. We chose to handle fairness by modifying the definitions of the BTL operators such that the new operators would provide equivalent results over all computations as would the normal operators over only the fair computations. This approach has the advantage that only those computations which could form an argument to either prove or deny the validity of BTL formula need to be checked for fairness. In Section 5.3 we showed that only the $\exists\Box$ and $\forall\Diamond$ operators needed to be redefined to handle fairness constraints. We developed and implemented fair versions of the model checking algorithms for these two operators.

To actually determined the fairness of a computation on the state graph we developed a set of fairness constraints, based on PAL semantics, against which to

judge the fairness of the computation. Possibilities for unfairness arise from the non-determinism in the PAL execution model and from certain PAL semantics. We showed that non-determinism in the state graph model of the PAL system was due to the PAL select construct and the PAL semantics defining access to global variables. We developed and implemented methods of detecting unfairness resulting from either source. The complexity of our fairness detection methods is squared in the number of states in the computation.

Because of the need to determine the fairness of a computation, the complexity of the model checking algorithms for the fair BTL operators $\exists\Box^f$ and $\forall\Diamond^f$ is no longer linear. We showed that in the very worst case the complexity of these two operators is now $O((n-1)!)$ where n is the number of states in the state graph. However, we also argued that in practice the complexity would be closer to being exponential in the number of processes in the system because only a small percentage of the total number of computations in the state graph would actually have to be tested for fairness.

A Verification Environment

The model checking algorithms provide an efficient means of determining the validity of a correctness property on the state graph model of the system. They do not however provide an explanation as to why a property is not satisfied or give any information that could help the user diagnose the error. We felt that these were important considerations in making the verification method accessible and of some benefit to system designers. Therefore, in our implementation we attempted to show how the model checking algorithms and the state graph generator could be integrated into a single verification environment complete with a user interface capable of providing the necessary services to the user. The user interface is capable of interpreting the model checker's results and if possible giving an explanation of any errors encountered. We found that although the user interface in our implementation was fairly primitive it

greatly enhanced the effectiveness of the verification system.

8.2 Evaluation of Results

An evaluation of our implementation of our verification system was performed and the results discussed in Chapter 7. Generally the evaluation concluded that while our implementation did fall short of completely meeting all of our goals it did demonstrate that our overall approach to the problem was capable of fully meeting the goals. The evaluation results are summarized below.

Provides a Proof of Correctness

The model checking algorithms on which our verification method is based do provide a proof of correctness of the system under analysis. That is, they directly determine the validity of temporal formulae expressing correctness properties on a state graph model of the system. Using the verifier we were able to detect starvation in the *readers/writers* problems and a safety error in the *mixer* problem.

Applicable to Practical Problems

As was expected our choice of Prolog as an implementation language and our emphasis on function rather than efficiency restricted the size of problem that our verifier could analyse. Therefore we were not able to demonstrate that our approach was suitable for problems of a practical size and complexity. However, we argued that the factors limiting the size of the problem were, with one exception, limitations of our implementation rather than our approach. The one exception is the state explosion problem common to all state based analysis methods. In Section 7.2.2 we discussed a number of analysis heuristics that could be applied to aid in dealing with the state explosion problem.

Obviously, our verification method is limited to the analysis of finite state systems. However, many useful systems fall into this class of problem. We could not establish a necessary minimum number of states that a full scale implementation of our verification method should be capable of handling if it were to be of any practical value. However, based on the number of states that were needed to analyse even the small example problems, i.e., over 300 states for one version of the *readers/writers* problem, we estimate a need for a capacity on the order of 10,000 states.

Useable by System Designers

Based on our analysis of the example problems we argued that our approach to verification does meet the criteria that it should be capable of being used by knowledgeable system designers. The accessibility of our approach is based on two factors: first, the model checking approach which with its easily mechanized algorithms hides the formalisms of the verification method from the user, and second, the user interface which provides services to interpret and diagnose the model checker's results. Both features contributed to the ease of use of the verifier.

A question left open by our evaluation was how well this ease of use would extend to a larger problem space. We argued that the user interface would be a vital factor in this respect. While the primitive user interface services provided by our implementation were adequate for the simple example problems that were analysed, we did perceived a need for improved user oriented features if the verifier was to be effective in dealing with larger problems. In the evaluation in Section 7.2 we suggested a number of ways of extending the features of the user interface such as animation of the **playback** facility and utilities to aid in the application of the analysis heuristics. We suggest investigation of these methods as areas of further research in Section 8.4.

In General

In general, it is felt that the verification approach investigated in this thesis does provide a suitable means of verifying the correctness of PAL expressions. Although our implementation of this approach was limited it did show that the approach is feasible. With greater emphasis on the user interface and on a more efficient implementation we believe that such a verification approach could provide a valuable tool to aid in producing correct systems.

8.3 Value of this Research

Our main contribution in this work was the realization of a verification method to aid in the correct development of PAL systems. We successfully integrated elements of two previously unrelated verification methods, Karam's state graph generation method and the model checking approach. With the appropriate emphasis on a suitable implementation the approach developed in this thesis would provide a valuable tool to aid in designing correct PAL systems. However, the development of this tool is not the only benefit of this research. We identify three other contributions below.

User Acceptance

An important result of our research was the realization that, independent of the verification method that is actually employed, the user's willingness to accept and act on the verification results depends on the user's confidence in how those results are obtained. i.e., the usefulness of the verification system depends on the confidence the user has in its results. Therefore, the development environment, i.e., user support and development tools, etc, are as directly related and as important to the quality of a system as the underlying design methodology or verification method. The user interface in our implementation, while being fairly primitive, did provide sufficient information to build confidence in the model checker's results. However, even the

primitive user interface features provided by our implementation represented half of the total programming effort required to implement the system. For the verifier to be effective in dealing with larger problems we realize that even more emphasis must be placed on the user interface. From our experiences in using our verification tool we can generalize to state that a major emphasis of any practical verification method should be user acceptance of that method.

Generality of Approach

While our verification method was developed specifically for PAL the approach that we have taken is equally applicable to other concurrent system description methods. PAL's semantics and execution model are based on common concurrent system concepts, therefore we should be able to apply our overall verification approach to implement verification systems for other concurrent languages.

Development of PAL

As Ralph London states in [Lon79], "Overall, the most important result of research in program verification is understanding the concept of verification itself–specifically, what it means to verify a program." In this regard one area in which this research has some been of value is in the further development of PAL itself. The development of our verification method underlined the fact that the structure and semantics of a language are important to verification efforts. Certain aspects of the language aided while others hindered verification efforts. For example, in Ada, many calling tasks can queue at a single rendezvous entry. The unknown ordering of the callers to this entry is a source of non-determinism that any verification method would have to deal with. The use of unidirectional, private message channels in PAL simplified verification efforts by removing this source of non-determinism. On the other hand, PAL's modelling power, i.e., its ability to specify concise solutions to a problem, is correspondingly reduced because of this lack of non-determinism.

We also confirmed that certain features of the language lead more easily to temporal errors than other features. For example, the use of global variables as an interprocess communication mechanism must be carefully controlled due to the non-determinism associated with the access to the global variable value. This was demonstrated in the *mixer* problem where the non-deterministic access to a global variable contributed to the overflow of the hopper.

Verification research has already and will continue to have an influence on languages and design methodology. Our effort in developing the verification method has underscored certain trade-offs, such as the two mentioned above, between methodology and verification. Further development of PAL should begin with a more thorough analysis of the verification aspects of the language. We suggest this as a possible area for further research.

8.4 Further Research

We identify certain areas below which present possible avenues of further research.

Improved user interface

One area of research which deserves further attention is the user's acceptance of verification methods, i.e., an ability for the method to actually be used effectively. We have already discussed a number of ways that the accessibility of our verifier could be improved. We list them again below:

- animated trace/playback facility to better convey an understanding of system events leading to temporal errors;
- a more intelligent explanation facility to provide better analysis of the verification methods results;
- development of utilities to ease the application of analysis heuristics.

For a system to be used it must be capable of being used. The emphasis here is to convey understanding to the user about the interactions in his system. Further research in this area is required to determine ways of effectively conveying that understanding.

Automatic generation of connective implications

To be able to analyse a PAL design we first had to translate the PAL process expressions into their equivalent connective implication form. In actually analysing the PAL designs we found that many of the 'temporal errors' discovered by the model checker were actually due to syntax errors in the connective implications. An obvious solution to this problem is to develop a pre-processor to automatically transform the PAL expressions into connective implication form. This would not only help to eliminate syntax errors in the specification but would also allow the user to easily submit modifications for analysis as the verifier discovered errors. This pre-processor should be integrated with other development tools such as editors and syntax checkers into a complete development environment.

Further development of PAL

As was discussed in Section 8.3 above, the development of a verification method for PAL has suggested ways that PAL could be improved to aid verification efforts. A more thorough analysis of PAL should be conducted with an aim of finding the best compromise between expressive power and verifiability consistent with the overall design philosophy on which PAL is based. Perhaps this research could suggest ways that PAL could be changed such that verification efforts are aided without losing any of the expressive power of the language. On the other hand, the research might also be able to suggest ways that the expressive power of PAL could be increased without significantly increasing the verification effort.

Appendix A

Axiom System for LTL

This appendix presents an axiom system for the Linear Time Temporal Logic system DX [Pnu81] as given in [Hai82]. The labels from [Hai82] are given to the left of each axiom. In this thesis we will refer to specific axioms by the labels to the right of each axiom.

$$(A0) \quad \models \quad \Diamond \mathbf{w} \equiv \neg \Box \neg \mathbf{w} \tag{A.1}$$

$$(A1) \quad \models \quad \Box(\mathbf{w}_1 \Rightarrow \mathbf{w}_2) \Rightarrow (\Box \mathbf{w}_1 \Rightarrow \Box \mathbf{w}_2) \tag{A.2}$$

$$(A2) \quad \models \quad \Box \mathbf{w} \Rightarrow \mathbf{w} \tag{A.3}$$

$$(A3) \quad \models \quad \bigcirc(\neg \mathbf{w}) \equiv \neg \bigcirc \mathbf{w} \tag{A.4}$$

$$(A4) \quad \models \quad \bigcirc(\mathbf{w}_1 \Rightarrow \mathbf{w}_2) \Rightarrow (\bigcirc \mathbf{w}_1 \Rightarrow \bigcirc \mathbf{w}_2) \tag{A.5}$$

$$(A5) \quad \models \quad \Box \mathbf{w} \Rightarrow \bigcirc \mathbf{w} \tag{A.6}$$

$$(A6) \quad \models \quad \Box \mathbf{w} \equiv \bigcirc \Box \mathbf{w} \tag{A.7}$$

$$(A7) \quad \models \quad \Box(\mathbf{w}_1 \Rightarrow \bigcirc \mathbf{w}_2) \equiv (\mathbf{w}_1 \Rightarrow \Box \mathbf{w}_2) \tag{A.8}$$

$$(A0\prime) \quad \models \quad \Box \mathbf{w} \equiv \neg \Diamond \neg \mathbf{w} \tag{A.9}$$

$$(A1\prime) \quad \models \quad (\Box \mathbf{w}_1 \wedge \Diamond \mathbf{w}_2) \Rightarrow \Diamond(\mathbf{w}_1 \wedge \mathbf{w}_2) \tag{A.10}$$

$$(A2\prime) \quad \models \quad \mathbf{w} \Rightarrow \Diamond \mathbf{w} \tag{A.11}$$

$$(A5\prime) \quad \models \bigcirc \mathbf{w} \Rightarrow \Diamond \mathbf{w} \tag{A.12}$$

$$(A6\prime) \quad \models \bigcirc \Diamond \mathbf{w} \Rightarrow \Diamond \mathbf{w} \tag{A.13}$$

$$(A7\prime) \quad \models (\mathbf{w} \wedge \Diamond(\neg \mathbf{w})) \Rightarrow \Diamond(\mathbf{w} \wedge \bigcirc(\neg \mathbf{w})) \tag{A.14}$$

$$(A7\prime\prime) \quad \models \Box(\bigcirc \mathbf{w} \Rightarrow \mathbf{w}) \Rightarrow (\Diamond \mathbf{w} \Rightarrow \mathbf{w}) \tag{A.15}$$

The rules of inference for this deductive system are:

$$\textit{If } \mathbf{p} \textit{ is an instance of a propositional tautology then } \models \mathbf{p} \tag{A.16}$$

$$\textit{If } \models \mathbf{p} \textit{ and } \models \mathbf{p} \Rightarrow \mathbf{q} \textit{ then } \models \mathbf{q} \textit{ (modus ponens)} \tag{A.17}$$

$$\textit{If } \models \mathbf{p} \textit{ then } \models \Box \mathbf{p} \tag{A.18}$$

We also include the following derived rules from various other sources as indicated.

From Feys[Fey65], page 12, equation(a10):

$$\models (\mathcal{P} \wedge \mathcal{Q}) \Rightarrow \mathcal{P} \tag{A.19}$$

From Karam[Kar87], page 85, equation (4.29):

$$\textit{if } \models \mathcal{P} \wedge \Diamond \mathcal{Q} \Rightarrow \textit{ and } \models \mathcal{Q} \Rightarrow \bigcirc \Diamond \mathcal{R} \textit{ then } \models \mathcal{P} \wedge \bigcirc \Diamond \mathcal{R} \tag{A.20}$$

From Karam[Kar87], page 85, equation (4.30):

$$\textit{if } \models \mathcal{P} \wedge \bigcirc \Diamond \mathcal{Q} \Rightarrow \textit{ and } \models \mathcal{Q} \Rightarrow \bigcirc \Diamond \mathcal{R} \textit{ then } \models \mathcal{P} \wedge \bigcirc \Diamond \mathcal{R} \tag{A.21}$$

From Karam[Kar87], page 85, equation (4.31):

$$\textit{if } \models \mathcal{P} \wedge \bigcirc \mathcal{Q} \Rightarrow \textit{ and } \models \mathcal{Q} \Rightarrow \bigcirc \Diamond \mathcal{R} \textit{ then } \models \mathcal{P} \wedge \bigcirc \Diamond \mathcal{R} \tag{A.22}$$

From Feys[Fey65], page 14, equation (a661):

$$\models [(\mathcal{P} \Rightarrow \mathcal{Q}) \wedge (\mathcal{R} \Rightarrow \mathcal{S})] \Leftrightarrow [(\mathcal{P} \wedge \mathcal{R}) \Rightarrow (\mathcal{Q} \wedge \mathcal{S})] \tag{A.23}$$

From Karam[Kar87], page 210, equation (B.40):

$$\textit{if } \models \mathcal{P} \textit{ and } \models \mathcal{Q} \textit{ then } \models \mathcal{P} \wedge \mathcal{Q} \tag{A.24}$$

Appendix B

Axiom System for BTL

This appendix presents an axiom system for the Unified Branching Time Temporal Logic UB as defined in [BAMP81]. The labels to the left of the axioms are taken from [BAMP81]. We will refer to the axioms by the labels to the right of each axiom.

$$(D1) \quad \models \quad \forall\Diamond\mathbf{p} \equiv \neg\exists\Box\neg\mathbf{p} \tag{B.1}$$

$$(D2) \quad \models \quad \exists\Diamond\mathbf{p} \equiv \neg\exists\Box\neg\mathbf{p} \tag{B.2}$$

$$(D3) \quad \models \quad \exists \bigcirc \mathbf{p} \equiv \neg\forall \bigcirc \neg\mathbf{p} \tag{B.3}$$

$$(A1) \quad \models \quad \forall\Box(\mathbf{p} \Rightarrow \mathbf{q}) \Rightarrow (\forall\Box\mathbf{p} \Rightarrow \forall\Box\mathbf{q}) \tag{B.4}$$

$$(A2) \quad \models \quad \forall \bigcirc (\mathbf{p} \Rightarrow \mathbf{q}) \Rightarrow (\forall \bigcirc \mathbf{p} \Rightarrow \forall \bigcirc \mathbf{q}) \tag{B.5}$$

$$(A3) \quad \models \quad \forall\Box\mathbf{p} \Rightarrow \forall \bigcirc \mathbf{p} \wedge \forall \bigcirc \forall\Box\mathbf{p} \tag{B.6}$$

$$(A4) \quad \models \quad \forall\Box(\mathbf{p} \Rightarrow \forall \bigcirc \mathbf{p}) \Rightarrow (\mathbf{p} \Rightarrow \forall\Box\mathbf{p}) \tag{B.7}$$

$$(E1) \quad \models \quad \forall\Box(\mathbf{p} \Rightarrow \mathbf{q}) \Rightarrow (\exists\Box\mathbf{p} \Rightarrow \exists\Box\mathbf{q}) \tag{B.8}$$

$$(E2) \quad \models \quad \exists\Box\mathbf{p} \Rightarrow \mathbf{p} \wedge \exists \bigcirc \exists\Box\mathbf{p} \tag{B.9}$$

$$(E3) \quad \models \quad \forall\Box\mathbf{p} \Rightarrow \exists\Box\mathbf{p} \tag{B.10}$$

$$(E4) \quad \models \quad \forall\Box(\mathbf{p} \Rightarrow \exists \bigcirc \mathbf{p}) \Rightarrow (\mathbf{p} \Rightarrow \exists\Box\mathbf{p}) \tag{B.11}$$

The rules of inference for this deductive system are:

$$\textit{If } \mathbf{p} \textit{ is an instance of a propositional tautology then } \models \mathbf{p} \tag{B.12}$$

$$\textit{If } \models \mathbf{p} \textit{ and } \models \mathbf{p} \Rightarrow \mathbf{q} \textit{ then } \models \mathbf{q} \ (\textit{modus ponens}) \tag{B.13}$$

$$\textit{If } \models \mathbf{p} \textit{ then } \models \forall\Box\mathbf{p} \tag{B.14}$$

Appendix C

Message Passing Example

In this appendix we present a series of axiomatic deductions to show how the PTR consisting of connective implications, an initial state assertion and the PAL language implications can be manipulated such that a message is passed from one process to another.

We will assume that the system is presently at the system state given as SSA (C.1): Relavent connective implications for both processes are given below as connective implications (c1) and (c2).

$$\models \left\{ \begin{array}{l} \bigcirc\Diamond(send\_msg(A, chan1) \wedge ps(A, prms[M], \{\})) \wedge \\ \bigcirc\Diamond(recv\_msg(B, chan1) \wedge ps(B, prms[], \{\})) \wedge \\ \{\} \wedge \\ \Diamond chan(chan1, [], 0, 1) \end{array} \right\} \qquad \text{(C.1)}$$

$$(c1) \quad \models end\_send(A, chan1) \wedge ps(A, prms([]), \{\}) \Rightarrow \bigcirc\Diamond(init(A) \wedge ps(A, prms([]), \{\}))$$

$$(c2) \quad \models end\_recv(B, chan1) \wedge ps(B, prms([M]), \{\}) \Rightarrow \bigcirc\Diamond(init(B) \wedge ps(B, prms([]), \{\}))$$

We will also refer to the PAL language implications 4.8 and 4.9 and to the LTL axioms given in Appendix A.

We will present four series of deductions in proof format. The first to show the transition of process A from a *send_msg* state to an *end_send* state. The second showing process A progressing from *end_send* to the *init* state. The third showing process B receiving the message, *recv_msg* to *end_recv* and the fourth showing process B progressing to the *init* state.

Before each series of deductions we will first give a list of major substitutions which should allow for a clearer presentation of the deductions. Minor substitutions within the proof will be shown as $[x_1/y_1, x_2/y_2, \ldots]$ where x_1 is being substituted for y_1, etc. The four series of deductions are presented on the following four pages.

C.0.1 Transition of $send\_msg(A, chan1)$ to $end\_send(A, chan1)$

Substitutions:

(s1) $\mathcal{Q}_1$ for $\Diamond(send\_msg(A, chan1) \wedge ps(A, prms([M]), \{\}))$

(s2) $\mathcal{Q}_2$ for $\Diamond chan(chan1, [], 0, 1)$

(s3) $\mathcal{Q}_3$ for $\Diamond(recv\_msg(B, chan1) \wedge ps(B, prms([]), \{\}))$

(s4) $\mathcal{R}_1$ for $end\_send(p, ch) \wedge ps(p, prms([]), \{\})$

(s5) $\mathcal{R}_2$ for $chan(ch, [q \| msg], n+1, m)$

Axiomatic deductions showing process A advancing from a *send_msg* state to an *end_send* state:

(1)	$\models \equiv$	$\bigcirc\mathcal{Q}_1 \wedge \mathcal{Q}_2 \wedge \bigcirc\mathcal{Q}_3$	(C.1) [s1,s2,s3]
(2)	$\models \equiv$	$\bigcirc\mathcal{Q}_1 \wedge \mathcal{P}_1$	(1) $[\mathcal{P}_1/(\mathcal{Q}_2 \wedge \bigcirc\mathcal{Q}_3)]$
(3)	$\models \equiv$	$(\mathcal{Q}_1 \wedge \mathcal{Q}_2) \Rightarrow (\bigcirc\Diamond\mathcal{R}_1 \wedge \mathcal{R}_2)$	(4.8) $\left[\begin{array}{l} A/p, M/msg, chan1/ch, \\ 0/n, 1/m, []/q, s1, s2, s4, s5 \end{array}\right]$
(4)	$\models \equiv$	$(\mathcal{Q}_1 \Rightarrow \bigcirc\Diamond\mathcal{R}_1) \wedge (\mathcal{Q}_2 \Rightarrow \bigcirc\Diamond\mathcal{R}_2)$	(3,A.23)
(5)	$\models \equiv$	$\mathcal{Q}_1 \Rightarrow \bigcirc\Diamond\mathcal{R}_1$	(4,A.19)
(6)	$\models \equiv$	$\mathcal{Q}_2 \Rightarrow \bigcirc\Diamond\mathcal{R}_2$	(4,A.19)
(7)	$\models \equiv$	$\mathcal{P}_1 \wedge \bigcirc\Diamond\mathcal{R}_1$	(2,5,A.22)
(8)	$\models \equiv$	$\mathcal{Q}_2 \wedge \bigcirc\mathcal{Q}_3 \wedge \bigcirc\Diamond\mathcal{R}_1$	(7) remove substitution (1)
(9)	$\models \equiv$	$\mathcal{Q}_2$	(8,A.19)
(10)	$\models \equiv$	$\bigcirc\mathcal{Q}_3 \wedge \bigcirc\Diamond\mathcal{R}_1$	(8,A.19)
(11)	$\models \equiv$	$\bigcirc\Diamond\mathcal{R}_2$	(6,9,A.17)
(12)	$\models \equiv$	$\bigcirc\Diamond\mathcal{R}_1 \wedge \bigcirc\mathcal{Q}_3 \wedge \bigcirc\Diamond\mathcal{R}_2$	(10,11,A.24)

With all substitutions removed (12) becomes:

$$\models \equiv \left\{ \begin{array}{l} \bigcirc\Diamond(end\_send(A, chan1) \wedge ps(A, prms[], \{\})) \wedge \\ \bigcirc\Diamond(recv\_msg(B, chan1) \wedge ps(B, prms[], \{\})) \wedge \\ \{\} \wedge \\ \bigcirc\Diamond chan(chan1, [M], 1, 1) \end{array} \right\} \qquad \text{(C.2)}$$

C.0.2 Transition of $end\_send(A, chan1)$ to $init(A)$

Substitutions:

(s1) $\mathcal{Q}_1$ for $\Diamond(end\_send(A, chan1) \wedge ps(A, prms([]), \{\}))$

(s2) $\mathcal{Q}_2$ for $\Diamond chan(chan1, [M], 1, 1)$

(s3) $\mathcal{Q}_3$ for $\Diamond(recv\_msg(B, chan1) \wedge ps(B, prms([]), \{\}))$

(s4) $\mathcal{R}$ for $init(A) \wedge ps(A, prms([]), \{\})$

Axiomatic deductions showing process A progressing from an *end_send* state back to the *init* state:

(1) $\models$ $\bigcirc\mathcal{Q}_1 \wedge \mathcal{Q}_2 \wedge \bigcirc\mathcal{Q}_3$ (C.2) [s1,s2,s3]

(2) $\models$ $\bigcirc\mathcal{Q}_1 \wedge \mathcal{P}_1$ (1) $[\mathcal{P}_1/(\mathcal{Q}_2 \wedge \bigcirc\mathcal{Q}_3)]$

(3) $\models$ $\mathcal{Q}_1 \Rightarrow \bigcirc\Diamond\mathcal{R}$ (c1) [s1,s4]

(4) $\models$ $\mathcal{P}_1 \wedge \bigcirc\Diamond\mathcal{R}$ (2,3,A.22)

(5) $\models$ $\bigcirc\Diamond\mathcal{R} \wedge \bigcirc\mathcal{Q}_3 \wedge \mathcal{Q}_2$ (4) remove substitution (2)

With all substitutions removed (5) becomes:

$$\models \left\{ \begin{array}{l} \bigcirc\Diamond(init(A) \wedge ps(A, prms[], \{\})) \wedge \\ \bigcirc\Diamond(recv\_msg(B, chan1) \wedge ps(B, prms[], \{\})) \wedge \\ \{\} \wedge \\ \bigcirc\Diamond chan(chan1, [M], 1, 1) \end{array} \right\} \qquad \text{(C.3)}$$

C.0.3 Transition of $recv\_msg(B, chan1)$ to $end\_recv(B, chan1)$

Substitutions:

(s1) $\mathcal{Q}_1$ for $\Diamond(recv\_msg(B, chan1) \wedge ps(B, prms([]), \{\}))$

(s2) $\mathcal{Q}_2$ for $\Diamond chan(chan1, [M], 1, 1)$

(s3) $\mathcal{Q}_3$ for $\Diamond(init(A) \wedge ps(A, prms([]), \{\}))$

(s4) $\mathcal{R}_1$ for $end\_recv(p, ch) \wedge ps(p, prms([msg]), \{\})$

(s5) $\mathcal{R}_2$ for $chan(ch, q, n-1, m)$

Axiomatic deductions showing process B progressing from a *recv_msg* state to the *end_recv* state:

(1)	$\models$	$\bigcirc\mathcal{Q}_1 \wedge \mathcal{Q}_2 \wedge \bigcirc\mathcal{Q}_3$	(C.3) [s1,s2,s3]
(2)	$\models$	$\bigcirc\mathcal{Q}_1 \wedge \mathcal{P}_1$	(1) $[\mathcal{P}_1/(\mathcal{Q}_2 \wedge \bigcirc\mathcal{Q}_3)]$
(3)	$\models$	$(\mathcal{Q}_1 \wedge \mathcal{Q}_2) \Rightarrow (\bigcirc\Diamond\mathcal{R}_1 \wedge \mathcal{R}_2)$	(4.9) $\left[\begin{array}{l} B/p, M/msg, chan1/ch, \\ 1/n, 1/m, [M]/q, s1, s2, s4, s5 \end{array}\right]$
(4)	$\models$	$(\mathcal{Q}_1 \Rightarrow \bigcirc\Diamond\mathcal{R}_1) \wedge (\mathcal{Q}_2 \Rightarrow \bigcirc\Diamond\mathcal{R}_2)$	(3,A.23)
(5)	$\models$	$\mathcal{Q}_1 \Rightarrow \bigcirc\Diamond\mathcal{R}_1$	(4,A.19)
(6)	$\models$	$\mathcal{Q}_2 \Rightarrow \bigcirc\Diamond\mathcal{R}_2$	(4,A.19)
(7)	$\models$	$\mathcal{P}_1 \wedge \bigcirc\Diamond\mathcal{R}_1$	(2,5,A.22
(8)	$\models$	$\mathcal{Q}_2 \wedge \bigcirc\mathcal{Q}_3 \wedge \bigcirc\Diamond\mathcal{R}_1$	(7) remove substitution (1)
(9)	$\models$	$\mathcal{Q}_2$	(8,A.19)
(10)	$\models$	$\bigcirc\mathcal{Q}_3 \wedge \bigcirc\Diamond\mathcal{R}_1$	(8,A.19)
(11)	$\models$	$\bigcirc\Diamond\mathcal{R}_2$	(6,9,A.17)
(12)	$\models$	$\bigcirc\Diamond\mathcal{R}_1 \wedge \bigcirc\mathcal{Q}_3 \wedge \bigcirc\Diamond\mathcal{R}_2$	(10,11,A.24)

With all substitutions removed (12) becomes:

$$\models \left\{\begin{array}{l} \bigcirc\Diamond(init(A) \wedge ps(A, prms[], \{\})) \wedge \\ \bigcirc\Diamond(end\_recv(B, chan1) \wedge ps(B, prms[M], \{\})) \wedge \\ \{\} \wedge \\ \bigcirc\Diamond chan(chan1, [], 0, 1) \end{array}\right\} \qquad \text{(C.4)}$$

C.0.4 Transition of $end\_recv(B, chan1)$ to $init(B)$

Substitutions:

(s1) $\mathcal{Q}_1$ for $\Diamond(end\_recv(B, chan1) \wedge ps(B, prms([]), \{\}))$

(s2) $\mathcal{Q}_2$ for $\Diamond chan(chan1, [], 0, 1)$

(s3) $\mathcal{Q}_3$ for $\Diamond(init(A) \wedge ps(A, prms([]), \{\}))$

(s4) $\mathcal{R}$ for $init(B) \wedge ps(B, prms([]), \{\})$

Axiomatic deductions showing process B progressing from the *end_recv* state back to the *init* state:

(1) $\models$ $\bigcirc\mathcal{Q}_1 \wedge \mathcal{Q}_2 \wedge \bigcirc\mathcal{Q}_3$ (C.4) [s1,s2,s3]

(2) $\models$ $\bigcirc\mathcal{Q}_1 \wedge \mathcal{P}_1$ (1) $[\mathcal{P}_1/(\mathcal{Q}_2 \wedge \bigcirc\mathcal{Q}_3)]$

(3) $\models$ $\mathcal{Q}_1 \Rightarrow \bigcirc\Diamond\mathcal{R}$ (c2) [s1,s4]

(4) $\models$ $\mathcal{P}_1 \wedge \bigcirc\Diamond\mathcal{R}$ (2,3,A.22)

(5) $\models$ $\bigcirc\mathcal{Q}_3 \wedge \bigcirc\Diamond\mathcal{R} \wedge \mathcal{Q}_2$ (4) remove substitution (2)

With all substitutions removed (5) becomes:

$$\models \left\{ \begin{array}{l} \bigcirc\Diamond(init(A) \wedge ps(A, prms[], \{\})) \wedge \\ \bigcirc\Diamond(init(B) \wedge ps(B, prms[], \{\})) \wedge \\ \{\} \wedge \\ \bigcirc\Diamond chan(chan1, [], 0, 1) \end{array} \right\} \qquad \text{(C.5)}$$

Appendix D

Model Checking Code

This appendix contains a listing of the the Prolog clauses implementing the model checking algorithms for each of the BTL operators. A listing of the Prolog source code for the complete verification system can be found in [Har89].

The $\forall\Box$ Operator

```
/************************************/
/* For ALL paths F is ALWAYS true  */
/************************************/
/* For ALL paths starting at this state */
/* F is ALWAYS true                     */

check(ag(F),S) :-
        check(ag(F),[],S),
        label_state(ag(F),S).

check(ag(F),_,S) :-
        visited1(S,ag(F)).
check(ag(F),_,S) :-
```

```
        label(S,L),
        member(ag(F),L).
check(ag(F),Path,S) :-
        (check(F,S);
           add_statement(result(ag(F),Path,S)),-),
        add_statement(visited1(S,ag(F))),
        next_states(S,B),!,
        all_paths(ag(F),B,[S|Path]).

all_paths(_,[],_).
all_paths(F,[S|R],[S|Path]) :-
        !,all_paths(F,R,[S|Path]).
all_paths(F,[S|R],Path) :-
        check(F,Path,S),!,
        all_paths(F,R,Path).
```

The $\exists\Box$ and $\exists\Box^f$ Operators

```
/************************************/
/* For SOME path F is ALWAYS true  */
/************************************/
/* There EXISTS a path starting at this   */
/* state along which F is ALWAYS true     */

check(eg(F),S) :-
        check(eg(F),[],S).
check(eg(F),_,S) :-
        label(S,L),
        member(eg(F),L).
```

```
check(eg(F),Path,S) :-
        member(S,Path),
        tree(S,[_]),
        (check_fair(S,Path),
         add_statement(result(eg(F),Path,S));-).
check(eg(F),Path,S) :-
        check(F,S),
        next_states(S,B),!,
        some_path(eg(F),B,[S|Path]),
        label_state(eg(F),S).

some_path(F,[],_) :- -.
some_path(F,[S|R],Path) :-
        (check(F,Path,S);
        !,some_path(F,R,Path)).
```

The $\forall\Diamond$ and $\forall\Diamond^f$ Operators

```
/*************************************/
/* For ALL paths F is EVENTUALLY true */
/*************************************/
/* For ALL paths starting at this state   */
/* F is EVENTUALLY true.                  */

check(af(F),S) :-
        check(af(F),[],S),
        label_state(af(F),S).
check(af(F),_,S) :-
        label(S,L),
```

```
        member(af(F),L).
check(af(F),Path,S) :-
        member(S,Path),
        tree(S,[_]),
        (check_fair(S,Path),
         add_statement(result(af(F),Path,S)),-;
        succeed).
check(af(F),_,S) :-
        check(F,S),!,
        label_state(af(F),S).
check(af(F),Path,S) :-
        next_states(S,B),!,
        all_paths(af(F),B,[S|Path]),
        all_paths_labelled(af(F),B,S).

all_paths_labelled(F,[S1|Rest],S) :-
        label(S1,L),
        member(F,L),
        !,all_paths_labelled(F,Rest,S).
all_paths_labelled(F,[],S) :-
        label_state(F,S).
all_paths_labelled(_,_,_).
```

The all_paths_labelled/3 predicate ensures that all next states to the current state have been labelled with af(F) before the current state is labelled. Without this test it is possible that the current state would be labelled with af(F) based solely on unfair computations.

The $\exists\Diamond$ Operator

```
/*************************************/
/* For SOME path F is EVENTUALLY true */
/*************************************/
/* There exists a path starting at this state  */
/* along which F is eventually true.           */

check(ef(F),S) :-
        check(ef(F),[],S).
check(ef(_),_,S) :-
        visited2(S,ef(F)),-.
check(ef(F),_,S) :-
        label(S,L),
        member(ef(F),L).
check(ef(F),Path,S) :-
        check(F,S),!,
        label_state(ef(F),S),
        add_statement(result(ef(F),Path,S)).
check(ef(F),Path,S) :-
        add_statement(visited2(S,ef(F))),
        next_states(S,B),!,
        some_path(ef(F),B,[S|Path]),
        label_state(ef(F),S).
```

The $\forall\bigcirc$ Operator

```
/*************************************/
/* F is true for all NEXT STATES      */
/*************************************/
```

```
/* For all next states from the current  */
/* F is true.                            */

check(ax(F),S) :-
        label(S,L),
        member(ax(F),L).
check(ax(F),S) :-
        next_states(S,B),!,
        all_paths(F,B,[S]),
        label_state(ax(F),S).
```

The $\exists\bigcirc$ Operator

```
/***************************************/
/* F is true for some NEXT STATE       */
/***************************************/
/* There exists a next state from the  */
/* current state at which F is true.   */

check(ex(F),S) :-
        label(S,L),
        member(ex(F),L).
check(ex(F),S) :-
        next_states(S,B),!,
        some_path(F,B,[S]),
        label_state(ex(F),S).
```

Appendix E

Example Problems

In this appendix we give the PAL expressions, connective implications and a brief description of each of the example system problems discussed in this thesis.

E.1 Simple Messaging System

PAL Process Expressions

Process S

S::=
$\mathcal{S}_{on}$
send(chan1,m);
$\mathcal{E}$

Process R

R::=
$\mathcal{S}_{on}$
receive(chan1,m);
$\mathcal{E}$

PAL Temporal Representation

```
init_state(1,[<>(init(s) & ps(s,[[],[count(0)]])),
              <>(init(r) & ps(r,[[],[]]))],
             [],
             [<>chan(c1,[],0,2)]).
```

```
init(r) & ps(r,[[],[]]) ==>
   *<>(recv_msg(r,c1) & ps(r,[[],[]])).
end_recv(r,c1) & ps(r,[[m],[]]) ==>
   *<>(init(r) & ps(r,[[],[]])).

init(s) & ps(s,[[],[]]) ==>
   *<>(send_msg(s,c1) & ps(s,[[m],[]])).
end_send(s,c1) & ps(s,[[],[]]) ==>
   *<>(init(s) & ps(s,[[],[]])).
```

E.2 Simple Global Variable System

PAL Process Expressions

Process Read

Read::=
$\mathcal{S}_{nn}$
RDS(svar,rc);
$\mathcal{E}$

Process Write

Write::=
$\mathcal{S}_{nn}$
case wc
wc $<$ 2 : STP(wc,wc + 1) |
wc $\geq$ 2 : STP(wc,0)
endcase;
STS(svar,wc);
$\mathcal{E}$

PAL Temporal Representation

```
init_state(1,[<>(init(r) & ps(r,[[],[rc(0)]])),
              <>(init(w) & ps(w,[[],[wc(0)]]))],
             [sc(0)],
             []).

init(r) & ps(r,[[],[]]) ==>
```

```
   *<>(rds(r,r1) & ps(r,[[],[]])).
rds(r,r1,sc(X),[rc(X)]) & ps(r,[[],[]]) ==>
   *<>(end_rds(r,r1) & ps(r,[[],[]])).
end_rds(r,r1) & ps(r,[[],[]]) ==>
   *<>(init(r) & ps(r,[[],[]])).

init(w) & ps(w,[[],[wc(X),@(X<2)]]) ==>
   *<>(sts(w,w1) & ps(w,[[],[@(X1 is X + 1),wc(X1)]])).
init(w) & ps(w,[[],[wc(X),@(X=2)]]) ==>
   *<>(sts(w,w1) & ps(w,[[],[wc(0)]])).
sts(w,w1,[wc(Y),sc(Y)]) & ps(w,[[],[]]) ==>
   *<>(end_sts(w,w1) & ps(w,[[],[]])).
end_sts(w,w1) & ps(w,[[],[]]) ==>
   *<>(init(w) & ps(w,[[],[]])).
```

E.3 Readers/Writers with starvation

PAL Process Expressions

Scheduler Process

```
sch::=
    S_on
    select
        write_ok →
            receive(ch1,req(wr));
            send(ch2,ack);
            STP(read_ok,false);
            STP(write_ok,false) |
        read ok →
            receive(ch5,req(rd1));
            send(ch6,ack);
            STP(nr,nr + 1);
            STP(write_ok,false) |
        read_ok →
            receive(ch9,req(rd2));
            send(ch10,ack);
            STP(nr,nr + 1);
            STP(write ok,false) |
        true →
            receive(ch3,rel(wr));
            send(ch4,ack);
            STP(write_ok,true);
            STP(read_ok,true) |
        true →
            receive(ch7,rel(rd1));
            send(ch8,ack);
            STP(nr,nr - 1);
            case nr
                nr = 0: STP(write_ok,true) |
                nr ≠ 0: true
            endcase |
        true →
            receive(ch11,rel(rd2));
            send(ch12,ack);
            STP(nr,nr - 1);
            case nr
                nr = 0: STP(write ok,true) |
                nr ≠ 0: true
            endcase
    endselect;
    ε
```

Writer Process

```
wr::=
    S_on
    send(ch1,req(wr));
    receive(ch2,ack);
    do_modify_activity;
    send(ch3,rel(wr));
    receive(ch4,ack);
    ε
```

First Reader Process

```
rd1::=
    S_on
    send(ch5,req(rd1));
    receive(ch6,ack);
    do_reading_activity;
    send(ch7,rel(rd1));
    receive(ch8,ack);
    ε
```

Second Reader Process

```
rd2::=
    S_on
    send(ch9,req(rd2));
    receive(ch10,ack);
    do_read_activity;
    send(ch11,rel(rd2));
    receive(ch12,ack);
    ε
```

PAL Temporal Representation

```
init_state(1,[<>(init(sch) &
                   ps(sch,[[],[nr(0),write_ok(t),read_ok(t)]])),
               <>(init(wr) & ps(wr,[[],[]])),
               <>(init(rd1) & ps(rd1,[[],[]])),
               <>(init(rd2) & ps(rd2,[[],[]]))],
              [],
              [<>chan(ch1,[],0,1),<>chan(ch2,[],0,1),<>chan(ch3,[],0,1),
               <>chan(ch4,[],0,1),<>chan(ch5,[],0,1),<>chan(ch6,[],0,1),
               <>chan(ch7,[],0,1),<>chan(ch8,[],0,1),<>chan(ch9,[],0,1),
               <>chan(ch10,[],0,1),<>chan(ch11,[],0,1),<>chan(ch12,[],0,1)
              ]).

init(rd1) & ps(rd1,[[],[]]) ==>
    *<>(send_msg(rd1,ch5) & ps(rd1,[[req(rd1)],[]])).
end_send(rd1,ch5) & ps(rd1,[[],[]]) ==>
    *<>(recv_msg(rd1,ch6) & ps(rd1,[[],[]])).
end_recv(rd1,ch6) & ps(rd1,[[ack],[]]) ==>
    *<>(send_msg(rd1,ch7) & ps(rd1,[[rel(rd1)],[]])).
end_send(rd1,ch7) & ps(rd1,[[],[]]) ==>
    *<>(recv_msg(rd1,ch8) & ps(rd1,[[],[]])).
end_recv(rd1,ch8) & ps(rd1,[[ack],[]]) ==>
    *<>(init(rd1) & ps(rd1,[[],[]])).

init(rd2) & ps(rd2,[[],[]]) ==>
    *<>(send_msg(rd2,ch9) & ps(rd2,[[req(rd2)],[]])).
end_send(rd2,ch9) & ps(rd2,[[],[]]) ==>
    *<>(recv_msg(rd2,ch10) & ps(rd2,[[],[]])).
end_recv(rd2,ch10) & ps(rd2,[[ack],[]]) ==>
```

```
    *<>(send_msg(rd2,ch11) & ps(rd2,[[rel(rd2)],[]])).
end_send(rd2,ch11) & ps(rd2,[[],[]]) ==>
    *<>(recv_msg(rd2,ch12) & ps(rd2,[[],[]])).
end_recv(rd2,ch12) & ps(rd2,[[ack],[]]) ==>
    *<>(init(rd2) & ps(rd2,[[],[]])).

init(wr) & ps(wr,[[],[]]) ==>
    *<>(send_msg(wr,ch1) & ps(wr,[[req(wr)],[]])).
end_send(wr,ch1) & ps(wr,[[],[]]) ==>
    *<>(recv_msg(wr,ch2) & ps(wr,[[],[]])).
end_recv(wr,ch2) & ps(wr,[[ack],[]]) ==>
    *<>(send_msg(wr,ch3) & ps(wr,[[rel(wr)],[]])).
end_send(wr,ch3) & ps(wr,[[],[]]) ==>
    *<>(recv_msg(wr,ch4) & ps(wr,[[],[]])).
end_recv(wr,ch4) & ps(wr,[[ack],[]]) ==>
    *<>(init(wr) & ps(wr,[[],[]])).

init(sch) & ps(sch,[[],[]]) ==>
    *<>(select(sch,sch1) & ps(sch,[[],[]])).
::=(select(sch,sch1) & ps(sch,[[],[]]),
        [ [read_ok(t)],*<>(recv_msg(sch,ch5) & ps(sch,[[],[]])),
        [read_ok(t)],*<>(recv_msg(sch,ch9) & ps(sch,[[],[]])),
        [write_ok(t)],*<>(recv_msg(sch,ch1) & ps(sch,[[],[]])),
        [],*<>(recv_msg(sch,ch7) & ps(sch,[[],[]])),
        [],*<>(recv_msg(sch,ch11) & ps(sch,[[],[]])),
        [],*<>(recv_msg(sch,ch3) & ps(sch,[[],[]]))  ]).
end_select(sch,sch1) & ps(sch,[[],[]]) ==>
    *<>(init(sch) & ps(sch,[[],[]])).

end_recv(sch,ch5) & ps(sch,[[req(rd1)],[nr(X)]]) ==>
```

```
    *<>(send_msg(sch,ch6) &
        ps(sch,[[ack],[@(X1 is X + 1),nr(X1),write_ok(f)]])).
end_send(sch,ch6) & ps(sch,[[],[]]) ==>
    *<>(end_select(sch,sch1) & ps(sch,[[],[]])).

end_recv(sch,ch9) & ps(sch,[[req(rd2)],[nr(X)]]) ==>
    *<>(send_msg(sch,ch10) &
        ps(sch,[[ack],[@(X1 is X + 1),nr(X1),write_ok(f)]])).
end_send(sch,ch10) & ps(sch,[[],[]]) ==>
    *<>(end_select(sch,sch1) & ps(sch,[[],[]])).

end_recv(sch,ch1) & ps(sch,[[req(wr)],[]]) ==>
    *<>(send_msg(sch,ch2) & ps(sch,[[ack],[read_ok(f),write_ok(f)]])).
end_send(sch,ch2) & ps(sch,[[],[]]) ==>
    *<>(end_select(sch,sch1) & ps(sch,[[],[]])).

end_recv(sch,ch7) & ps(sch,[[rel(rd1)],[nr(1)]]) ==>
    *<>(send_msg(sch,ch8) & ps(sch,[[ack],[nr(0),write_ok(t)]])).
end_send(sch,ch8) & ps(sch,[[],[]]) ==>
    *<>(end_select(sch,sch1) & ps(sch,[[],[]])).

end_recv(sch,ch7) & ps(sch,[[rel(rd1)],[nr(X),@(X > 0)]]) ==>
    *<>(send_msg(sch,ch8) & ps(sch,[[ack],[@(X1 is X - 1),nr(X1)]])).
end_send(sch,ch8) & ps(sch,[[],[]]) ==>
    *<>(end_select(sch,sch1) & ps(sch,[[],[]])).

end_recv(sch,ch11) & ps(sch,[[rel(rd2)],[nr(1)]]) ==>
    *<>(send_msg(sch,ch12) & ps(sch,[[ack],[nr(0),write_ok(t)]])).
end_send(sch,ch12) & ps(sch,[[],[]]) ==>
    *<>(end_select(sch,sch1) & ps(sch,[[],[]])).
```

```
end_recv(sch,ch11) & ps(sch,[[rel(rd2)],[nr(X),@(X > 0)]]) ==>
    *<>(send_msg(sch,ch12) & ps(sch,[[ack],[@(X1 is X - 1),nr(X1)]])).
end_send(sch,ch12) & ps(sch,[[],[]]) ==>
    *<>(end_select(sch,sch1) & ps(sch,[[],[]])).

end_recv(sch,ch3) & ps(sch,[[rel(wr)],[]]) ==>
    *<>(send_msg(sch,ch4) & ps(sch,[[ack],[read_ok(t),write_ok(t)]])).
end_send(sch,ch4) & ps(sch,[[],[]]) ==>
    *<>(end_select(sch,sch1) & ps(sch,[[],[]])).
```

E.4 Readers/Writers without starvation

PAL Process Expressions

Scheduler Process

```
sch::=
    S_on
    select
        true →
            receive(ch1,req(wr));
            case nr
                nr > 0: STP(write_pend,true) |
                nr = 0: send(ch2,ack);
                    STP(writing,true) |
            endcase |
        (not writing) and (not write_pend) →
            receive(ch5,req(rd1));
            send(ch6,ack);
            STP(nr,nr + 1) |
        (not writing) and (not write pend) →
            receive(ch9,req(rd2));
            send(ch10,ack);
            STP(nr,nr + 1) |
        true →
            receive(ch3,rel(wr));
            send(ch4,ack);
            STP(writing,false) |
        true →
            receive(ch7,rel(rd1));
            send(ch8,ack);
            STP(nr,nr - 1);
            case nr
                nr = 0:
                    case write_pend
                        write pend:
                            send(ch2,ack);
                            STP(writing,true);
                            STP(write pend,false) |
                        not write_pend: true
                    endcase |
                nr ≠ 0: true
            endcase |
        true →
            receive(ch11,rel(rd2));
            send(ch12,ack);
            STP(nr,nr - 1);
            case nr
                nr = 0:
                    case write_pend
                        write pend:
                            send(ch2,ack);
                            STP(writing,true);
                            STP(write pend,false) |
                        not write_pend: true
                    endcase |
                nr ≠ 0: true
            endcase
    endselect;
    S
```

Writer Process

```
wr::=
    S_on
    send(ch1,req(wr));
    receive(ch2,ack);
    do_modify_activity;
    send(ch3,rel(wr));
    receive(ch4,ack);
    ε
```

First Reader Process

```
rd1::=
    S_on
    send(ch5,req(rd1));
    receive(ch6,ack);
    do_reading_activity;
    send(ch7,rel(rd1));
    receive(ch8,ack);
    ε
```

Second Reader Process

```
rd2::=
    S_on
    send(ch9,req(rd2));
    receive(ch10,ack);
    do_read_activity;
    send(ch11,rel(rd2));
    receive(ch12,ack);
    ε
```

PAL Temporal Representation

```
init_state(1,[<>(init(rd1) & ps(rd1,[[],[]])),
              <>(init(rd2) & ps(rd2,[[],[]])),
              <>(init(wr) & ps(wr,[[],[]])),
              <>(init(sch) &
                 ps(sch,[[],[nr(0),writing(f),write_pend(f)]]))],
             [],
             [<>chan(ch1,[],0,1),<>chan(ch2,[],0,1),<>chan(ch3,[],0,1),
              <>chan(ch4,[],0,1),<>chan(ch5,[],0,1),<>chan(ch6,[],0,1),
              <>chan(ch7,[],0,1),<>chan(ch8,[],0,1),<>chan(ch9,[],0,1),
              <>chan(ch10,[],0,1),<>chan(ch11,[],0,1),<>chan(ch12,[],0,1)]).

init(wr) & ps(wr,[[],[]]) ==>
    *<>(send_msg(wr,ch1) & ps(wr,[[req(wr)],[]])).
end_send(wr,ch1) & ps(wr,[[],[]]) ==>
    *<>(recv_msg(wr,ch2) & ps(wr,[[],[]])).
end_recv(wr,ch2) & ps(wr,[[ack],[]]) ==>
    *<>(send_msg(wr,ch3) & ps(wr,[[rel(wr)],[]])).
end_send(wr,ch3) & ps(wr,[[],[]]) ==>
    *<>(recv_msg(wr,ch4) & ps(wr,[[],[]])).
end_recv(wr,ch4) & ps(wr,[[ack],[]]) ==>
    *<>(init(wr) & ps(wr,[[],[]])).

init(rd1) & ps(rd1,[[],[]]) ==>
    *<>(send_msg(rd1,ch5) & ps(rd1,[[req(rd1)],[]])).
end_send(rd1,ch5) & ps(rd1,[[],[]]) ==>
    *<>(recv_msg(rd1,ch6) & ps(rd1,[[],[]])).
end_recv(rd1,ch6) & ps(rd1,[[ack],[]]) ==>
    *<>(send_msg(rd1,ch7) & ps(rd1,[[rel(rd1)],[]])).
```

```
end_send(rd1,ch7) & ps(rd1,[[],[]]) ==>
    *<>(recv_msg(rd1,ch8) & ps(rd1,[[],[]])).
end_recv(rd1,ch8) & ps(rd1,[[ack],[]]) ==>
    *<>(init(rd1) & ps(rd1,[[],[]])).

init(rd2) & ps(rd2,[[],[]]) ==>
    *<>(send_msg(rd2,ch9) & ps(rd2,[[req(rd2)],[]])).
end_send(rd2,ch9) & ps(rd2,[[],[]]) ==>
    *<>(recv_msg(rd2,ch10) & ps(rd2,[[],[]])).
end_recv(rd2,ch10) & ps(rd2,[[ack],[]]) ==>
    *<>(send_msg(rd2,ch11) & ps(rd2,[[rel(rd2)],[]])).
end_send(rd2,ch11) & ps(rd2,[[],[]]) ==>
    *<>(recv_msg(rd2,ch12) & ps(rd2,[[],[]])).
end_recv(rd2,ch12) & ps(rd2,[[ack],[]]) ==>
    *<>(init(rd2) & ps(rd2,[[],[]])).

init(sch) & ps(sch,[[],[]]) ==>
    *<>(select(sch,sch1) & ps(sch,[[],[]])).
::=(select(sch,sch1) & ps(sch,[[],[]]),
        [ [write_pend(f),writing(f)],*<>(recv_msg(sch,ch5) &
                                         ps(sch,[[],[]])),
          [write_pend(f),writing(f)],*<>(recv_msg(sch,ch9) &
                                         ps(sch,[[],[]])),
          [],*<>(recv_msg(sch,ch1) & ps(sch,[[],[]])),
          [],*<>(recv_msg(sch,ch7) & ps(sch,[[],[]])),
          [],*<>(recv_msg(sch,ch11) & ps(sch,[[],[]])),
          [],*<>(recv_msg(sch,ch3) & ps(sch,[[],[]]))  ]).
end_select(sch,sch1) & ps(sch,[[],[]]) ==>
    *<>(init(sch) & ps(sch,[[],[]])).
```

```
end_recv(sch,ch5) & ps(sch,[[req(rd1)],[nr(X)]]) ==>
    *<>(send_msg(sch,ch6) &
        ps(sch,[[ack],[@(X1 is X + 1),nr(X1)]])).
end_send(sch,ch6) & ps(sch,[[],[]]) ==>
    *<>(end_select(sch,sch1) & ps(sch,[[],[]])).

end_recv(sch,ch9) & ps(sch,[[req(rd2)],[nr(X)]]) ==>
    *<>(send_msg(sch,ch10) &
        ps(sch,[[ack],[@(X1 is X + 1),nr(X1)]])).
end_send(sch,ch10) & ps(sch,[[],[]]) ==>
    *<>(end_select(sch,sch1) & ps(sch,[[],[]])).

end_recv(sch,ch1) & ps(sch,[[req(wr)],[nr(X),@(X > 0)]]) ==>
    *<>(end_select(sch,sch1) & ps(sch,[[],[write_pend(t)]])).
end_recv(sch,ch1) & ps(sch,[[req(wr)],[nr(0)]]) ==>
    *<>(send_msg(sch,ch2) &
        ps(sch,[[ack],[writing(t),write_pend(f)]])).

end_recv(sch,ch7) & ps(sch,[[rel(rd1)],[]]) ==>
    *<>(send_msg(sch,ch8) & ps(sch,[[ack],[]])).
end_send(sch,ch8) & ps(sch,[[],[nr(1),write_pend(f)]]) ==>
    *<>(end_select(sch,sch1) & ps(sch,[[],[nr(0)]])).
end_send(sch,ch8) & ps(sch,[[],[nr(1),write_pend(t)]]) ==>
    *<>(send_msg(sch,ch2) &
        ps(sch,[[ack],[nr(0),writing(t),write_pend(f)]])).
end_send(sch,ch2) & ps(sch,[[],[]]) ==>
    *<>(end_select(sch,sch1) & ps(sch,[[],[]])).
end_send(sch,ch8) & ps(sch,[[],[nr(X),@(X>0)]]) ==>
    *<>(end_select(sch,sch1) &
        ps(sch,[[],[@(X1 is X - 1),nr(X1)]])).
```

```
end_recv(sch,ch11) & ps(sch,[[rel(rd2)],[]]) ==>
    *<>(send_msg(sch,ch12) & ps(sch,[[ack],[]])).
end_send(sch,ch12) & ps(sch,[[],[nr(1),write_pend(f)]]) ==>
    *<>(end_select(sch,sch1) & ps(sch,[[],[nr(0)]])).
end_send(sch,ch12) & ps(sch,[[],[nr(1),write_pend(t)]]) ==>
    *<>(send_msg(sch,ch2) &
        ps(sch,[[ack],[nr(0),writing(t),write_pend(f)]])).
end_send(sch,ch12) & ps(sch,[[],[nr(X),@(X>1)]]) ==>
    *<>(end_select(sch,sch1) &
        ps(sch,[[],[@(X1 is X - 1),nr(X1)]])).

end_recv(sch,ch3) & ps(sch,[[rel(wr)],[]]) ==>
    *<>(send_msg(sch,ch4) & ps(sch,[[ack],[writing(f)]])).
end_send(sch,ch4) & ps(sch,[[],[]]) ==>
    *<>(end_select(sch,sch1) & ps(sch,[[],[writing(f)]])).
```

E.5 Readers/Writers with starvation and global variables

PAL Process Expressions

Scheduler Process

```
sch::=
    S_on
    select
        write_ok →
            receive(ch1,req(wr));
            send(ch2,ack);
            STP(read_ok,false);
            STP(write_ok,false) |
        read ok →
            receive(ch5,req(rd1));
            send(ch6,ack);
            STP(nr,nr + 1);
            STP(write ok,false) |
        read ok →
            receive(ch9,req(rd2));
            send(ch10,ack);
            STP(nr,nr + 1);
            STP(write ok,false) |
        true →
            receive(ch3,rel(wr));
            send(ch4,ack);
            STP(write_ok,true);
            STP(read_ok,true) |
        true →
            receive(ch7,rel(rd1));
            send(ch8,ack);
            STP(nr,nr - 1);
            case nr
                nr = 0: STP(write_ok,true) |
                nr ≠ 0: true
            endcase |
        true →
            receive(ch11,rel(rd2));
            send(ch12,ack);
            STP(nr,nr - 1);
            case nr
                nr = 0: STP(write ok,true) |
                nr ≠ 0: true
            endcase
    endselect;
    ε
```

Writer Process

```
wr::=
    S_on
    send(ch1,req(wr));
    receive(ch2,ack);
    case my_var
        my_var: STP(my_var,false) |
        not my_var: STP(my_var,true)
    endcase;
    STS(data var,my var);
    send(ch3,rel(wr));
    receive(ch4,ack);
    ε
```

First Render Process

```
rd1::=
    S_on
    send(ch5,req(rd1));
    receive(ch6,ack);
    RDS(data var,read var);
    send(ch7,rel(rd1));
    receive(ch8,ack);
    ε
```

Second Render Process

```
rd2::=
    S_on
    send(ch9,req(rd2));
    receive(ch10,ack);
    do_read_activity;
    send(ch11,rel(rd2));
    receive(ch12,ack);
    ε
```

PAL Temporal Representation

```
init_state(1,[<>(init(sch) & ps(sch,[[],[nr(0),write(t),read(t)]])),
              <>(init(wr) & ps(wr,[[],[pvw(t)]])),
              <>(init(rd1) & ps(rd1,[[],[pvr(f)]])),
              <>(init(rd2) & ps(rd2,[[],[]]))],
             [sv1(t)],
             [<>chan(ch1,[],0,1),<>chan(ch2,[],0,1),<>chan(ch3,[],0,1),
              <>chan(ch4,[],0,1),<>chan(ch5,[],0,1),<>chan(ch6,[],0,1),
              <>chan(ch7,[],0,1),<>chan(ch8,[],0,1),<>chan(ch9,[],0,1),
              <>chan(ch10,[],0,1),<>chan(ch11,[],0,1),<>chan(ch12,[],0,1)
             ]).

init(rd1) & ps(rd1,[[],[]]) ==>
   *<>(send_msg(rd1,ch5) & ps(rd1,[[req(rd1)],[]])).
end_send(rd1,ch5) & ps(rd1,[[],[]]) ==>
   *<>(recv_msg(rd1,ch6) & ps(rd1,[[],[]])).
end_recv(rd1,ch6) & ps(rd1,[[ack],[]]) ==>
   *<>(rds(rd1,rd) & ps(rd1,[[],[]])).
rds(rd1,rd,sv1(X),[pvr(X)]) & ps(rd1,[[],[]]) ==>
   *<>(end_rds(rd1,rd) & ps(rd1,[[],[]])).
end_rds(rd1,rd) & ps(rd1,[[],[]]) ==>
   *<>(send_msg(rd1,ch7) & ps(rd1,[[rel(rd1)],[]])).
end_send(rd1,ch7) & ps(rd1,[[],[]]) ==>
   *<>(recv_msg(rd1,ch8) & ps(rd1,[[],[]])).
end_recv(rd1,ch8) & ps(rd1,[[ack],[]]) ==>
   *<>(init(rd1) & ps(rd1,[[],[]])).

init(rd2) & ps(rd2,[[],[]]) ==>
   *<>(send_msg(rd2,ch9) & ps(rd2,[[req(rd2)],[]])).
```

```
end_send(rd2,ch9) & ps(rd2,[[],[]]) ==>
   *<>(recv_msg(rd2,ch10) & ps(rd2,[[],[]])).
end_recv(rd2,ch10) & ps(rd2,[[ack],[]]) ==>
   *<>(send_msg(rd2,ch11) & ps(rd2,[[rel(rd2)],[]])).
end_send(rd2,ch11) & ps(rd2,[[],[]]) ==>
   *<>(recv_msg(rd2,ch12) & ps(rd2,[[],[]])).
end_recv(rd2,ch12) & ps(rd2,[[ack],[]]) ==>
   *<>(init(rd2) & ps(rd2,[[],[]])).

init(wr) & ps(wr,[[],[]]) ==>
   *<>(send_msg(wr,ch1) & ps(wr,[[req(wr)],[]])).
end_send(wr,ch1) & ps(wr,[[],[]]) ==>
   *<>(recv_msg(wr,ch2) & ps(wr,[[],[]])).
end_recv(wr,ch2) & ps(wr,[[ack],[pvw(f)]]) ==>
   *<>(sts(wr,w1) & ps(wr,[[],[pvw(t)]])).
end_recv(wr,ch2) & ps(wr,[[ack],[pvw(t)]]) ==>
   *<>(sts(wr,w1) & ps(wr,[[],[pvw(f)]])).
sts(wr,w1,[pvw(X),sv1(X)]) & ps(wr,[[],[]]) ==>
   *<>(end_sts(wr,w1) & ps(wr,[[],[]])).
end_sts(wr,w1) & ps(wr,[[],[]]) ==>
   *<>(send_msg(wr,ch3) & ps(wr,[[rel(wr)],[]])).
end_send(wr,ch3) & ps(wr,[[],[]]) ==>
   *<>(recv_msg(wr,ch4) & ps(wr,[[],[]])).
end_recv(wr,ch4) & ps(wr,[[ack],[]]) ==>
   *<>(init(wr) & ps(wr,[[],[]])).

init(sch) & ps(sch,[[],[]]) ==>
   *<>(select(sch,sch1) & ps(sch,[[],[]])).
::=(select(sch,sch1) & ps(sch,[[],[]]),
       [ [read(t)], *<>(recv_msg(sch,ch5) & ps(sch,[[req(rd1)],[]])),
```

```
          [read(t)], *<>(recv_msg(sch,ch9) & ps(sch,[[req(rd2)],[]])),
          [write(t)], *<>(recv_msg(sch,ch1) & ps(sch,[[req(wr)],[]])),
          [], *<>(recv_msg(sch,ch7) & ps(sch,[[rel(rd1)],[]])),
          [], *<>(recv_msg(sch,ch11) & ps(sch,[[rel(rd2)],[]])),
          [], *<>(recv_msg(sch,ch3) & ps(sch,[[rel(wr)],[]]))  ]).
end_select(sch,sch1) & ps(sch,[[],[]]) ==>
   *<>(init(sch) & ps(sch,[[],[]])).

end_recv(sch,ch5) & ps(sch,[[req(rd1)],[nr(X)]]) ==>
    *<>(send_msg(sch,ch6) &
        ps(sch,[[ack],[@(X1 is X + 1),nr(X1),write(f)]])).
end_send(sch,ch6) & ps(sch,[[],[]]) ==>
    *<>(end_select(sch,sch1) & ps(sch,[[],[]])).

end_recv(sch,ch9) & ps(sch,[[req(rd2)],[nr(X)]]) ==>
    *<>(send_msg(sch,ch10) &
        ps(sch,[[ack],[@(X1 is X + 1),nr(X1),write(f)]])).
end_send(sch,ch10) & ps(sch,[[],[]]) ==>
    *<>(end_select(sch,sch1) & ps(sch,[[],[]])).

end_recv(sch,ch1) & ps(sch,[[req(wr)],[]]) ==>
    *<>(send_msg(sch,ch2) & ps(sch,[[ack],[read(f),write(f)]])).
end_send(sch,ch2) & ps(sch,[[],[]]) ==>
    *<>(end_select(sch,sch1) & ps(sch,[[],[]])).

end_recv(sch,ch7) & ps(sch,[[rel(rd1)],[nr(1)]]) ==>
    *<>(send_msg(sch,ch8) & ps(sch,[[ack],[nr(0),write(t)]])).
end_send(sch,ch8) & ps(sch,[[],[]]) ==>
    *<>(end_select(sch,sch1) & ps(sch,[[],[]])).
```

```
end_recv(sch,ch7) & ps(sch,[[rel(rd1)],[nr(X),@(X > 0)]]) ==>
    *<>(send_msg(sch,ch8) &
        ps(sch,[[ack],[@(X1 is X - 1),nr(X1)]])).
end_send(sch,ch8) & ps(sch,[[],[]]) ==>
    *<>(end_select(sch,sch1) & ps(sch,[[],[]])).

end_recv(sch,ch11) & ps(sch,[[rel(rd2)],[nr(1)]]) ==>
    *<>(send_msg(sch,ch12) & ps(sch,[[ack],[nr(0),write(t)]])).
end_send(sch,ch12) & ps(sch,[[],[]]) ==>
    *<>(end_select(sch,sch1) & ps(sch,[[],[]])).

end_recv(sch,ch11) & ps(sch,[[rel(rd2)],[nr(X),@(X > 0)]]) ==>
    *<>(send_msg(sch,ch12) &
        ps(sch,[[ack],[@(X1 is X - 1),nr(X1)]])).
end_send(sch,ch12) & ps(sch,[[],[]]) ==>
    *<>(end_select(sch,sch1) & ps(sch,[[],[]])).

end_recv(sch,ch3) & ps(sch,[[rel(wr)],[]]) ==>
    *<>(send_msg(sch,ch4) & ps(sch,[[ack],[read(t),write(t)]])).
end_send(sch,ch4) & ps(sch,[[],[]]) ==>
    *<>(end_select(sch,sch1) & ps(sch,[[],[]])).
```

E.6 Mixer

PAL Process Expressions

Environment Modelling Process

```
env::=
    S_on
    select
        true →
            receive(ch5,dumped);
            STP(batches,batches + 1);
            case batches
                batches > 2: STS(level,high) |
                batches ≤ 2: true
            endcase |
        batches > 0 →
            receive(ch6,use);
            send(ch7,ack);
            STP(batches,batches - 1);
            case batches
                batches = 1: STS(level,low) |
                batches > 1: true
            endcase
    endselect;
    E
```

Buffer Process

```
buf::=
    S_on
    send(ch1,buf(empty));
    receive(ch2,buf(full));
    send(ch3,buf(full));
    receive(ch4,buf(empty));
    E
```

Mixture Consumer Process

```
consumer::=
    S_on
    send(ch6,use);
    receive(ch7,ack);
    use mixture;
    E
```

First Mixer Process

```
mix1::=
    S_on
    repeat
        RDS(level,test)
    until test = low;
    repeat
        make_mixture;
        receive(ch1,buffer(empty));
        dump mixture;
        send(ch2,buffer(full));
        RDS(level,test)
    until test = high;
    E
```

Second Mixer Process

```
mix2::=
    S_on
    receive(ch3,buffer(full));
    make mixture;
    send(ch4,buffer(empty));
    dump mixture;
    send(ch5,dumped);
    E
```

PAL Temporal Representation

```
init_state(1,[<>(init(mix1) & ps(mix1,[[],[level(low)]])),
              <>(init(mix2) & ps(mix2,[[],[]])),
              <>(init(env) & ps(env,[[],[batches(1)]])),
              <>(init(consumer) & ps(consumer,[[],[]])),
              <>(init(buf) & ps(buf,[[],[]]))],
             [level(low)],
             [<>chan(ch1,[],0,1),<>chan(ch2,[],0,1),<>chan(ch3,[],0,1),
              <>chan(ch4,[],0,1),<>chan(ch5,[],0,1),<>chan(ch6,[],0,1),
              <>chan(ch7,[],0,1)
             ]).

init(mix1) & ps(mix1,[[],[]]) ==>
    *<>(rds(mix1,s1) & ps(mix1,[[],[]])).
rds(mix1,s1,level(X),[level(X)]) & ps(mix1,[[],[]]) ==>
    *<>(end_rds(mix1,s1) & ps(mix1,[[],[]])).
end_rds(mix1,s1) & ps(mix1,[[],[level(low)]]) ==>
    *<>(recv_msg(mix1,ch1) & ps(mix1,[[],[]])).
end_rds(mix1,s1) & ps(mix1,[[],[]]) ==>
    *<>(rds(mix1,s1) & ps(mix1,[[],[]])).
end_recv(mix1,ch1) & ps(mix1,[[buf(empty)],[]]) ==>
    *<>(send_msg(mix1,ch2) & ps(mix1,[[buf(full)],[]])).
end_send(mix1,ch2) & ps(mix1,[[],[]]) ==>
    *<>(rds(mix1,s2) & ps(mix1,[[],[]])).
rds(mix1,s2,level(X),[level(X)]) & ps(mix1,[[],[]]) ==>
    *<>(end_rds(mix1,s2) & ps(mix1,[[],[]])).
end_rds(mix1,s2) & ps(mix1,[[],[level(high)]]) ==>
    *<>(init(mix1) & ps(mix1,[[],[]])).
end_rds(mix1,s2) & ps(mix1,[[],[]]) ==>
```

```
    *<>(recv_msg(mix1,ch1) & ps(mix1,[[],[]])).

init(mix2) & ps(mix2,[[],[]]) ==>
    *<>(recv_msg(mix2,ch3) & ps(mix2,[[],[]])).
end_recv(mix2,ch3) & ps(mix2,[[buf(full)],[]]) ==>
    *<>(send_msg(mix2,ch4) & ps(mix2,[[buf(empty)],[]])).
end_send(mix2,ch4) & ps(mix2,[[],[]]) ==>
    *<>(send_msg(mix2,ch5) & ps(mix2,[[dumped],[]])).
end_send(mix2,ch5) & ps(mix2,[[],[]]) ==>
    *<>(init(mix2) & ps(mix2,[[],[]])).

init(env) & ps(env,[[],[]]) ==>
    *<>(select(env,env1) & ps(env,[[],[]])).
::=(select(env,env1) & ps(env,[[],[]]),
        [ [],*<>(recv_msg(env,ch5) & ps(env,[[],[]])),
          [batches(X),@(X>0)],*<>(recv_msg(env,ch6) &
                                  ps(env,[[],[]]))
        ]).
end_select(env,env1) & ps(env,[[],[]]) ==>
    *<>(init(env) & ps(env,[[],[]])).

end_recv(env,ch5) & ps(env,[[dumped],[batches(X),@(X=2)]]) ==>
    *<>(sts(env,s2) & ps(env,[[],[@(X1 is X + 1),batches(X1)]])).
sts(env,s2,[level(high)]) & ps(env,[[],[]]) ==>
    *<>(end_sts(env,s2) & ps(env,[[],[]])).
end_sts(env,s2) & ps(env,[[],[]]) ==>
    *<>(init(env) & ps(env,[[],[]])).
end_recv(env,ch5) & ps(env,[[dumped],[batches(X)]]) ==>
    *<>(init(env) & ps(env,[[],[@(X1 is X + 1),batches(X1)]])).

end_recv(env,ch6) & ps(env,[[use],[batches(X),@(X=2)]]) ==>
```

```
    *<>(sts(env,s3) & ps(env,[[],[@(X1 is X - 1),batches(X1)]])).
sts(env,s3,[level(low)]) & ps(env,[[],[]]) ==>
    *<>(end_sts(env,s3) & ps(env,[[],[]])).
end_sts(env,s3) & ps(env,[[],[]]) ==>
    *<>(send_msg(env,ch7) & ps(env,[[ack],[]])).
end_recv(env,ch6) & ps(env,[[use],[batches(X)]]) ==>
    *<>(send_msg(env,ch7) &
        ps(env,[[ack],[@(X1 is X - 1),batches(X1)]])).
end_send(env,ch7) & ps(env,[[],[]]) ==>
    *<>(init(env) & ps(env,[[],[]])).

init(consumer) & ps(consumer,[[],[]]) ==>
    *<>(send_msg(consumer,ch6) & ps(consumer,[[use],[]])).
end_send(consumer,ch6) & ps(consumer,[[],[]]) ==>
    *<>(recv_msg(consumer,ch7) & ps(consumer,[[],[]])).
end_recv(consumer,ch7) & ps(consumer,[[ack],[]]) ==>
    *<>(init(consumer) & ps(consumer,[[],[]])).

init(buf) & ps(buf,[[],[]]) ==>
    *<>(send_msg(buf,ch1) & ps(buf,[[buf(empty)],[]])).
end_send(buf,ch1) & ps(buf,[[],[]]) ==>
    *<>(recv_msg(buf,ch2) & ps(buf,[[],[]])).
end_recv(buf,ch2) & ps(buf,[[buf(full)],[]]) ==>
    *<>(send_msg(buf,ch3) & ps(buf,[[buf(full)],[]])).
end_send(buf,ch3) & ps(buf,[[],[]]) ==>
    *<>(recv_msg(buf,ch4) & ps(buf,[[],[]])).
end_recv(buf,ch4) & ps(buf,[[buf(empty)],[]]) ==>
    *<>(init(buf) & ps(buf,[[],[]])).
```

Bibliography

[Aba87] M. Abadi. The power of temporal proofs. In *Proceedings of the Symposium on Logic in Computer Science*, pages 123–130, Ithaca, NY, June 1987.

[BAMP81] M. Ben-Ari, Z. Manna, and A. Pnueli. The temporal logic of branching time. In *Eighth Annual ACM Symposium on the Principles of Programming Languages*, pages 164–176, Williamsburg, VA, January 1981.

[BBFM77] J.C. Bossy, P. Brard, P. Faugere, and C. Merlaud. *Le Grafcet: sa pratique et ses applications.* educalivre, Paris, France, 1977.

[BBFM82] H.K Berg, W.E. Boebert, W.R. Franta, and T.G. Moher. *Formal Methods of Program Verification and Specification.* Prentice-Hall Inc., Engelwood Cliffs, NJ, 1982.

[BGG84] P.G. Bosco, G. Giandonato, and E. Giovannetti. Verification of concurrent systems with temporal logics in prolog. *CSELT Technical Reports*, 12(6):557–563, December 1984.

[Bro86] M.C. Browne. An improved algorithm for the automatic verification of finite state systems using temporal logic. In *IEEE Symposium on Logic in Computer Science*, pages 260–266, 1986.

[Buh84] R.J. Buhr. *System Design with Ada.* Prentice-Hall Inc., Engelwood Cliffs, NJ, 1984.

[CBES85] E.M. Clarke, M.C. Browne, E.A. Emerson, and A.P. Sistla. Using temporal logic for automatic verification of finite state systems. In K.R. Apt, editor, *Logics and Models of Concurrent Systems*, NATO ASI Series, Vol. F13, pages 1–2. Springer-Verlag, New York, NY, 1985.

[CES83] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications: A practical approach. In *Tenth Annual ACM Symposium on the Principles of Programming Languages*, pages 117–126, Austin, Texas, January 1983.

[CES86] E.M. Clarke, E.A.Emerson, and A.P. Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[CK83] P. Cousineau and M. Krieger. A distributed query architecture for a hospital information system. In *Proceedings of the Seventeenth Annual Conference on Information Sciences and Systems*, pages 44–49, John Hopkins University, Baltimore, Maryland, March 1983.

[CP88] P. Camurati and P. Prinetto. Formal verification of hardware correctness: Introduction and survey of current research. *IEEE Computer*, 21(7):8–19, July 1988.

[DGU87] J.P. Diaz-Gonzalez and J.E. Urban. Envisager: A visual, object-oriented specification environment for real-time systems. In *Proceedings of the 4th International Workshop on Software Specification and Design*, pages 13–20, Monterey, CA, April 1987.

[Dij75] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8), August 1975.

[EC82] E.A. Emerson and E.M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2(3):241–266, December 1982. Published July, 1983.

[EH83] E.A. Emerson and J.Y. Halpern. 'sometimes' and 'not never' revisited: on branching vs. linear time. In *Proceedings of the Tenth Annual ACM Symposium on the Principles of Programming Languages*, pages 127–140, January 1983.

[Fey65] Robert Feys, editor. *Modal Logics*. Collection de Logique Mathématique. Published with the support of the Fondation Universitaire de Belgique by E. Nauwelaerts, Louvain, France, 1965.

[Flo67] R.W. Floyd. Assigning meanings to programs. *Mathematical Aspects of Computer Science*, 19:19–32, 1967. This paper was not available but the essence of Floyd's work is presented in [BBFM82] and [Gal87].

[Gal87] Antony Galton, editor. *Temporal Logics and Their Applications*. Academic Press, London, 1987.

[GG75] John B. Goodenough and Susan L. Gerhart. Toward a theory of test data selection. *IEEE Transactions on Software Engineering*, SE-1(2), June 1975.

[Hai82] B.T. Hailpern. Verifying concurrent processes using temporal logic. *Lecture Notes in Computer Science*, 129, 1982.

[Hal85] Roger Hale. Modelling a ring network in interval temporal logic. In *Microcomputers, Usage and Design: Eleventh EUROMICRO Symposium on Microprocessing and Microprogramming*, pages 77–84, Brussels, Belgium, September 1985.

[Har89] R.A. Harvey. *PAL Interactive Verification System: Users Guide.* University of Ottawa, Ottawa, Ontario, 1989.

[Hau84] H.-L. Hausen, editor. *Software Validation.* North-Holland, New York, NY, 1984.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, October 1969.

[HW73] C.A.R. Hoare and N. Wirth. An axiomatic definition of the programming language pascal. *Acta Informatica*, 2(4), 1973.

[JK86] R. Joannis and M. Krieger. Process activity diagrams: A tool for the specification and design of multiple microprocessor systems. In *ISMM International Symposium*, Beverly Hills, CA, February 1986.

[Joa86] R. Joannis. Specification and design of multiple microprocessor systems using process activity diagrams. Master's thesis, University of Ottawa, Ottawa, Ontario, 1986.

[JT79] R.W. Jensen and C.C. Tonies. *Software Engineering.* Prentice-Hall Inc., Engelwood, NJ, 1979.

[Kar87] G.M. Karam. *A Tool-Set for Temporal Analysis in a Design Environment for Concurrent Systems.* PhD thesis, Carleton University, Ottawa, Ontario, 1987.

[KHL88] M. Krieger, R.A. Harvey, and J.-P. Lachance. Process activity language 'pal' for planning and coordination of fms. In *Proceedings of the Symposium on Manufacturing Application Languages*, pages 127–135, Winnipeg, Manitoba, June 1988.

[KJH88] M. Krieger, A. Junaid, and R.A. Harvey. Process activity language (pal). unpublished, November 1988.

[Kow74] R. Kowalski. Predicate logic as a programming language. *Proceedings of the IFIP Congress*, 1974.

[Lac89] J.P. Lachance. A simulator to check the effects of various scheduling algorithms on real-time constraints. Master's thesis, University of Ottawa, Ottawa, Ontario, 1989. this work has not yet been submitted for approval.

[Lon79] R.L. London. Program verification. In P. Wegner, editor, *Research Directions in Software Technology*, pages 302–315. MIT Press, London, 1979.

[MP81a] Z. Manna and A. Pnueli. *Verification of Concurrent Programs: Temporal Proof Principles*, volume 131 of *Lecture Notes in Computer Science*, pages 200–252. Springer-Verlag, New York, NY, 1981.

[MP81b] Z. Manna and A. Pnueli. *Verification of Concurrent Programs: the Temporal Framework*, pages 215–273. Academic Press, 1981.

[OL82] S. Owicki and L. Lamport. Proving liveness properties of concurrent programs. *ACM Transactions on Programming, Languages, and Systems*, 4(3):445–495, July 1982.

[Pet77] James L. Peterson. Petri nets. *Computing Surveys*, 9(3):223–252, September 1977.

[PKGP86] Phillippe R. Piche, M. Krieger, C. Gauthier, and E. Petriu. Pad: A new tool for fms design. In *IECON'86 Twelfth Annual IEEE Industrial Electronics Society Conference*, Milwaukee, Wisconsin, September 1986.

[Pnu81] A. Pnueli. The temporal semantics of concurrent programs. *Theoretical Computer Science*, 13:45–60, 1981.

[Pri67] Arthur Prior, editor. *Past, Present and Future.* The Clarendon Press, Oxford, UK, 1967.

[QS83] J.P. Queille and J. Sifakis. Fairness and related properties in transition systems - a temporal logic to deal with fairness. *Acta Informatica*, 19:195–220, 1983.

[RU71] N. Rescher and A. Urquhart, editors. *Temporal Logic*. Springer-Verlag, New York, NY, 1971.

[SZ87] A.P. Sistla and L.D. Zuck. On the eventuality operator in temporal logic. In *Proceedings of the Symposium on Logic in Computer Science*, pages 153–166, Ithaca, NY, June 1987.

[Wol83] Pierre Wolper. Temporal logic can be more expressive. *Information and Control*, 56:72–99, 1983.

[Zav82] Pamela Zave. An operational approach to requirement specification for embedded systems. *IEEE Transactions on Software Engineering*, 8(3):250–269, March 1982. Reprinted in N. Gehani and A.D. McGettrick, editors, *Software Specifications Techniques*, pages 131-170, Addison-Wesley, Workingham, England, 1986.