1

MICRO

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service

Services des thèses canadiennes

Ottawa, Canada
K1A 0N4

CANADIAN THESES

THÈSES CANADIENNES

NOTICE

The quality of this microfiche 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.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

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

AVIS

La qualité de cette microfiche 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.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, examens publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de ce microfilm est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30.

THIS DISSERTATION
HAS BEEN MICROFILMED
EXACTLY AS RECEIVED

LA THÈSE A ÉTÉ
MICROFILMÉE TELLE QUE
NOUS L'AVONS REÇUE

**A Tool-Set for Temporal Analysis in a
Design Environment For Concurrent Systems**

by

Gerald Michael Karam, B.A.Sc., M.Eng.

A thesis submitted to the
Faculty of Graduate Studies and Research
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

Department of Systems and Computer Engineering,
Carleton University,
Ottawa, Ontario, Canada

December 3, 1986

© Gerald Michael Karam 1986

Permission has been granted to the National Library of Canada to microfilm this thesis and to lend or sell copies of the film.

The author (copyright owner) has reserved other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without his/her written permission.

L'autorisation a été accordée à la Bibliothèque nationale du Canada de microfilmer cette thèse et de prêter ou de vendre des exemplaires du film.

L'auteur (titulaire du droit d'auteur) se réserve les autres droits de publication; ni la thèse ni de longs extraits de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation écrite.

ISBN 0-315-35857-2

The undersigned hereby recommend to the Faculty of Graduate Studies and Research acceptance of the thesis

*A Tool-Set for Temporal Analysis in a
Design Environment for Concurrent Systems*

submitted by Gerald Michael Karam, B.A.Sc., M.Eng., in partial fulfilment of the requirements for the degree of Doctor of Philosophy.

A handwritten signature in dark ink, appearing to read "D.H. James", written over a horizontal line.

Chairman,
Department of Systems and
Computer Engineering

A handwritten signature in dark ink, appearing to read "G. E. Butler", written over a horizontal line.

Thesis Supervisor

A handwritten signature in dark ink, appearing to read "Joseph E. Urban", written over a horizontal line.

External Examiner

Department of Systems and Computer Engineering
Faculty of Engineering
Carleton University
December 3, 1986

Abstract —

This thesis describes a tool-set for the temporal analysis of concurrent systems for use within CAEDE, an iconic, Ada-based, concurrent system design environment. As a formal model for our analysis tools, we introduce COL, the CAEDE Operational Language, a temporal logic-based specification language for Ada. The tool-set includes analyzers for deadlock, starvation and critical races, and a COL interactive interpreter. The deadlock analysis tool uses the axiomatic properties of COL's operational semantics to demonstrate that Ada designs specified in COL are system-wide deadlock-free. This analyzer is shown to be decidable for systems with a finite data space, and acceptable in general. We further show that it has an exponential computational complexity relative to the number of tasks in the system; however, we demonstrate that the actual performance can be acceptable. The starvation analyzer is based on a set-theoretic model of starvation that we develop. It uses the deductions produced by the deadlock analyzer to define the possible computation space of the design. A pre-processing phase optimizes the analysis such that, for many cases, it is proportional to the cost of the deadlock analysis. The starvation analyzer is decidable if the deadlock analyzer is decidable. The critical race analysis tool provides semi-automatic detection of critical races by identifying non-deterministic rendezvous (races) from the deductions produced by the deadlock analyzer and then assisting the human operator in identifying which of these constitute critical races. The COL interpreter provides a facility for executing interactively the design specification using the operational interpretation of the COL language. It provides an ongoing trace of the execution and allows the user to control the selection of execution path alternatives. Several design examples are included to demonstrate the capabilities of the tool-set. Finally, we suggest a set of analysis heuristics to assist the user in applying the analysis methods to large or complex designs.

Acknowledgements

I have a suspicion that acknowledgements should be a chapter unto itself — there are simply that many people to thank.

On top of the list is Dr. Ray Buhr, a.k.a. RJAB, or just good ol' Ray, supervisor, advice-giver, cottage owner and friend. If you consider this thesis at all readable then thank him. I can best summarize Ray's assistance in this work by paraphrasing his most frequent comment to me, "Gerald, if you think its obvious, then its not". He was generally right.

Next, a host of faculty members from Carleton Univ. and the Univ. of Ottawa who answered questions, gave advice and provided support. They are: (1) U. of O., C.S. — Dr. Hasan Ural, who examined much of my theoretical work, and Dr. Luigi Logrippo, (2) U. of O., E.E. — Dr. Morris Goldberg, who read a draft of this thesis, (3) C. U., S.C.E. — Dr. Bernie Pagurek, who acted as my thesis advisor in Ray's absence and Drs. Murray Woodside, Roger Kaye and Archie Bowen and (4) C. U., S.C.S. — Dr. Nicola Santoro, who both read a draft of my thesis and answered many, many questions.

Of the outside Drs. I would also like to thank: Dr. Joseph Urban from the Univ. of Southwestern Louisiana, for serving as my external examiner, Dr. Lew Baxter of Logicware, for timely responses to my always difficult Prolog questions and Dr. Dick Peacocke of Bell-Northern Research.

I have tremendously valued my experience at Carleton, much of it due to colleagues such as Carol Hayes, Ron Casselman and Trevor Pearce of the CAEDE Project and Kim Rowe of the ARTT project.

Of the administration, I thank, Naren Mehta, the system manager and general saviour, Dave Sword, Carolyn Morrison and Diane Dodds.

Since I have been at Carleton for some time I have seen a great many grad students come and go. Of these I must single out Mr. Rafik Goubran, who as a co-PhD student, provided a continuous moral support. As for the rest, I will always remember the ways in which they reminded me "not to take it all so seriously".

A few private thanks to my family for providing unswerving support for many years as I pursued my holy grail.

Perhaps the one I should thank the most is Annie, who endured my mental or physical absence many nights, listened to my complaints, triumphs, losses and general rantings, read and learned my thesis. I think she should get a PhD too.

Finally, I thank the money: the gracious support of the NSERC for a substantial scholarship and Carleton University for financial support.

For anyone I've missed, a profound thanks!

I dedicate this thesis to my father, the late Frederick Karam, Mus.D.

Contents

1	Introduction	1
1.1	A Revolution In Software Design	1
1.2	Thesis Objective	5
1.3	A Research Overview	7
1.4	Summary of Contributions	12
1.5	Thesis Outline	13
2	The Design Of Concurrent Systems: A Framework	14
2.1	The Design of Embedded, Concurrent Systems	14
2.1.1	Ada: A Concurrent System Design Model	16
2.1.2	Design Examples Illustrating Issues And Notation	19
2.2	Design Environments	25
2.2.1	CAEDE: A Design Environment for Software CAD	26
2.2.2	Other Design Environments	29
3	Problems and Methods in Temporal Analysis	33
3.1	Undesirable Temporal Behaviour	33
3.1.1	Deadlock	34
3.1.2	Starvation	36
3.1.3	The Critical Race	38
3.2	Methods for Temporal Analysis	40
3.2.1	State-based Analysis Methods	43
3.2.2	Event Sequence-based Analysis Methods	44
3.2.3	Selecting a Formal Model	45
4	COL: An Operational Specification Language	48
4.1	Language Motivation	48
4.1.1	Temporal Logic: An Overview	49
4.1.2	Ada and Temporal Logic	53
4.1.3	A State Interpretation of LTL	59
4.2	Language Development	62

4.2.1	Program State Predicates	66
4.2.2	Describing the Ada Rendezvous	70
4.2.3	Procedure Calls in Ada	74
4.2.4	Specifying Data and Functions	75
4.2.5	Language Statements: LTL Formulæ	79
4.2.6	Basic Operational Semantics	80
4.2.7	Extensions for Non-Deterministic Rendezvous	91
4.3	Practical Language Definition	100
4.3.1	A Prolog-based COL Language Syntax	100
4.3.2	Data and Function Handling	102
4.3.3	A COL Language Example	104
5	A Deadlock Analysis Method	107
5.1	Automatic Theorem Proving: An Example	107
5.2	A Theorem for Absence of Deadlock	110
5.3	A COL-Based Deadlock Prover	112
5.3.1	Proofs of Non-Deterministic Programs	113
5.3.2	Terminating the Inference Engine	121
5.3.3	Summary of Proof Procedure	122
5.4	Analysis Tool Implementation	124
5.4.1	Efficient Recording of PSA	124
5.4.2	General Organization of Tool	128
5.4.3	Examples	129
5.5	Decidability	131
5.6	Computational Complexity	132
6	A Starvation Analysis Method	134
6.1	A Description of Starvation	134
6.1.1	Kwong's Starvation	135
6.1.2	An Ada Interpretation	136
6.2	A Generate and Test Starvation Detector	139
6.2.1	A Set-Theoretic Starvation Test	140
6.2.2	A Repeatable Sequence Generator	147
6.2.3	A Basic Analysis Algorithm	150
6.3	Optimization	151
6.4	Analysis Tool Implementation	155
6.4.1	Input From Deadlock Analysis	155
6.4.2	Pre-Processing	156
6.4.3	Starvation Algorithm	156
6.4.4	Examples	158
6.5	Decidability	159

6.6	Computational Complexity	160
7	A Critical Race Analysis Method	161
7.1	Proof of a Critical Race	161
7.2	The ND-Analyzer	164
8	The COL Interpreter	169
8.1	Functional Description	169
8.1.1	Operational Interpretation of COL	170
8.1.2	COL Interpreter Features	170
8.1.3	COL Interpreter Algorithm	174
8.2	Interpreter Implementation	176
9	Practical Considerations of Analysis	177
9.1	The Limitations of Analysis	177
9.2	Analysis Heuristics	179
10	Conclusions	181
10.1	A Research Summary	181
10.2	Further Research	189
	References	192
A	CAEDE Icon Set	202
B	Temporal Logic Axioms and Rules	206
C	Design Examples	213
C.1	Readers and Writers	214
C.2	Five Dining Philosophers	216
C.3	Gas-Station	220
C.4	Dialogue	221
C.5	Remote Mailbox System	223
C.6	Alternating Bit Protocol	230
C.7	Ten Task	231
D	COL Language and Interpreter Example	234
D.1	Readers and Writers COL Specification	234
D.2	Readers and Writers Interpreter Transcript	238

List of Figures

1.1	Y-Chart Illustrating Components of the CAEDE Design Process . . .	4
2.1	Basic Ada Accept, Task Call Structures	17
2.2	Ada Selective Accept Structure	18
2.3	Structure Diagram for Word Processor Design	20
2.4	Remote Mailbox System: Service Layer Design With Deadlock . . .	22
2.5	The Structural Organization Of CAEDE	27
3.1	Hidden Cycle Deadlock	35
3.2	Readers and Writers Example: One Writer	37
3.3	Agent-Pool Example Illustrating a Critical Race	40
4.1	Rain Example Expressed in LTL	52
4.2	Two Task Design Illustrating Temporal Logic	53
4.3	Ada Code for Two Task Design Example	54
4.4	State Machine Implementation of Two Task Design	60
4.5	Program State Machine Segment for Two Task Design	61
4.6	A Task Deterministic Ada Program	63
4.7	Task Perspective Total Ordering of Rendezvous	64
4.8	Total Orderings of Rendezvous in Program	65
4.9	Ada Code Fragment Illustrating Variable Notation	79
4.10	Ada Language Implications	81
4.11	Connective Implications for Two Task Example	82
4.12	COL Language System	84
4.13	Three Task Design with Non-Deterministic Rendezvous	92
4.14	Timing Diagrams for Three Task Program	93
4.15	Semaphore Access Program	95
6.1	Readers and Writers Example: One Writer	141
6.2	PSA-Tree for Readers and Writers	142
6.3	Program State Assertions (PSA) for Readers and Writers	143
6.4	Assertion Sets M , O , I of PSA 2, 3, 6	144

6.5	Grand Sets M_g , O_g , I_g Of Sequence (2,3,6)	144
6.6	Example σ_r from Readers and Writers	146
6.7	PSA-Tree With σ_e Overlaid	149
6.8	A σ_e Trée	150
6.9	Design Illustrating Deterministic Successor Rendezvous	153
6.10	MProlog Implementation of Starvation Analysis Algorithm	157
7.1	Gas-Station Design Example	165
7.2	Execution Log of Gas-Station Design Analysis	167
A.1	Full CAEDE Icon Set	205
C.1	Structure Graph: Readers and Writers	216
C.2	Structure Graph: Five Dining Philosophers, w/deadlock	217
C.3	Structure Graph: Five Dining Philosophers, w/starvation	218
C.4	Structure Graph: Five Dining Philosophers, corrected	219
C.5	Structure Graph: Gas-Station, w/critical race	220
C.6	Structure Graph: Gas-Station, corrected	221
C.7	Structure Graph: Dialogue Example	222
C.8	Structure Graph: Dialogue Event Decoder Package	224
C.9	Structure Graph: Remote Mailbox System	225
C.10	Structure Graph: Remote Mailbox System, Service Layer	226
C.11	Structure Graph: Remote Mailbox System, Link Layer	228
C.12	Structure Graph: Remote Mailbox System, Driver Layer	229
C.13	Structure Graph: Alternating Bit Design	232
C.14	Structure Graph: Ten Task Design	233

List of Tables

5.1	Summary of Deadlock Analysis Results	130
6.1	Summary of Starvation Analysis Results	159
7.1	Summary of Non-Determinism Analysis Times	168
C.1	Summary of Analysis Results for All Design Examples	215
C.2	Summary of Analysis Results for RMS Design Examples	230

Glossary of Acronyms and Symbols

BTL	— Discrete Branching Time Temporal Logic
CAD	— Computer Aided Design
CAEDE	— Carleton Embedded system Design Environment
COL	— CAEDE Operational Language
FCFS	— First Come, First Served
FSM	— Finite State Machine
LSM	— Logical Structure Manager
LTL	— Discrete Linear Time Temporal Logic
NL	— Natural Language
PM	— Picture Manager
PSA	— Program State Assertion
RMS	— Remote Mailbox System
SDWA	— System Design With Ada
U	— LTL Until Operator
○	— LTL Next Operator
◇	— LTL Eventually Operator
□	— LTL Henceforth Operator
^	— logical conjunction
∨	— logical disjunction
¬	— logical negation
⇒	— logical implication
⇔	— logical equivalence
∃	— existential quantification
⊨	— a valid formula
⊨	— a valid formula within some domain
$t_0 ⊨$	— a valid formula within some domain starting from t_0
$\pi(v)$	— an array of variables, v
ϕ	— inference engine rule and formula selection algorithm

ϕ	— selection algorithm with expansion
$\{\phi\}$	— empty set
ϵ	— set membership
$::=$	— defined to be
O	— option-to-do set
I	— is-done set
M	— must-do set
O_e	— option-to-do set for elementary sequence
I_e	— is-done set for elementary sequence
M_e	— must-do set for elementary sequence
O_g	— grand option-to-do set
I_g	— grand is-done set
M_g	— grand must-do set
A_e	— active task set for elementary sequence
A_g	— grand active task set
R	— set of starving rendezvous
σ	— sequence of successive PSA
σ_e	— elementary sequence of successive PSA
σ_r	— repeatable sequence of successive PSA
j_p	— PSA-tree join point
l_p	— PSA-tree looping point

Chapter 1

Introduction

1.1 A Revolution In Software Design

The variety of software design techniques and tools has expanded by leaps and bounds since the development of the first compilers. Unfortunately these efforts often pale in comparison to what has been achieved for integrated circuit (IC) design, a field with a similar length of history. The cornerstone of IC circuit design is Computer Aided Design, or CAD. The IC CAD process allows engineers to develop and test new designs of moderate complexity in relatively short periods of time [105]. The major components of IC CAD, work stations and software tools, appear as commercial products that now pervade the hardware industry. Furthermore, the extension of CAD to accommodate the needs of Very Large Scale Integration (VLSI) circuit design looks positive (although somewhat challenging) [46].

Software design techniques and tools do not exist in the same sense as IC CAD. The stress for software tool development has been directed primarily towards programming (editing), debugging and source code maintenance products. Complete environments for creating and testing software designs are still only research projects [6,36,49,78,93,102,103,116].

In examining the relationship between hardware and software design, Smith and Gross [99] cite several instances where the common practices of IC CAD

should be exploited for software design:

- behavioural, structural and physical design should be present at all levels of design abstraction;
- techniques should be used to demonstrate correctness prior to implementation;

~~•~~ design should be graphics-driven through the use of colours and shapes for representation.

Following the lead of IC CAD, we believe that it is time for a revolution: *Software CAD*, [21].

The start of the revolution is a hardware-like design paradigm in which software designs are specified iconically. Indeed, we believe the start of the revolution is already behind us. One example is the creation of an icon set based on *Ada*¹ [104] presented in Buhr's *System Design With Ada* [17]. In this approach, *Ada* was chosen as the design model because:

- it is an important new language with a need for design support;
- it captures the notion of interconnected *black boxes* that is typical of CAD design;
- it provides a coherent, standardized and widely disseminated notation for specifying and implementing sophisticated, concurrent systems.

The next stage of the revolution is the development of a computer-based design environment founded on the iconic design paradigm. Specifically, this environment should:

- support entry and manipulation of the iconic designs;
- produce software automatically from the design;

¹Ada is a registered trademark of the U.S. Government (Ada Joint Program Office)

- assist the designer in evaluating and experimenting with the consequences of design decisions, prior to implementation.

This thesis is concerned with the development of design evaluation techniques for an experimental version of such an environment: *CAEDE*², the *CARleton Embedded system Design Environment* [18,20,19,16,21]. Explicit in the genesis of the name CAEDE is its primary application domain: concurrent software that is typical of embedded systems. The rigidly constrained interfaces and subsystems of embedded systems make them natural candidates for the software CAD revolution.

The underlying software design philosophy of CAEDE is the *CAEDE Design Process*, illustrated using the "Y-chart" in Figure 1.1. The x-y-z axes of the chart (in thick lines) identify our perception of the primary aspects of design: *structure*, *temporal behaviour* and *functional behaviour*. We define briefly these terms below.

Structure — the identification of black boxes, interfaces and interconnections at all levels of nesting.

Temporal Behaviour — the temporal sequencing rules, variables and functions affecting the patterns of interactions among the black boxes, along the interconnection paths; for example, variables used in Ada guard conditions are elements of temporal behaviour.

Functional Behaviour — the details of the functional behaviour of the design components, variables and function that do not have a direct influence on the patterns of interaction; for instance, suppose the result of a numerical integration function is transmitted from one task to another, the temporal action, the task call, takes place regardless of the outcome of the integration.

We believe that the design process naturally starts with structure, proceeds to temporal behaviour and then concludes with functional behaviour. Furthermore, considerable initial effort is attributed to developing a good design from the structural and temporal viewpoint. Corroboration for these notions is found in design

²pronounced *kay-da*

Figure 1.1: Y-Chart Illustrating Components of the CAEDE Design Process

experiments involving practicing engineers [1]. The findings of these experiments revealed that engineers build abstract mental models of the system under design, simulate them and then add lower level details once they were satisfied that the designs behaved correctly. Thus, the first priority of CAEDE's development is to support creation and evaluation of structure and temporal behaviour without committing to detailed functional behaviour.

The outputs of the CAEDE Design Process, as indicated by the thin lines in Figure 1.1 are executable design specifications, target software and target hardware. The latter output is a distant objective of the CAEDE project relating to the creation of hardware implementations of software designs [42] (a broad description of this research topic is given in [50]). The generation of Prolog-based [28,55] executable design specifications [19] and Ada code skeleton generation [18], address the first two manner of output. The Prolog-based executable design specifications are available during any stage of design; they form the basis for design experimentation tools.

1.2 Thesis Objective

Design evaluation techniques are of considerable benefit for the design of embedded, concurrent systems because of the interaction complexity between concurrent entities. They are particularly useful for studying the temporal behaviour of the system under design. Undesirable or fatal temporal behaviours include deadlock, starvation and critical races [17,29,33,32,51,56,57,62,64,71]. The combination of a system's structure, temporal behaviour and functional behaviour could lead to one or more of these unwanted situations. We believe that early decisions on the structure of a design often dictate the temporal patterns for execution. Thus, the quality of the design should be explored at significant steps in the design process as this may prevent a massive re-design effort at a later stage due to incorrect temporal behaviour [3].

The objective of this thesis is to develop an operational specification language for Ada and a comprehensive design evaluation tool-set that together, can be used to identify undesirable temporal behaviour in Ada-oriented concurrent system designs. The path to the thesis objective is influenced by the factors outlined below.

- (1) — *The tools should be readily comprehensible to a designer with a practical orientation, but little facility for advanced methods in computer science.*

All too often, analysis methods are proposed that correctly represent and analyze a design, but in a manner that is well beyond the training of the typical designer. Avrunin et al. [4] concur with this opinion when they state that, if a technique is to be of any practical value, "it must be amenable to use by software developers who have little or no training in advanced mathematics or theoretical computer science". Riddle's notion of the "sophisticated user" [91] also alludes to this point.

- (2) — *The design should represent an operational rather than a declarative (nonconstructive) specification of the system.*

An operational specification defines *how* a problem should be solved, whereas a declarative specification states *what* properties a design should have [116]. The

latter approach is more desirable in principle because it represents the design at a higher level; however, it is harder to write and implement. The operational specification is necessarily at a lower level of detail, but considerably easier to manipulate. Also, transformational techniques to convert automatically from declarative to operational specifications have yet to be effective for sequential, let alone concurrent, systems [6].

(3) — *The specification and evaluation methods should be based on formal techniques as much as possible.*

Formal methods allow us to reason on the design with complete confidence in the results [3,4]; however, we stress that this goal must be balanced against the desire to have the analysis process amenable to the typical designer (*factor 1*, above).

(4) — *Design analysis should be expressed in terms of CAEDE's Ada-oriented design elements.*

Since the designer enters and manipulates the design using the Ada-oriented icons, it is only natural that design evaluation techniques adhere to the designer's view of the system. This factor also encourages the exploitation of the limitations imposed by the Ada concurrency model (which is consistent with the guidelines for tool development suggested by Razouk and Hirschberg [87]).

We envision that our specification language and temporal analysis tool-set will integrate fully CAEDE's iconic principles. In particular, the operational specification language would be entered iconically (with perhaps some text annotation) and the control and output of the analysis tools would be iconic. Research into iconic specification based on state machine notations has been addressed [58,80,106]. The iconic representation of analysis, in most cases program animation, has been investigated primarily for sequential systems [13,58,89,106] and for very simple concurrent systems [69]. In this research, we limit ourselves to primarily text³ input and output; although iconic input and output is an important research area, it is beyond the scope of our current objectives.

³CAEDE's structure icons are used to generate some preliminary text statements.

1.3 A Research Overview

In response to the thesis objective, we overview the key developments arising from our research:

- an operational specification language (dubbed COL⁴) that can express the operational specification of Ada designs derived from CAEDE icons and uses *Temporal Logic* as a formal model;
- a temporal logic-based deadlock analysis tool;
- an efficient starvation analysis tool based on a *set-theoretic* starvation test, a tree of temporal logic assertions describing program behaviour and a *generate and test* analysis algorithm;
- a critical race analysis assistant that uses a tree of temporal logic assertions describing program behaviour to identify *race conditions* and a user-aid to locate critical races;
- an interpreter for the operational specification language.

We outline each of these research areas below.

CAEDE Operational Language (COL)

The COL operational specification language provides us with a strong base upon which we can construct tools for temporal analysis because it supports both operational and axiomatic reasoning. The design of the language has been shaped by the four factors of influence stated in the previous section:

- the language is based on first order, linear temporal logic, which as an extension of classic logic [90] has an obvious natural language interpretation (*factor 1*);
- the language can express operational specifications of Ada designs (*factors 2,4*);

⁴COL, pronounced *coal*, an acronym for CAEDE Operational Language

- temporal logic is a demonstrated formalism for concurrent system analysis [11,47,73,79,96,97] (*factor 9*).

The COL language expresses Ada programs by: (1) temporal logic implication formulæ that describe the behaviour of tasks and procedures, (2) temporal logic implication formulæ that describe the Ada concurrency model and (3) an assertion specifying the initial state of the program. The implication formulæ define allowable sequences of program states by stating legal transformations of the program state from the perspective of a single task; for example, if task A executes the procedure call `Proc1(...)`; followed by a procedure call `Proc2(...)`; we would state this as:

"All allowable sequences of program states such that if it is true in some state that task A has just returned from `Proc1(...)`; then it will be true in a future program state that task A is calling `Proc2(...)`."

The initial program state assertion (initial PSA) describes the state of the program relative to some arbitrary temporal origin, t_0 . The formulæ describing the Ada concurrency model describe the mechanics of the Ada rendezvous and procedure calls [104].

The operational semantics of COL are defined by a *bottom-up* [60,61] inference engine. The inference engine applies the implication formulæ describing the program to a *current PSA* in order to deduce a new PSA; the first current PSA is the initial PSA. The inference engine uses a set of temporal logic inference rules to perform deductions. Like Prolog, the execution of the inference engine can be interpreted axiomatically (declaratively) or operationally (procedurally).

The basic description of the COL inference engine is open-ended; that is, certain elements of its behaviour are left undefined. The description of the inference engine is completed when it is used for a specific application (such as the deadlock analyzer tool or COL Interpreter).

The definition of the COL language draws on the efforts of Pnueli and DeRoeve [85] who describe temporal logic semantics for a subset of the Ada language and suggest ad hoc proof methods for reasoning liveness properties of Ada programs.

It differs from other operational specification languages, such as PAISLey [116] and Gist [6], in that it used Ada as the concurrency model and addresses specification at the design rather than functional level. It also approaches the notion of Ada program specification differently than ANNA [70], which provides declarative specifications of Ada program data structures and code through the annotation (by encoded comments) of an actual Ada program.

Deadlock Analysis Tool

The deadlock analysis technique uses temporal logic as the theoretical basis for automatically proving the absence of *system-wide deadlock*⁵ in Ada concurrent system designs. In this method, the design under review is expressed in COL and then analyzed automatically using a special purpose implementation of the COL inference engine. The inference engine generates a tree of deduced program state assertions (PSA-tree) that describe all possible behaviours of the design under analysis. If there exists any PSA from which no further deductions are possible, then a system-wide deadlock is proven.

The method is *decidable* [67] for designs in which the data space is finite and it is *acceptable* in general. When the algorithm is decidable, it has a computational complexity that is exponential with the number of tasks in the program. The tool's performance is comparable to that of *reachability analysis* for an equivalent design of communicating finite state machines [10].

Starvation Analysis Tool

We develop an efficient method for identifying starvation in Ada designs using: (1) a set-theoretic starvation test based on the finite delay property defined by Kwong [62], (2) an analysis algorithm that follows the generate and test paradigm [7,52,109] and (3) several optimizations reasoned from the tree of program state assertions (PSA-tree) that is generated by the deadlock analyzer. The deadlock analysis is assumed to have shown successfully absence of system-wide deadlock.

⁵A *system-wide deadlock* is one in which all tasks cannot progress; if some tasks can continue execution while others are permanently blocked, then there is *local deadlock*.

Kwong's finite delay property allows us to decide if a computation, (a sequence of successive program states) is *fair*; some notions of fairness in Ada programs are discussed in [85]. We interpret Kwong's work from an Ada perspective and then define a new set-theoretic starvation test for Ada programs that uses the program state assertions generated by the deadlock analysis as an alternate description of computations.

Our starvation analysis method is an example of the generate and test analysis paradigm; that is, we generate a repeating sequence of successive program state assertions, σ , and then use our set-theoretic test to determine if there exists some task that is blocked waiting for a *rendezvous* [104] and if an infinite repetition of this sequence can represent a fair computation. A depth-first traversal [109] of the PSA-tree is used to generate the σ , to be tested. Any σ , in which all tasks are active or that represent unfair computations are ineligible. Optimizations obtained by pre-processing the program state assertions allow us to prune many ineligible σ . The method is shown to be decidable. The cost of the pre-processing is linearly proportional to the cost of the deadlock analysis, but the cost of the analysis algorithm is exponential with the cost of the deadlock analysis; however, because of the optimizations, there exist designs (such as an implementation of the Five Dining Philosophers [57]) for which the cost of the analysis algorithm is negligible.

Critical Race Analysis Tool

Critical races are very elusive errors in Ada programs. They occur for fairly simple reasons: a mutual exclusion constraint for some resource has been violated, or, the program has been designed using some invalid assumption of rendezvous behaviour. However, they are chameleon-like in their appearance: as temporal errors, such as deadlock and starvation, or as incorrect computations. Critical races that lead to deadlock or starvation are detected by the previously described analysis techniques; that is, we detect the effect rather than the cause. However, the detection of incorrect computations is far more challenging since it is difficult to state the problem in abstract terms (unlike deadlock and starvation).

If we presume that errors due to violation of mutual exclusion do not exist (since ensuring mutual exclusion in Ada is relatively easy), then the cause of a critical race must be an invalid assumption of rendezvous behaviour; in particular, the design assumes wrongly that the rendezvous will occur in some deterministic way. We can locate this cause by: (1) identifying a non-deterministic rendezvous (a race) and (2) proving that the design has assumed that it was deterministic. Unfortunately there is no general algorithm for performing this task.

In this research, we develop a compromise solution: an analysis assistant that automatically detects non-deterministic rendezvous and then aids the user in proving that it invalidates some design assumption. The set of all non-deterministic rendezvous is derived from the tree of program state assertions produced by the deadlock analysis tool. This approach is effective for designs with a complexity in the order of 10 tasks. Karam and Buhr [59] suggest a non-deterministic rendezvous detector using design heuristics [17] embedded in a knowledge-based system [7,52,109]. This method is not appropriate for designs that are reasonably analyzed with the deadlock tool; however, for very large designs (perhaps 50-100 tasks) it would become practical.

COL Interpreter

The COL interpreter we develop provides the designer with the most flexible temporal analysis tool: the ability to interactively "execute" the design. Using the interpreter, the designer explores the design in order to gain confidence in the correctness of its behaviour (beyond anything demonstrated by the three previous tools). This type of analysis is strongly advocated throughout the literature [5,6,14,22,31,58,87,106,114,116]. The interpreter is a special purpose implementation of the COL inference engine. It features: user-controlled inference progress and direction, and an english-like display of the behaviour of the specified program.

1.4 Summary of Contributions

The primary research contribution of this thesis is the development of a comprehensive temporal analysis system for Ada designs; however, within this broad accomplishment there are several specific achievements.

- The development of a new operational specification language.
- The use of temporal logic as the basis for an operational specification language.
- A new automatic, temporal logic-based deadlock analysis technique using an operational specification as the design description.
- A new, set-theoretic description of starvation in Ada designs.
- A new, efficient method for automatically detecting starvation in Ada designs.
- A characterization of critical races in Ada programs.
- A new, semi-automated critical race analysis tool for Ada designs.
- The development of an interpreter for the temporal logic-based operational specification language.

In addition to this list of primary research, several other contributions have been made on the CAEDE project itself. In particular we identify: (1) co-design of the CAEDE architecture, (2) preliminary design of CAEDE's graphical interface, (3) design of the Prolog executable specifications, (4) design and implementation of CAEDE's logical structure management subsystem, (5) enhancement of CAEDE's icon set, (6) porting the Ada code generator to MProlog and (7) enhancing the performance and capabilities of the Ada code generator. These efforts shaped the temporal analysis system described in this thesis.

1.5 Thesis Outline

In the following chapter, we describe a framework for the design of embedded, concurrent systems. This framework identifies aspects of the design process and design environments that are relevant to this thesis. In particular, we describe CAEDE's architecture and its impact on the organization of the tool-set.

Chapter three presents a detailed characterization of temporal behaviour problems in concurrent systems: deadlock, starvation and critical races. An overview of the current analysis techniques is given.

In chapter four, we present the development of COL, the operational specification language for Ada. Following, in chapters five through eight, we describe the four temporal analysis techniques outlined earlier: analyzers for deadlock, starvation and critical races, and the COL interactive interpreter.

Given some understanding of the constraints imposed by the analysis tools, we provide a discussion on practical analysis considerations in chapter nine. Finally, chapter ten presents conclusions and suggested areas of further research.

Four appendices contain a summary of the CAEDE icon set, temporal logic axioms, the suite of design examples and a COL language example demonstrating the use of the COL Interpreter.

Chapter 2

The Design Of Concurrent Systems: A Framework

In this chapter, we present a framework for concurrent ~~system~~ design. Our framework consists of two sections: a discussion of the design issues, models and notation relevant to this thesis and an overview of design environments, with emphasis on CAEDE, the target environment for our analysis tool-set.

2.1 The Design of Embedded, Concurrent Systems

Embedded, concurrent systems are the target application for both the CAEDE design environment and our temporal analysis tool-set. We characterize an embedded, concurrent system as follows:

1. it implements a single or special-purpose application (it is not a general purpose computer);
2. there is real or apparent concurrency;
3. it exhibits a primarily reactive behaviour to external events;
4. it usually consists of rigidly constrained sub-systems and interfaces.

Examples of such systems are: point-of-sale terminals, stand-alone word processors, communications switching equipment, industrial process control systems, avionics systems and weapons systems. Many of these applications are real-time, non-terminating and require high levels of reliability.

Given a particular concurrency model (such as Ada or Modula-2), the designer must wrestle with (at least) two classes of design issues that have resulted from the presence of concurrency: the implementation of inter-task protocols and undesirable inter-task temporal behaviour.

The implementation of inter-task protocols is concerned with the approach by which application specific information is communicated to the required tasks. Some specific examples are described below.

Waiting for service — in what way do we express the idea that some task is waiting for another task to provide it with some services?

Server replies — in what way does a task performing some service indicate to another task that it has completed a service, or, that it is available for service?

Flow control — how can flow of control be exercised by one task upon another?

For example, if one task does not have an empty message buffer, how can it inform the producer of messages that it can no longer accept messages.

Time deadlines — how can one task inform another task that some timing deadline has passed?

The manner in which inter-task protocol issues are handled depends on the chosen concurrency model; for instance, Burr addresses many of these concerns for Ada-oriented designs in [17].

Undesirable inter-task temporal behaviour covers problems that arise in a design during execution; these include: deadlock, starvation, critical races and unnecessary waiting. The temporal analysis tool-set described in this thesis addresses the first three issues directly; they are discussed further in chapter 3. Unnecessary waiting is a situation in which some (usually time critical) task is forced to

wait for service from another task because of contention for the service task's resources. Unlike deadlock, starvation and critical races, unnecessary waiting may not produce incorrect system behaviour, but only poor performance.

In the remainder of this section, we present the Ada language as our concurrent system design model and introduce CAEDE's iconic representation of Ada through two design examples.

2.1.1 Ada: A Concurrent System Design Model

In this section, we describe aspects of the Ada programming language [104] from the perspective of its use as a design model for CAEDE and this thesis. Initially, we limit our concern to Ada's *task* and *rendezvous* features.

The task represents a concurrent entity in the Ada language. It consists of: (1) a specification part, which defines task entries (communication ports) and (2) a body part, which contains the program code that implements the task. The Ada language definition does not specify the run-time character of tasks, e.g. the relative speed, implementation, scheduling, etc. This is convenient from a programming standpoint since it permits many interpretations of the language, however, as we shall later see, it is an obstacle for formal analysis (cref. chapter 6).

Tasks communicate with each other by the rendezvous mechanism. Semantically, the rendezvous is equivalent to a message-based *send-receive-reply* structure [23]; i.e. a source task sends a message to a target task and then remains suspended until it gets a reply message. The appearance of a rendezvous request (send) in Ada code is a procedure call to an entry of a task. If the task call appears in a program section outside of the task, for example, within a passive procedure, the source task is the task that was active the code when the call occurred. A rendezvous acceptance (receive) by a target task is denoted by an accept statement qualified by a task entry name and an argument list. The accept statement is eventually followed with an end statement (reply). Data is passed between communicating tasks through the argument list of the accept statement. Each task entry is assigned a queue in which source tasks are queued (when calling). Calling

```

...
accept Entry(A : in INTEGER) do
...
end;
...
Test.Entry2(A);
...

```

Figure 2.1: Basic Ada Accept, Task Call Structures

tasks are removed (accepted) from the queue using a First Come First Served (FCFS) discipline. The text appearance of the Ada accept structure and and Ada task call are illustrated in the program code fragment of Figure 2.1. In this code fragment, (representing part of the body of some task), the task entry Entry is accepted and then a call on the entry Entry2 of task Test is called.

The detailed semantics of the rendezvous are described below (adopting the convention of Buhr [17], the source and target tasks in the rendezvous shall be referred to as the caller and acceptor tasks).

- If a caller executes a call on a task entry and the acceptor task is not executing the accept statement for that entry, then the caller joins the entry queue and is suspended.
- If an acceptor task executes an accept statement of an entry for which there are no callers, then the acceptor is suspended.
- If an acceptor task executes an accept statement of an entry for which there is a caller waiting, then the caller is removed from the entry queue and the rendezvous begins; the caller remains suspended.
- If a caller executes a call on a task entry and the acceptor has executed the accept statement for that entry and is currently suspended, then the caller is suspended, the acceptor is resumed and the rendezvous begins.
- If the acceptor executes the end statement for the accept structure, then the rendezvous is completed, the caller task resumes execution and the acceptor task continues execution.

```

select
    accept Entry1(....) do
--      do entry stuff
--      critical section
--      rendezvous in progress
    end;
--    can do more non-entry stuff
--    non-critical section
--    rendezvous is terminated
or
--  guard condition
--  when Flag =>
    accept Entry2(....) do
--      do entry stuff
--      critical section
--      rendezvous in progress
    end;
--    can do more non-entry stuff
--    non-critical section
--    rendezvous is terminated
or
--  and so on...
end select;

```

Figure 2.2: Ada Selective Accept Structure

The Ada language provides a wide range of variations on this basic theme. These include: selective accepts, timed calls, timed accepts and guard conditions. Of these extensions, we shall use the selective accept structure, with guard conditions, in many of our design examples. We study this structure in detail below.

The selective accept allows the acceptor task to wait for callers on a disjunction of task entries. Each entry may be qualified by a *guard*. A guard is a boolean condition that determines whether an entry is to be eligible for receiving callers. The guards are evaluated when the `select` statement is executed. An entry whose guard has been computed to be *true* is eligible for callers; it is said to be an *open entry*. If the guard is *false*, then the entry cannot accept callers; it is a *closed entry*. All entries without guards are automatically open entries. The text appearance of the selective accept construct is shown in Figure 2.2.

The execution semantics of the selective accept are summarized below; we assume that the `select` statement has been executed and the guard conditions evaluated.

- If there are no callers for all open entries, then the acceptor task is suspended until a caller arrives; it will accept the first entry for which a caller appears.
- If there are waiting callers on one open entry only, then the acceptor task will choose this entry.
- If there are waiting callers on several open entries, then one of the entries is selected using some algorithm not specified by the language standard [104].

When a selective accept is entered only one of the disjunction of accept statements may be eventually executed. The code that appears after an entry's termination statement (`end;`), is executed only if that entry was accepted; however, once the entry termination statement is executed, the rendezvous is ended and the caller can resume execution. We distinguish between these two stages in a selectively accepted entry as: the *critical section* (the entry during rendezvous) and the *non-critical section* (the entry after rendezvous but before the next or statement or the `end select;` statement). These stages are indicated in the example of Figure 2.2.

2.1.2 Design Examples Illustrating Issues And Notation

As an example of a relatively simple embedded, concurrent system let us consider a fictitious word processor system with one disk drive, a printer and buffered keyboard input. The design structure diagram for this application is illustrated in Figure 2.3. The graphical notation is from Buhr's *System Design with Ada* [17]. An explanation of the symbols is provided in Appendix A; however, for the benefit of this discussion, we describe the symbols appearing in Figure 2.3:

- parallelograms represent Ada tasks;
- small parallelograms situated at the edges of tasks correspond to entries;
- arrows connecting tasks and task entries represent task calls;

Figure 2.3: Structure Diagram for Word Processor Design

- isolated, labeled arrows represent data flows that occur during task calls.

The word processor design ¹ has four tasks, one for each of the potentially concurrent activities: keyboard type-ahead, printing, disk I/O and console operation. Under normal circumstances, these tasks will cycle endlessly through their duties. Furthermore, for the remainder of this document, we consider all tasks to execute in endless cycles unless stated otherwise. The four tasks function as follows:

Key_bd — The keyboard task buffers keystrokes from the keyboard of the word processor. Once a buffer is full, it is passed to the Console task for processing via the `Console.Request` entry.

Console — The console task manages the word processor screen and most of the word processing functions. It executes print and disk I/O requests by calling the `Invoke` entries of the **Printer** and **Disk_mgr** tasks, respectively. It receives data and status information from the other three tasks through its `Request` entry. The `Type` argument of the `Request` entry enables the

¹The actual hardware access components of this design have not been shown in order to simplify the example.

console task to identify the calling task. The usual execution cycle for the console task is: accept `Request`, process the request and, if necessary, call `Printer` or `Disk_mgr`.

`Disk_mgr` — The disk manager performs disk read and write operations for the console. It accepts `Console` requests through its `Invoke` entry and returns status and possibly data to the `Console` by calling the `Console.Request` entry.

`Printer` — The printer manager performs printing for the console. It accepts `Console` requests through its `Invoke` entry and returns status and the `Console` by calling the `Console.Request` entry.

This relatively simple example illustrates the nature of embedded, concurrent systems — self-containment, concurrency and endless cycling. However, there are also several interesting design issues that can be revealed.

- Does the console task have too much central control? All peripheral activities must go through a central point for almost all operations.
- Is there any deadlock in this design since there are mutual calls between tasks? This is not likely under normal conditions, however, should the console become deadlocked because of printer or disk drive failure then the keyboard becomes frozen and no error messages can be displayed.

Many of these design issues are dealt with in [17]. One can imagine other more complex designs in which, problems such as deadlock, starvation and critical races could appear in subtle ways. For example, let us consider the critical race/deadlock example shown in Figure 2.4.

The design illustrated in Figure 2.4 is the service layer package from the *Remote Mailbox System* (RMS) design found in Appendix C.5. This particular design illustrates both a critical race and a deadlock; i.e. the critical race manifests its presence as a deadlock. Figure 2.4 introduces further graphic notation which we describe below.

Figure 2.4: Remote Mailbox System: Service Layer Design With Deadlock

- A group of task entries (such as `Rec_Msg_Tx` and `Ack_Tx` in task `Ack_Status`) connected by an intersecting line define a selective accept.
- A rectangular box represents an Ada package and smaller rectangular boxes at the edge of a package are exported procedures defined within the package; i.e. *visible procedures*.
- Octogonal icons represent Ada variables or *data areas*.

The service layer package supports the following functions: it transmits mail messages provided to its `S_RM` procedure to the service layers of other RMS facilities, provides end-to-end acknowledgements of mail transmission and receives mail messages from the service layers of other RMS facilities. All communication with other RMS facilities takes place through an underlying link layer package (not shown) which provides guaranteed message transmission (messages are sent to the link layer through `S_Link` and received from the link layer by the procedure `L_Serv`). There are implicit user tasks and mailbox tasks that provide and accept mail messages, respectively. We describe the behaviour of the design elements below.

`S_RM` — *Send Remote Mailbox* — This procedure implements the send mail function on behalf of the calling user task. It performs the following steps: get the routing address from the `Route_tbl`, format a service layer message, send the `Serv_Msg` to the link layer (through `S_Link`) and register an outstanding mail message acknowledgement with the `Ack_Status` task through the `Rec_Msg_Tx` entry.

`L_Serv` — *Link to Service* — This procedure implements the receive mail and acknowledgement function on behalf of the link layer. When a message arrives it is routed to the mailbox task (through `Put`) if it is a mail message and to the `Ack_Status` task through the `Ack_Tx` entry if it is mail message acknowledgement.

`Ack_Status` — *Acknowledge Status Task* — This task manages the outstanding acknowledgements for outgoing mail messages. Each message is registered with the task by a call on `Rec_Msg_Tx`; when the acknowledgement arrives it

is matched against a registered message and then an acknowledgement mail message is sent to the mailbox task (through Put). If an acknowledge arrives for which there is no registered message then it is ignored.

This design can function correctly under some conditions and realize a deadlock under others; this is because of the critical race. We illustrate the critical race leading to deadlock by the following set of execution steps (we assume that there are no outstanding outgoing messages).

- A user task calls S_R_M which invokes the following two steps: a reference the routing table and a completed call to the link layer; the call to Rec_Msg_Tx is not performed as yet.
- The link layer transmits the message and eventually returns an acknowledgement message by a call on L_Serv which generates a task call on the Ack_Tx entry of the Ack_Status task. This message is rejected because there is no registered outgoing message.
- The user task makes a call (again through S_R_M) on Ack_Status to register the outgoing message.
- The user task does not send any more mail messages until it receives an acknowledgement; thus, it will remain deadlocked since the acknowledgement had already arrived and was discarded.

The critical race occurs between the user task and the link layer to call Ack_Status task; if the link layer arrives ahead of the user task then a deadlock results, otherwise the system operates normally.

Like the word processor design, this relatively simple design can be studied manually; however, when placed in the context of the complete design, the behaviour we have just described may go unnoticed until the system is implemented and in operation.

As we have seen in this section, one of the primary concerns of embedded, concurrent system design is the temporal behaviour. This is shown by the need to:

1. guard against problem-specific temporal attributes (such as the keyboard delays due to congestion at the CONSOLE task in Figure 2.3);
2. locate generally unacceptable temporal behaviour (such as the deadlock in Figure 2.4).

It is the objective of this thesis to develop a tool-set that addresses the latter need

2.2 Design Environments

Design environments are computer-based systems that support the designer in the task of transforming the functional specifications of a system into a software design. An effective design environment will provide the user with an integrated set of tools that permit specification, review, modification, presentation and management of a design. Structurally, the typical design environment will have the following components: (1) a design entry system, (2) a design data base, (3) design analysis tools and (4) design transformation tools [19]. The basic objectives of these components are outlined below.

Design entry system — a facility for specifying and editing the system under design.

Design data base — repository of information about the current state of the design; this includes: the actual design, design decisions, analysis data and management information.

Design analysis tools — provide quantitative and qualitative review of the design: examples include: verification systems and tools to study design complexity, reliability, performance and temporal behaviour.

Design transformation tools — provide a means of generating the different representations of the design from its internal representation; for instance, creating implementation language code skeletons and producing structure graphs.

Given this model of a design environment, we describe next the CAEDE design environment and the relationship of other design environments to CAEDE.

2.2.1 CAEDE: A Design Environment for Software CAD

The analysis tools that are described in later chapters are to be an integral part of a design environment for concurrent systems, CAEDE [16,18,19,20,21]; thus, the characteristics of the design environment necessarily impact the nature of the analysis tools and their underlying models. In this section, we describe the CAEDE environment and, in particular, examine those features that are relevant to our work.

CAEDE is an experimental, iconic, design environment for embedded, concurrent systems. Its aim [19,21] is to provide the basis for a design environment that fills three specific roles:

1. The *Expert's Assistant* — in this role CAEDE will aid the system designer in building, editing, analyzing and transforming designs.
2. The *Tutor* — as a tutor CAEDE assists in teaching system design, thus enlarging the pool of expert designers.
3. The *Design Expert* — CAEDE will eventually, through intelligent design entry tools and analysis tools, directly incorporate design expertise.

For this research, we are concerned with the role of CAEDE as an expert's assistant. Given our target use for CAEDE, we list its key features below.

- CAEDE implements the *System Design with Ada* (SDWA) design methodology and graphical notation [17]. Following the SDWA approach, CAEDE uses the Ada language as the underlying design model.
- CAEDE provides the basic functionality of a design environment: an iconic entry system, a design storage system, analysis tools and design transformation tools.

The structural organization of CAEDE is depicted in Figure 2.5. Through the use of this diagram, we can study the mechanism for integrating analysis tools.

CAEDE is composed of six subsystems and five data stores. A short description of the most relevant components appears below.

Figure 2.5: The Structural Organization Of CAEDE

Graphics Interface — The graphics interface interacts with the user to manipulate icons, invoke commands and access tools. The graphics interface follows the *verb-object* interaction paradigm; that is, the user selects an action to be performed and then chooses the icon on which to apply the action. Action selection is menu-driven.

Logical Structure Manager (LSM) — The logical structure manager performs manipulation and run-time storage of the logical construction of the design; for example, "task entry E1 belongs to task T1" is logical information. Permanent storage of the logical structure is in two forms: a logic representation (logic store) and a binary representation (system store). The binary representation is used because of run-time performance considerations. The logic representation is used for the analysis and transformation processes.

Picture Manager (PM) — The picture manager is responsible for representation and manipulation of the physical picture. For instance, "task entry E1 is at the left-hand edge of task T1" is physical picture data. There is no logical association between E1 and T1 from this statement. In fact, E1 may belong to task T2 and be incorrectly positioned. For non-volatile storage, the picture manager saves the picture construction information in a binary file (picture store).

Text Editor — The text editor is a readily available operating system editing facility. It is used primarily to complete coding of program skeletons that are produced by the transformation tools. Secondary use is found for design annotation.

Transformation Tools — Transformation tools convert the internal representation of the system under design to formats more useful for analysis or presentation. This includes: (1) the generation of picture hardcopies from the picture information and (2) the generation of code skeletons from logic information and graphical annotation. The generated code skeletons are stored in text files (code store).

Analysis Tools — The analysis tools are integrated into the design environment through two interfaces: (1) the logic store, the source of design information and (2) the graphical interface through which the user specifies the parameters of the analysis and analysis results are displayed, graphically or textually, as appropriate.

ate.

Logic Store — The logic store is a file of Prolog facts describing the logical associations of each icon. There are five types of facts: object, component, abstraction, reference and information. Object facts represent icons that are used for abstracting sets of atomic icons; in Ada these correspond to packages and tasks. Component facts are representations all other icons with the exception of interconnection icons which are expressed in reference facts. Parent-child relationships between icons are stated explicitly in object and component facts. Abstraction facts are used to group child components of a parent object in the appropriate object perspective: internal or external. External components are those that represent the access to the components hidden within the object, the internal components. In Ada, examples of external components are visible package procedures and task entries. Ada internal components include nested tasks, packages and procedures. Information facts store annotation corresponding to objects, components and references. Temporal and functional information representation and storage is not currently implemented.

The CAEDE development vehicle is a SUN graphics work station on an Ethernet local area network. The local area network includes a laser printer and a VAX-11/750 computer. The operating system for all computers is Berkeley Software Distribution UNIX version 4.2 (BSD UNIX 4.2). The graphics interface, the LSM and PM are implemented in 'C' using SUNCORE graphics for iconic support. The analysis tools, (with the exception of the performance analyzers) and the code skeleton generation tools are written in MProlog [68].

2.2.2 Other Design Environments

The range of design environments and/or tool-sets described in recent journals is highly diverse. In some cases, completed research is presented, in others only descriptions or goals are provided. In the following discussion we outline several research efforts in order to identify the typical support tools. We do not attempt to cover all tools or environments, only those that are representative of particular approaches.

The *SARA* design environment [36,37] is very similar to CAEDE in its basic functions. *SARA* supports three complementary graphical design notations: *UCLA* control graphs (akin to Petri-nets [36]), data flow graphs and sub-system modules. A special PL/I derivative called PLIP is used to specify data types and data transformations. The chief design evaluation tools are a simulator [22,88] and a control graph analyzer [86]. The simulator supports the interactive execution of a design specification and performance analysis of the design. The control graph analyzer performs a variation of reachability analysis.

The *P-NUT* tool-set [87] is an outgrowth of the *SARA* experience. Unlike CAEDE, it is strictly a concurrent system experimentation facility, rather than a software development environment. Based on Petri-nets, *P-NUT* supports graphical entry, deadlock analysis (using reachability analysis), performance analysis and a Petri-net interpreter.

Bruno and Marchetto [14] describe a tool-set based on a variant of Petri-nets, called *PROT-nets*. Although this tool-set is not operational, a tool complement similar to that of *P-NUT* is suggested: a graphical entry tool and a discrete event simulator. Unlike *P-NUT*, an Ada code generator tool is identified.

The *PAISley* language and environment [116] define an executable specification language and tool-set to support concurrent system development. Although the *PAISley* environment does not support target language code generation, it possesses several tools: a consistency checker, a cross referencer and a powerful interactive interpreter; a graphical input tool is suggested.

The *Formal System Development* (FSD) testbed described by Balzer [6] supports a high level executable specification language called *GIST*. Support tools for *GIST* include a symbolic evaluator and a language transformation tool; the latter tool can be used to generate target language code, such as LISP.

DREAM [91,92,93,108] was an early concurrent system design environment based on *event expressions* and state transitions. Support tools included a simulator and a consistency checker based on the event expressions. The generation of code for a target programming language was never addressed. Like *P-NUT*, *DREAM* was primarily an evaluation and modelling system.

Constrained expressions (derived from DREAM's event expressions) are the formal model of a design environment suggested by Avrunin et al. [4]. The proposed environment would include tools for behaviour generation (an interpreter) and analysis (for deadlock and other conditions).

Arcadia, a future Ada development environment described by Taylor et al. [103], will use constrained expressions and *ANNA* declarative specifications as some of its formal models. A particularly interesting aspect of *Arcadia* will be its tool extensibility provided by a "tool building tool" call *GRAPHITE* [27]. *GRAPHITE* will generate an attributed graph data structure that will be central to all tools.

Arcturus [102], an Ada programming environment, incorporates static analysis tools for deadlock analysis [101] and dynamic debugging tools for Ada tasking [41,53]. *TOOL-PACK* [78] is an early FORTRAN programming environment that also uses static analysis tools (such as syntax checkers) and a sophisticated data flow analyzer. A general purpose, intelligent programming environment called *KBEmacs* [107] will eventually support an automatic deduction tool to perform contradiction detection; for example, it would detect a violation of data types.

Several specialized environments for the design of user interfaces have been proposed [54,58,106]. Jacob [58] and Wasserman [106] both describe systems based on finite state machine specifications supported by: a graphical entry tool and an animated, interactive, state machine interpreter. (Several programming environments have demonstrated animated, interactive language interpreters [13,89]). Hoffman [54] describes a user interface design environment that uses specification by example and an expert system evaluation tool for human factors analysis of the user interface.

Harandi and Lubars [49] describe a knowledge-based design environment utilizing data flow design principles. The knowledge-based design aid within the environment performs continuous consistency checks on the design. Future tools include a code generator and a simulator.

The *PegaSys* environment [76] supports graphical documentation of completed programs. An interactive consistency checker ensures that the program and graph-

ical representation reflect the same information. A graphics animator (interpreter) is suggested as a future tool.

Finally, the *MASCOT* environment [15] supports graphical design and code generation similar to CAEDE but no analysis tools.

Chapter 3

Problems and Methods in Temporal Analysis

In this chapter we characterize unsatisfactory temporal behaviours in concurrent systems; these includes: deadlock, starvation and critical races. These descriptions are followed by a survey of currently applicable formal models and temporal analysis techniques. Finally, we present the formal model used as the basis of our operational specification language and analysis tools: a temporal logic representation that describes allowable sequences of program states by expressing legal transformations of program state information.

3.1 Undesirable Temporal Behaviour

Concurrent systems can exhibit unsatisfactory temporal behaviours that are not found in sequential systems. Specifically, we identify the following behaviours: deadlock, starvation and critical races. Most of the research on this subject stems from the design of multiprogramming operating systems since they were the earliest examples of concurrent systems. In these operating systems, failure due to improper temporal behaviour was usually a consequence of multiple user tasks competing for a set of shared resources (e.g. memory, I/O devices). With the advent and popular use of concurrent programming languages such as Modula-2

[110] and Ada [104], these problems now plague a much wider audience.

The need to eliminate unsatisfactory or fatal temporal behaviour from a concurrent system is obvious; however, practical methods by which this can be achieved are not as clear. As a starting point for developing appropriate techniques, we investigate the nature of the problem.

3.1.1 Deadlock

Deadlock is a notion that has been studied intensively since the mid 1960's. Several landmark papers appeared towards the late 1960's and early 1970's that clearly articulated the deadlock concept [29,51,56]. From this background we find a definition for deadlock (paraphrased from [32]): "a task in a multiprogramming system is said to be in a state of deadlock if it is waiting for a particular event that will not occur". Coffman et al. [29] described four necessary conditions for deadlock from the perspective of resource management in a multiprogramming operating system (paraphrased version from [32]):

1. tasks claim exclusive control of the resources they require (*mutual exclusion condition*);
2. tasks hold resources already allocated to them while waiting for additional resources (*wait for condition*);
3. resources cannot be removed from the tasks holding them until the resources are used to completion (*no pre-emption condition*);
4. a circular chain of tasks exists in which each task holds one or more resources that are requested by the next task in the chain (*circular wait condition*).

We now recast this definition from the perspective of the Ada language by the example of Figure 3.1 taken from [17, pg. 73].

In Figure 3.1, the three tasks, T1, T2 and T3 behave as follows: (1) task T1 executes a cycle in which it calls T3.E1 and then accepts entry E1, (2) task T2 executes a cycle in which it calls T3.E2 followed by a call on T1.E1 and (3) task

Figure 3.1: Hidden Cycle Deadlock

T3 endlessly executes the selective accept of entries T3.E1 and T3.E2. The guard on entry T3.E1 is initially false and is set true by a call on T3.E2 and set false by a call on T3.E1. The mechanics of this system with regard to the four deadlock conditions stated earlier are described below.

mutual exclusion condition — The processing services and data within an Ada task are the resources to be allocated; since only the Ada task has direct access to these resources, then the mutual exclusion condition is always true.

wait for condition — When a caller wants to rendezvous with an acceptor that is not currently accepting then it is suspended. Furthermore, an accepting acceptor is suspended if there are no pending callers; thus, the wait for condition is always true;

no pre-emption condition — An Ada task cannot be pre-empted from an attempt to rendezvous once the call or accept is executed, therefore the no pre-emption condition is always true.

circular wait condition — Assume the following events occur: (1) task T1 calls T3.E1, (2) T3 selectively accepts T3.E1 and T3.E2 but only T3.E2 is open and (3) task T2 calls T1.E1; since, T1 is waiting for T3, T3 is waiting for T2 and T2 is waiting for T1, then the circular waiting condition is true.

The *wait for* and *no pre-emption* conditions should be qualified because Ada does provide language elements that can eliminate these conditions (e.g. timed or

conditional calls).

3.1.2 Starvation

Starvation is a temporal anomaly that has the appearance of deadlock. A task is said to experience starvation if some scheduling mechanism creates a situation in which the task can be indefinitely prevented from executing; thus, the task seems to be in a state of deadlock although it isn't according to the definition of deadlock in section 3.1.1.

Starvation has been discussed under a variety of guises; such as, *livelocks* [33,62] and *indefinite postponement* [32]. Starvation has not been as heavily explored as deadlock because it depends on the rather ill-defined notion of *fairness* [62,85]. Although, there exists a formal description of starvation [62], an informal presentation of the nature of starvation is sufficient for this discussion. Buhr [17, pg. 157] provides a suitable description of starvation through a detailed examination of a classic starvation example: the readers and writers problem [30]. A short study of this problem will reveal the subtle nature of starvation.

The readers and writers problem involves the design of a system that will regulate concurrent read and write access to a secondary storage resource (such as a disk). Tasks play either one of the two access roles; a task is a reader or a writer but never both, simultaneously. The resource scheduling rules are defined as follows [17, pg. 198]:

- if the resource is idle then any task may gain access;
- if the task currently accessing the resource is a reader then any number of other reader tasks are also permitted access;
- if the task currently accessing the resource is a writer then no other task may be granted access.

As a simplification, we study only the case in which there is one writer.

The Ada structure graph for an implementation of the readers and writers example from [17, pg. 158] is shown in Figure 3.2. Figure 3.2 introduces a new iconic

Figure 3.2: Readers and Writers Example: One Writer

representation: a labeled set of overlapping parallelograms implies a pool of tasks performing the same sequence of operations; e.g. an array of tasks instantiated from the same task type.

The resource scheduler task in Figure 3.2 manages access to a shared resource. It has three selectively accepted entries: (1) *Req_read* — request for read, (2) *Req_write* — request for write and (3) *End_acc* — end access. Reader tasks call the *Req_read* entry to gain permission to access the resource. Similarly, the writer task calls the *Req_write* entry. Both the readers and the writer call the *End_acc* entry when resource access is no longer required. The entry guard *W_fin* (write finished) for the *Req_read* entry is false if the writer currently has access to the resource and is true otherwise; readers are not selected if the writer is assigned the resource. The *R_fin* (read finished) guard for the *Req_write* entry is false if any readers have access to the resource and is true otherwise. Initially, both guards are true. The *R_fin* flag is set false when a reader calls *Req_read* and it is set true when the last reader has called *End_acc*. The *W_fin* guard is set false when the writer calls the *Req_write* entry and it is set true when the writer subsequently calls the *End_acc* entry.

The starvation of the single writer by the set of readers is illustrated by the following execution pattern:

- the resource is idle;
- readers continually gain and relinquish access to the resource in an overlapping manner, therefore, the resource is never idle;
- the writer requests access but is blocked because the `W_fin` guard is false.

The writer can be indefinitely denied access to the resource because of scheduling bias; i.e. it experiences starvation. There is no deadlock in this situation since there may be an opportunity for the writer to proceed if all of the readers eventually relinquish access to the resource.

3.1.3 The Critical Race

The critical race is a well-known problem in concurrent systems [17,32,33,57,71]. We define a critical race as follows:

a system is said to contain a critical race if the execution rate of one or more tasks affects the validity of the system's expected results.

Although this definition seems to imply physical concurrency, systems that are only logically concurrent can also suffer from critical races; for example, task preemption can introduce unpredictable execution delays.

Critical races may occur when two (or more) tasks access some shared resource in the course of their operation. Each task assumes that the shared resource is in a particular state whenever it is accessed. However, since there is no mechanism that guarantees this assumption then the task execution times will dictate the actual outcome of the computation.

Controlling access to a shared resource in these circumstances implies two duties: (1) providing mutual exclusive access and (2) providing scheduled access. Mutually exclusive access may be sufficient in some case, but not in others. In general, scheduled access suggests mutually exclusive access; however, the reverse is not true.

In Ada, the task provides mutually exclusive access to any variables within its scope; another other task that wishes to modify these variables can do so only

indirectly, by way of the owner task. Scheduled access to a target task (the task controlling some variable), is not provided directly by the Ada language:

- competing callers arrive at task entries unpredictably, although they are accepted in FIFO order;
- competing callers at different entries of a selective accept arrive unpredictably and the selected entry may be chosen unpredictably.

Faced with this situation, the designer must exercise the scheduling at the application level. To illustrate the nature of critical races in Ada programs, we shall discuss briefly the *agent-pool* example from [17, pg. 167].

The Ada structure graph for the agent-pool example is shown in Figure 3.3. In this diagram there is a pool of User tasks, a pool of Agent tasks and a single Dispatcher task. An Agent task performs a specific set of services for a User task. The Agent tasks are dynamically assigned to the User tasks by the Dispatcher task. Every Agent task is uniquely identified with an Id number. The User tasks request Agent tasks by calling the `Reserve` entry of the Dispatcher. Once they are assigned an Agent's Id, User tasks call the `Agent(Id).Ask` entry for any required services. The `Agent(Id).Cancel` entry permits the User to release the Agent from service. A released Agent will call the `Dispatcher.Free` in order to identify itself as available. Following this, the Agent calls the `Dispatcher.Wait` entry where it is delayed by the `Wait_guard` until it is assigned to a User. The `Wait_guard` is set to true when `Dispatcher.Reserve` is called and there is a free agent; it is set to false when an Agent is accepted on the `Dispatcher.Wait` if there are no other Agent tasks to be dispatched.

We demonstrate the critical race between two Agent tasks, `Agent(1)` and `Agent(2)` to reach a target task, `Dispatcher`, as follows:

- `Agent(1)` and `Agent(2)` both become free;
- `Agent(1)` calls `Dispatcher.Free` and completes the rendezvous;
- `Agent(2)` calls `Dispatcher.Free` and completes the rendezvous;

Figure 3.3: Agent-Pool Example Illustrating a Critical Race

- Agent(2) calls `Dispatcher.Wait` and is blocked (`Wait_guard` is false);
- Agent(1) calls `Dispatcher.Wait` and is blocked (`Wait_guard` is false);
- User(1) calls `Dispatcher.Reserve` and is assigned the Id for Agent(1) (the first Agent to call `Free`);
- Dispatcher accepts the call by Agent(2) on `Wait`.

Observe that the wrong Agent has been dispatched for the Agent Id assigned to User(1). This resulted from the critical race between Agent(1) and Agent(2) in their attempts to call the `Free` and `Wait` entries. The thrust of this example is that, although the Ada task provides basic scheduled access to critical control variables through mutual exclusion, it does not provide automatically more sophisticated control, such as *fair scheduling*.

3.2 Methods for Temporal Analysis

Current methods for the temporal analysis of concurrent systems are based primarily on techniques for the proof of program correctness. The proof of concurrent

program correctness involves the study of two classes of program properties: *safety* and *liveness* [47]. Safety properties express the notion that if some part of a program executes, then it will yield correct results. A liveness property conveys the idea that a program will execute. Absence of deadlock, starvation and critical races are classified by Manna & Pnueli [72] according to program property as follows:

- absence of deadlock: a safety property;
- absence of starvation: a liveness property;
- absence of critical races: a safety property.

Proofs of program correctness can be established by primarily two means: *operational proofs* and *axiomatic proofs*. Operational proofs include *symbolic execution* of the program [48] and the evaluation of an executable specification of the program [5,6,14,58,106,114,116,117]. Axiomatic proofs are mathematical proofs based on some formal model, such as logic or algebra.

In general, we prefer axiomatic proofs over operational proofs for analysis because axiomatic proofs can establish truth without question. Operational proofs are not proofs in the mathematical sense; they are in many respects, similar to program testing. Nevertheless, operational proofs are essential when axiomatic proofs are not possible, or, not practical.

Formal models used in the axiomatic proof of concurrent program properties fall in, or between, the following two categories [40,92,98]:

1. those using a program state representation (where a program state is the N -tuple of the control and data states of N executing tasks);
2. those using an event sequence representation (where possibly, events are program states).

Petri-nets [2,9,66,81,82,112], State Machines [10,12,44], Taylor's control flow graph [101] and UCLA Graphs [36,37,86] are examples of models that use a state representation of program behaviour. Temporal logic [72,83,84], Milner's Calculus of

Communicating Systems (CCS) [74] and constrained expressions [40,4] all assume a model of event sequences, where the events range from message transmissions to program states. We do not attempt to consider all known formal models in this discussion since the set of models we have mentioned is fairly representative; for example, Harel's Dynamic Logic is equivalent to branching time temporal logic [8] and constrained expressions [4] are generalized forms of traces [75,77].

The two approaches to modelling are very different: in state-based models the program is represented by the set of program states and transitions between program states; in event sequence models, the program is expressed by allowable sequences of atomic events. However, to some degree, the models within a category express equivalent ideas, as we illustrate below.

- UCLA graphs have been shown to be equivalent to Petri-nets [36] and state machines are a special type of Petri-net [81]; Taylor's control flow graph is similar to the program state machine for a concurrent system.
- Constrained expressions use a regular expression notation to define allowable sequences of events. Discrete linear temporal logic uses temporal operators to express truths regarding an infinite sequence of program states [72]. CCS uses algebra to represent sequences of messages between communicating tasks; algebraic operators and formula manipulation are used to prove specific program properties.

The distinctions between apparently equivalent models appear in their emphasis, complexity and completeness. We will explore these issues shortly when we examine the merits of the various representations.

The analysis capabilities of the above formal models address the temporal behaviour problems we identified in the previous section: deadlock, starvation and critical races. Below, we describe the relevant analysis methods used by the different models.

3.2.1 State-based Analysis Methods

The state-based formal models, state machines, Petri-nets and UCLA graphs all use a structure known as the *reachability tree* to perform deadlock and starvation analysis [10,81,37]; analysis based on a reachability tree is called *reachability analysis*. Since the three models are similar in this regard, we shall discuss reachability from the perspective of state machines, the simplest of the models.

A finite state machine (FSM) for a single task is a bounded network of nodes, denoting task states and directed arcs, which identify successor states that result from changes of state. In communicating finite state machines, changes of state generally occur because of the transmission or reception of a message (to or from other tasks). A message channel (or buffer) of some pre-described size (possibly infinite) is considered part of a transmitting task's data state; there is usually one message channel per sender-receiver pair.

The program (or global) state machine is the cross product of the N task state machines. Transitions between program states represent the change of state of one task only. A reachability tree is derived from a program state machine by constructing a spanning tree starting from some designated *initial program state*. The previous outgoing branches of the terminating nodes are identified, but never traversed; for example, if node x leads to the initial program state, this relationship will be stored, however, when transversing the tree, the x to initial program state branch will never be travelled.

Reachability analysis begins with the identification of any program state x in the reachability tree for which there exists a directed path from the initial program state to state x ; such a state is called a *reachable state*. The analysis performed on a reachable state depends on the program property to be proven;

deadlock analysis — if there does not exist an outgoing transition from a reachable state, then the reachable state is a *deadlocked state*;

starvation analysis — if there does not exist a directed path from a reachable state back to the initial program state (or some other state representing progress of the program), then the reachable state is said to be *livelocked*.

The reachability-based deadlock analysis corresponds exactly the notion of deadlock we illustrated in § 3.1.1; however, the starvation analysis does not exactly represent the notion of starvation we presented in § 3.1.2. Livelock and starvation both express the idea that the program is not progressing for some reason. The difference lies in the interpretation of "progress". For starvation we have adopted the definition of progress to mean that all tasks must execute eventually; i.e. in the absence of deadlock, all tasks cannot be blocked forever. The concept of livelock in communicating state machines defines progress to be the eventual return of the system to some "stable" program state (such as the initial program state). If we assume (quite reasonably) that starvation implies livelock, then livelock analysis can be used to show absence of starvation.

State-based models do not provide an effective proof framework for absence of critical races for two several reasons: (1) pure state machines are not sufficiently expressive to describe mutual exclusion [9] and (2) although UCLA graphs and Petri-nets can describe mutual exclusion (using the *boundedness* property [81]), there is no general definition of scheduled access (cref. § 3.1.3) that can be used to identify critical races.

3.2.2 Event Sequence-based Analysis Methods

The event sequence-based formal models possess tremendous reasoning powers for temporal analysis. For example, liveness and safety properties usually can be proven with only a few axiomatic operations (as illustrated in [4,73,74]). These capabilities are particularly impressive when compared to the exponential computational complexity of reachability analysis; i.e. "state space explosion" [10].

Unfortunately, mechanical procedures for proving absence of deadlock, starvation and critical races do not exist (in general) for event sequence-based formal models. All of the proof methods are either, mechanical for some restricted set of problems, or completely manual. To investigate this issue further, we summarize below, the state of research into proof methods using event sequence-based formal models.

Constrained Expressions — Avrunin et al. [4] present manual, algebraic proofs

of absence of deadlock using constrained expressions that describe an Ada program; a mechanical proof procedure does not exist as yet.

Milner's CGS — Milner [74] describes several manual safety and liveness proofs, however, no mechanical proof procedures exist.

Temporal Logic — Mechanical proof procedures (called *decision procedures*) exist for propositional branching time temporal logic [8] and propositional linear time temporal logic [24,25]. A wide range of manual proofs and proof strategies for safety and liveness properties are presented in [47,73,77,79,96]. Clarke et al. [26] describe a mechanical technique to verify that a state-based representation of a program matches a propositional branching time temporal logic specification of the program.

3.2.3 Selecting a Formal Model

In selecting a formal model for our temporal analysis tool-set, we use the following set of guidelines:

- the chosen model should suit the Ada-oriented design model;
- the chosen model should permit axiomatic proofs of temporal properties;
- the chosen model should permit operational proofs, if desired.

Of the formal models we have considered, we choose a hybrid model of temporal logic and state machines similar to the *bounded state* class of models described by Schwartz & Melliar-Smith [98] and illustrated in [97]. In this model, we use the power of temporal logic to describe sequences of program states, but express the behaviour as allowable transformations of program state information, such as program variables and program location. Below, we contrast, formally and informally, this approach with pure event sequence models and pure state models.

Pure event sequence model — a program is defined by the allowable histories of interface events with no consideration of program state information; i.e. the program is described by how it behaves in response to outside stimuli.

Pure state model — program behaviour is defined by the current program state and the outside stimuli; the current state encodes history information.

Bounded state model — the program history is described by both bounded history, as provided by program state information, and by temporal logic operators, which describe allowable transformations of program state information; in effect, a complete history is presented in terms of (repeated) sections of bounded history.

An illustration comparing our use of temporal logic in a bounded state (temporal logic) model to the pure state model is presented in the chapter 4, § 4.1.3.

We have not considered either of the algebraic event sequence models, constrained expressions or CCS, because when compared to temporal logic, they are syntactically complex and their mechanical axiomatic powers are less developed. The benefits of the temporal logic-state model over the pure state model are reflected through our selection guidelines, as shown below.

Suitability of Representation — Temporal logic is a natural means of expressing program behaviour because it does not impose any artificial models on the concurrency model, in this case, Ada; that is, temporal logic can describe the behaviour of an Ada program in its own terms. For example, we describe the end of a rendezvous on entry *A.E* between two tasks, *A* and *B*, as follows (in english rather than symbolic form):

if ending a rendezvous between task *A* and task *B* on entry *A.E* then
eventually, task *A* will be after the end statement of its accept statement for *A.E* and eventually task *B* will be after its call statement to *A.E*.

The equivalent description in terms of communicating state machines would be:

Task A — if in end of rendezvous state for *A.E* then transmit reply message to *B* followed by a transition to "after end accept" state;

Task B — if in a waiting for *A.E* reply message and message is received then make a transition to the "after the call" state.

Both of these descriptions are correct, however, the temporal logic version expresses the program behaviour strictly in Ada terms, unlike the communicating state machines, which introduce message passing.

Axiomatic Proofs — Axiomatic proofs for absence of deadlock and starvation using this hybrid model are demonstrated as part of this research. Although a similar deadlock analysis could be performed using communicating state machines, the temporal logic approach can save some processing steps because it describes the program in terms of allowable sequence of program states. By contrast, communicating state machines express program behaviour as the cross product of individual task states, thus, a cross product must be generated before analysis can begin. Techniques have been developed to generate simplified program state machines efficiently [37,45,95] in general, and under certain constraints [43,44,113]; however, many of these techniques can be avoided simply by using the reasoning power of temporal logic. Starvation analysis based on livelock in communicating state machines is awkward because of the requirement to identify stable or home program states.

Operational Proofs — Operational proofs are enhanced in the bounded state class by the presence of program state information [98]. Communicating state machines possess an excellent operational interpretation because they describe program behaviour at its lowest level [115]; however, temporal logic enhanced with state information provides a suitable and more natural operational interpretation than state machines.

Chapter 4

COL: An Operational Specification Language

In this chapter we present the CAEDE Operational Language (COL). This language, based on *Discrete Linear Temporal Logic* (LTL), specifies Ada program behaviour in a manner amenable to both axiomatic and operational analysis. The subject is covered in three sections. In section 4.1 we motivate the use of LTL as the formal basis of COL through background and example.

Section 4.2 is a description of the language development from its LTL roots in two phases: (1) a preliminary language based on a single caller, single acceptor rendezvous and (2) the final language, in which the rendezvous restriction is relaxed.

Finally, in section 4.3, we define a practical implementation of COL syntax using the structures and conventions of Prolog; this permits easy implementation of the operational semantics using Prolog. An example of an Ada program specified in COL is provided.

4.1 Language Motivation

The CAEDE Operational Language is based on a temporal logic representation of Ada programs. In this section, we motivate the relationship between temporal

logic and Ada: first, through a brief, informal introduction to temporal logic and second, by way of a simple example that illustrates the reasoning power of temporal logic in an Ada perspective.

4.1.1 Temporal Logic: An Overview

Temporal Logic is an extension of classic logic that includes operators that express the validity of formulæ with respect to time; it provides the capability to make logic assertions that involve a temporal component. From the family of temporal logics, *Discrete Linear Time Temporal Logic* (LTL) uses the following interpretation of time:

time is expressed as a sequence of discrete instances referenced relative to a temporal instance t_0 or *now* that follows a deterministic future (hence it does not include the past).

Rescher and Urquhart [90] provide descriptions of many other possibilities.

Interest in the LTL system stems from its use in proving properties of concurrent systems [34,35,63]. However, there exists a popular alternative temporal logic: *Discrete Branching Time Temporal Logic* (BTL) [8]. BTL allows for multiple non-deterministic futures relative to some instance t_0 . Since concurrent systems are generally non-deterministic, it would seem natural that BTL is more appropriate [63]. This conclusion is true in general, but if we are attempting to prove global properties about a system (such as absence of deadlock) then we must examine all possible futures [34,35,111]. Thus, we can avoid the additional complexity of BTL by converting non-deterministic programs into a set of all possible deterministic programs and reason instead with LTL.

We convert a non-deterministic program to a set of deterministic programs by enumerating all of the possible program behaviors; i.e. all possible orderings of events. An analogous method is used to convert classic predicate logic to propositional logic. We illustrate this method by example:

— Suppose that for all x , $is_man(x)$ is valid and the domain of x is the subset

of male names: john, fred and david. The predicate variable x represents a non-deterministic value.

— Given a finite, enumerable domain for x , we can assert the equally valid statement: $is_man(john)$ and $is_man(fred)$ and $is_man(david)$; that is, a conjunction of the predicate is_man substituted with all of the individuals in the domain. The substitutions create propositions, which are deterministic statements of validity.

— Therefore, if we use $is_man(x)$ for all x in some proof of some property about members of x , then we can generate an equivalent proof by showing that the property holds for all substitutions of x in is_man .

We will use a similar approach to convert a non-deterministic program into its set of equivalent deterministic programs.

There are four basic temporal operators in LTL: *eventually* ($\Diamond$), *next* ($\bigcirc$), *until* (U) and *henceforth* ($\Box$). Below we describe informally the semantics of these operators.

eventually — The formula $\Diamond P$ is true if, in the sequence of instances starting from t_0 , there is some instance in the future at which the formula P is true.

next — The formula $\bigcirc P$ is true if, in the sequence of instances starting from t_0 , the formula P is true in the sequence of instances starting from t_1 .

until — The formula $P U Q$ is true if, in the sequence of instances starting from t_0 there is an instance in which the formula Q is true and for all instances up until then the formula P is true.

henceforth — The formula $\Box P$ is true if, P is true in all instances of the sequence of instances starting from t_0 .

Any formula that is not qualified by a temporal operator is assumed to be true in t_0 only. There are many axioms and several deduction rules for LTL; we list a subset of those (which are relevant to this work) in Appendix B. Complete formal definitions of LTL (also called DX temporal logic) can be found in [47,72,84].

As an illustration of the use of LTL consider a simple example. We assume the following formulae are valid for all time:

(1) "if it is raining now then it will eventually be both sunny and not cloudy."

The two temporal references in (1) are "now" and "will eventually".

From this statement, LTL will allow us to deduce (2).

(2) "if it is raining now then it will eventually be sunny and it will eventually be not cloudy."

If statement (2) was the original assertion instead of (1), then LTL would NOT allow us to deduce (1) simply on the strength of (2) (since it may never be sunny and not cloudy simultaneously).

Before discussing the symbolic representation of this example, below we define a set of logic symbols that will be used in the remainder of this thesis.

$\Diamond$	— eventually operator	$\Box$	— henceforth operator
$\bigcirc$	— next operator	U	— until operator
$\wedge$	— logical conjunction	$\vee$	— logical disjunction
$\Rightarrow$	— logical implication	$\Leftrightarrow$	— logical equivalence
$\neg$	— logical negation	$\models$	— denotes a valid formula
a, b, z	— predicate variables	A, B, Z	— formula placeholder

Character strings of (possibly subscripted) capitalized letters ($A - Z$) shall be used for symbols representing specific individuals; e.g. T_1 . In our use of predicate variables, we assume that all variables are universally quantified, unless stated otherwise. We assume also that whenever two subscripted predicate variables, using the same name but different subscripts occur in the same formula, they do not represent the same instance; e.g. $(t_1 \wedge t_2)$ implies t_1 is not the same instance as t_2 .

The notion that a formula is valid implies a strong meaning: $\models P$ states that the formula P is true under *all* circumstances, e.g. all time, all space. Thus, it follows that

$$\text{if } \models P \text{ then } \models \Box P$$

since $\models P$ implies that P is true for all time. Observe that this is not the same as $\models P \Rightarrow \Box P$ which is interpreted to be

"it is always true that if P is true in t_0 then P will be true always."

- $$\begin{aligned}
 (1) \quad & \models is_raining \Rightarrow \Diamond(\neg is_cloudy \wedge is_sunny) && \text{(given)} \\
 (2) \quad & \models \Diamond(P \wedge Q) \Rightarrow \Diamond(P) \wedge \Diamond(Q) && \text{(B.25)} \\
 (3) \quad & \models is_raining \Rightarrow \Diamond(\neg is_cloudy) \wedge \Diamond(is_sunny) && (1, 2, \text{B.41})
 \end{aligned}$$

Figure 4.1: Rain Example Expressed in LTL

Following the convention of Manna and Pnueli [72], we use the symbol $\models$ to define formulæ that are valid within some particular domain, as opposed to universally valid; that is, $\models P$ is equivalent to $S \models P$ for some domain S . Manna and Pnueli [72] also show that any axioms or deduction rules that are valid under $\models$ are also valid under $\models$. Thus, to our existing set of symbols we add $\models$:

$\models$ — denotes a valid formula in some domain

The LTL description of the rain example appears in Figure 4.1. Statement (3) is derived statement from statement (1), an LTL axiom and an LTL syllogism deduction rule. There are no LTL axioms or deduction rules that will allow (1) to be derived from (3). From this example, we see the symbolic logic proof style to be used for the remainder of this work:

- each statement is labeled, e.g. (5);
- the source of a statement is provided: it may be given, an assumption, a statement within the proof and statements, axioms or deduction rules within the body of the text or Appendix B.

For convenience, multiple steps may be combined into a single step, whenever the transformation is relatively straightforward.

LTL has been applied in a number of application areas to specify and/or verify properties of concurrent systems. Examples are found for concurrent systems software [26,47,73,77,79,83,84,85], hardware design [11] and communications protocols [24,96,97,98]. It has also been used as a basis for automatic software synthesis from

Figure 4.2: Two Task Design Illustrating Temporal Logic

temporal logic specifications [111]. Since we are concerned primarily with verifying properties of concurrent programs, below we present Manna and Pnueli's interpretation of discrete time instants from the perspective of concurrent programs [72].

A concurrent program is defined to be a set of N concurrent, infinitely executing tasks. The state of a concurrent program is an N -tuple of the control and data states of the N tasks. A change in program state is caused by a change in at most one of the tasks. Using this definition of a concurrent program, discrete time is interpreted as an infinite sequence of program states; that is, each state of a program corresponds to a time instant. Thus, a formula is true if it is true in a particular program state as opposed to an actual time instant.

4.1.2 Ada and Temporal Logic

As a means of motivating the use of temporal logic descriptions of Ada programs, let us examine the application of temporal logic to a very simple Ada program.

The design of Figure 4.2 consists of two communicating tasks. Task A produces data, transmits it to task B and then receives data from task B. Task B accepts data from task A and then returns it to task A. Both tasks perform these duties endlessly. The Ada code for this design is presented in Figure 4.3.

Below we give the temporal logic assertions that describe this design; following each assertion is an english interpretation. Note the use of the predicates *at* X_n and *after* X_n ; these are defined as follows:

- *at* X_n — task X is ready to execute statement n ;


```

1      task A is
2          entry A_entry(N : in INTEGER);
3      end;

4      task body A is
5          I : INTEGER;
6      begin
7          I := 1;
8          loop
9              B.B_entry(I);
10             accept A_entry(N : in INTEGER) do
11                 end A_entry;
12                 I := (I + 1) mod 10;
13             end loop;
14     end A;

1      task B is
2          entry B_entry(N : in INTEGER);
3      end;

4      task body B is
5      begin
6          loop
7              accept B_entry(N : in INTEGER) do
8                  end B_entry;
9                  A.A_entry(N);
10             end loop;
11     end B;

```

Figure 4.3: Ada Code for Two Task Design Example

- *after* X_n — task X has completed the execution of statement n and is ready to execute the next statement.

Two Task Design: LTL Description

$$\models \text{at } A_8 \Rightarrow \bigcirc \Diamond (\text{at } A_9 \cup (\text{at } A_9 \wedge \text{at } B_7) \wedge \neg \text{at } A_9) \quad (4.1)$$

It is always true that if task A is at the top of its loop in the current program state then, in the sequence starting from the next program state, task A will eventually execute the task call on entry B.B.entry until a rendezvous with task B occurs.

$$\models \text{at } A_9 \wedge \text{at } B_7 \Rightarrow \bigcirc \Diamond \text{after } A_9 \wedge \bigcirc \Diamond \text{after } B_7 \quad (4.2)$$

It is always true that if task A and task B rendezvous on the entry B.B.entry in the current state then, in the sequence starting from the next program state, task A will eventually return from the task call and eventually task B will end the accept.

$$\models \text{after } A_9 \Rightarrow \bigcirc \Diamond (\text{at } A_{10} \cup (\text{at } A_{10} \wedge \text{at } B_8) \wedge \neg \text{at } A_{10}) \quad (4.3)$$

It is always true that if task A has returned from the task call on B.B.entry in the current program state then, in the sequence starting from the next program state, task A will eventually execute an accept of the entry A.entry until a rendezvous with task B occurs.

$$\models \text{at } A_{10} \wedge \text{at } B_8 \Rightarrow \bigcirc \Diamond \text{after } A_{10} \wedge \bigcirc \Diamond \text{after } B_8 \quad (4.4)$$

It is always true that if task A and task B rendezvous on the entry A.A.entry in the current state then, in the sequence starting from the next program state, task B will eventually return from the task call and eventually task A will end the accept.

$$\models \text{after } A_{10} \Rightarrow \bigcirc \Diamond \text{at } A_8 \quad (4.5)$$

It is always true that if task A has ended its accept of entry A.entry in the current state then, in the sequence starting from the next program state, task A will eventually be after the top of its loop.

$$\models \text{at } B_8 \Rightarrow \bigcirc \Diamond (\text{at } B_7 \cup (\text{at } B_7 \wedge \text{at } A_9) \wedge \neg \text{at } B_7) \quad (4.6)$$

It is always true that if task B is at the top of its loop in the current program state then, in the sequence starting from the next program state, task B will eventually execute an accept of the entry B_entry until a rendezvous with task A occurs.

$$\models \text{after } B_7 \Rightarrow \bigcirc \Diamond (at B_9 \cup (at B_9 \wedge at A_{10}) \vee \square at B_9) \quad (4.7)$$

It is always true that if task B has ended the accept on B_entry in the current program state then, in the sequence starting from the next program state, task B will eventually execute a task call on A.A_entry until a rendezvous with task A occurs.

$$\models \text{after } B_9 \Rightarrow \bigcirc \Diamond (at B_6) \quad (4.8)$$

It is always true that if task B returned from its call on A.A_entry in the current state then, in the sequence starting from the next program state, task B will eventually be after the top of its loop.

Given this LTL description of our example, we can prove the absence of deadlock in this two task system. The approach we take is to show that if the two tasks can eventually be at the top of their loops, then they will return to them infinitely often; that is, we prove absence of deadlock using induction

$$\models \Diamond at A_8 \wedge \Diamond at B_6 \Rightarrow \square (\Diamond at A_8 \wedge \Diamond at B_6) \quad (4.9)$$

which is implied by

$$\models \square \{ \Diamond at A_8 \wedge \Diamond at B_6 \Rightarrow \bigcirc (\Diamond at A_8 \wedge \Diamond at B_6) \} \quad (4.10)$$

under the LTL induction axiom (B.27). The key element of the proof is to demonstrate that the rendezvous between task A and task B must occur. Intuitively, this is acceptable because there is only one caller for any acceptor and both callers and acceptors remain suspended until a rendezvous can begin. We leave this proof step as an informal argument; a generalized version of this implication will be dealt with formally in § 4.2.2 in the context of language development.

The proof of absence of deadlock in the two task system is as follows:

- $$\begin{aligned}
 & \Diamond at A_8 \wedge \Diamond at B_6 \Rightarrow \\
 (1) \quad & \equiv \bigcirc \left[\begin{array}{l} \Diamond (at A_9 U (at A_9 \wedge at B_7) \vee \square at A_9) \wedge \\ \Diamond (at B_7 U (at B_7 \wedge at A_9) \vee \square at B_7) \end{array} \right] \quad \left(\begin{array}{l} 4.1, 4.2, B.43, \\ B.40, B.9, B.23 \end{array} \right) \\
 (2) \quad & \equiv \Diamond at A_8 \wedge \Diamond at B_6 \Rightarrow \bigcirc \Diamond (at A_9 \wedge at B_7) \quad (1, \text{rendez}, B.35) \\
 (3) \quad & \equiv \Diamond at A_8 \wedge \Diamond at B_6 \Rightarrow \bigcirc \Diamond after A_9 \quad \left(\begin{array}{l} 2.4.2, \\ B.3, B.42 \end{array} \right) \\
 (4) \quad & \equiv \Diamond at A_8 \wedge \Diamond at B_6 \Rightarrow \bigcirc \Diamond after B_7 \quad \left(\begin{array}{l} 2.4.2, B.3, \\ B.42 \end{array} \right) \\
 (5) \quad & \equiv \Diamond at A_8 \wedge \Diamond at B_6 \Rightarrow \bigcirc \Diamond (at A_{10} U (at A_{10} \wedge at B_9) \vee \square at A_{10}) \quad (3, 4.3, B.42) \\
 (6) \quad & \equiv \Diamond at A_8 \wedge \Diamond at B_6 \Rightarrow \bigcirc \Diamond (at B_9 U (at B_9 \wedge at A_{10}) \vee \square at B_9) \quad (4, 4.7, B.42) \\
 (7) \quad & \equiv \bigcirc \left[\begin{array}{l} \Diamond (at A_{10} U (at A_{10} \wedge at B_9) \vee \square at A_{10}) \wedge \\ \Diamond (at B_9 U (at B_9 \wedge at A_{10}) \vee \square at B_9) \end{array} \right] \quad \left(\begin{array}{l} 5, 6, B.40, \\ B.8, B.23 \end{array} \right) \\
 (8) \quad & \equiv \Diamond at A_8 \wedge \Diamond at B_6 \Rightarrow \bigcirc \Diamond (at A_{10} \wedge at B_9) \quad (7, \text{rendez}, B.35) \\
 (9) \quad & \equiv \Diamond at A_8 \wedge \Diamond at B_6 \Rightarrow \bigcirc \Diamond after A_{10} \quad \left(\begin{array}{l} 8, 4.4, B.3, \\ B.42 \end{array} \right) \\
 (10) \quad & \equiv \Diamond at A_8 \wedge \Diamond at B_6 \Rightarrow \bigcirc \Diamond after B_9 \quad \left(\begin{array}{l} 8, 4.4, B.3, \\ B.42 \end{array} \right) \\
 (11) \quad & \equiv \Diamond at A_8 \wedge \Diamond at B_6 \Rightarrow \bigcirc (\Diamond at A_8 \wedge \Diamond at B_6) \quad \left(\begin{array}{l} 9, 4.5, 10, 4.8, \\ B.42, B.40, \\ B.8, B.23 \end{array} \right) \\
 (12) \quad & \equiv \Diamond at A_8 \wedge \Diamond at B_6 \Rightarrow \square (\Diamond at A_8 \wedge \Diamond at B_6) \quad (11, B.29, B.27)
 \end{aligned}$$

Q.E.D.

Note: *rendez* as a proof step source signifies our informal rendezvous implication formula.

Two Task Design: Discussion

The two task design example, simple as it is, illustrates many interesting points. Below, we discuss some useful observations.

Program variables and functions operating on program variables (such as $I := (I + 1) \bmod 10$ from line A_{12}) play no part in the LTL description or the temporal reasoning used in the absence of deadlock proof (they represent *functional behaviour*, as defined in chapter 1). While some variables and their functions may not be required for certain types of reasoning, they must be represented in some form, in general.

The temporal rules of our design do not attempt to capture the complex behaviour of the Ada accept process; i.e. it was not explicit that a rendezvous begins, a critical section is executed and the caller released. Statements 4.2 and 4.4 summarize the accept process rather simply: if the rendezvous begins then both tasks eventually execute past the accept and call structures. Furthermore, this example does not attempt to describe the non-deterministic aspects of the rendezvous since it involves only two tasks.

The LTL statements describing the design are rather cryptic even though they correctly represent the design. Furthermore, predicates that are based purely on line numbers are difficult to interpret and lack generality.

The structure of the LTL formulæ is quite regular; i.e. most formulæ are of the form $pred_1(\dots) \Rightarrow \bigcirc \Diamond pred_2(\dots)$. In addition, each statement indicates the progress of a task (or tasks) in the program.

The absence of deadlock proof demonstrates the axiomatic power of LTL; however, it also shows the operational behaviour of the program. Each inference of the proof represents progress in the system, relative to some initial state. The inference derives its operational flavour from the progress-based formulæ of the LTL description. In fact, the sequence of steps in the proof was deliberately chosen in order to suggest an operational behaviour. The same proof could have been reasoned by other approaches that would have obscured an operational interpretation.

All of these observations lead to steps in the development of the CAEDE Op-

erational Language: (1) integrating program variables and functions with the LTL framework, (2) simplifying, generalizing and extending the program predicates, (3) standardizing the structure of formulæ and (4) defining the operational semantics

4.1.3 A State Interpretation of LTL

The behaviour of the two task example in Figure 4.3 as defined in LTL formulae can be equivalently expressed by communicating finite state machines (FSMs). This comparison is useful because it highlights the differences between the two approaches. The particular contrast we show is:

- LTL formulæ specify and analyze program behaviour in terms of allowable sequences of programs states;
- communicating FSMs describe program behaviour by specifying the behaviour of individual tasks by FSMs assuming message channels between communicating FSMs; analysis is done using the program (global) state machine, which is the product of the states of the individual state machines and the states of the message channels.

The FSM specifications for the two task design are provided in Figure 4.4. We have implemented the Ada rendezvous by message passing:

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

We adopt the convention of marking transitions defining message transmission with " $- m$ " (where m is the message) and " $+ m$ " for message reception. We assume a non-blocking send for message transmission and a wait-receive for message reception. There are two implied message channels between the tasks.

The program state machine is generated by taking the product of the two FSMs and their communication channels; thus, the program state is a tuple of

Figure 4.4: State Machine Implementation of Two Task Design

Figure 4.5: Program State Machine Segment for Two Task Design

the states of the individual tasks and the states of each message channel. The transitions between program states represent the change of state of one task.

A segment of the program state machine for the FSMs in Figure 4.4 is shown in Figure 4.5. This segment describes the beginning of the rendezvous on entry $B.B_entry$ to the release of the caller task. The equivalent LTL formula is equation 4.2, repeated below.

$$\models at A_9 \wedge at B_7 \Rightarrow \bigcirc \Diamond after A_9 \wedge \bigcirc \Diamond after B_7$$

In Figure 4.5 there are two state sequences: (1,2,3,5) and (1,2,4,5). The different sequences are generated from the non-deterministic rates of execution of the individual tasks; however, these different sequences yield identical results since the each task performs a deterministic set of operations. Both of the sequences are represented by the same LTL formula:

(1,2,3,5) — in state 1 (the t_0 state), $at A_9 \wedge at B_7$ is true, therefore, in the sequence starting from state 2 (the next state), $after A_9$ must be true eventually and

after B₇ must be true eventually; these predicates are satisfied in states 2 and 5, respectively;

(1,2,4,5) — in state 1 (the t_0 state), *at A₀ $\wedge$ at B₇* is true, therefore, in the sequence starting from state 2 (the next state), *after A₀* must be true eventually and *after B₇* must be true eventually; these predicates are satisfied in states 2 and 4, respectively.

Thus, we have expressed with one LTL formula two computationally equivalent state sequences. Pre-processing of state machines can be performed in order to eliminate such equivalent state sequences, but, through the use of LTL, we have achieved the same objective with no cost.

4.2 Language Development

In this section, we present the development of the CAEDE Operational Language from its LTL roots. The language is constructed in two steps: (1) a preliminary definition of the language assuming a deterministic model of rendezvous and (2) the extension of the preliminary language for a non-deterministic model of rendezvous. In both steps, we assume that there are no sources of non-deterministic behaviour other than rendezvous partners and the relative rates of task execution.

In the first step, we build the language from LTL predicates and formulæ that reflect the nature of the Ada language. To this LTL formalism we attach operational semantics. In the definition of this preliminary language we assume that there are only single accepts (no selective accepts) and that there is only one caller for any accept; thus, the partners of all rendezvous are deterministic. We describe any program created under this assumption as *task deterministic* program; that is, each task perceives the system to be deterministic, at least in as much as every task experiences a *total ordering* of its rendezvous. There are two corollaries of this statement:

1. the program can guarantee only a *partial ordering* of rendezvous;

Figure 4.6: A Task Deterministic Ada Program

2. the different total orderings of rendezvous in the program do not change total orderings of rendezvous perceived by the individual tasks.

These ideas are illustrated in Figures 4.6, 4.7 and 4.8. In Figure 4.6, we show a design of four tasks in which there is only one caller and one acceptor for each rendezvous. The total orderings of rendezvous from the task's perspective are shown by timing diagrams in Figure 4.7. Finally, in Figure 4.8, we show two different total orderings of rendezvous from the perspective of the program.

So long as the program, is task deterministic, we may treat separately the actions of individual tasks, with the exception of rendezvous. Furthermore, if we choose any specific total ordering of rendezvous within the program, (since they are all functionally equivalent), and reason with this ordering only, then we can regard the program to be deterministic. Thus, LTL-based reasoning is valid for the task deterministic programs with which we are concerned.

In the second step of language development, we extend the preliminary language and basic operational semantics to include non-deterministic rendezvous structures: multiple callers on an entry and multiple open entries in a selective accept. We accommodate such structures by converting them to equivalent enumerated sets of deterministic rendezvous; that is, we create a set of task deterministic programs from one non-deterministic program. This procedure is analogous to the predicate logic to propositional logic conversion described in § 4.1.1.

We do not attempt to include all of the Ada language features in order to

Figure 4.7: Task Perspective Total Ordering of Rendezvous

Note: time scales are not necessarily linear

Figure 4.8: Total Orderings of Rendezvous in Program

simplify our ability to develop proofs. In particular, we do not support: dynamic task creation and destruction, timed or conditional task calls, delay or else selective accept alternatives, exceptions and dynamic data creation (which eliminates procedural recursion). We justify these choices below.

- Dynamic task creation and destruction violates our assumption that tasks cycle forever.
- Timed or conditional task calls and else or delay alternatives in selective accepts impeded our ability to prove that rendezvous must occur (since it is possible with these structures that rendezvous do not occur).
- Exceptions complicate our ability to prove that rendezvous take place because they represent computations that interrupt the expected behaviour of the system.
- Dynamic data creation complicates assertions of the data state of a task.

The extension of our language to accommodate these features is left as an open research topic.

4.2.1 Program State Predicates

Below, we define some basic LTL predicates describing Ada programs that are used in language statements (formulae). Examples of predicate use are drawn from the two task example of Figure 4.3.

$tcall(t, x, y)$ — The $tcall$ predicate is true if a caller task t is making a call to some acceptor task. The call and the accept locations in the program are uniquely identified by x and y , respectively. This predicate represents the notion that task t is at a task call. For example, $tcall$ would be used to define $at A_0$ as follows: $tcall(A, A_0, B_{B_entry})$. Note: we have used the X_n notation to specify the task caller and a X_{entry} notation for the acceptor (which assumes that the entry occurs in one location only).

$acc(t, y)$ — The *acc* predicate is true if an acceptor task t is accepting a call from some caller task. The accept locations in the program is uniquely identified y . This predicate represents the fact that task t is at a lone (not selective) accept statement. For example, *acc* would be used to define at B_7 as follows:
 $acc(B, B_{B_entry})$.

$end_rendez(t, y)$ — The *end_rendez* predicate is true if an acceptor task t has reached the end of its accept structure (i.e. the end of the critical section) and the caller of the most recent rendezvous has not been released (as yet). The accept location in the program is uniquely identified by y . For example, *end_rendez* would be used to define at B_8 as follows: $end_rendez(B, B_{B_entry})$.

$end_accept(t, y)$ — The *end_accept* predicate is true if an acceptor task t has reached the end of its accept structure and the caller of the most recent rendezvous has been released. The accept location in the program is uniquely identified by y . For example, *end_accept* would be used to define after B_8 as follows: $end_accept(B, B_{B_entry})$.

$return_task_call(t, x, y)$ — The *return_task_call* predicate is true if a caller task t has returned from a call with some acceptor task. The call and the accept locations in the program are uniquely identified by x and y , respectively. For example, *return_task_call* would be used to define after A_9 as follows:
 $return_task_call(A, A_9, B_{B_entry})$.

$proc_call(t, c, p)$ — The *proc_call* predicate is true if a caller task t is executing a procedure call on procedure p from program location c .

$return_proc_call(t, c, p)$ — The *proc_call* predicate is true if a caller task t has returned to program location c from a procedure call on procedure p .

$start_proc(t, p)$ — The *start_proc* predicate is true if a caller task t is beginning execution of procedure p .

$end_proc(t, p)$ — The *end_proc* predicate is true if a caller task t is ready to return from execution of procedure p .

$ts(t, \pi(v), q)$ — The ts predicate is true if v , the array of values of all variables for task t satisfies some predicate $\pi(v)$ and the queue of accepted rendezvous or procedure call return points for task t is q (to be known as the *caller queue*). The elements of the caller queue are the rendezvous or procedure return point predicates described below.

$rdz(t, x, y, \pi(v), q)$ — The rdz predicate is true if some acceptor task is engaged in a rendezvous with caller task t originating with a task call at location x in the caller task, an accept at location y in the acceptor task, the array of values of all variables for task t , v satisfies some predicate $\pi(v)$ and the caller queue for task t is q . For example, after B_7 , the accept statement for entry B_{entry} , the following predicate would be true: $rdz(A, A_9, B_{entry}, \pi(\{I, V\}), [])$ where $\{\dots\}$ represents an array and $[]$ represents an empty queue.

$prc(c)$ — The prc predicate is true if c is the return point of an active procedure call.

These predicates define the elementary predicates of our language; however, as we saw in the absence of deadlock proof for Figure 4.3, even simple predicates can result in complex formulae. As a means of capturing the notions of rendezvous, we define several statements that create *compound predicates*; i.e. predicates constructed from the elementary predicates described above.

$$\begin{aligned} & \models (tcall(t_1, x, y) \cup [tcall(t_1, x, y) \wedge acc(t_2, y)] \vee \square tcall(t_1, x, y)) \\ & \Leftrightarrow task_call(t_1, x, y) \end{aligned} \quad (4.11)$$

Equation 4.11 describes the behaviour of a caller task when it executes a task call: it either stays in the call until it is accepted or it stays in the call forever.

$$\begin{aligned} & \models (acc(t_1, y) \cup [tcall(t_2, x, y) \wedge acc(t_1, y)] \vee \square acc(t_1, y)) \\ & \Leftrightarrow start_accept(t_1, y) \end{aligned} \quad (4.12)$$

Equation 4.12 describes the behaviour of an acceptor task when it executes an accept; it either stays in the accept statement until a caller arrives or it stays in the accept statement forever.

$$\models tcall(t_1, x, y) \wedge acc(t_2, y) \Leftrightarrow start_rendez(t_2, x, y) \quad (4.13)$$

Equation 4.13 defines the start of a rendezvous as the eventual conjunction of *tcall* and *accept* for some t_1, t_2, x and y . From our example design, *at A₀* $\wedge$ *at B₇* would be represented by *start_rendez(B, A₀, B₈, ...)*.

Below, several more statements express characteristics of both elementary and compound predicates that are of particular importance for later proofs.

$$\vdash \models \text{start_rendez}(t, x, y) \Rightarrow \Diamond \Diamond \neg \text{start_rendez}(t, x, y) \quad (4.14)$$

Equation 4.14 declares that a rendezvous must progress past its start; that is, the acceptor task performs some action. For instance, in the two task example, this property implies the formula $\models \text{at } A_0 \wedge \text{at } B_7 \Rightarrow \Diamond \Diamond \text{after } A_3$.

$$\models ts(t, \pi(v), q) \Rightarrow \bigwedge_{\text{all } p \text{ in } q} p \quad (4.15)$$

In equation 4.15, we state that if the *ts* predicate is true, then all of the *rdz* or *prc* predicates in the caller queue q are also true. This reiterates the notion that state of a caller task is suspended while the call is processed by the acceptor.

A corollary of equation 4.15 defines the *call_accepted* predicate:

$$\models (ts(t_1, \pi_1(v_1), q_1) \wedge rdz(t_2, x, y, \pi_2(v_2), q_2) \wedge q_1) \Rightarrow \text{call_accepted}(t_2, x, y) \wedge ts(t_2, \pi_2(v_2), q_2) \quad (4.16)$$

If the caller task has been accepted, then its state is frozen on the caller queue of the acceptor task; thus, we may regard the caller task to be in a state of "call accepted". Although this state does not exist in Ada explicitly, the *call_accepted* predicate will serve as a notational convenience later.

The *tcall* and *acc* predicates imply that there exists some fixed *ts* for both the caller and acceptor. This follows from the nature of the Ada rendezvous mechanism; that is, while the *tcall* and *acc* predicates are true, the caller and acceptor tasks are suspended, thus the respective task states cannot change. These notions are described below.

$$\begin{aligned} \models \text{tcall}(t, x, y) &\Rightarrow ts(t, \pi(v), q) \\ \models \text{acc}(t, y) &\Rightarrow ts(t, \pi(v), q) \end{aligned} \quad (4.17)$$

An extension of this idea to the *task.call* and *start.accept* predicates yields:

$$\begin{aligned} &\equiv \text{task.call}(t, x, y) \Rightarrow ts(t, \pi(v), q) \\ &\equiv \text{start.accept}(t, y) \Rightarrow ts(t, \pi(v), q) \end{aligned} \quad (4.18)$$

The extension for *task.call* can be reasoned informally as follows:

- if *task.call* is true; then either $tcall \vee (tcall \wedge acc)$ is true, or $\neg tcall$ is true (ignoring predicate variables);
- if $\neg tcall$ is true then it follows that *ts* is true by equation 4.17;
- if $tcall \vee (tcall \wedge acc)$ is true, then either *tcall* is true or *tcall* $\wedge$ *acc* is true (by definition); both of these imply *ts* by equation 4.17.

A similar reasoning can be made for the *start.accept* extension.

4.2.2 Describing the Ada Rendezvous

A description of the Ada rendezvous must capture two notions: (1) if a single caller and a single acceptor both wait for a rendezvous then the rendezvous must begin eventually and (2) a caller task is suspended until it is released by the acceptor task at the end of the rendezvous. These properties are expressed by the two statements below.

Rendezvous Entry —

$$\begin{aligned} &\left(\begin{array}{l} \Diamond(\text{task.call}(t_1, x, y) \wedge ts(t_1, \pi_1(v_1), q_1)) \wedge \\ \Diamond(\text{start.accept}(t_2, y) \wedge ts(t_2, \pi_2(v_2), q_2)) \end{array} \right) \Rightarrow \\ \equiv &\Diamond \left(\begin{array}{l} \text{start.rendex}(t_2, x, y) \wedge \\ ts(t_2, \pi_2(v_2), [rdx(t_1, x, y, \pi_1(v_1), q_1) : q_2]) \end{array} \right) \end{aligned} \quad (4.19)$$

Rendezvous Exit —

$$\begin{aligned} &\left(\begin{array}{l} \text{end.rendex}(t_2, y) \wedge \\ ts(t_2, \pi_2(v_2), [rdx(t_1, x, y, \pi_1(v_1), q_1) : q_2]) \end{array} \right) \Rightarrow \\ \equiv &\left[\begin{array}{l} \Diamond(\text{end.accept}(t_2, y) \wedge ts(t_2, \pi_2(v_2), q_2)) \\ \wedge \Diamond \left(\begin{array}{l} \text{return.task.call}(t_1, x, y) \wedge \\ ts(t_1, \pi_1(v_1), q_1) \end{array} \right) \end{array} \right] \end{aligned} \quad (4.20)$$

In equation 4.19 we specify that a rendezvous will start in the next state, eventually if the conjunction of an eventual *task_call* and an eventual *start_accept* occurs and the state of the accepted caller is frozen in rendezvous. We define the start of the rendezvous to occur at least in the next program state following the conjunction of the call and the accept in order to freeze the state of the accepted caller task on the call queue of the acceptor task; although this may not be true in practice, it does describe (without any erroneous consequences) the property that the caller is suspended while the rendezvous is active. The compound predicates, *task_call* and *start_accept* define how callers and acceptors wait for rendezvous to occur. We have introduced in this equation the list notation $\{x|y\}$, which we interpret as a list with head x and tail y .

Equation 4.20 specifies the release of the caller task when the end of the rendezvous is reached. When the caller is released, the state of the caller task is removed from the call queue, thus, the caller is no longer suspended.

Below we justify that the rendezvous entry and exit formulae (equations 4.19 and 4.20) are consistent with our program state predicate definitions of § 4.2.1. Equation 4.19 is demonstrated by an LTL proof and equation 4.20 is argued informally from the basic definitions of the predicates.

The proof of equation 4.19 requires the prior proof of the equation shown below.

$$\models \neg(\Diamond \Box tcall(t_1, x, y) \wedge \Diamond \Box acc(t_2, y)) \quad (4.21)$$

This equation states that a single caller and a single acceptor cannot forever await the start of a rendezvous. This is implied by the rendezvous definition in the Ada language reference manual, however, we show that it follows also from our predicate definitions. For proof convenience, we assign the program state predicates to symbols for these proofs, as follows:

$tcall(t_1, x, y)$ to P	$task_call(t_1, x, y)$ to S
$acc(t_2, y)$ to R	$start_accept(t_2, y)$ to T
$start_rendez(t_2, x, y)$ to Q	

Below we prove equation 4.21.

- (1) $\models Q \Rightarrow \Diamond \neg Q$ (4.14)
- (2) $\models \Box Q \Rightarrow \Box \Diamond \neg Q$ (1, B.17, B.26)
- (3) $\models \Box Q \Rightarrow \Diamond \Box \neg Q$ (2, B.19)
- (4) $\models \Box Q \Rightarrow \Box \Box Q$ (B.14, B.17)
- (5) $\models \Box Q \Rightarrow \Box \Box \Box Q$ (4, B.17)
- (6) $\models \Box Q \Rightarrow \Box \Diamond \neg Q \wedge \Box Q \Rightarrow \Box \Box \Box Q$ (3, 5, B.40, B.21)
- (7) $\models \Box Q \Rightarrow \Box (\Diamond \neg Q \wedge \Box Q)$ (6, B.8, B.21, B.23)
- (8) $\models \Box Q \Rightarrow \Box \neg \Box Q \wedge \Box Q$ (7, B.12)
- (9) $\models \neg \Box Q$ (8, see note below)
- (10) $\models \neg \Diamond \Box Q$ (9, B.17, B.11)
- (11) $\models \neg \Diamond \Box (P \wedge R)$ (10, 4.13, B.38)

Q.E.D.

Note on Step (9): since the left-hand side of the implication in step (8) is always false ($\neg A \wedge A$ is false by the definition of the $\wedge$ operator), then $\Box Q$ is also false by definition of the $\Rightarrow$ operator.

Below we prove that equation 4.19 is consistent with the program predicate

definitions.

- (1) $\models \Diamond \Box P \wedge \Diamond (R U [P \wedge R]) \Rightarrow \Diamond (P \wedge R)$ (B.3, B.29, B.15, B.33)
- (2) $\models \Diamond \Box R \wedge \Diamond (P U [P \wedge R]) \Rightarrow \Diamond (P \wedge R)$ (B.3, B.29, B.15, B.33)
- (3) $\models \Diamond (P U [P \wedge R]) \wedge \Diamond (R U [P \wedge R]) \Rightarrow \Diamond (P \wedge R)$ (B.3, B.29, B.15, B.33)
- (4) $\models \neg(\Diamond \Box P \wedge \Diamond \Box R)$ (4.21)
- (5) $\models \neg(\Diamond \Box P \wedge \Diamond \Box R) \Rightarrow \neg(\Diamond \Box P \wedge \Diamond \Box R) \vee \Diamond (P \wedge R)$ (B.5, B.29)
- (6) $\models (\Diamond \Box P \wedge \Diamond \Box R) \Rightarrow \Diamond (P \wedge R)$ (4.5, B.36, B.6)
- (7) $\models \left[\begin{array}{l} (\Diamond (P U [P \wedge R]) \wedge \Diamond \Box R) \vee \\ (\Diamond (P U [P \wedge R]) \wedge \Diamond (R U [P \wedge R])) \vee \\ (\Diamond \Box P \wedge \Diamond (R U [P \wedge R])) \vee \\ (\Diamond \Box P \wedge \Diamond \Box R) \end{array} \right] \Rightarrow \Diamond (P \wedge R)$ (1, 2, 3, 6, B.40, B.10)
- (8) $\models \left(\begin{array}{l} \Diamond (P U [P \wedge R] \vee \Box P) \wedge \\ \Diamond (R U [P \wedge R] \vee \Box R) \end{array} \right) \Rightarrow \Diamond (P \wedge R)$ (7, B.4, B.22)
- (9) $\models (\Diamond S \wedge \Diamond T) \Rightarrow \Diamond (P \wedge R)$ (8, 4.11, 4.12)

Q.E.D.

Below, the proof of equation 4.20 is argued from the definitions of the program state predicates.

1. The $end_rendez(t_2, y)$ predicate is true if the acceptor task has reached the end of the accept structure and the caller has not been released; the unreleased caller is expressed by

$$ts(t_2, \pi_2(v_2), [rdz(t_1, x, y, v_1, q_1) \cdot q_2])$$

which shows that the caller state is still suspended in the acceptor's call queue.

2. If the acceptor is at the end of the accept structure, then in the next state, eventually the acceptor must have released the caller and be ready to execute the statement following the end of accept; this is denoted by the *end_accept* predicate and $ts(t, \pi(v), q)$ which define the acceptor to be after the end accept statement with the caller state no longer frozen on the caller queue.
3. The released caller is denoted by the *return_task_call* predicate, which is true when the caller is released from its rendezvous.

Q.E.D.

4.2.3 Procedure Calls in Ada

Procedures in Ada are program objects that contain passive code. This code becomes active when some active object, namely a task, directly or indirectly calls the procedure. (Note: we consider only procedures without nested tasks). There are two types of procedures:

- those that are shared between tasks;
- those that can be executed by one task only.

We limit our representation to shared procedures because the non-shared procedure is identical to a shared procedure that is called by a single task.

There is no task synchronization required for a procedure call, thus, the LTL statements describing the procedure call and return are relatively simple:

Procedure Call —

$$\begin{aligned} \models \text{proc_call}(t, c, p) \wedge ts(t, \pi(v), q) \Rightarrow \\ \bigcirc \Diamond (\text{start_proc}(t, p) \wedge ts(t, \pi(v), [prc(c)|q])) \end{aligned} \quad (4.22)$$

Return From Procedure Call —

$$\begin{aligned} \models \text{end_proc}(t, p) \wedge ts(t, \pi(v), [prc(c)|q]) \Rightarrow \\ \bigcirc \Diamond (\text{return_proc_call}(t, c, p) \wedge ts(t, \pi(v), q)) \end{aligned} \quad (4.23)$$

Equation 4.22 describes the start of a procedure call: if a procedure call is begun then in the next state, eventually the execution will begin at the start of the named procedure with the return point on the top of the caller queue. In equation 4.23, we express the return from a procedure call: if the end of a procedure is reached, then the caller returns to the statement after the procedure call, as specified by the caller queue. In both of these descriptions we have deliberately sacrificed the recursive nature of Ada procedure calls for the sake of simplicity.

4.2.4 Specifying Data and Functions

So far we have treated task data and operations on data (functions) as predicates on the array of task variables. Given an implication such as

$$\begin{aligned} & \text{pred}_1(t, \dots) \wedge ts(t, \pi_1(v), q) \Rightarrow \\ \equiv & \quad \bigcirc \Diamond (\text{pred}_2(t, \dots) \wedge ts(t, \pi_2(v), q)) \end{aligned}$$

we consider $\pi_1(v)$ to be a pre-condition predicate of the implication and $\pi_2(v)$ to be a post-condition predicate. The meaning of the pre-condition predicate is fairly intuitive: unless the task variables satisfy $\pi_1(v)$ then the premise of the implication is false. The interpretation of the post-condition is somewhat different: if $\pi_2(v)$ is a post condition then the values of the variables in v must be such that $\pi_2(v)$ is true.

While the pre- and post-condition predicates can precisely specify operations on data, they do not suggest a systematic or understandable treatment of variables and functions. In this section, we present a notation that eliminates these shortcomings.

An Alternate Variable Notation

The current pre- and post-condition predicates have a single argument: an array of task variables. A similar, but not totally equivalent representation would be a *conjunction of predicates* where the single argument of each predicate is the value of a variable; i.e. "an array of predicates rather than a predicate of an array". We

define this relationship below:

$$\pi(v) \Rightarrow \bigwedge_{i=1}^n \pi_i(v_i)$$

or, in an array-like notation,

$$\pi(v) \Rightarrow \{\pi_1(v_1), \pi_2(v_2), \dots\}$$

In order to make this notation equivalent we would need to support predicates with multiple arguments so that complex conditions could be stated; e.g. $\pi_n(v_1, v_2, v_k)$. We shall address this problem shortly.

Our new notation allows us to attach a convenient meaning to the $\pi_i(v_i)$ predicate. We interpret π_i to be the names of task variables. Thus, the predicate $\pi_i(v_i)$ is read as:

"the value of the variable π_i is v_i ."

To further illustrate this notion, consider the following examples¹ and their interpretations:

$\text{var}_1(7)$ — the value of var_1 is the integer 7;

$\text{var}_2(\text{YES})$ — the value of var_2 is string YES;

$\text{var}_3(x)$ — the value of var_3 is some (arbitrary) x ;

The meaning of $\pi_i(v_i)$ predicates varies slightly with the context of use; i.e. within a pre-condition or within a post-condition. We interpret $\pi_i(v_i)$, if used in a pre-condition, as:

"if the value of π_i is v_i ,"

and if used in a post-condition, as:

"the value of π_i must be v_i ,"

¹ Although the Ada programming convention suggests that variables begin with a capital letter, we shall continue to use our convention of lower case letters for predicate names.

Returning to our earlier problem of predicates with multiple arguments, we denote such "extra" predicates by pre-pending the "@" symbol to the predicate name; e.g. $@text(v_1, v_2)$. These supplementary predicates can be added to the array of variable predicates. For example, as a pre-condition, we interpret:

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

as

"if the value of var_1 is some x and the value of var_2 is some y and x is greater than y ".

The use of supplementary predicates in post-conditions allows us to specify variable mapping functions by representing them in a relational form; i.e. we can express $z \leftarrow f(x)$ by the relation $r(x, z)$. For example, as a post-condition, we interpret:

$$\{var_1(x), var_2(y), var_3(z), @plus(x, y, z)\}$$

as

"the value of var_1 must be some x and the value of var_2 must be some y and the value of var_3 must be some z such that x plus y is z ".

We leave the definition of supplementary predicates to § 4.3 under the discussion of language implementation.

As a means of supporting parameter passing during task calls, we require that each task contain at least the following predicate on a single well-known variable: $prm([])$. This predicate represents a parameter list. As will be indicated later in § 4.2.5, the parameter list of the caller task becomes the parameter list of the acceptor task when the rendezvous is begun, and vice-versa when the rendezvous is terminated.

An Example Statement

To illustrate the power of our new variable notation, we specify below the LTL formulae that describe the Ada code fragment shown in Figure 4.9.

Statement —

$$\begin{aligned} & \left(\text{return_task_call}(A, A_s, B_E) \wedge \right. \\ & \left. ts(A, \{v1(x), v2(y), v3(z), \bullet_{\text{gtreq}}(y, z)\}, q) \right) \Rightarrow \dots \\ \equiv & \quad \circ \diamond \left(\text{proc_call}(A, A_{11}, PROC1) \wedge \right. \\ & \left. ts(A, \{v1(x), v2(a), v3(z), \bullet_{\text{plus}}(y, z, a)\}, q) \right) \end{aligned} \quad (4.24)$$

Interpretation — If task A has returned from task call A_s on entry $B.E$ and the value of $v1$ is some x , the value of $v2$ is some y , the value of $v3$ is some z and y is *greater than or equal to* z , then in the next state, eventually, task A will execute a procedure call from A_{11} on procedure PROC1 with the value of $v1$ equal to x , the value of $v3$ equal to z and the value of $v2$ equal to a where a is y plus z .

Statement —

$$\begin{aligned} & \left(\text{return_task_call}(A, A_s, B_E) \wedge \right. \\ & \left. ts(A, \{v1(x), v2(y), v3(z), \bullet_{\text{less}}(y, z)\}, q) \right) \Rightarrow \dots \\ \equiv & \quad \circ \diamond \left(\text{proc_call}(A, A_{11}, PROC1) \wedge \right. \\ & \left. ts(A, \{v1(x), v2(a), v3(z), \bullet_{\text{minus}}(y, 1, a)\}, q) \right) \end{aligned} \quad (4.25)$$

Interpretation — If task A has returned from task call A_s on entry $B.E$ and the value of $v1$ is some x , the value of $v2$ is some y , the value of $v3$ is some z and y is *less than* z , then in the next state, eventually, task A will execute a procedure call from A_{11} on procedure PROC1 with the value of $v1$ equal to x , the value of $v3$ equal to z and the value of $v2$ equal to a where a is y minus 1.

The predicate $v1(x)$ in the example above appears in both the pre- and post-conditions and it is not used in any supplementary predicates. We conclude that it describes a "don't care" value within the context of these statements; i.e. it specified the value of a variable that is neither altered or used in a statement. We shall exploit this conclusion in § 4.3 in order to create a more compact array notation for pre- and post-condition predicates.

In the following sections, we use both the $\pi(v)$ and $\{\dots\}$ notations to represent pre- and post-condition predicates, as a matter of convenience; however, in the final implementation of the language, we will use only the $\{\dots\}$ notation.

(Task A)

```

5      B.E(X : out XTYPE);
6      if V2 >= V3 then
7          V2 := V2 + V3;
8      else
9          V2 := V2 - 1;
10     end if;
11     PROC1(V1,V2);

```

Figure 4.9: Ada Code Fragment Illustrating Variable Notation

4.2.5 Language Statements: LTL Formulae

The COL statements are primarily LTL formulae that express the execution behaviour of Ada programs. These formulae are classified as one of two types: (1) Ada language implications and (2) connective implications. The Ada language implications, presented in Figure 4.10, are formulae that describe the properties of the Ada rendezvous and procedure call; they are derived from equations 4.16, 4.19, 4.20, 4.22, 4.23. Equation 4.16 allows us to introduce the redundant *call accepted* predicate into the implication statements. In this manner, there is a predicate specifying directly the control state of each task, at all times.

Connective implications formulae "connect" program state predicates by precedence implications. They have the general form:

$$\models \text{predicate}_1(\dots) \wedge ts(t, \pi_1(v), q) \Rightarrow \bigcirc \Diamond (\text{predicate}_2(\dots) \wedge ts(t, \pi_2(v), q)) \quad (4.26)$$

which is interpreted as follows:

"for all time, for some task t , if some predicate_1 is true and the ts for t is such that $\pi_1(v)$ is valid and the caller queue is q then in the next state, eventually predicate_2 will be true and the ts for t will be such that $\pi_2(v)$ is valid and the caller queue is unchanged".

For example, the connective implications of the two task design in Figure 4.3 are shown in Figure 4.11.

The complete program behaviour is specified by the connective implications, the Ada language implications and an assertion of the initial program state, the *initial PSA*. The initial PSA is a statement that all tasks eventually start at the top of their endless loop structure with some given initial values for task variables; that is,

$$\bigwedge_{\text{all } t_i} \Diamond (\text{init}(t_i) \wedge \text{ts}(t_i, \pi_0(v_i), []))$$

where $\pi_0(v_i)$ is a predicate defining the initial values of all variables in v_i . This assertion is valid only for the sequence of program states starting from some arbitrary t_0 ; i.e.

$$t_0 \models \bigwedge_{\text{all } t_i} \Diamond (\text{init}(t_i) \wedge \text{ts}(t_i, \pi_0(v_i), [])) \quad (4.27)$$

The sequence of states starting from t_0 would be an instance of the sequences valid under $\models$. With these LTL formulae and several LTL inference rules, we define the operational semantics of our preliminary version of COL, next:

4.2.6 Basic Operational Semantics

The axiomatic properties of LTL statements like those of § 4.2.5 were illustrated by the two task example of Figure 4.3. The same design example also suggested an operational interpretation of the axiomatic proof. The formulae of the example contributed to only part of the analysis process. The remainder was provided by the inference rules and proof procedure that were employed; that is, the *operational semantics*.

In this section, we attach operational semantics to the formulae of § 4.2.5 by specifying a set of inference rules and an inference engine that are suitable for both axiomatic and operational analysis. The operational semantics describe how the implication formulae are applied to the initial program state assertion to derive axiomatic or operational information; in this case, the inference process defines the operational semantics of the formulae.

Our approach to presenting the operational semantics will involve two steps: (1) a discussion of the inference process and (2) an in-depth example indicating the axiomatic and operational analysis powers of the prescribed inference process.

$$\begin{aligned}
& \left\{ \begin{array}{l} \Diamond(\text{task_call}(t_1, x, y) \wedge \text{ts}(t_1, \{\text{prm}(p), \dots\}, q_1)) \\ \Diamond(\text{start_accept}(t_2, y) \wedge \text{ts}(t_2, \{\text{prm}(q), \dots\}, q_2)) \end{array} \right\} \\
(\text{ren}) & \equiv \left\{ \begin{array}{l} \bigcirc \Diamond \left(\begin{array}{l} \text{start_rendez}(t_2, x, y) \wedge \\ \text{ts}(t_2, \{\text{prm}(p), \dots\}, \text{rdz}(t_1, x, y, \{\text{prm}(p), \dots\}, q_1), q_2) \end{array} \right) \\ \wedge \bigcirc \Diamond(\text{call_accepted}(t_1, x, y) \wedge \text{ts}(t_1, \{\text{prm}(p), \dots\}, q_1)) \end{array} \right\} \\
(\text{rex}) & \equiv \left\{ \begin{array}{l} \Diamond \left(\begin{array}{l} \text{end_rendez}(t_2, y) \wedge \\ \text{ts}(t_2, \{\text{prm}(q), \dots\}, \text{rdz}(t_1, x, y, \{\text{prm}(p), \dots\}, q_1), q_2) \end{array} \right) \\ \wedge \Diamond(\text{call_accepted}(t_1, x, y) \wedge \text{ts}(t_1, \{\text{prm}(p), \dots\}, q_1)) \end{array} \right\} \\
& \Rightarrow \left\{ \begin{array}{l} \bigcirc \Diamond(\text{end_accept}(t_2, y) \wedge \text{ts}(t_2, \{\text{prm}(q), \dots\}, q_2)) \\ \bigcirc \Diamond(\text{return_task_call}(t_1, x, y) \wedge \text{ts}(t_1, \{\text{prm}(q), \dots\}, q_1)) \end{array} \right\} \\
(\text{pen}) & \equiv \text{proc_call}(t, c, p) \wedge \text{ts}(t, \{\dots\}, q) \Rightarrow \\
& \bigcirc \Diamond(\text{start_proc}(t, p) \wedge \text{ts}(t, \{\dots\}, [\text{prc}(c), q])) \\
(\text{pex}) & \equiv \text{end_proc}(t, p) \wedge \text{ts}(t, \{\dots\}, [\text{prc}(c), q]) \Rightarrow \\
& \bigcirc \Diamond(\text{return_proc_call}(t, c, p) \wedge \text{ts}(t, \{\dots\}, q))
\end{aligned}$$

Figure 4.10: Ada Language Implications

Connective Implications for Task "A"

- (c1) $\equiv \text{init}(A) \wedge \text{ts}(A, \{\dots\}), q \Rightarrow$
 $\quad \bigcirc \Diamond (\text{task_call}(A, A_0, B_{B_entry}) \wedge \text{ts}(A, \{\dots\}), q))$
- (c2) $\equiv \text{return_task_call}(A, A_0, B_{B_entry}) \wedge \text{ts}(A, \{\dots\}), q \Rightarrow$
 $\quad \bigcirc \Diamond (\text{start_accept}(A, A_{A_entry}) \wedge \text{ts}(A, \{\dots\}), q))$
- (c3) $\equiv \text{start_rendez}(A, c, A_{A_entry}) \wedge \text{ts}(A, \{\dots\}), q \Rightarrow$
 $\quad \bigcirc \Diamond (\text{end_rendez}(A, A_{A_entry}) \wedge \text{ts}(A, \{\dots\}), q))$
- (c4) $\equiv \text{end_accept}(A, A_{A_entry}) \wedge \text{ts}(A, \{\dots\}), q \Rightarrow$
 $\quad \bigcirc \Diamond (\text{init}(A) \wedge \text{ts}(A, \{\dots\}), q))$

Connective Implications for Task "B"

- (c5) $\equiv \text{init}(B) \wedge \text{ts}(B, \{\dots\}), q \Rightarrow$
 $\quad \bigcirc \Diamond (\text{start_accept}(B, B_{B_entry}) \wedge \text{ts}(B, \{\dots\}), q))$
- (c6) $\equiv \text{start_rendez}(B, c, B_{B_entry}) \wedge \text{ts}(B, \{\dots\}), q \Rightarrow$
 $\quad \bigcirc \Diamond (\text{end_rendez}(B, B_{B_entry}) \wedge \text{ts}(B, \{\dots\}), q))$
- (c7) $\equiv \text{end_accept}(B, B_{B_entry}) \wedge \text{ts}(B, \{\dots\}), q \Rightarrow$
 $\quad \bigcirc \Diamond (\text{task_call}(B, B_0, A_{A_entry}) \wedge \text{ts}(B, \{\dots\}), q))$
- (c8) $\equiv \text{return_task_call}(B, B_0, A_{A_entry}) \wedge \text{ts}(B, \{\dots\}), q \Rightarrow$
 $\quad \bigcirc \Diamond (\text{init}(B) \wedge \text{ts}(B, \{\dots\}), q))$

Note: the predicate $\{\dots\}$ is true if all variables in v
 are equal to any values

Figure 4.11: Connective Implications for Two Task Example

The Inference Process

The ability to provide both axiomatic and operational analysis using the LTL formulae describing a program can be attained by performing an axiomatic manipulation of the formulae that produces a reasonable operational interpretation. This validity of this approach is found in the design of the Prolog language. Statements in Prolog represent predicate logic formulae, however, they have both an axiomatic (declarative) or operational (procedural) interpretation. This property results from the top-down inference engine used for executing Prolog programs.

The structure of our language system is shown in Figure 4.12. The elements of the language are: (1) the program description components and (2) the inference engine components. We indicate an larger structure called the *COL Executor* that incorporates the inference engine components and the Ada Design Implication formulae. The COL Executor represents the language elements that are common to all programs specified in COL.

The program description components, *Initial PSA*, *Connective Implications* and *Ada Language Implications*, have been described in detail in § 4.2.5, below, we outline briefly the inference engine components.

Program State Assertion — The Program State Assertion (PSA) is a conjunction of sub-formulae that describe the behaviour of a task with respect to an arbitrary starting program state to

Inference Rules — Inference Rules specify how logical formulae may be combined to deduce information. For instance, the implication rule from classic logic if $\models P$ and $\models P \Rightarrow Q$ then $\models Q$ describes how an implication formula can be used to deduce the conclusion assertion given a premise assertion.

Inference Engine — The Inference Engine applies inference rules to logical formulae in order to generate deductions; for example, an inference engine would apply the implication rule above to $\models A$ and $\models A \Rightarrow A \wedge B$ to deduce $\models A \wedge B$. The inference engine uses some systematic approach to matching rules, formulae and assertions to produce deductions.

2

1.0

1.1

1.25

1.4

1.6

1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0

2.8

3.2

3.6

4.0

2.5

2.2

2.0

1.8

METRO

Figure 4.12: COL Language System

We discuss these three components in more detail below.

The notion of a program state assertion was introduced in § 4.2.5 to specify the initial conditions of a program. In that case, as for all PSA, the assertion is true with reference to the sequence of states starting from some arbitrary t_0 . The general form of a PSA is:

$$t_0 \models \bigwedge_{\text{all tasks}} \text{predicate}_i(\dots) \wedge ts(i, \dots) \quad (4.28)$$

where each task description may be prefaced by temporal operator (such as $\bigcirc \Diamond$). We combine PSA with LTL implication formulæ and inference rules to generate new PSA. Since each PSA describes the behaviour of each task in the program (relative to t_0) we can interpret a PSA as a "snapshot" of program behaviour. Therefore, a sequence of successive deductions of PSA can be interpreted as the operational behaviour of the specified program. This will be demonstrated with an example later.

The inference rules that we use to combine PSA and implication formulæ to deduce new PSA are the following:

$$\text{if } \models P \wedge \Diamond Q \text{ and } \models Q \Rightarrow \bigcirc \Diamond R \text{ then } \models P \wedge \bigcirc \Diamond R \quad (4.29)$$

$$\text{if } \models P \wedge \bigcirc \Diamond Q \text{ and } \models Q \Rightarrow \bigcirc \Diamond R \text{ then } \models P \wedge \bigcirc \Diamond R \quad (4.30)$$

$$\text{if } \models P \wedge \bigcirc Q \text{ and } \models Q \Rightarrow \bigcirc \Diamond R \text{ then } \models P \wedge \bigcirc \Diamond S \quad (4.31)$$

The proofs of these deduction rules are provided in Appendix B.

We use pattern matching by *unification* [28,55,94] to match LTL formulæ during inference; for example, in inference rule 4.29 unification matches the Q in:

$$\models P \wedge \Diamond Q$$

with the Q in:

$$\models Q \Rightarrow \bigcirc \Diamond R$$

Unification matches two formulæ on a predicate by predicate, argument by argument basis, given that logical connectives match. The basic rules of unification are summarized below:

- two predicates match if they have the same name, same number of arguments and corresponding arguments match;
- two arguments match if they are identical symbols;
- two arguments match if one is a symbol and the other is a predicate variable; in this case, the variable is *instantiated* to the symbol in every occurrence of the variable within the formula;
- two arguments match if they are both predicate variables; in this case, if either of the variables becomes instantiated (perhaps by unifying another predicate in the formula) then all occurrences of both variables become instantiated.

We illustrate the unification process with the following example. Suppose we have an element from a PSA:

$$\Diamond(\text{start_rendez}(A, B_1, A_E) \wedge \text{ts}(A, \{\text{prm}(p), g(T)\}, []))$$

and the implication premise:

$$\Diamond(\text{start_rendez}(A, x, A_E) \wedge \text{ts}(A, \{\text{prm}(p), g(y)\}, [])).$$

The unification of these two predicates succeeds with the variable x instantiated to B_1 and the variable y instantiated to T . (Note: the data predicates in the data predicate array are unified on a predicate by predicate basis.) If we had attempted to unify the same PSA element with:

$$\Diamond(\text{start_rendez}(A, x, A_E) \wedge \text{ts}(A, \{\text{prm}(p), g(F)\}, []))$$

then the unification would have failed since the symbol F does not match identically with the symbol T . We illustrate the use of inference rules and unification with another example.

If we assume that the initial PSA for the two task design of Figure 4.3 is:

$$t_0 \equiv \left\{ \begin{array}{l} \Diamond(\text{init}(A) \wedge \text{ts}(A, \{\text{prm}([])\}, [])) \wedge \\ \Diamond(\text{init}(B) \wedge \text{ts}(B, \{\text{prm}([])\}, [])) \end{array} \right\} \quad (4.32)$$

and the connective implication statements of Figure 4.11 describe the behaviour of the program, then we can infer:

$$t_0 \models \left\{ \begin{array}{l} \bigcirc \Diamond (task_call(A, A_0, B_{B_entry}) \wedge ts(A, \{prm(\{\})\}, \{\})) \wedge \\ \Diamond (init(B) \wedge ts(B, \{prm(\{\})\}, \{\})) \end{array} \right\} \quad (4.33)$$

by applying inference rule 4.29 to the initial PSA (eqn. 4.32) and statement (c1) from Figure 4.11. An operational interpretation of this assertion is:

“task A is calling entry B.B_entry and task B is at the top of its control loop”.

The application of inference rule 4.30 to statement (c5) and the above PSA yields:

$$t_0 \models \left\{ \begin{array}{l} \bigcirc \Diamond (task_call(A, A_0, B_{B_entry}) \wedge ts(A, \{prm(\{\})\}, \{\})) \wedge \\ \bigcirc \Diamond (start_accept(B, B_{B_entry}) \wedge ts(B, \{prm(\{\})\}, \{\})) \end{array} \right\} \quad (4.34)$$

which we interpret operationally as:

“task A is calling entry B.B_entry and task B is accepting entry B_entry”.

The axiomatic reasoning we have just applied can be interpreted from an operational perspective:

- connective implications (c1) and (c5) describe the progress of tasks A and B respectively;
- the order of use of the implications suggests relative execution speeds (e.g. task A is faster than task B).

Regarding the second point, we can assume any relative rates of execution because the program is task deterministic, thus, different relative orderings of events are equivalent (cref. § 4.2).

The inference engine chooses inference rules and implication formulæ that can be applied to the current PSA; we designate the selection algorithm by the symbol ϕ . If the set of implication formulæ is large this can be a costly process. Fortunately, only a few implication formulæ are acceptable for a given PSA, as we show below.

Task-deterministic programs imply that for every task in the program, if the task is at a particular program step, then the next program step is deterministic. From the axiomatic perspective, this means that only one implication formula can be valid for any given task inferred to be at some program step (as defined by the control state predicate). If two implication formulæ were valid then a non-deterministic situation would exist.

Given a set of acceptable implication formulæ the inference engine must select one for use in an inference. A ϕ that progresses all tasks fairly from an operational perspective is desirable. For now, we do not specify a particular ϕ ; instead, special purpose ϕ will be introduced in later chapters as they are required.

An Example

We illustrate the operational semantics of COL by means of the two task example from Figure 4.3. The operational semantics are defined by the inference rules defined in equations 4.29, 4.30, 4.31 and by an inference engine that will select implication formulæ for inference in such a manner that, from an operational perspective, the tasks in the program will progress at similar rates. The program description is composed of the connective implication formulæ in Figure 4.11, the Ada Language Implications in Figure 4.10 and the initial PSA defined in equation 4.32.

Below, we present several steps of axiomatic inference followed by an operational interpretation of the deduced PSA. Note: some inferences are combined for brevity and we have left out the *prm*([]) predicates from the assertions on the data state.

Axiomatic Deductions:

$$(1) \quad t_0 \equiv \begin{array}{l} \Diamond(\text{init}(A) \wedge ts(A, \{\}, [])) \wedge \\ \Diamond(\text{init}(B) \wedge ts(B, \{\}, [])) \end{array} \quad (4.32)$$

$$(2) \quad t_0 \equiv \begin{array}{l} \Diamond \Diamond (\text{task_call}(A, A_0, B_{B_entry}) \wedge ts(A, \{\}, [])) \wedge \\ \Diamond \Diamond (\text{start_accept}(B, B_{B_entry}) \wedge ts(B, \{\}, [])) \end{array} \quad \left(\begin{array}{l} 1, c1, 4.29, \\ c5, 4.30 \end{array} \right)$$

$$(3) \quad t_0 \equiv \begin{array}{l} \Diamond \Diamond (\text{call_accepted}(A, A_0, B_{B_entry}) \wedge ts(A, \{\}, [])) \wedge \\ \Diamond \Diamond \left(\begin{array}{l} \text{start_rendez}(B, A_0, B_{B_entry}) \wedge \\ ts(B, \{\}, [\text{rdz}(A, A_0, B_{B_entry}, \{\}, [])]) \end{array} \right) \end{array} \quad (2, \text{ren}, 4.31)$$

$$(4) \quad t_0 \equiv \begin{array}{l} \Diamond \Diamond (\text{call_accepted}(A, A_0, B_{B_entry}) \wedge ts(A, \{\}, [])) \wedge \\ \Diamond \Diamond \left(\begin{array}{l} \text{end_rendez}(B, B_{B_entry}) \wedge \\ ts(B, \{\}, [\text{rdz}(A, A_0, B_{B_entry}, \{\}, [])]) \end{array} \right) \end{array} \quad (3, c6, 4.30)$$

$$(5) \quad t_0 \equiv \begin{array}{l} \Diamond \Diamond \left(\begin{array}{l} \text{return_task_call}(A, A_0, B_{B_entry}) \wedge \\ ts(A, \{\}, []) \end{array} \right) \wedge \\ \Diamond \Diamond (\text{end_accept}(B, B_{B_entry}) \wedge ts(B, \{\}, [])) \end{array} \quad (4, \text{rex}, 4.31)$$

$$(6) \quad t_0 \equiv \begin{array}{l} \Diamond \Diamond (\text{start_accept}(A, A_{A_entry}) \wedge ts(A, \{\}, [])) \wedge \\ \Diamond \Diamond (\text{task_call}(B, B_0, A_{A_entry}) \wedge ts(B, \{\}, [])) \end{array} \quad \left(\begin{array}{l} 5, c2, 4.30, \\ c7, 4.30 \end{array} \right)$$

$$(7) \quad t_0 \equiv \begin{array}{l} \Diamond \Diamond \left(\begin{array}{l} \text{start_rendez}(A, B_0, A_{A_entry}) \wedge \\ ts(A, \{\}, [\text{rdz}(B, B_0, A_{A_entry}, \{\}, [])]) \end{array} \right) \wedge \\ \Diamond \Diamond (\text{call_accepted}(B, B_0, A_{A_entry}) \wedge ts(B, \{\}, [])) \end{array} \quad (6, \text{ren}, 4.31)$$

$$(8) \quad t_0 \equiv \begin{array}{l} \Diamond \Diamond \left(\begin{array}{l} \text{end_rendez}(A, A_{A_entry}) \wedge \\ ts(A, \{\}, [\text{rdz}(B, B_0, A_{A_entry}, \{\}, [])]) \end{array} \right) \wedge \\ \Diamond \Diamond (\text{call_accepted}(B, B_0, A_{A_entry}) \wedge ts(B, \{\}, [])) \end{array} \quad (7, c3, 4.30)$$

$$(9) \quad t_0 \equiv \begin{array}{l} \Diamond \Diamond (\text{end_accept}(A, A_{A_entry}) \wedge ts(A, \{\}, [])) \wedge \\ \Diamond \Diamond \left(\begin{array}{l} \text{return_task_call}(B, B_0, A_{A_entry}) \wedge \\ ts(B, \{\}, []) \end{array} \right) \end{array} \quad (8, \text{rex}, 4.31)$$

$$(10) \quad t_0 \equiv \begin{array}{l} \Diamond \Diamond (\text{init}(A) \wedge ts(A, \{\}, [])) \wedge \\ \Diamond \Diamond (\text{init}(B) \wedge ts(B, \{\}, [])) \end{array} \quad \left(\begin{array}{l} 9, c4, 4.30, \\ c8, 4.30 \end{array} \right)$$

Operational Interpretation:

- (1) — tasks A and B are both at the top of their control loops;
- (2) — task A is calling entry B.B_entry and task B is accepting entry B_entry;
- (3) — task A's call on B.B_entry has been accepted (and it is suspended until the rendezvous is completed) and task B has begun its rendezvous with task A on entry B_entry;
- (4) — task A's call on B.B_entry has been accepted (and it is suspended until the rendezvous is completed) and task B is ending its rendezvous with task A on entry B_entry;
- (5) — task A has returned from its call on B.B_entry and task B has ended its accept of entry B_entry;
- (6) — task A is accepting entry A_entry and task B is calling entry A.A_entry;
- (7) — task A has begun its rendezvous with task B on entry A_entry and task B's call on A.A_entry has been accepted (and it is suspended until the rendezvous is completed);
- (8) — task A is ending its rendezvous with task B on entry A_entry and task B's call on A.A_entry has been accepted (and it is suspended until the rendezvous is completed);
- (9) — task A has ended its accept of entry A_entry and task B has returned from its call on A.A_entry;
- (10) — again, tasks A and B are both at the top of their control loops.

The operational interpretation of the sequence of LTL deductions provides one possible perception of the programs operational behaviour.

The axiomatic steps of the example besides rendering operational behaviour have also demonstrated that the program is deadlock-free as we illustrate below.

Proof of Absence of Deadlock:

We can derive from the deduced $\tilde{P}SA$ that:

$$t_0 \models \Diamond(\text{init}(A) \wedge ts(A, \{\}, [])) \wedge \Diamond(\text{init}(B) \wedge ts(A, \{\}, [])) \Rightarrow \quad (4.35)$$

$$\quad \quad \quad \bigcirc\{\Diamond(\text{init}(A) \wedge ts(A, \{\}, [])) \wedge \Diamond(\text{init}(B) \wedge ts(A, \{\}, []))\}$$

Since this will be true for any arbitrary starting state t_0 then we can state:

$$\models \square \left\{ \begin{array}{l} \Diamond(\text{init}(A) \wedge ts(A, \{\}, [])) \wedge \Diamond(\text{init}(B) \wedge ts(A, \{\}, [])) \Rightarrow \\ \bigcirc\{\Diamond(\text{init}(A) \wedge ts(A, \{\}, [])) \wedge \Diamond(\text{init}(B) \wedge ts(A, \{\}, []))\} \end{array} \right\} \quad (4.36)$$

from which we deduce:

$$\models \Diamond(\text{init}(A) \wedge ts(A, \{\}, [])) \wedge \Diamond(\text{init}(B) \wedge ts(A, \{\}, [])) \Rightarrow \quad (4.37)$$

$$\quad \quad \quad \square\{\Diamond(\text{init}(A) \wedge ts(A, \{\}, [])) \wedge \Diamond(\text{init}(B) \wedge ts(A, \{\}, []))\}$$

which is equivalent to the absence of deadlock proof we presented in § 4.1.2.

From this example, we have demonstrated axiomatic and operational interpretations of COL using operational semantics defined by an LTL inference engine, LTL inference rules and program state assertions.

4.2.7 Extensions for Non-Deterministic Rendezvous

Until this point we have considered only a limited type of rendezvous — a single caller for a single accept. Programs constrained to this type of rendezvous were classified as task deterministic; furthermore, such programs could be interpreted as completely deterministic. Our concern with deterministic programs is founded on the inability of LTL to model or analyze non-deterministic programs.

In this section we extend the simple, deterministic rendezvous model to include two non-deterministic rendezvous structures: multiple (concurrent) callers of a task entry and multiple open entries (with callers) in a selective accept. In the first case there is non-determinism in the arrival order of callers on the entry queue. In the selective accept, we assume that some algorithm non-deterministically selects one open entry. The gist of our approach is to "convert" a non-deterministic rendezvous into a set of deterministic rendezvous, thus defining a set of task deterministic programs. The conversion process is an enumeration of all possible rendezvous alternatives whenever a non-deterministic rendezvous exists; this is

Figure 4.13: Three Task Design with Non-Deterministic Rendezvous

analogous to the predicate logic to propositional logic conversion discussed in § 4.1.1.

Each task deterministic program is created by selecting one of the enumerated alternatives, whenever a choice exists. For example, in Figure 4.13, we have two tasks calling a single entry. In this case, we enumerate two alternatives:

1. task A arrives ahead of task B and begins a rendezvous with task C;
2. task B arrives ahead of task C and begins a rendezvous with task C

The non-deterministic choice is the arrival order of callers on the task entry. We create a task deterministic program by deciding that the acceptor can rendezvous with one caller only. Timing diagrams for the two the programs defined by the enumerated rendezvous are presented in Figure 4.14.

The extension of our basic rendezvous model is described in three parts: (1) modifications to our current set of program state predicates (defined in § 4.2.1) and Ada Language Implication formulæ (defined in § 4.2.5), (2) a notation to specify enumerated rendezvous alternatives and (3)¹ changes to the basic operational semantics.

Modifications to Current Predicates and Formulæ

The program state predicates in § 4.2.1 and the Ada Language Implication formulæ of § 4.2.5 are defined under the assumption that rendezvous were restricted to

Note: time scales are not necessarily linear

Figure 4.14: Timing Diagrams for Three Task Program

single caller, single acceptor structures. If we allow for rendezvous structures outside of this restriction, some formulæ will be invalid. Specifically, we refer to the Rendezvous Entry formula (ren) in Figure 4.10:

$$\begin{aligned} & \left\{ \begin{array}{l} \Diamond(\text{task_call}(t_1, x, y) \wedge ts(t_1, \{prm(p), \cdot\}, q_1)) \wedge \\ \Diamond(\text{start_accept}(t_2, y) \wedge ts(t_2, \{prm(q), \cdot\}, q_2)) \end{array} \right\} \Rightarrow \\ \equiv & \left\{ \begin{array}{l} \bigcirc \Diamond \left(\begin{array}{l} \text{start_rendez}(t_2, x, y) \wedge \\ ts(t_2, \{prm(p), \cdot\}, [rdz(t_1, x, y, \{prm(p), \cdot\}, q_1) | q_2]) \end{array} \right) \\ \wedge \bigcirc \Diamond(\text{call_accepted}(t_1, x, y) \wedge ts(t_1, \{prm(p), \cdot\}, q_1)) \end{array} \right\} \end{aligned}$$

This formula states that a start of rendezvous is inferred by the presence of an acceptor and *some* caller. If there are two eligible callers we argue that a rendezvous will start but "With Which Caller?"

A solution to this uncertainty is to introduce another predicate variable, x , in the *start_accept* predicate, where x explicitly names the caller task. Thus, *start_accept* is re-defined as follows:

$$\begin{aligned} & \equiv (\text{acc}(t_1, y) \underline{U} [\text{tcall}(t_2, x, y) \wedge \text{acc}(t_1, y)] \vee \Box \text{acc}(t_1, y)) \\ & \Leftrightarrow \text{start_accept}(t_1, x, y) \end{aligned} \quad (4.38)$$

Extending this definition to the Rendezvous Entry equation we get a formula that expresses a deterministic start of rendezvous.

$$\begin{aligned} & \left\{ \begin{array}{l} \Diamond(\text{task_call}(t_1, x, y) \wedge ts(\dots)) \wedge \\ \Diamond(\text{start_accept}(t_2, x, y) \wedge ts(\dots)) \end{array} \right\} \Rightarrow \\ \equiv & \left\{ \begin{array}{l} \bigcirc \Diamond(\text{start_rendez}(t_2, x, y) \wedge ts(\dots)) \wedge \\ \bigcirc \Diamond(\text{call_accepted}(t_1, x, y) \wedge ts(\dots)) \end{array} \right\} \end{aligned} \quad (4.39)$$

The effect of the caller variable, x , is to select one task caller for rendezvous out of the set of eligible callers; that is, it forces one and only one of the enumerated rendezvous to be selected. For instance, in Figure 4.13, suppose we have the following program state assertion:

$$t_0 \equiv \left\{ \begin{array}{l} \bigcirc \Diamond(\text{task_call}(A, A_1, C_E) \wedge ts(A, \dots)) \wedge \\ \bigcirc \Diamond(\text{task_call}(B, B_1, C_E) \wedge ts(B, \dots)) \wedge \\ \bigcirc \Diamond(\text{start_accept}(C, A_1, C_E) \wedge ts(C, \dots)) \end{array} \right\}$$

Figure 4.15: Semaphore Access Program

If we apply equation 4.39 (with the assistance of LTL inference rules) we can deduce:

$$t_0 \models \left\{ \begin{array}{l} \bigcirc \Diamond (\text{call_accepted}(A, A_1, C_E) \wedge ts(A, \dots)) \wedge \\ \bigcirc \Diamond (\text{task_call}(B, B_1, C_E) \wedge ts(B, \dots)) \wedge \\ \bigcirc \Diamond (\text{start_rendez}(C, A_1, C_E) \wedge ts(C, \{\dots\}, [\text{rdz}(A, A_1, C_E, \dots)])) \end{array} \right\}$$

whereas without the modification we would deduce:

$$t_0 \models \left\{ \begin{array}{l} \bigcirc \Diamond (\text{call_accepted}(A, A_1, C_E) \wedge ts(A, \dots)) \wedge \\ \bigcirc \Diamond (\text{call_accepted}(B, B_1, C_E) \wedge ts(B, \dots)) \wedge \\ \bigcirc \Diamond (\text{start_rendez}(C, A_1, C_E) \wedge ts(C, \{\dots\}, [\text{rdz}(A, A_1, C_E, \dots)])) \wedge \\ \bigcirc \Diamond (\text{start_rendez}(C, B_1, C_E) \wedge ts(C, \{\dots\}, [\text{rdz}(B, B_1, C_E, \dots)])) \end{array} \right\}$$

which may not be true. (Since $\Diamond$ implies a formula is true in some state in the future, it could be possible that both callers get accepted eventually, but not concurrently, so the above statement may be valid coincidentally).

Specification of Rendezvous Alternatives: Expansion

The modification of the *start_accept* predicate permits us to generate task deterministic programs by explicitly naming rendezvous callers as part of the accept. Unfortunately, blind substitution can produce task deterministic programs that do not describe realizable behaviour. To illustrate this point, we use the *semaphore access* program shown in Figure 4.15.

In this program, two user tasks, A and B, call a binary semaphore² task S in order to gain access to a resource task R. Each user task calls S.W, R.A and S.S in an endless loop; i.e. they wait for access, access the resource and then release access. The semaphore task implements a binary semaphore by using wait (W) and signal (S) entries; the wait entry is closed after it is called and opened after a call on the signal entry.

The presence of the semaphore task in this program, guarantees that only the task accepted by its wait entry task can ever be queued at the access entry (A) of the resource. If we create a task deterministic program in which the semaphore task accepts task B and the resource task accepts task A, then we have described an impossible program.

Our solution to this problem requires several steps:

1. introduce new program state predicates, called *define predicates*, that specify the possibility of alternative *start_accept* predicates; there should be no implication formula that use these predicates as premises;
2. replace the *start_accept* predicates with the define predicates in the non-deterministic program description; this produces a task deterministic program that is incompletely specified;
3. when the reference engine encounters define predicates in a PSA, they can be replaced by generated *start_accept* predicates for whichever tasks are described to be in task calls in the PSA.

The last step, which we call *expansion*, requires some modification of the basic operational semantics defined in § 4.2.6. We will address these changes shortly

The define predicates are as follows:

sel_accept(t, s) — The *sel_accept* predicate is true if an acceptor task *t* is executing the selective accept structure specified by *s*.

²This same design can in fact be easily a general counting semaphore.

$\text{accept}(t, y)$ — The *accept* predicate is true if an acceptor task t is executing an accept on entry y for which there may be multiple (concurrent) callers.

For completeness we add another program state predicate:

$\text{end_sel_accept}(t, s)$ — The *end_sel_accept* predicate is true if an acceptor task t is leaving the selective accept structure specified by s .

The substitution of define predicates in place of *start_accept* predicates in the program description formulae results in a task deterministic program. We prove this informally below.

Proof: Any program is task deterministic if and only if for each task there is only one path of progress. Since we have replaced any predicates for which more than one path of progress exists with predicates for which no path of progress exists, then the resultant program must be task deterministic.

Q.E.D.

The substituted program is an incomplete description of the original program because there is no specification of progress from the define predicates. The re-substitution of *start_accept* will be carried out by the inference engine, however, in the case of selective accepts, we still need to specify how the guard conditions define open and closed entries:

We introduce the *defined to be* symbol " $::=$ " to denote a set of tuples of the form:

$$(\pi(v), \bigcirc \Diamond (\text{accept}(t, e) \dots))$$

for a *sel_accept* define predicate. These tuples imply that the entry defined by an *accept* predicate is open if $\pi(v)$ is valid; thus, the $\pi(v)$ predicate expresses the guard condition. The set contains one tuple for each entry of the selective accept. The complete syntax is illustrated below.

$$\bigcirc \Diamond \left(\begin{array}{l} \text{sel_accept}(t, s) \\ \wedge ts(t, v, q) \end{array} \right) ::= \left\{ \begin{array}{l} \vdots \\ (\pi_1(v), \bigcirc \Diamond (\text{accept}(t, e_1) \wedge ts(t, v, q))), \\ \vdots \end{array} \right\}$$

The entry set specified by this expression is used by the inference engine to decide which entries can be considered for *start_accept* substitution.

Modification of Basic Operational Semantics

The basic operational semantics (cref. § 4.2.6) described how the LTL description of a task deterministic program is interpreted for axiomatic or operational analysis. Our intent in axiomatic analysis is the proof of global properties such as absence of deadlock. In the case of non-deterministic programs, a global property can be demonstrated by proving the property over each deterministic instance of program behaviour (as discussed in § 4.1.1).

Operational analysis is concerned primarily with observing the program in execution; thus, we are not necessarily interested in all possible behaviours of the program, as was the case for global properties. These different objectives have a substantial effect on the *start_accept* re-substitution, or expansion process described next.

In the basic operational semantics, an inference engine deduces a new PSA from the current PSA using inference rules and program description implications identified by some selection algorithm ϕ . As we stated in the previous section, the re-substitution of *start_accept* predicates in the program is performed by the inference engine using the expansion process. This process is as follows:

if the current PSA implies some defined predicate, then a new set of PSA can be created by re-substituting start_accept predicates for which task callers are implied by the current PSA; that is, expanding the define predicate into a set of predicates.

For example, in the three task example of Figure 4.13 we have the following PSA.

$$t_0 \equiv \left\{ \begin{array}{l} \bigcirc \Diamond (\text{task_call}(A, A_1, C_E) \wedge ts(A, \dots)) \wedge \\ \bigcirc \Diamond (\text{task_call}(B, B_1, C_E) \wedge ts(B, \dots)) \wedge \\ \bigcirc \Diamond (\text{accept}(C, C_E) \wedge ts(C, \dots)) \end{array} \right\}$$

then the inference engine can generate two PSA by expansion:

$$t_0 \models \left\{ \begin{array}{l} \bigcirc \Diamond (task_call(A, A_1, C_E) \wedge ts(A, \dots)) \wedge \\ \bigcirc \Diamond (task_call(B, B_1, C_E) \wedge ts(B, \dots)) \wedge \\ \bigcirc \Diamond (start_accept(C, A_1, C_E) \wedge ts(C, \dots)) \end{array} \right\}$$

$$t_0 \models \left\{ \begin{array}{l} \bigcirc \Diamond (task_call(A, A_1, C_E) \wedge ts(A, \dots)) \wedge \\ \bigcirc \Diamond (task_call(B, B_1, C_E) \wedge ts(B, \dots)) \wedge \\ \bigcirc \Diamond (start_accept(C, B_1, C_E) \wedge ts(C, \dots)) \end{array} \right\}$$

Expansion generates two separate inferences; i.e. inferences for different task deterministic programs.

A more complicated example illustrates expansion involving a selective accept structure. In the semaphore access program of Figure 4.15, the situation in which user A is waiting for access and user B is relinquishing access is defined by the following PSA (we deliberately omit $prm(p)$ from the assertion on the data state):

$$t_0 \models \left\{ \begin{array}{l} \bigcirc \Diamond (task_call(A, A_1, S_w) \wedge ts(A, \dots)) \wedge \\ \bigcirc \Diamond (task_call(B, B_1, S_s) \wedge ts(B, \dots)) \wedge \\ \bigcirc \Diamond (sel_accept(S, S_1) \wedge ts(S, \{g(F)\}, [])) \wedge \\ \bigcirc \Diamond (accept(R, R_A) \wedge ts(R, \dots)) \end{array} \right\}$$

The selective accept entry list description for this program is:

$$\bigcirc \Diamond \left(\begin{array}{l} sel_accept(S, S_1) \\ \wedge ts(S, v, q) \end{array} \right) ::= \left\{ \begin{array}{l} (\{g(T)\}, \bigcirc \Diamond (accept(S, S_w) \wedge ts(S, v, q))), \\ (\{g(x)\}, \bigcirc \Diamond (accept(S, S_s) \wedge ts(S, v, q))) \end{array} \right\}$$

In this case, the only possible expansion yields the following PSA (user A cannot be selected because the entry is closed):

$$t_0 \models \left\{ \begin{array}{l} \bigcirc \Diamond (task_call(A, A_1, S_w) \wedge ts(A, \dots)) \wedge \\ \bigcirc \Diamond (task_call(B, B_1, S_s) \wedge ts(B, \dots)) \wedge \\ \bigcirc \Diamond (start_accept(S, B_1, S_s) \wedge ts(S, \{g(F)\}, [])) \wedge \\ \bigcirc \Diamond (accept(R, R_A) \wedge ts(R, \dots)) \end{array} \right\}$$

Like the selection of appropriate inference rules and implication formulæ for deductions, the inference engine decides when to invoke expansion, given valid

circumstances. We designate the selection algorithm, ϕ , that also supports expansion, as ϕ_r .

The choice of ϕ_r defines the manner in which task deterministic programs are generated. For instance, in the axiomatic absence of deadlock analysis technique we present in chapter 5, ϕ_r is designed to generate all possible task deterministic programs; however, in the COL Interpreter described in chapter 8, ϕ_r is tailored to produce a useful operational analysis.

4.3 Practical Language Definition

In this section, we conclude our presentation of COL by: (1) describing a practical implementation of the language syntax and (2) demonstrating the use of COL (and its new syntax) by the specifying the behaviour of the semaphore access Ada program presented earlier in this chapter.

4.3.1 A Prolog-based COL Language Syntax

Thus far, the COL syntax has been described using typeset symbols and a variety of structures (queues, arrays and sets). We now re-define this syntax using Prolog structures and conventions. Through the use of Prolog language features, such as operators and lists, we transform COL statements into legal Prolog expressions. Our purpose for this activity is threefold:

1. we need a language syntax that is practical for the character sets of conventional computers;
2. we will implement later the operational semantics of COL (for two different objectives) using Prolog;
3. Prolog supports features that are useful for constructing logical inferences; e.g. predicates and pattern matching by unification.

We begin our transformation process by re-defining the basic symbols:

◇	—	<>	○	—	*
^	→	&	⇒	—	⇒
<i>ts</i>	—	<i>ts</i>	@	—	@
::=	—	::=			

Any symbols not re-defined above will not be required. Some further changes are itemized below.

- The validity of connective formulae will be implicit, therefore $\equiv$ and $t_0 \equiv$ are not used.
- Predicate variables are expressed by uppercase character strings; e.g. *sel_accept(t,s)* becomes *sel_accept(T,S)*.
- Character strings are surrounded by single or double quotes (e.g. "ABC"); however, they may be used without quotes if they are lowercase only (e.g. *task_call(t1,...)*).
- The caller and acceptor locations formerly described using subscripts are now defined using a dyadic ! operator; e.g. *task_call(A, A₁, S_w)* becomes *task_call("A", "A"!1, "S"! "W")*.
- The LIFO caller queue is implemented by a Prolog list; e.g. [...] becomes [...].
- The data predicate array is implemented by a Prolog list; e.g. {...} becomes [...].
- The set of entry alternatives in selective accept entry lists is implemented by a Prolog list; e.g. {(,), ...} becomes [(,), ...].
- All language statements are terminated by a period.

The initial PSA which has been defined as an LTL formula is re-defined as a Prolog fact *init_set* consisting of a list of 2-tuples. Each 2-tuple defines a task name and the initial assertion for the task. For example, the initial PSA from the two

task example in Figure 4.3 is transformed from:

$$t_0 \equiv \left\{ \begin{array}{l} \Diamond(\text{init}(A) \wedge \text{ts}(A, \{\text{prm}(\{\})\}, \{\}) \wedge \\ \Diamond(\text{init}(B) \wedge \text{ts}(B, \{\text{prm}(\{\})\}, \{\})) \end{array} \right\}$$

to:

```
init_set([
    ("A", <>(init("A") & ts("A", [prm({})], {}))),
    ("B", <>(init("B") & ts("B", [prm({})], {})))
]).
```

This format is chosen for practical convenience.

4.3.2 Data and Function Handling

The conjunction of data predicates in array notation was defined in § 4.2.4. The enumeration of the data predicates of each task variable in pre- and post-conditions was shown. In this section, we describe a modified unification algorithm that allows us to represent more compactly the conjunction of data predicates when used in pre- and post-conditions. We introduce the new data predicate unification algorithm by an example.

Suppose we wish to match the data predicate array:

`[v1(1), v2(a), v3(7)]`

against the pre-condition:

`[v1(X), v2(a), v3(Y)]`

Normal unification would match the formulæ by instantiating *X* to 1 and *Y* to 7. In fact, the pre-condition does not specifically address the values of task variables *v1* and *v3* other than to imply "I don't care".

Let us adopt a new notation for the pre-condition, called *compact array*:

`= [v2(a)]`

and interpret this as:

"if the value of $v2$ is $\bar{a}$ and the values of all other task variables are anything then...".

Assuming that we have a process that can implement this interpretation, we then apply regular unification only to $v2$. We designate this new unification process Un^* . The same process can be applied to data predicate arrays that define post-conditions; for example,

$= [v2(a)]$

is interpreted as:

"the value of $v2$ must be a and the values of all other task variables are unchanged".

Thus, the behaviour of Un^* is dependent on its context of use:

pre-condition: $[...] \quad Un^* = [...]$

post-condition: $[...] \leftarrow [...] \quad Un^* = [...]$

The post-condition behaviour uses the data predicate array of the pre-condition as a Un^* argument.

We integrate easily the Un^* process with our inference engine because all implication pre-conditions are matched against the data predicate arrays of the current PSA; thus, there is always one Un^* argument of the pre-condition that is a data predicate array.

To illustrate the operation of the compact array notation, consider the following example:

pre-condition: $= [v2(X), v3(a)]$

post-condition: $= [v3(X)]$

PSA array: $[v1(7), v2(8), v3(a)]$

new PSA array: $[v1(7), v2(8), v3(8)]$

The pre-condition compact array matched against the PSA array and instantiated X to 8; thus, the compact post-condition becomes $=[v3(8)]$ which creates the new PSA array.

The use of supplementary predicates (cref. § 4.2.4) within compact arrays is identical to their use in the data predicate arrays; for example,

```
= [v2(X), @ (greater(X, 7))]
```

defines a condition that tests that the value of task variable $v2$ is greater than 7.

As yet we have not specified how the supplementary predicates are defined. Given the Prolog implementation of COL syntax, we define these predicates as Prolog code to be resolved using the Prolog expression evaluator; i.e. we add Prolog as an extension to COL for the specification of supplementary predicates. Therefore, we can create supplementary predicates that are as complex as allowed by the Prolog language.

4.3.3 A COL Language Example

Below, we illustrate various features of COL by specifying the semaphore access program shown earlier in Figure 4.15. Each task is specified individually with features of interest identified. We will use lowercase strings for all program symbols as a matter of convenience.

Task A

Task A illustrates the task calls.

```
init(a) & ts(a,=[],Q) =>
    *<>(task_call(a,a!1,s!w) & ts(a,=[],Q)).

return_task_call(a,a!1,s!w) & ts(a,=[],Q) =>
    *<>(task_call(a,a!2,r!a) & ts(a,=[],Q)).

return_task_call(a,a!2,r!a) & ts(a,=[],Q) =>
    *<>(task_call(a,a!3,s!s) & ts(a,=[],Q)).
```

```

return_task_call(a,a!3,s!s) & ts(a,=[],Q) =>
  *<>(init(a) & ts(a,=[],Q)).

```

Task B

Task B is identical in behaviour to task A.

```

init(b) & ts(b,=[],Q) =>
  *<>(task_call(b,b!1,s!w) & ts(b,=[],Q)).

return_task_call(b,b!1,s!w) & ts(b,=[],Q) =>
  *<>(task_call(b,b!2,r!a) & ts(b,=[],Q)).

return_task_call(b,b!2,r!a) & ts(b,=[],Q) =>
  *<>(task_call(b,b!3,s!s) & ts(b,=[],Q)).

return_task_call(b,b!3,s!s) & ts(b,=[],Q) =>
  *<>(init(b) & ts(b,=[],Q)).

```

Task S

Task S illustrates the selective accept entry list definition and pre- and post-condition using the guard variable g.

```

init(s) & ts(s,=[],Q) =>
  *<>(sel_accept(s,s!1) & ts(s,=[],Q)).

start_rendez(s,X,s!w) & ts(s,=[],Q) =>
  *<>(end_rendez(s,s!w) & ts(s,=[g(F)],Q)).

end_accept(s,s!w) & ts(s,=[],Q) =>
  *<>(end_sel_accept(s,s!1) & ts(s,=[],Q)).

start_rendez(s,X,s!s) & ts(s,=[],Q) =>
  *<>(end_rendez(s,s!s) & ts(s,=[g(T)],Q)).

end_accept(s,s!s) & ts(s,=[],Q) =>
  *<>(end_sel_accept(s,s!1) & ts(s,=[],Q)).

```

```

end_sel_accept(s,s!1) & ts(s,=[],Q) =>
    *<>(init(s) & ts(s,=[],Q)).

*<>(sel_accept(s,s!1) & ts(s,V,Q)) ::=
    [
        (=[],*<>(accept(s,s!s) & ts(s,V,Q))),
        (=[g(T)],*<>(accept(s,s!w) & ts(s,V,Q)))
    ]:

```

Task R

Task R illustrates a simple accept.

```

init(r) & ts(r,=[],Q) =>
    *<>(accept(r,r!a) & ts(r,=[],Q)).

start_rendez(r,X,r!a) & ts(r,=[],Q) =>
    *<>(end_rendez(r,r!a) & ts(r,=[],Q)).

end_accept(r,r!a) & ts(r,=[],Q) =>
    *<>(init(r) & ts(r,=[],Q)).

```

Chapter 5

A Deadlock Analysis Method

In this chapter we present an automated, axiomatic method to prove the absence of deadlock in Ada programs: a *deadlock analyzer*. We motivate the method by examining briefly, the structure of a classic mechanical theorem prover: Prolog. Following this discussion, we describe the development of a theory for proving absence of deadlock based on *discrete linear temporal logic* (LTL) that uses a COL description of the Ada program as input. We implement the theory by defining COL operational semantics; specifically, we describe the inference engine algorithm, ϕ_r , that characterizes the operational semantics (cref. § 4.2.7). Finally, we conclude the chapter with discussions on the deadlock analyzer tool implementation (with examples), decidability and computational complexity.

5.1 Automatic Theorem Proving: An Example

Techniques for automatic (mechanical) theorem proving with classical logic attack the following basic problem:

“given a set of valid formulæ that describe some system and a formula to be proven, the *theorem*, is there an algorithm that can derive the theorem from the set of valid formulæ?”

The most widely known solution to this problem is the top-down, resolution-based theorem proving technique that forms the basis of the programming language

Prolog [28,55,60]. The key elements of this method are itemized below.

- All formulæ are expressed in a *clausal form* known as *Horn Clauses* so that formulæ are represented using a restricted set of simple structures.
- Proof by contradiction is used to prove the theorem; i.e. the negation of the theorem is assumed to be true but shown to imply an inconsistent system.
- Pattern matching by unification is used to identify clauses that form the sequence of deduction steps (this was discussed earlier in § 4.2.6).
- Possible deduction paths are explored using a depth-first search, starting with the theorem to be proven; i.e. *top-down*.

Formulæ in the Horn clause format are implications with one conclusion and zero or more premises; a clause with no premises is an *assertion* (also known as a *fact*) and a clause with premises and conclusions is called a *rule*. In an assertion, the premise is implied to be the predicate *true* which is always valid. As a convenience, we can treat assertions as rules with the premise *true*. A special Horn clause with premises but no conclusion represents a *denial* (negation of a formula).

Unification matches the predicates and predicate arguments of denials and rules as described in § 4.2.6. In the case where there are no universally quantified variables, unification reduces to exact symbol matching.

A theorem is proven if the assumption that the theorem is false implies a contradiction in the clauses describing the system, otherwise, nothing can be proven; thus, we express the theorem as a denial Horn clause and all other clauses as rules. The inference rule, *modus tollens*:

$$\text{if } \models \neg A \text{ and } \models B \Rightarrow A \text{ then } \models \neg B$$

is used to derive results. The inference mechanism is applied as follows:

1. starting from the initial denial, use unification to find rule in which the conclusion matches the predicate of the denial;
2. use modus tollens to infer the negation of the premise side of the rule;

3. continue (1) until $\neg \text{true}$ is deduced, which implies the system is false, or contradictory;
4. if the system is contradictory then the theorem is proven;
5. if the system is not shown to be contradictory then nothing is proven.

The depth-first search is required to prove a denial consisting of a conjunction of predicates. We prove a contradiction of this denial by showing that each predicate in the conjunction is valid; e.g. in order to prove:

$$\models \neg(a(1) \wedge b(2))$$

we show:

$$\models a(1) \text{ and } \models b(2)$$

which implies:

$$\models a(1) \wedge b(2)$$

which contradicts the original denial. Each predicate of the conjunction is treated as another theorem or goal to be proven; we refer to them as *subgoals*. We apply the identical inference process to each subgoal; i.e. we assume the denial and show a contradiction. the set of subgoals to be proven form a "tree" of deductions. For example, the proof of a conclusion X may require the proof of premises A, B and C, which in turn, require more subgoals to be proven. The tree of deductions is explored depth-first, thus, one predicate in a conjunction is completely proven before the next predicate in the same conjunction is proven. An alternate method, *breadth-first* search, would perform proofs on all predicates of a conjunction simultaneously.

A variation on the top-down proof strategy used by Prolog is the *bottom-up* deduction method described by Kowalski [60]. In this strategy, a contradiction of the goal denial is deduced from the assertions using the rules and the inference rule *modus ponens*:

$$\text{if } \models A \text{ and } \models A \Rightarrow B \text{ then } \models B$$

The contradiction of the denial is constructed (or synthesized) from the assertions of the system given the appearance that we start at the bottom of a deduction tree and progress towards the top (goal).

The COL-based, absence of deadlock theorem prover that we describe in the following sections, uses many of notions just introduced: proof by contradiction, a systematic format for logic statements, unification, a bottom-up deduction strategy and a set of inference rules. In fact, most of them are an integral part of COL itself.

5.2 A Theorem for Absence of Deadlock

In chapter 4, § 4.2.6, we proved absence of deadlock in an Ada program by using the axiomatic interpretation of the COL specification of the program; that is, we proved absence of deadlock using LTL. In that proof, we defined a program to be deadlock-free if all of its tasks were cycling endlessly. We represent this notion in LTL, for task deterministic programs, as follows:

$$\models \Box \left\{ \begin{array}{l} \Diamond(\text{init}(t) \wedge \text{ts}(t, \pi_1(v), q_1)) \Rightarrow \\ \bigcirc \Diamond(\text{init}(t) \wedge \text{ts}(t, \pi_2(v), q_2)) \end{array} \right\} \quad (5.1)$$

which we interpret as:

"for all sequences, if task t is eventually at the top of its cycle then, in the next state, it will eventually be at the top of its cycle (again)".

For example, if task t is at the top of its cycle "now" and in the next program state it is not, then if the program is deadlock-free, task t will eventually be at the top of its cycle again.

Equation 5.1 is used in an inductive, rather than deductive, proof for absence of deadlock. If equation 5.1 can be shown to be valid, then the LTL induction formula:

$$\models \Box(P \Rightarrow \bigcirc P) \Rightarrow (P \Rightarrow \Box P)$$

will deduce:

$$\models \Diamond(\text{init}(t) \dots) \Rightarrow \Box \Diamond(\text{init}(t) \dots)$$

which proves absence of deadlock by induction; that is, if we show that a task eventually reaches the top of its cycle once, then it will return always to the top of its cycle, eventually.

We do not choose to prove absence of deadlock in this manner since it requires a considerable bookkeeping effort to manage simultaneous proofs that all tasks satisfy equation 5.1; instead, we take an indirect approach: prove that the program is not deadlock-free by demonstrating that it has a *system-wide* deadlock.

A system-wide deadlock implies that no task in the program can progress; a non-system-wide, or, *local* deadlock occurs when some task can never progress, but does not hinder the progress of other tasks. All deadlocks can be classified as either system-wide or local. The lack of system-wide deadlock in a program does not imply that we have proven absence of deadlock; we must show also that a program is free of local deadlocks. We accomplish this objective by treating local deadlocks as instances of *starvation* and using starvation analysis, as we demonstrate below.

- Suppose we have a program where there exists an infinite sequence of program states in which at least one task is not progressing and all other tasks are progressing.
- Further, suppose that within the same program, there exists an infinite sequence of program states in which the non-progressing task from the sequence above, progresses.
- We say that the task in the first sequence is starving.
- A local deadlock implies that for all infinite sequences of program states, some task cannot progress.
- We prove absence of starvation by showing that there is no infinite sequence of program states in which a task is not progressing; thus, if we prove starvation by locating such a sequence then the non-progressing task is either a local deadlock or a starving task. Human reasoning can be used to make the distinction.

The topics of starvation and proving absence of starvation are presented in chapter 6.

We use contradiction to prove that a program is not system-wide deadlock-free as follows:

"Assume that the program is deadlock-free; if we can show that there exists a program state in which the program exhibits system-wide deadlock, then we have a contradiction; thus the program is NOT deadlock-free. Otherwise, we conclude that the program is system-wide deadlock-free."

We have deliberately chosen failure of the proof to imply that the goal is not true. This concept, called *failure as negation*, is valid since we consider the COL specification to represent a "closed world" or system [61].

In the next section, we show how the COL language and operational semantics can be used to implement our theorem prover.

5.3 A COL-Based Deadlock Prover

The COL language and its operational semantics provide us with the basis for an automatic theorem prover:

- COL statements are simple LTL implications in a few restricted formats;
- the COL operational semantics are defined by a bottom-up inference engine and a set of inference rules;
- the inference engine uses unification-based pattern matching.

The COL inference engine can identify a system-wide deadlock in a task deterministic program if there exists a program state assertion (PSA) in which each task is described to be in one of the states below, eventually.

1. the task is in a task call for which there is no acceptor available;
2. the task is in a call accepted state;

3. the task is in an accept for which there are no callers.

We need not explicitly identify such a PSA; it is implied if no implication formulæ can be applied to the PSA.

The COL-based proof that a task deterministic program is not deadlock-free is summarized as follows:

- assume that the program is deadlock-free;
- starting with the initial PSA, apply the COL inference engine to successively deduced PSA;
- if there exists a PSA for which no implication formulæ apply, then a system-wide deadlock is deduced;
- the program is NOT deadlock-free by contradiction of its assumption.

The above procedure suggests two important questions: (1) how is the procedure extended to permit analysis of non-deterministic programs and (2) how does the inference engine stop if a system-wide deadlock cannot be deduced. These issues are addressed next.

5.3.1 Proofs of Non-Deterministic Programs

In chapter 4, § 4.1.1, we indicated that the proof of a global property of a program (such as absence of deadlock) requires that we demonstrate the property for all possible behaviours of the program. In proving that a program is not deadlock-free, we must examine all of the possible deterministic programs that represent the non-deterministic program. In the COL framework, the deterministic alternatives are generated by the *expansion* of PSA (cref. § 4.2.7) that is executed by the COL inference engine.

Expansion converts a PSA of a deterministic program into a set of PSA representing non-deterministic alternatives. The identification of non-deterministic alternatives in the original PSA is provided by the *define* predicates: *sel_accept* and *accept*. The expansion process substitutes one or more define predicates in a

PSA with a *start_accept* predicate to produce a deterministic alternative. A set of alternatives is the set of PSA resulting from different substitutions.

The proof of a global property requires that we explore each alternative generated by expansion, provided that expansion generates all possible deterministic alternatives. The inference algorithm, ϕ , that we specify for the COL inference engine, must guarantee the latter condition. The ϕ implements two procedures: (1) a procedure that selects the implication formula to be used for the next inference and (2) a procedure that decides whether to invoke expansion or apply inference. The placement of expansion in the inference process is fundamental to the correct enumeration of all possible deterministic alternatives. We show this by example, below. (Note: throughout this chapter we omit the $\text{pr}\mathbf{a}(\mathbf{[]})$ predicate in the data state information).

Suppose that we have the following PSA and COL implication statement:

PSA:

```
[
  (a.*<>(init(a) & ts(a,[],[]))),
  (b.*<>(task_call(b,b!1,c!e) & ts(b,[],[]))),
  (c.*<>(accept(c,c!e) & ts(c,[],[])))
]
```

COL statement:

```
init(a) & ts(a,=[],Q) =>
  *<>(task_call(a,a!1,c!e) & ts(a,=[],Q)).
```

If we apply expansion to the PSA then we will generate only one deterministic alternative because only one caller is implied by the PSA:

```
[
  (a.*<>(init(a) & ts(a,[],[]))),
  (b.*<>(task_call(b,b!1,c!e) & ts(b,[],[]))),
  (c.*<>(start_accept(c,b!1,c!e) & ts(c,[],[])))
]
```

Had we applied the implication statement instead, we would get a new PSA:

```
[
  (a.*<>(task_call(a,a!1,c!e) & ts(a,[],[]))),
  (b.*<>(task_call(b,b!1,c!e) & ts(b,[],[]))),
  (c.*<>(accept(c,c!e) & ts(c,[],[])))
]
```

which, when expanded, creates two deterministic alternatives, one for each.

* * *

A variation of this problem occurs when a PSA implies that two or more tasks can be expanded. We illustrate this in the example below.

Suppose that we have the following PSA and COL statement:

PSA:

```
[
  (a.*<>(task_call(a,a!1,d!e) & ts(a,[],[]))),
  (b.*<>(task_call(b,b!1,d!e) & ts(b,[],[]))),
  (c.*<>(task_call(c,c!1,e!e) & ts(c,[],[]))),
  (d.*<>(accept(d,d!e) & ts(d,[],[]))),
  (e.*<>(accept(e,e!e) & ts(e,[],[])))
]
```

COL statement:

```
return_task_call(c,c!1,e!e) & ts(c,=[],Q) =>
  *<>(task_call(c,c!2,d!e) & ts(c,=[],Q)).
```

If we apply expansion to both acceptor tasks in the PSA then we will generate two deterministic alternatives:

```
[
  (a.*<>(task_call(a,a!1,d!e) & ts(a,[],[]))),
  (b.*<>(task_call(b,b!1,d!e) & ts(b,[],[]))).
]
```

```

(c,*<>(task_call(c,c!1,e!e) & ts(c,[],[]))),
(d,*<>(start_accept(d,a!1,d!e) & ts(d,[],[]))),
(e,*<>(start_accept(e,c!1,e!e) & ts(e,[],[])))
]

[
(a,*<>(task_call(a,a!1,d!e) & ts(a,[],[]))),
(b,*<>(task_call(b,b!1,d!e) & ts(b,[],[]))),
(c,*<>(task_call(c,c!1,e!e) & ts(c,[],[]))),
(d,*<>(start_accept(d,b!1,d!e) & ts(d,[],[]))),
(e,*<>(start_accept(e,c!1,e!e) & ts(e,[],[])))
]

```

If we had expanded only the accept of task *e* and then applied several inferences, we would generate the following PSA:

```

[
(a,*<>(task_call(a,a!1,d!e) & ts(a,[],[]))),
(b,*<>(task_call(b,b!1,d!e) & ts(b,[],[]))),
(c,*<>(task_call(c,c!2,d!e) & ts(c,[],[]))),
(d,*<>(accept(d,d!e) & ts(d,[],[]))),
(e,*<>(...))
]

```

If we apply expansion to task *d* we generate three PSA, each representing a deterministic accept of one of the three callers.

* * *

Both of these examples illustrate the complex relationship between expansion and inference. Below we summarize the salient points of this issue.

- The task deterministic programs that exist before and after expansion are indifferent to assumptions of relative rates of task execution, therefore, applicable implication formulæ can be used for inference in any order. The

expansion process will generate alternative deterministic accepts for those callers implied by the current PSA. Therefore, in order to generate all possible alternative deterministic accepts we must be sure that all implications that deduce task calls have been applied. This can be guaranteed if all implications have been applied; i.e. no further inference is possible.

- The order of expansion of multiple define predicates affects the outcome of the expansion process because the inference following expansion of one acceptor may generate more eligible callers for other acceptors; thus, more eligible callers will generate more PSA from the expansion of the other acceptors.

The operational interpretation of the relationship provides an alternate perspective to the discussion; we paraphrase the two points discussed above as follows:

- The expansion process generates deterministic alternatives by selecting one of the callers that has arrived at a non-deterministic accept; thus, in order to generate alternatives with all possible callers, the callers must have executed the task call before any selections are made. If all tasks have progressed as far as possible then we have guaranteed that all possible callers have arrived.
- The order of expansion of multiple define predicates suggests the execution of one acceptor before another. The execution rates of acceptor tasks may affect other acceptors by producing additional callers. Therefore, we evaluate the non-deterministic execution order of acceptors by expanding define predicates in all possible orders.

The expansion control procedure to be used by the COL inference engine, as suggested by this discussion, is summarized below.

1. Apply all possible inferences to PSA before using expansion; this will guarantee that all arrival orders of callers are considered.
2. Expand define predicates under the limitation that only one task is expanded in any new PSA; this will guarantee that all execution orders of acceptors are considered.

3. If expansion fails, then a system-wide deadlock is identified.

A further example illustrates this procedure.

Suppose that we have the following PSA and COL implication statements:

PSA:

```
[
  (a,*<>(init(a) & ts(a,[],[]))),
  (b,*<>(task_call(b,b!1,f!e) & ts(b,[],[]))),
  (c,*<>(task_call(c,c!1,f!e) & ts(c,[],[]))),
  (d,*<>(task_call(d,d!1,e!e) & ts(d,[],[]))),
  (e,*<>(accept(e,e!e) & ts(e,[],[]))),
  (f,*<>(accept(f,f!e) & ts(f,[],[])))
]
```

COL statements:

```
init(a) & ts(a,=[],Q) =>
  *<>(task_call(a,a!1,e!e) & ts(a,=[],Q)).

return_task_call(d,d!1,e!e) & ts(d,=[],Q) =>
  *<>(task_call(d,d!2,f!e) & ts(d,=[],Q)).
```

By applying procedure step 1, we exhaust all possible inferences (specifically, the `init(a)` implication) and generate:

```
[
  (a,*<>(task_call(a,a!1,e!e) & ts(a,[],[]))),
  (b,*<>(task_call(b,b!1,f!e) & ts(b,[],[]))),
  (c,*<>(task_call(c,c!1,f!e) & ts(c,[],[]))),
  (d,*<>(task_call(d,d!1,e!e) & ts(d,[],[]))),
  (e,*<>(accept(e,e!e) & ts(e,[],[]))),
  (f,*<>(accept(f,f!e) & ts(f,[],[])))
]
```

By applying procedure step 2, we generate four alternative PSA:

```
[
  (a,*<>(task_call(a,a!1,e!e) & ts(a,[],[]))),
  (b,*<>(task_call(b,b!1,f!e) & ts(b,[],[]))),
  (c,*<>(task_call(c,c!1,f!e) & ts(c,[],[]))),
  (d,*<>(task_call(d,d!1,e!e) & ts(d,[],[]))),
  (e,*<>(start_accept(e,a!1,e!e) & ts(e,[],[]))),
  (f,*<>(accept(f,f!e) & ts(f,[],[])))
]
```

```
[
  (a,*<>(task_call(a,a!1,e!e) & ts(a,[],[]))),
  (b,*<>(task_call(b,b!1,f!e) & ts(b,[],[]))),
  (c,*<>(task_call(c,c!1,f!e) & ts(c,[],[]))),
  (d,*<>(task_call(d,d!1,e!e) & ts(d,[],[]))),
  (e,*<>(start_accept(e,d!1,e!e) & ts(e,{},[]))),
  (f,*<>(accept(f,f!e) & ts(f,[],[])))
]
```

```
[
  (a,*<>(task_call(a,a!1,e!e) & ts(a,[],[]))),
  (b,*<>(task_call(b,b!1,f!e) & ts(b,[],[]))),
  (c,*<>(task_call(c,c!1,f!e) & ts(c,[],[]))),
  (d,*<>(task_call(d,d!1,e!e) & ts(d,[],[]))),
  (e,*<>(accept(e,e!e) & ts(e,[],[]))),
  (f,*<>(start_accept(f,b!1,f!e) & ts(f,[],[])))
]
```

```
[
  (a,*<>(task_call(a,a!1,e!e) & ts(a,[],[]))),
  (b,*<>(task_call(b,b!1,f!e) & ts(b,[],[]))),
]
```

```

(c,*<>(task_call(c,c!1,f!e) & ts(c,[],[]))),
(d,*<>(task_call(d,d!1,e!e) & ts(d,[],[]))),
(e,*<>(accept(e,e!e) & ts(e,[],[]))),
(f,*<>(start_accept(f,c!1,f!e) & ts(f,[],[])))
]

```

By applying successive inferences to the second PSA, (which includes the use of the second COL implication statement), until no more inferences are possible (step 2) we generate:

```

[
(a,*<>(task_call(a,a!1,e!e) & ts(a,[],[]))),
(b,*<>(task_call(b,b!1,f!e) & ts(b,[],[]))),
(c,*<>(task_call(c,c!1,f!e) & ts(c,[],[]))),
(d,*<>(task_call(d,d!2,f!e) & ts(d,[],[]))),
(e,*<>(...)),
(f,*<>(accept(f,f!e) & ts(f,[],[])))
]

```

The successive inference and expansions of PSA can be represented by a directed tree called a *PSA-tree*. Each node of the tree represents a PSA generated by inference or expansion; the directed arc points to the successor PSA. Nodes that are expanded into sets of PSA are called *expansion points*. The leaves of the PSA-tree are the set of current PSA; any path along the tree from a non-expansion point to an expansion point represents the deductions of a task deterministic program, or a *task deterministic path*. The path from the initial PSA to any leaf PSA is a *deduction path*.

The proof that a program is not deadlock-free requires the examination of multiple paths of deduction; thus, we need a procedure by which each path is evaluated. Depth-first search and breadth-first search are two reasonable candidates. We select breadth-first search for the following reasons: (1) breadth-first search will discover the shortest path of deduction from the initial PSA to a system-wide deadlock (if a deadlock exists) and (2) it will discover a system-wide deadlock if

one exists, whereas a depth-first search may not terminate¹ even if there is a system-wide deadlock. The breadth-first evaluation of deduction alternatives will be handled by the COL inference engine by placing the alternative current PSA in a queue and processing them in FIFO order.

5.3.2 Terminating the Inference Engine

In the theorem prover described above, the COL inference engine is used to continually generate PSA until it produces a PSA for which it cannot create a successor; such a PSA represents a system-wide deadlock. If there is no system-wide deadlock in the program then the inference engine runs forever because the program represented by the COL specification never terminates. If we assume that only task deterministic programs are allowed, then we propose the following:

1. record every PSA that is deduced;
2. if a PSA is deduced that has been deduced previously, then there is no value in continuing the inference process (because the program is task deterministic).

This leads us to another question: will a duplicate PSA be deduced? The answer is "yes" if the data space of the program is finite; i.e. each variable in the program can accept only a finite set of values. For the moment, we assume that this is the case; further treatment of this subject is left to the discussion on decidability in § 5.5.

Given our assumption of a finite data space, we modify our previous proof procedure to include the following:

- ...
- starting with the initial PSA, apply the COL inference engine to each deduced PSA and record the deduced PSA;

¹ Although, the termination issue is addressed in the next section, the problem of non-termination is presented in § 5.5 on decidability.

- ...
- if a PSA is deduced that matches a recorded, previously deduced PSA, then terminate the inference engine.

The only significant drawback to this approach is the need to store PSA. In this regard, efficient methods of recording PSA will be addressed in § 5.4 under the description of tool implementation.

If we extend our termination procedure to the proof of a non-deterministic program then the interpretation of a duplicate PSA is more complex:

1. if the first occurrence of the PSA is in the deduction path of the duplicate PSA (i.e. it is a predecessor), then we treat the duplicate as a repetition in a task deterministic program;
2. if the first occurrence of the PSA is in another deduction path then it represents the merging of the deduction paths of two task deterministic programs;

In the first case, we terminate the deduction for that path as we did for a simple task deterministic program; i.e. we remove the duplicate from the queue of PSA to be evaluated by the inference engine. We also terminate inference on duplicate PSA in the second case; since the two task deterministic programs have merged to a single behaviour, there is no value in pursuing both paths of deduction. Thus, if all deduction paths generate duplicate PSA, then eventually, the queue of PSA to be evaluated will be empty and the inference engine will terminate.

5.3.3 Summary of Proof Procedure

In this section, we specify the complete algorithm of COL-based deadlock theorem prover; we refer to this algorithm in the remainder of this chapter as the *deadlock analysis algorithm*.

```
(1)      assume that the program is deadlock-free;
(2)      queue_of_current_PSA <== initial PSA;
(3)      start COL inference engine;
(4)      apply inference to PSA at head of queue;
(5)      if inference fails then
(6)          expand PSA;
(7)          if expansion fails then
(8)              system-wide deadlock;
(9)              goto (27); [contradiction]
(10)         else
(11)             record each new PSA;
(12)             if not already deduced then
(13)                 add PSA to end of queue;
(14)             end if;
(15)         end if;
(16)     else
(17)         record new PSA;
(18)         if not already deduced then
(19)             add PSA to end of queue;
(20)         end if;
(21)     end if;
(22)     if queue_of_current_PSA is empty then
(23)         goto (27); [proof failed, no deadlock]
(24)     else
(25)         goto (4); [continue inference engine]
(26)     end if;
(27)     terminate inference engine;
```

5.4 Analysis Tool Implementation

In this section, we describe the implementation of our COL-based deadlock analysis tool using Prolog. Prolog's pattern matching facility provides us with the essential part of our inference engine and its dynamic assertions support the recording program state assertions (PSA). For the most part, the implementation is a direct translation of the inference algorithm, ϕ , specified in the last section; the sole exception is the storage of recorded PSA. The efficient recording of PSA is of primary importance because of its impact on the usefulness of the tool. Most of the remaining discussion is addressed to this issue. Following the descriptions of tool implementation, we provide several examples of use.

5.4.1 Efficient Recording of PSA

In section § 5.3.2, we introduced PSA recording as a requirement of terminating the inference engine when no system-wide deadlock could be deduced. The essence of the technique was the recognition of a previously followed path of deduction, where a path of deduction is a sequence of successively deduced PSA. A successor PSA is deduced from its predecessor by the application of one implication formula to the predecessor PSA by the inference engine; we also consider the PSA generated by expansion to be a successor although it is not deduced in the logic sense.

All of the single implication deductions that are performed by the inference engine are applied to task deterministic programs; thus, any PSA on a task deterministic path can identify the path. Furthermore, a deduction path is composed of a chain of task deterministic paths. Therefore, if we recorded the starting PSA of each task deterministic path, then we would have sufficient information to describe the deduction path; this represents the minimum level of information: one PSA per task deterministic path. We term the number of PSA recorded to express a task deterministic path to be the *PSA recording density*. Given the upper and lower bounds of the PSA recording density, we now discuss its impact on the analysis tool.

The PSA recording density is chosen such that two opposing objectives are

reasonably satisfied: (1) the number of recorded PSA must be as small as possible and (2) the number of recorded PSA must be large enough to produce a suitable explanation of deadlock should a system-wide deadlock be discovered. The explanation of a derived system-wide deadlock is the operational interpretation of the deduction path in the PSA-tree that leads from the initial PSA to the terminating PSA. If we use the lower bound of PSA recording density, the designer would have a difficult time deducing the remainder of any task deterministic path if the number of PSA in any such path was much more than 3-5. Such a poor explanation would frustrate the user of the tool. Unfortunately the upper bound of PSA recording density, which generates the most detailed explanation, is unacceptable also because of the significant number of PSA that would be recorded for even the simplest programs. We propose a compromise:

apply successive inferences to a current PSA until a PSA is deduced in which every task is implied to be waiting for a rendezvous and such that any set of inferences applied to any task of the first current PSA do not represent the deduction of more than one rendezvous; this PSA is recorded.

This procedure will highlight the occurrence of rendezvous between tasks, generally the most useful information for an explanation of deadlock; there should be sufficient recording density for explanation since each task can progress no more than one rendezvous between successively recorded PSA. We illustrate the procedure below.

Suppose we have the following sequence of successive PSA recorded at maximum density:

```
[
  (a.*<>(task_call(a.a!1,b!e1) & ts(a.[].[ ]))),
  (b.*<>(start_accept(b,a!1,b!e1) & ts(b.[].[ ]))),
  (c.*<>(task_call(c.c!1,b!e2) & ts(c.[].[ ])))
]
```

```
[
  (a.*<>(call_accepted(a,a!1,b!e1) & ts(a,[],[]))).
  (b.*<>(start_rendez(b,a!1,b!e1) & ts(b,[],[...]))).
  (c.*<>(task_call(c,c!1,b!e2) & ts(c,[],[])))
]
```

```
[
  (a.*<>(call_accepted(a,a!1,b!e1) & ts(a,[],[]))).
  (b.*<>(end_rendez(b,b!e1) & ts(b,[],[...]))).
  (c.*<>(task_call(c,c!1,b!e2) & ts(c,[],[])))
]
```

```
[
  (a.*<>(return_task_call(a,a!1,b!e1) & ts(a,[],[]))).
  (b.*<>(end_accept(b,b!e1) & ts(b,[],[]))).
  (c.*<>(task_call(c,c!1,b!e2) & ts(c,[],[])))
]
```

```
[
  (a.*<>(task_call(a,a!2,c!e1) & ts(a,[],[]))).
  (b.*<>(start_accept(b,c!1,b!e2) & ts(b,[],[]))).
  (c.*<>(task_call(c,c!1,b!e2) & ts(c,[],[])))
]
```

which when recorded using the suggested procedure would be limited to the following two PSA:

```
[
  (a.*<>(task_call(a,a!1,b!e1) & ts(a,[],[]))).
  (b.*<>(start_accept(b,a!1,b!e1) & ts(b,[],[]))).
  (c.*<>(task_call(c,c!1,b!e2) & ts(c,[],[])))
]
```

```
[
  (a,*<>(task_call(a,a!2,c!e1) & ts(a,[],[]))),
  (b,*<>(start_accept(b,c!1,b!e2) & ts(b,[],[]))),
  (c,*<>(task_call(c,c!1,b!e2) & ts(c,[],[])))
]
```

which provides a reasonable operational explanation of the program activity assuming that the user has some understanding of the programs intended behaviour.

We include our procedure in the inference engine's current algorithm, ϕ , as the implication formula selection procedure:

```

(1)   temp_current_PSA <== current_PSA;
(2)   do until no inference
(3)       apply rendezvous entry implication formula;
(4)       if PSA deduced then
(5)           temp_current_PSA <== deduced_PSA;
(6)       else
(7)           no inference;
(8)       end if;
(9)   end do;
(10)  do until no inference
(11)      apply all other implication formula;
(12)      if PSA deduced then
(13)          temp_current_PSA <== deduced_PSA;
(14)      else
(15)          no inference;
(16)      end if;
(17)  end do;
(18)  if no deduced PSA then
(19)      inference failed;
(20)  else
(21)      record temp_current_PSA;
(22)  end if;

```

The current PSA is taken from the FIFO queue of current PSA; the recorded PSA is placed at the end of this queue if it is not a duplicate PSA. The failure of inference invokes expansion as specified earlier.

5.4.2 General Organization of Tool

The Prolog implementation of our deadlock analysis tool is composed of primarily four subsystems: the PSA manager, the GOL language storage manager, the inference engine and the task animator. The PSA manager provides access to Prolog dynamic assertions that represent the FIFO queue of current PSA and the

recorded PSA; pointers between deduced PSA are maintained. The COL language storage manager, provides access to the COL language statements specifying the program under analysis. These language statements include: connective implication formulæ Ada language implications and the initial PSA; the Ada language implications are provided automatically by the tool. The inference engine uses the current PSA and the implication formulæ to generate new PSA; it implements the deduction and expansion procedures that produce new PSA. Finally, the task animator is used to generate an operational interpretation of the path of deduction used to derive a system-wide deadlock.

5.4.3 Examples

A suite of design examples presented in Appendix C is used to study the behaviour of the deadlock analyzer. The key designs of interest are the Five Dining Philosophers (cref. § C.2), the Remote Mailbox System (RMS) (cref. § C.5) and Ten-Task (cref. § C.7). In this section, we present only the results of the analysis since detailed descriptions of the experiments are found in the appendix.

The dining philosophers and RMS designs are presented in two variations for deadlock analysis: one with deadlock and one without. The deadlock in the RMS design was discussed in chapter 2. In both examples, a system-wide deadlock is identified by the deadlock analysis tool. The corrected versions of the designs were shown to be system-wide deadlock-free. Of more interest is the number of PSA recorded and total execution time required to prove that correct systems are deadlock-free. A summary of these results is presented in Table 5.1; also included are several other design examples.

In the case of the dining philosophers and RMS-examples, a considerable level of non-determinism in the programs produced a great many PSA and consequently a lengthy execution time (1-2 hours). Although a factor of 10 improvement could be gained with modest improvements in hardware and software, it is apparent that the basic computational complexity is the culprit. As an experiment, we divided the RMS design into two halves, RMS-Link and RMS-Service. Each half used a simulation of the missing half to re-create the total system. The results indicated

Name	#Task	#PSA	Time
Diner	6	1137	7589
Diner-Dead	6	439	1937
RMS	10	1356	11048
RMS-Dead	10	1978	17736
RMS-Link	6	66	158
RMS-Service	5	38	75
Readers	4	36	29
Alter-Bit	7	150	360
Gas-Station	4	60	58
Dialogue	5	70	183
10 Task	10	4	11

All times quoted in seconds

Table 5.1: Summary of Deadlock Analysis Results

a considerable savings by partitioning the analysis. This experiment represents an application of the *Divide and Conquer* analysis heuristic which will be described in more detail later in chapter 9. Other practical concerns of computational complexity also will be addressed in chapter 9.

The power of the COL language to permit fast analysis of task deterministic programs is illustrated by Ten-Task. This design consists of ten tasks which behave in a completely task deterministic manner. The deadlock analyzer proves that it is system-wide deadlock-free using 4 recorded PSA in 11 seconds. This substantial performance was obtained without any pre-processing costs.

Several other designs, smaller by number of tasks, or where there is a lower level of non-determinism, are included for comparison. These include: the readers and writers example, Dialogue, Gas-Station and an alternating bit protocol implementation.

5.5 Decidability

The decidability of the deadlock analysis tool is resolved by answering the two following questions:

1. Will the analysis algorithm terminate if a system-wide deadlock can be derived?
2. Will the analysis algorithm terminate if a system-wide deadlock does not exist?

If the response to both of these is yes then the analysis algorithm is *decidable*. If the response to the question (1) is yes, but no to question (2), then the algorithm is *acceptable*. If the answer to both questions is no, then we do not have a useful tool.

We have already addressed these questions to some extent in § 5.3.2; however, in this section we conclude that our analysis algorithm is decidable if the data space of the program under analysis is finite and acceptable, otherwise. These conclusions will follow from the answers provided for the decidability questions, above.

Termination If Deadlock Derived — We have shown in § 5.3.1 that the deadlock analysis algorithm *can generate* all possible task deterministic programs that represent the program under analysis; furthermore, we have shown that the inference engine will terminate if a PSA is deduced that implies system-wide deadlock. What we must determine is whether the analysis algorithm *will generate* all possible task deterministic programs, or not. This problem reduces to the following:

Will the analysis algorithm explore all paths of the PSA-tree (which defines all task deterministic programs)?

Exploration in this context implies the generation of the PSA-tree branches.

Our breadth-first search procedure of the analysis algorithm explores all deduction paths simultaneously; therefore, if there exists a deduction path that derives a system-wide deadlock, then the breadth-first search procedure will encounter

it. By contrast, the equivalent situation analyzed using a depth-first would not necessarily discover the path terminating in deadlock if it began to follow another non-terminating deduction path. If all paths terminated then both exploration strategies would be suitable.

Termination If No Deadlock — In § 5.3.2 we stated that the inference engine would terminate if we did not explore duplicated deduction paths and the data space of the program was finite. The supporting argument for this statement is as follows:

- the control state space of the program under analysis is finite (each task has a finite number of language statements);
- the data state space is finite (assumed);
- the task state space for any task is finite (i.e. the set of combinations of control and data states are finite);
- the program state space is finite (i.e. the set of N-tuples that represent all combinations of task states for N tasks, is finite);
- a finite program state space implies a finite set of unique PSA;

The constraint of a finite data space results in a finite PSA-tree; since the PSA-tree is bounded then the breadth-first search procedure will exhaustively explore all of the tree and then terminate. If the data space of the program is infinite, then we cannot guarantee that the PSA-tree is finite; thus, if the analysis algorithm is applied to a program in which a system-wide deadlock cannot be derived, then the algorithm may not terminate.

5.6 Computational Complexity

The computational complexity of the deadlock analysis algorithm is of interest only if we consider input programs for which the algorithm is decidable. In this case, the computational complexity is equivalent to the cost of exploring the finite

PSA-tree. The cost of exploring the PSA-tree is the effort required to produce each node (PSA) of the tree; therefore, the cost is proportional to the number of PSA in the tree. The number of PSA is bounded by the finite program state space; thus, we shall use the size of the program state space as an estimate of the number of PSA.

The size of the program state space is equal to the set of N-tuples that express all combinations of task states for N tasks. If T_i^* represents the set of all difference task states for task i , then the program state space, P_S , is defined as:

$$P_S = T_S^1 \times T_S^2 \times \dots \times T_S^N$$

The worst case estimate for P_S would be if all T_i^* were the same size as the largest task state set, T_S^* ; in this case, we would have:

$$P_S = (T_S^*)^N$$

Since T_S^* is a constant, we observe that the program state space increases *exponentially* with the number of tasks in the program. Therefore, we conclude that the deadlock analysis algorithm has an exponential computational complexity relative to the number of tasks.

Chapter 6

A Starvation Analysis Method

In this chapter, we describe a starvation analysis method based on: (1) the program state assertions (PSA) generated by the deadlock analysis of chapter 5, (2) a set-theoretic test for starvation based on Kwong's finite delay property [62] and (3) an optimized generate and test analysis method [52]. The discussion begins with a description of starvation, a generate and test approach to analysis and an interpretation of starvation for Ada programs. Given this basis, a set-theoretic starvation test and repeatable sequence generator are defined. These components constitute a basic analysis algorithm. Further development of this result yields an optimized analysis algorithm. Finally, issues of implementation (with examples), decidability and computational complexity are addressed.

6.1 A Description of Starvation

In this section, we develop a description of starvation in Ada programs. As a foundation, we begin by reviewing a characterization of starvation given by Kwong [62] and identify the generate and test analysis approach used for the remainder of this chapter. Following this, we recast Kwong's efforts in an Ada perspective.

6.1.1 Kwong's Starvation

A *computation* is defined to be a sequence of successive program states, where a program state is the n -tuple of task control and data states. For simplicity, all computations are presumed to be infinite sequences of a finite set of program states. A *fair* computation is one in which any continuously enabled task is eventually executed; that is, the execution scheduler will never delay the execution of an enabled task indefinitely. Any scheduler that generates only fair computations is said to satisfy the *finite delay property*, i.e. it is a *fair scheduler*¹.

If a program is assumed to be deadlock-free, then starvation is described as follows:

If there exists a fair computation in which some task is never enabled, we say that the task is experiencing starvation and that the computation is a "starving computation".

Any starving computation is necessarily a repeating sequence of program states (since there are only finitely many program states to represent an infinite sequence of program states).

We weaken the deadlock-free assumption to *system-wide deadlock-free* since the deadlock analyzer of chapter 5 can prove only absence of system-wide deadlock. The implication on the starvation definition is that if we identify starvation, we are detecting either real starvation or *local deadlock*. We leave the reasoning of the cause of the detected starvation with the human operator since we are interested in both anomalies.

Kwong suggests two different procedures to detect starvation: (1) for deterministic programs, it suffices to show that the program is deadlock-free and (2) an approach based on Floyd's Well-Ordered Sets. Rather than using his techniques, we state an algorithm for detecting starving computations based on the *generate and test* paradigm [52]:

1. generate a repeatable sequence of successive program states;

¹Other perceptions of fairness can be found in a detailed study by Frances [39].

2. test if this sequence could be produced by a fair scheduler and if there is some task that is never enabled;
3. if the test is valid then a computation composed of an infinite repetition of the sequence would be a starving computation.

The generate and test approach to analysis is usually inefficient when the problem space is large. This is the case for starvation analysis, however, later in this chapter we give optimizations that can substantially reduce the cost of sequence generation.

6.1.2 An Ada Interpretation

As a first step towards an Ada interpretation, we introduce an alternate representation of a computation: an infinite sequence of successively deduced program state assertions (PSA). Each deduced PSA describes an allowable, deterministic progression of an Ada program. The deadlock analyzer of chapter 5 produces a PSA-tree which defines the deductions which represent all possible, different program behaviours; thus, each path along the PSA-tree describes a unique, deterministic program behaviour. The PSA-tree represents an infinitely executing program with a finite set of program behaviours, therefore, an infinite sequence of deduced PSA can be created by repeatedly connecting finite sequences (we discuss this in more detail in § 6.2.2). We show that a sequence of successively deduced PSA represents a computation below.

- According to the Ada language reference manual, each task is assumed to have its own logical processor; thus, there is no unfairness due to interleaving in an equivalent interleaved model of task execution.
- Any computations which differ only by the interleaving of independent task executions are equivalent; i.e. the program is task deterministic.
- We need consider only one of the equivalent computations for analysis.
- The sequence of successively deduced PSA represents one of the equivalent computations.

Therefore, a sequence of successively deduced PSA can serve as a computation.

Given this alternate representation of a computation, what is a "starving computation" and what is a "fair computation"? Before answering these questions we digress to clarify the requirements of proving starvation in some application program.

In order to prove starvation in an application program, we must reason on the whole system: *application + computing model*. If analysis demonstrates starvation in a system, then we can state that it originates with the application only if we assume that the computing model upon which an application is constructed, does not in itself introduce starvation; that is, starvation occurs in a computation because of the application, not the computing model. This notion is expressed by the statements of "fairness" that accompany our discussion on starvation; i.e. any computation that introduces starvation because of the computing model, is unfair. We shall see shortly, that the assumption of a starvation-free Ada computing model can be demonstrated.

Returning to our earlier questions, the definition of a starving computation seems relatively intuitive: if there exists some task that is waiting for a rendezvous that cannot take place over some computation and there is no deadlock in the system, then there must be some task that is never given an opportunity to rendezvous². From this definition we say that a fair computation is one in which the Ada computing model does not create such a situation; thus, we must examine the behaviour of the Ada rendezvous scheduling mechanisms in order to satisfy ourselves that only fair computations exist³.

We identify three rendezvous schedulers in Ada: the task call scheduler, a scheduler of multiple callers on a task entry and a scheduler of select alternatives in a selective accept. As stated earlier, the notion of a task execution scheduler does not exist in the Ada language definition, although there may be a scheduler

²We assume implicitly that non-concurrency related elements of the program do not introduce indefinite delays; e.g. no endless loops.

³A reader willing to accept fairness in the Ada language may skip ahead to the end of this section.

in the underlying run-time system.

Task call scheduling occurs when a task call is made with a delay or else alternative. In the case of a delay alternative, if there is no acceptor for a call within some period of time, then the task call scheduler chooses to execute the delay alternative. If there is an acceptor waiting then the task call will proceed. We treat the else alternative as equivalent to a delay alternative of zero time. The task call scheduler generates fair computations because:

1. if the acceptor waits until it receives a task call and there is only one caller for the acceptor and the caller infinitely often executes the task call, then the rendezvous must eventually take place;
2. if there are multiple callers, or the acceptor does not wait, then there is no guarantee that the rendezvous will ever occur, but neither the caller or acceptor are ever forced to wait forever.

Since the present implementation of the deadlock analyzer does not permit delay or else alternatives this argument is academic; however, it is included for completeness.

For the second scheduler, the Ada language reference manual states that callers on a single entry are processed using an FCFS discipline. Pnueli and DeRoever [85] demonstrate that this is a fair scheduling method to all waiting callers; thus, any callers that wait until they are accepted on an entry that is accepted infinitely often, will rendezvous with the acceptor task, eventually.

The Ada reference manual does not specify how to select an entry of a selective accept if there are callers on different open entries. Furthermore, a selective accept may have a delay or else alternative analogous to the task call. Let us assume that the alternative selection mechanism is fair; i.e. if a selective accept is executed infinitely often, then any continuously open entry with a caller waiting must be selected for rendezvous, eventually. Under this assumption, the selective accept generates fair computations for any caller that waits. Furthermore, for any callers that do not wait and the selective accept contains a delay or else alternative then both the caller and acceptor tasks will continue to execute. The deadlock analyzer

does not support else or delay alternatives for selective accepts; again, they are included for completeness.

With the notions of fairness established, the starvation analysis algorithm using our new representation of a computation is:

1. identify a repeatable sequence of successively deduced PSA;
2. test if this sequence could be generated by fair rendezvous schedulers and if there is some task call for which there is no rendezvous;
3. if the test is valid then a computation composed of an infinite repetition of the sequence would be a starving computation.

We can consider the definitions of starvation strictly from the caller's perspective because we are not allowing delay or else alternatives in task calls; i.e. the Ada rendezvous schedulers select callers only, never acceptors. The only side effect of this decision will be the inability of the analyzer to detect local deadlock in an acceptor. Intuitively, this shouldn't happen because, in principle, there are callers for every entry, thus an error would be detected from the caller's perspective. If there were no callers then the design is structurally flawed. The other possibility, a completely closed selective accept, would also be a rather significant error to overlook during specification. Although the theory could be extended to accommodate this circumstance, it would introduce additional complexity.

6.2 A Generate and Test Starvation Detector

In this section, we describe the basic generate and test starvation analysis algorithm. We first define a set-theoretic starvation test for Ada programs. This test is based on the notion that, over some repeatable sequence of successively deduced PSA, there is a set of rendezvous that must occur (because of task calls), a set of rendezvous that should occur (because of fairness) and a set of rendezvous that actually occur. Should the first set less the other two sets be non-empty, then there exists some rendezvous that is never executed. The partner tasks in this

rendezvous are considered starving if the repeatable sequence is considered fair. The sequence is fair if the set of rendezvous that should occur is contained in the set of rendezvous that are done.

The repeatable sequence generator is based on depth-first tree traversal of the PSA-tree produced by the deadlock analyzer. For the sake of efficiency, we compact the information in the PSA-tree in order to search a smaller (but equivalent) tree.

6.2.1 A Set-Theoretic Starvation Test

The PSA-tree illustrated in Figure 6.2 represents the PSA generated by the deadlock analyzer when applied to the Readers and Writers design (2 readers, 1 writer), with starvation that was presented earlier in chapter 3, § 3.1.2 and is also summarized in Appendix C.1 with the complete COL specification provided in Appendix D. As a convenience, we reproduce the structure diagram for this example in Figure 6.1. The relationship between this design and its PSA-tree is not crucial to the understanding of the examples that follow; it exists only to add a realistic flavour to the discussion.

The deadlock analysis of chapter 5 generates all possible program behaviours (new branches in the PSA-tree) whenever a selective accept, or single accept is encountered. We refer to this process as *expansion* (cref. § 4.2.7 and § 5.3.1). The PSA to which expansion is applied are called *expansion points*. The directed paths in the PSA-tree point to successively deduced PSA; each PSA is unique and labeled. Expansion points are identified as those PSA with multiple successors; for example, PSA 2, 6, 14 and 7 are expansion points. The actual PSA for 2, 3, 4 and 6 are shown in Figure 6.3. As a notational shorthand, we abbreviate the labels of the design elements as follows:

Reader(1)	— r1	Req_read	— rr
Reader(2)	— r2	Req_write	— wr
Writer(1)	— w1	End_acc	— ea
Scheduler	— s		

The deadlock analysis algorithm invokes expansion because no other implications can be applied to the expansion point PSA. When expansion is applied, each

Figure 6.1: Readers and Writers Example: One Writer

new branch represents exactly one accept matching a waiting call; that is, one caller from one entry, or, one caller from one entry of one selective accept (since there may be more than one selective accept to expand); for example, PSA 3, 4 and 5 in Figures 6.2 and 6.3, each contain a unique *start_accept* for which there existed a *task_call* in the expansion point, PSA 2.

Given our PSA-tree model of program behaviour, we define below its rendezvous characteristics in set-theoretic terms.

Definition 6.1 The *option-to-do* set, O , of a successor PSA of an expansion point is the set of new rendezvous (each on a different branch) generated by the expansion process. For instance, PSA 3, 4 and 5 each have the same O set because they have the same predecessor. The rendezvous in the O set of PSA 3 are shown in Figure 6.4. Any PSA that was not generated from an expansion, such as PSA 6, has an empty O set. (Note: all rendezvous, calls or accepts are represented as (caller,acceptor) pairs when used as set elements.)

Definition 6.2 The *is-done* set, I , is the set of rendezvous implied by changes in control state predicates from *start_accept* in one PSA to *start_rendez* in the successor PSA; that is, the set of rendezvous that will occur. For example, in Figure 6.4, the I set of PSA 3 includes one rendezvous; that is, assertion 3 infers one rendezvous, as evidenced in PSA 6.

Figure 6.2: PSA-Tree for Readers and Writers

PSA 2:

```
*<>(task_call(r1,r1!l1,s!rr) & ts(...))
*<>(task_call(r2,r2!l1,s!rr) & ts(...))
*<>(task_call(w1,w1!l1,s!wr) & ts(...))
*<>(sel_accept(s,s!1) & ts(...))
```

PSA 3:

```
*<>(task_call(r1,r1!l1,s!rr) & ts(...))
*<>(task_call(r2,r2!l1,s!rr) & ts(...))
*<>(task_call(w1,w1!l1,s!wr) & ts(...))
*<>(start_accept(s,r1!l1,s!rr) & ts(...))
```

PSA 4:

```
*<>(task_call(r1,r1!l1,s!rr) & ts(...))
*<>(task_call(r2,r2!l1,s!rr) & ts(...))
*<>(task_call(w1,w1!l1,s!wr) & ts(...))
*<>(start_accept(s,r2!l1,s!rr) & ts(...))
```

PSA 6:

```
*<>(task_call(r1,r1!l2,s!ea) & ts(...))
*<>(task_call(r2,r2!l1,s!rr) & ts(...))
*<>(task_call(w1,w1!l1,s!wr) & ts(...))
*<>(sel_accept(s,s!1) & ts(...))
```

Note: data state predicates not shown

Figure 6.3: Program State Assertions (PSA) for Readers and Writers

$$\begin{aligned}
M_2 &= \{ (r1!l1, s!rr), (r2!l1, s!rr), (w1!l1, s!wr) \} \\
I_2 &= \{ \phi \} \\
O_2 &= \{ \phi \} \\
M_3 &= \{ (r1!l1, s!rr), (r2!l1, s!rr), (w1!l1, s!wr) \} \\
I_3 &= \{ (r1!l1, s!rr) \} \\
O_3 &= \{ (r1!l1, s!rr), (r2!l1, s!rr), (w1!l1, s!wr) \} \\
M_6 &= \{ (r1!l2, s!ea), (r2!l1, s!rr), (w1!l1, s!wr) \} \\
I_6 &= \{ \phi \} \\
O_6 &= \{ \phi \}
\end{aligned}$$

Figure 6.4: Assertion Sets M , O , I of PSA 2, 3, 6

$$\begin{aligned}
M_g &= \{ (r1!l1, s!rr), (r2!l2, s!ea), (r2!l1, s!rr), (w1!l1, s!wr) \} \\
I_g &= \{ (r1!l1, s!rr) \} \\
O_g &= \{ (r1!l1, s!rr), (r2!l1, s!rr), (w1!l1, s!wr) \}
\end{aligned}$$

Figure 6.5: Grand Sets M_g , O_g , I_g Of Sequence (2,3,6)

Definition 6.3 The *must-do* set, M , is the set of all rendezvous implied by *task_call* control state predicates within a PSA; that is, all rendezvous that should occur (because the program has been shown to be deadlock-free). In Figure 6.4 we find that PSA 2 has a M set of 3 elements.

Definition 6.4 Given some sequence of PSA, σ , we define the *grand sets* of the option-to-do, is-done and must-do sets, as the unions of the O , I and M -sets over all of the PSA in σ . These grand sets are designated O_g , I_g and M_g , respectively. For instance, the grand sets of the PSA sequence (2,3,6) are shown in Figure 6.5.

With these four definitions in hand, we describe below our set-theoretic approach to starvation detection.

Suppose that we have generated a repeatable sequence of successive PSA, σ ,

for example, the PSA sequence (2,5,8,13,18,2) from Figure 6.2 forms a valid σ_r . There are two conditions on a σ_r that must be true if it represents a starving computation in a deadlock-free program: (1) within the σ_r there must exist some task that executes a task call for which there is no rendezvous and (2) the σ_r must represent a fair computation. The set-theoretic interpretation of these conditions is provided below.

Cond. 1 The M_r set of a σ_r is the set of all task calls that are executed over σ_r . The set I_r is the set of all rendezvous that occur over the σ_r . Thus, if there are more task calls in M_r than rendezvous in I_r , then there exists some task call that is executed for which no rendezvous occurs; i.e. $I_r \subset M_r$, or $|I_r| < |M_r|$.

Cond. 2 Each expansion point of a σ_r represents a selective accept or a single accept that is executed with the same data state, infinitely often. According to our discussion on fairness in § 6.1.2, any task call on such a selective accept or single accept must rendezvous eventually. The O_r set is the set of all such task calls. If there exists a task call in O_r for which no rendezvous occurs in a σ_r , then the σ_r cannot be an infinitely repeating sequence representing a fair computation; i.e. eventually a branch leaving the σ_r would have to be taken in order to satisfy the task call. For instance, if two tasks called on a single task entry that is accepted infinitely often, then eventually both task must rendezvous with the entry because of fairness in Ada's FCFS acceptance scheduling algorithm. Any repeating sequence in which one task is deliberately ignored, cannot be an infinitely repeating sequence. In summary, a repeating sequence representing a fair computation must have each task call in O_r satisfied by a rendezvous in I_r ; i.e. $O_r \subseteq I_r$.

Three example σ_r from Figure 6.2 are shown in Figure 6.6: σ_r^1 represents an unfair computation with starving tasks (cond. 1 = true, cond. 2 = false), σ_r^2 shows a fair computation without starving tasks (cond. 1 = false, cond. 2 = true) and σ_r^3 defines a fair computation with starving tasks (cond. 1 = true, cond. 2 = true). Below we state formally our theorem of starvation based on sets.

$$\begin{aligned}\sigma_r^1 &= (2,5,8,13,18,2) \\ \sigma_r^2 &= (2,3,6,10,15,2,4,7,12,17,2,5,8,13,18,2) \\ \sigma_r^3 &= (6,9,14,19,21,7,11,16,14,20,22,6)\end{aligned}$$

Figure 6.6: Example σ_r from Readers and Writers

Theorem 6.1 *Given some σ_r , we test for starvation as follows:*

if $(|M_g| > |I_g|) \wedge (O_g \subseteq I_g)$ then
starvation exists;
 $R = M_g - I_g$;
else
no starvation;
end if

where R defines the set of rendezvous for which task calls occur, but not accepts.

Proof: If $|M_g| = |I_g|$ then there are no rendezvous that must be done that are not done; i.e. there is no starvation.

If it is not true that $O_g \subseteq I_g$, then it is true that $\exists e (e \in O_g \wedge e \in M_g \wedge \neg e \in I_g)$ which implies that there is some rendezvous that must take place that is being unfairly ignored by the rendezvous schedulers; i.e. a violation of the finite delay property.

The set of rendezvous that represent starving callers is $R = M_g - I_g$, because R is the set of all rendezvous for which task calls occur but no acceptors respond. Q.E.D.

Corollary 6.1 *The test $|M_g| > |I_g|$ is equivalent to showing that not all tasks are active over a σ_r .*

Corollary 6.2 *The test $|M_g| = |I_g|$ is a sufficient condition for proving absence of starvation in a σ_r .*

Corollaries 6.1 and 6.2 are fundamental to the optimization methods that appear later in § 6.3.

6.2.2 A Repeatable Sequence Generator

All sequences of successive PSA are represented by the PSA-tree generated by deadlock analysis. A σ_r could be generated by starting at any PSA and traversing the tree successor by successor until the starting PSA is encountered. Unfortunately, there is no way to know when all possible σ_r have been explored. A more methodical starting PSA selection approach is required. To discover this approach we return to the PSA-tree of the Readers and Writers Example in Figure 6.2.

In Figure 6.2, dashed arrows, called *join branches* indicate *join points*, j_p , between paths; that is, PSA with the same successor PSA. There are two types of j_p :

- those that represent converging but otherwise distinct sequences;
- those that represent a repeating occurrence of some PSA within a single sequence, which we shall regard as *looping points*, l_p .

We define *loop branches* to be those join branches that terminate on l_p . The loop branches to l_p are a notational convenience for describing the infinite PSA-tree structure in finite terms; they do not imply that the program can "go backwards in time".

Looping points are the starting PSA for σ_r generation because they are the only PSA that the program "branches back to". In effect, they are the "GOTO" labels of the PSA-tree. Thus, any repeatable sequence must include a l_p . This is summarized below, formally.

Theorem 6.2 *A sequence of successive PSA, σ_r , is a σ_r if and only if there exists some l_p, x , such that x is an element of σ_r .*

Proof: If x is a l_p , then there exists a directed path from x , traversing some loop branch that returns to x . Such a path is a σ_r ; i.e. $l_p \dots l_p$.

If σ_r is a repeatable sequence, σ_r , then there exists at least one PSA in σ_r that is the successor PSA of a loop branch. By definition, such an PSA is a l_p . Q.E.D.

Corollary 6.3 *The set of l_p in a PSA-tree can be used to generate all σ_r represented by the tree.*

Given suitable starting points for σ , generation, any of a variety of tree traversal methods [109] can be used. However, we can reduce the magnitude of the traversal by encoding the PSA-tree into sub-sequences; i.e. process collections of PSA rather than individual PSA.

We group sequences of successive PSA such that the first and last PSA are exclusively j_p and the only j_p of the sequence. We define these sequences to be *elementary sequences*, σ_e . The motivation for selecting j_p as the partitioning points is that by definition, computations converge and diverge through j_p only. We include also l_p as partitioning points because a repeatable sequence is part of a computation that converges on itself. The σ_e of the PSA-tree of Figure 6.2 are shown overlaying the tree in Figure 6.7. The σ_e of the original tree also form a tree, where the nodes are j_p and the branches are σ_e ; the σ_e from Figure 6.7 are shown in a tree format in Figure 6.8. Formally, we represent an σ_e tree branch by a 7-tuple:

$$\sigma_e = (n, first, last, seq, M_e, O_e, I_e)$$

where n is a unique id, *first* is the id of the first PSA in the sequence, *last* is the id of the last PSA in the sequence, *seq* is the sequence of PSA id's representing the successive PSA and M_e , O_e , I_e are the unions of the M , O , I sets over all of the PSA represented by *seq*. Given this definition, we state an equivalent calculation of the grand sets M_g , O_g , I_g (from § 6.2.1) to be the union of the M_e , O_e , I_e sets over all σ_e that comprise some σ .

We can construct computations from elementary sequences by concatenating σ_e such that the last assertion of a tail σ_e matches the first assertion of the successor σ_e . From the perspective of the σ_e tree, this is equivalent to selecting some starting l_p , following the branches of the tree and concatenating the sequences encoded within each branch, until the target l_p is reached. For example, in Figure 6.8, if we want to find a computation from l_p 2 back to itself, then the σ_e , 2...7 and 7...2, can be concatenated to form the computation 2...7...2.

A σ can be generated by applying a conventional *depth-first traversal* procedure [109] to the σ_e tree, using the l_p as starting and terminating nodes; however, since the σ_e tree is actually an infinite tree, this algorithm could execute forever.

Figure 6.7: PSA-Tree With σ_e Overlaid

Figure 6.8: A σ_i Tree

In order to force the search to stop, we require it to consider only those branches that are as yet untraversed; if it should not find any such branches then it should backtrack to find an alternate path (this issue will be dealt with in more detail when we discuss decidability issues in § 6.5). This limitation does not affect our starvation test because the grand sets do not change values with the union of duplicate elementary sequence sets (because $A = A \cup A$).

6.2.3 A Basic Analysis Algorithm

The basic starvation algorithm can now be expressed in terms of σ_i generator from § 6.2.2 and the set-theoretic starvation test from § 6.2.1:

```

for all looping points do
  loop until no more paths to search
    generate next computation;
    generate grand sets  $M_i$ ,  $O_i$ ,  $I_i$ ;
    test grand sets for starvation;
    if starvation then

```

```

        record starvation;
        backtrack to previous node in search;
    else
        continue generation from current node;
    end if;
end loop;
end for;

```

Although this algorithm searches a compactly represented PSA-tree, it is still impractical for the analysis of all but the simplest designs. In the next section, we introduce optimization to significantly improve this algorithm.

6.3 Optimization

Optimizing the generation of σ , involves reducing the scope of the σ , tree traversal; that is, identifying paths that will not yield successful results; without actually traversing the entire path. The approach we use is to shape our pure generate and test algorithm into a *generate with incremental test and final test* algorithm; that is, test partial σ , as they are generated in order to prune traversal paths if the test fails. The final test would be applied only if a σ , passed all of the incremental tests. The incremental test we use is based on Corollaries 6.1 and 6.2 from § 6.2.1:

The test $|M_\sigma| > |I_\sigma|$ is equivalent to showing that not all tasks are active over a σ .

The test $|M_\sigma| = |I_\sigma|$ is a sufficient condition for proving absence of starvation in a σ .

From these corollaries we deduce that any partial σ , that can be shown to have all tasks active implies that the complete σ , is starvation-free.

We define a *grand active task set*, A_σ , to be the set of all tasks that are proposed to be active in some σ (of which a partial σ , is an instance). The A_σ set would be continually updated and tested as each σ , is concatenated to a partial σ ,. In summary, the optimization technique is as follows:

1. associate an active task set, A_i , with each σ_i tuple (thus extending it from a 7-tuple to an 8-tuple); this set represents all tasks that are, or eventually will be, active as a result of including an σ_i as a subsequence of a σ_j ;
2. as the sequence of each σ_i is added to the partial σ_j (during the σ_j tree traversal), union the A_i set with the A_j set;
3. if at any time $|A_j|$ is equal to the number of tasks in the system then the path could not possibly contain any starving tasks, therefore, any further generation of the current σ_j is unnecessary.

Given the basic optimization technique, we turn our attention to the construction of the A_i sets.

Intuitively, the first source of active tasks for an A_i set is the partner tasks of rendezvous specified by the corresponding I_i set, since we have already concluded that these tasks must be active. As it turns out, partner tasks of the task calls specified by the O_i set can also be included. We justify this below.

The intent of optimization is to prune the path generating a σ_i . We naturally assume that any partial σ_i leads to a representation of a fair computation. We verify this assumption as part of the final test for starvation. If we eliminate σ_i that imply unfair computations by the incremental test then we have violated our assumption. Nonetheless, we are pleased because σ_i defining unfair computations would have been rejected by the final test anyway.

Since we are assuming that partial σ_i lead to fair computations, then any task calls in the O_i set of a concatenated σ_i must rendezvous eventually (cref. § 6.2.1); i.e. they must become active, eventually.

The third source of active task information is not present explicitly in the current σ_i representation: If we know the deterministic successor rendezvous of a task call the O_i set of some σ_j , or the rendezvous of the I_i set, then the partner tasks in the successor rendezvous must be active. We justify this as follows:

Figure 6.9: Design Illustrating Deterministic Successor Rendezvous

if some rendezvous, r , is executed eventually, (as is the case for task calls in the O_s set and rendezvous in the I_s set), then any rendezvous, r_d that follows deterministically from r also must be executed, eventually.

We motivate the need for this A_s source by the design segment illustrated in Figure 6.9.

In this design segment, task A repeatedly calls B.E1 followed by an accept of E3, task B repeatedly selectively accepts E1 and E2 and task C repeatedly calls A.E3. Furthermore, there are other tasks competing for the entries of task B. In the construction of a σ_s , it may be possible that many σ_s have to be concatenated before an σ_s , say σ_s^* , is encountered in which the rendezvous (A,B.E1) is shown to occur; that is, it is an element of an I_s set. This does not affect our prior knowledge that (A,B.E1) must occur eventually because it would have been represented in the O_s set of some earlier σ_s .

If we consider the rendezvous (C,A.E3), we find that it is only a member of M_s sets in σ_s up to σ_s^* since it is the sole caller of a single accept (and therefore does not appear in O_s sets produced by expansion). Thus, we cannot deduce that all tasks are active prior to σ_s^* (using only the O_s and I_s sets as active task sources) because task C cannot be considered active until after σ_s^* .

If we know that (C,A.E3) follows deterministically from (A,B.E1) then we know task C becomes active eventually if we can show that (A,B.E1) occurs eventually; a premise which is established by the O_s sets of earlier σ_s . We describe formally

the construction of the A_r set by the theorems below.

Theorem 6.3 *All tasks that are partners in rendezvous that are members of an I_r set are members of the A_r set.*

Proof: A rendezvous implies that both partners are active at least once in a sequence; however, since we assume that the sequence is indefinitely repeating, then both tasks are active infinitely often. Q.E.D.

Theorem 6.4 *All tasks that are partners in rendezvous that are members of an O_r set are members of the A_r set.*

Proof: Assume that they are not members; i.e. they are not now, nor will they ever be, active tasks. If they are not active tasks then they will never be members of the A_r set, but they will be members of the O_r set; however, this implies that $\exists c (c \in O_r \wedge c \in M_r \wedge \neg c \in I_r)$ is true, which means that the repeating sequence represents an unfair computation (cref. § 6.2.1). If we assume that they are members, (thus, they are, or eventually will be, active tasks), then the proof applied for theorem 6.3 is valid. Q.E.D.

Theorem 6.5 *All tasks that are partners in rendezvous that are deterministic successors to any rendezvous in the O_r or I_r sets are members of the A_r set.*

Proof: We have shown by theorems 6.3 and 6.4 that the rendezvous in the O_r and I_r sets are, or eventually will be, active. Thus, any rendezvous that follow deterministically must also occur. Therefore, the partner tasks are, or eventually will be, active. Q.E.D.

The deterministic successor information is actually a form of *lookahead*: that is, we deliberately encode knowledge about later branches in the tree in order to simplify the search. Fortunately, this information is obtained at a cost linear to the cost of the deadlock analysis since it is a relatively simple lookahead of one rendezvous in any task of the system. A more sophisticated lookahead may introduce too high an overhead to justify its value.

6.4 Analysis Tool Implementation

In this section, we describe relevant features of the MProlog implementation of the starvation analysis procedure. Our primary concern is to identify what inputs are required from the deadlock analysis, what pre-processing is required to generate the σ , and the approach we use to integrate efficiently the optimization theorems into the basic algorithm of § 6.2.3.

6.4.1 Input From Deadlock Analysis

The deadlock analysis tool of chapter 5 provides the information required to generate the σ , 8-tuples. Recall that the σ , 8-tuple is

$$\sigma = (n, first, last, seq, M_s, O_s, I_s, A_s).$$

The first 5 fields are derived directly from the PSA and successor PSA data recorded by deadlock analyzer; however, the information required to compute the O_s , I_s and deterministic successor rendezvous (used in the calculation of A_s) is not readily available. Gathering this information requires the recording of additional data, by the deadlock analyzer, as described below.

Is-Done Set: The *start_rendez* control state predicates occur only in *temporary* PSA, rather than the permanently recorded PSA produced by the deadlock analyzer (cref. § 5.4). To compensate, we record with each predecessor PSA, (in the PSA-tree), the set of rendezvous that are inferred in the the PSA's successor. This set is exactly the I set from § 6.2.1.

Option-To-Do Set: The O set of a PSA can be generated from the predecessor PSA and the sibling PSA; however, this is a cumbersome calculation. Instead, we record explicitly the O set when the PSA are generated.

Deterministic Successor Rendezvous: The successor rendezvous of a given rendezvous can be identified from the PSA, but such an approach would be equivalent to searching the PSA-tree, a process we are trying to avoid. An alternate method is to generate successor rendezvous relations each time a task progresses

from one rendezvous to the next. This method essentially "piggybacks" the generation of the successor relations with the tree generation. Additional processing selects the deterministic successor rendezvous from all of the successor relations.

6.4.2 Pre-Processing

The pre-processing phase of the starvation analyzer uses the information produced by the deadlock analyzer to generate the σ_e tuples and identify looping points. The σ_e tuples are asserted as Prolog facts.

Each σ_e tuple is generated, but, they are asserted only if the $|A_e|$ is less than the number of tasks in the program (since there is no value in retaining any σ_e that violate this test, according to theorem 6.1). This condition can lead to a desirable situation in which there are no outgoing σ_e branches from the looping point nodes of the σ_e tree, hence there are no repeating sequences to be generated that could lead to starvation. For instance, a correct implementation of the Five Dining Philosophers problem [57] (see Appendix C.2), requires pre-processing but little analysis.

6.4.3 Starvation Algorithm

The starvation algorithm with optimized search as expressed in MProlog is shown in Figure 6.10. The σ_e 8-tuple is represented by the `elem_seq/8` facts. The repeating sequences (as represented by the σ_e ids) are constructed in both list and fact format in order to eliminate any costs in conversion costs. The fact format is used to test for duplicate path traversal (cref. § 6.2.3) and the list format is used to construct the grand sets for the starvation test. All union operations are performed using a fact-based, rather than list-based [28] implementation of sets. This significantly improves the performance of the $|A_e|$ test and the generation of the grand sets.


```

% generate_and_test(SEQUENCE,START_LOOPING_POINT,MAX_NUM_OF_TASKS)

% Test for end of a repeating sequence;
% if a sequence is found but failed the test then
%   continue generating sequences from the current node;
%   the repeating sequence found thus far and the grand sets
%   that were computed are stored for future use so as to avoid
%   duplicating the effort to generate the grand sets;
% else
%   fail the starvation_test predicate and explore new sequences
%   since any further exploration of the current sequence
%   is unnecessary;
% end if;

generate_and_test([H|T],LOOPING_POINT,NUM_OF_TASKS) :-
    elem_seq(H,_,LOOPING_POINT,_,_,_,_,_),
    starvation_test([H|T]),
    elem_seq(ID,LOOPING_POINT,_,_,_,_,_,A_E),
    not traversed_elem_seq(ID),
    fast_union(A_E,active_task_set),
    test_active_task_set_magnitude(NUM_OF_TASKS),
    add_statement_b(traversed_elem_seq(ID)),
    generate_and_test([ID],LOOPING_POINT,NUM_OF_TASKS).

% Last elem seq did not terminate repeating sequence, thus find
% next elem seq

generate_and_test([H|T],LOOPING_POINT,NUM_OF_TASKS) :-
    elem_seq(H,_,LOOPING_POINT,_,_,_,_,_),
    elem_seq(ID,LOOPING_POINT,_,_,_,_,_,A_E),
    not traversed_elem_seq(ID),
    fast_union(A_E,active_task_set),
    test_active_task_set_magnitude(NUM_OF_TASKS),
    add_statement_b(traversed_elem_seq(ID)),
    generate_and_test([ID,H|T],LOOPING_POINT,NUM_OF_TASKS).

```

Figure 6.10: MProlog Implementation of Starvation Analysis Algorithm

6.4.4 Examples

A suite of design examples presented in Appendix C is used to study the behaviour of the starvation analyzer. The key designs of interest are the Readers and Writers (cref. § C.1), the Five Dining Philosophers (cref. § C.2), the Remote Mailbox System (RMS) (cref. § C.5). In this section, we present only the results of the analysis since detailed descriptions of the experiments are found in the appendix.

The dining philosophers and readers and writers' designs are presented in two variations for starvation analysis: one with starvation and one without. The starvation in the readers and writer design was discussed in chapter 3. In both examples, starvation is identified by the starvation analyzer; specifically, the writer task was shown to be starving in the readers and writers example and all five diners could starve in the dining philosophers design. The corrected versions of the designs were shown to be starvation-free.

The performance of the starvation analyzer is judged by the amount of time spent in pre-processing versus the time spent in actual analysis (searching the encoded PSA-tree). A summary of these results is presented in Table 6.1; also included are several other design examples.

In the case of the dining philosophers and RMS examples, a considerable level of non-determinism in the programs produced a great many PSA and consequently a lengthy pre-processing time (45 minutes); however, the analysis time for the dining philosophers design was very small and the analysis time for RMS was acceptable. The pre-processing time was about 20% of the deadlock analysis time (see Table C.1). In the partitioned RMS experiment, results indicated that the cost of starvation analysis was negligible. A discussion on the practical issues of computational complexity will follow later in chapter 9.

Several other designs, smaller by number of tasks, or where there is a lower level of non-determinism, are included for comparison. These include: ten-task, the readers and writers example, Dialogue, Gas-Station and an alternating bit protocol implementation. These designs were processed by the starvation analyzer in negligible time.

Name	#Task	#PSA	Pre-Proc Time	Analysis Time
Read	4	36	6	< 1
Read-Starve	4	22	3	1
Diner	6	1137	1767	164
Diner-Starve	6	62	18	24
Gas-Station	4	60	11	< 1
Dialogue	5	70	19	< 1
RMS	10	1356	2769	1454
RMS-Link	6	66	16	< 1
RMS-Service	5	38	7	< 1
Alter-Bit	7	150	71	10
10 Task	10	4	1	1

All times quoted in seconds

Table 6.1: Summary of Starvation Analysis Results

6.5 Decidability

To show that the starvation analysis algorithm is decidable, requires a proof that the algorithm will terminate for any given input. Our approach will be to reduce the algorithm to a complete depth-first traversal of a finite tree, a process which is decidable. We argue the proof of decidability informally, below.

Proof of Decidability: The input to the starvation algorithm is a finite set of PSA. These PSA are represented for starvation analysis as an infinite directed tree of a finite number of branches (the σ_i). We can alternately interpret this tree as a finite directed network with cycles. If we construct a path from some given node in the network such that we visit every node and traverse every edge only once, then there are less than $(N - 1)!$ different paths. (The upper bound for a fully connected network of N nodes is $(N - 1)!$ paths, thus a directed network must have an upper bound below this limit). Since we use the depth-first search algorithm to generate test sequences and there are a finite number of sequences, then our algorithm must terminate.

Q.E.D.

6.6 Computational Complexity

The computational complexity of the starvation analyzer is difficult to quantify in a single term because of the optimization methods; however, we can provide complexity bounds on the two steps of the analysis separately.

The complexity of the pre-processing stage is linearly proportional to the cost of the deadlock analysis because the cost of producing the σ_t is linearly proportional to the cost of examining each PSA in the PSA-tree. Since the deadlock analysis is exponential with the number of tasks in the system, N_t , we conclude that the pre-processor also is exponential with N_t .

The complexity of the starvation algorithm is equal to the complexity of the depth-first tree traversal algorithm on which it is based. This complexity is exponential with the number of nodes in the tree. In our case, the number of nodes in the tree corresponds to the number of j_p in the PSA-tree. The cost of producing the j_p is equal to the complexity of the deadlock analysis; thus, the starvation algorithm is also exponential with N_t .

Chapter 7

A Critical Race Analysis Method

In this chapter, we describe a method for identifying the potential for critical races in a COL specification of an Ada program. This method succeeds in proving the existence of the *race condition*, one of two necessary conditions for a critical race. The *assumption condition*, the second condition, is left to be demonstrated by the human operator. The analysis method uses the program state assertion (PSA) tree, or PSA-tree, generated by the deadlock analyzer of chapter 5 to locate instances of the race condition; interactive facilities allow the user to reason the presence of the assumption condition. We present the analysis method in two sections: (1) an overview of the proof of a critical race and (2) a functional description of the analysis tool called the *ND-Analyzer*.

7.1 Proof of a Critical Race

Avoiding critical races in Ada programs requires careful program design; for example, mutual exclusion is guaranteed simply by placing the target resource under the control of a single task (cref. § 3:1.3). Besides mutual exclusion failure, the only other source of critical races are assumptions on the arrival order of caller tasks at an acceptor task. Under the presumption that no critical races exist by violation of mutual exclusion, we identify two necessary conditions for a critical race to exist between two (or more) caller tasks and an acceptor task below.

race condition — the rendezvous between the callers and the acceptor must be non-deterministic; i.e. there must be a race.

assumption condition — the acceptor task must assume that the rendezvous is deterministic; i.e. it assumes a particular caller order.

Taken together, these conditions are sufficient to prove the presence of a critical race.

The detection of the race condition in a COL-specified Ada program is straightforward: every COL program state assertion (PSA) to which we can successfully apply expansion, (cref. § 4.2.7) implies the possibility of a non-deterministic rendezvous. Expansion does not always imply a non-deterministic rendezvous because in some cases, expansion yields a deterministic outcome; that is, operationally, the program may not produce a non-deterministic rendezvous where there exists a structural possibility. For example, two tasks may contain calls to the same entry of a third task, but execute the entry call in a mutually exclusive fashion because of the program logic.

The deadlock analyzer of chapter 5 generates a tree of PSA that represent all possible behaviours of an Ada program. Expansion points in the PSA-tree are those PSA for which there are several successor PSA produced by expansion (cref. § 6.2.1). We can examine each expansion point in the PSA-tree to establish if a non-deterministic rendezvous can occur; i.e. if the race condition holds.

We identify non-deterministic rendezvous in an expansion point as when either of the following conditions exist:

accept race — multiple callers exists for a single entry;

select race — callers exist for at least two open entries in a selective select.

Multiple races may exist within the same acceptor task since an accept race can occur within a select race; for example, if a selective accept has two open entries E1 and E2 and E1 has two callers and E2 has one caller, then there is an accept race between the callers of E1 and a select race between callers of E1 and E2.

Establishing the assumption condition (given that a race condition has been proven) is difficult to automate because we lack a program-independent way of

expressing the condition. An alternate, relatively easy approach is to use program-specific behaviour constraints; for instance, in the agent-pool example of Figure 3.3 of chapter 3, we can state the constraint on the queueing order of free agents as follows:

$$\models \text{last_element}(q_w, t) \wedge \text{last_element}(q_f, t) \quad (7.1)$$

where, q_w is the queue of agents waiting at the Dispatcher.Wait entry, q_f is the queue of ids for agents that have called the Dispatcher.Free entry and not yet allocated, and $\text{last_element}(q, t)$ is true if task t (or its id) is the last element of some queue q . If we can prove that our COL specification violates this constraint then the assumption condition is true. Assuming that such a proof is possible, we are still faced with a significant problem: how do we identify the constraints that should be tested?

We now have two difficult alternatives to proving the assumption condition:

1. develop a general expression of the assumption condition with which we can analyze the COL specification;
2. develop an algorithm to generate problem-specific constraints to be tested against the COL specification.

From an engineering perspective we suggest that it may not be necessary to automatically perform this proof because: (1) if the analysis is performed on an unfinished design (e.g. it may be missing some functional behaviour), then any proof could be invalid and (2) the number of unique types of acceptor tasks is small enough¹ that a human operator could cope with the proof, given that the races were identified. In short, an automatic technique for proving the assumption condition is desirable, but we can still perform a significant degree of analysis in spite of its absence.

¹Our tool-set is suited to designs that are typically less than about 15 tasks.

7.2 The ND-Analyzer

The non-deterministic rendezvous analyzer, *ND-Analyzer*, supports critical race analysis by identifying all expansion point PSA (in the PSA-tree produced by the deadlock analyzer tool) in which the race condition holds. Given this set of PSA, it interactively aids the user in exploring the race conditions and establishing the assumption condition.

In the first phase of analysis, the PSA-tree is scanned; if select or accept race conditions are identified, then they are recorded. In a secondary step of the first phase of analysis, we produce a list of tasks, T_r , in which the race condition holds. If T_r is empty then analysis is terminated, otherwise we proceed to the second phase of analysis: interactive assistance.

The interactive display of race conditions uses a two tier menu system (illustrated in Figure 7.2). In the first level menu we identify one of the tasks in T_r to be investigated. The task to be studied, t_r , may have races in several PSA; the second level menu helps the user to explore races in each of these PSA. Upon selecting t_r , the analyzer presents a *race characterization display* of the first PSA in which t_r has a race; the PSA are explored only in their order of discovery. The race characterization display shows the accept structure of t_r in which the race occurred; this is either a single accept or selective accept structure. All races within the accept structure are denoted with a text flag, ND (since there may be select and accept races within a selective accept, or only an accept race within some entry of the selective accept, or only a select race within a selective accept). The caller tasks that constitute the race are listed.

The second level menu appears after the race characterization display; its options include: display of the current PSA, study of the next PSA appropriate for t_r , return to the first menu and exit of the analysis tool. The display of the PSA presents the operational interpretation (cref. chapter 8) of the PSA currently under study; the second level menu is presented after display is completed. The selection of the next PSA changes the PSA under analysis to the next appropriate PSA. If another PSA exists, then a new race condition display is presented followed by the second level menu. If no other PSA exists then the user is returned

—Figure 7.1: Gas-Station Design Example

to the first level menu.

We illustrate the use of the ND-Analyzer and its various features with a study of the *gas-station* problem from Helmbold and Luckham [53]. The structure diagram for this example is shown in Figure 7.1; the complete design is presented in Appendix C.3. A synopsis of the design is discussed below.

In the gas-station problem customer tasks request a specific gas pump for a specific amount of gas (in dollars) by calling the *prepay* entry of an operator task. If the pump is idle, then the operator task calls the *activate* entry of the requested pump, stores the amount requested in a queue and releases the customer task. If the pump is in use, then the operator queues the amount in the request queue for the pump and releases the customer. The requests are queued first-come, first-served. Regardless of the state of the pump, the customer calls the *start* entry of the pump. A pump always sequentially accepts its *activate*, *start* and *finish* entries. When a customer has completed the use of the pump, it calls the *finish* entry and indicates a payment. The pump task, after releasing the customer, calls the operator's *charge* entry to indicate how much gas was actually sold. After accepting and releasing a pump caller on the *charge* entry, the operator will activate the pump for another customer, if any are waiting and call the *change* entry of the first customer task in the amount queue for that pump.

The critical race appears when two customers call the start entry of a single pump: the operator and pump tasks expect that the first customer to call the start entry is also the customer to whom the appropriate change is returned. This is not enforced by any scheduling logic, therefore customer might receive incorrect change. The complexity of this critical race is reflected in the way in which any observable error follows from its occurrence: the critical race occurs at the pump, the flaw is propagated to the operator task which further propagates the error to the customer task where the effects of the critical race are finally observed. (The flaw might be observed earlier at the operator if the amount pre-paid was less than the value of the gas actually pumped.)

In our version of this example, we use a single pump and two customers. An execution log of the deadlock and ND-Analyzer analysis of the design are presented in Figure 7.2. A corrected version of this design is presented in § 7.3. In the corrected version the customer callers of a given gas pump always arrive deterministically.

The cost of the searching the PSA-tree for non-determinism is quite inexpensive even when the tree is large. In worst case, it is proportional to the size of the PSA-tree. In Table 7.1 we present the time required to identify non-deterministic rendezvous in the set of test designs described in Appendix C. Of the entries, the worst was about 18 minutes for a PSA-tree of 1356 PSA. Another PSA-tree of similar size required only about 6 minutes of execution time. In most cases the analysis time was negligible.

*** TRAS Menu ***

Enter one of help load info col deadlock nda starvation
animate output sequence quit {help}

> deadlock

Executing TL-based deadlock analysis...

*** Execution Analysis Completed ***

Deadlock analysis complete; elapsed cpu time = 162 secs.

*** TRAS Menu ***

Enter one of help load info col deadlock nda starvation
animate output sequence quit {help}

> nda

*** Non-determinism has been detected in the design ***

Which task would you like to study?

Enter one of pump operator {bye}

> pump

ND Accept Entry "start":

customer(2)!9

customer(1)!9

More study?

Enter one of display_psa next_psa another_task bye {bye}

> display_psa

customer(1) : task call by customer(1)!9 on entry pump!start.
vars: [prn([])]

customer(2) : task call by customer(2)!9 on entry pump!start.
vars: [prn([])]

pump : accept call on pump!start. vars: [prn([])]

operator : selective accept 1. vars: [prn([]).cid(x).c_q([1,2])]

More study?

Enter one of display_psa next_psa another_task bye {bye}

> bye

Figure 7.2: Execution Log of Gas-Station Design Analysis

Name	#Task	#PSA	ND Detect Time
Readers	4	36	7
Diners	6	1137	381
Gas-Station	4	60	3
Gas-Station-CR	4	98	3
Dialogue	5	70	4
RMS	10	1256	1041
RMS-Link	6	66	3
RMS-Service	5	38	2
Alter-Bit	7	150	15
10 Task	10	4	< 1

All times quoted in seconds

Table 7.1: Summary of Non-Determinism Analysis Times

Chapter 8

The COL Interpreter

In this chapter, we describe an interactive language interpreter for the COL language. The *COL Interpreter* provides us with the capability to execute interactively the COL specification of an Ada program. From the COL perspective, the interpreter presents the operational interpretation of an executing COL program.

The behaviour of the interpreter is determined by the COL operational semantics. The open-ended operational semantics are described in chapter 4. We complete the description of the operational semantics by defining the execution algorithm, ϕ , for the *COL Inference Engine*. In particular, we establish procedures for implication formulae selection and expansion. Given the basic functionality of the interpreter we conclude the chapter with a discussion of its implementation. An example illustrating the execution of the COL Interpreter is provided in Appendix D.

8.1 Functional Description

The objective of the COL interpreter is to provide the user with a facility with which to explore the behaviour of an Ada program through its COL specification; i.e. the COL program is an *executable* or operational specification of the Ada program. From the COL perspective, "execution" is the operational interpretation of an axiomatic process, the COL inference engine.

In this section, we provide a functional description of the COL interpreter. We broach the subject with a review of how the operational interpretation of a COL program is extracted from the behaviour of the COL inference engine. This will be followed by descriptions of specific interpreter functions.

8.1.1 Operational Interpretation of COL

The COL inference engine treats COL language statements as *discrete, linear temporal logic* (LTL) implication formulæ that specify the behaviour of an Ada program. It applies the implication formulæ to a *program state assertion* (PSA) in order to deduce a new program state assertion. The new PSA describes the progress of the Ada program relative to the first PSA.

Each PSA is a result of the inference produced by the application of one implication formula and an inference represents a change in the state assertion of one or two tasks. A sequence of inferences generates a chain of successively deduced PSA. We interpret such a sequence as description of the execution of the specified Ada program.

With this brief review, we summarize the operational interpretation of COL below.

- PSA describe the execution state of the Ada program.
- Inference describes the progress of the Ada program from one execution state to a future execution state.
- A sequence of successively deduced PSA represents the discrete execution of the Ada program.

We shall interpret also the generation of a PSA expansion alternative (cref. § 4.2.7) as an inference step although it is not a logical deduction.

8.1.2 COL Interpreter Features

The COL interpreter features satisfy two objectives: (1) they provide the user with sufficient control over the interpretation process so that any path of exploration

may be followed and (2) they display the inference process to the user frequently enough so that it can be properly controlled, but not so often that it appears excessively drawn out.

The functionality of the COL interpreter is summarized as follows: (1) PSA display, (2) one step inference, (3) expansion process, and (4) inference backtracking. We address each of these features below.

PSA Display

A sequence of successively deduced PSA represents the discrete execution of a COL specification of an Ada program. The display of the deduction sequence is fundamental to the interpretation process since it reveals the behaviour of the executing Ada program. There are essentially two ways to display this information: graphically and textually. The graphical display of an executing program, called *program animation*, is still an open research area; it is indicated for future exploration in the conclusions (cref. chapter 10).

In our current work, the display of the program execution is of secondary interest, thus we have opted for a textual presentation: a natural language (NL) translation of the LTL formula represented by a PSA. We illustrate a NL translation below:

Actual PSA:

```
[
  (a.*<>(task_call(a,a!1,b!e) & ts(a,[prm([])],[[]])).
  (b.*<>(start_accept(b,a!1,b!e) & ts(b,[prm([])],[[]])).
  (c.*<>(proc_call(c,c!2,c!p1) & ts(c,[prm([])],[[]]))
]
```

NL Translation:

- a: executing a task call from a!1 on entry b!e
with variables [prm([])]
- b: starting the accept of entry b!e using caller a!1
with variables [prm([])]

c: executing procedure call from c!2 on procedure c!p1
with variables [prm([])]

We have represented the conjunction of task state assertions as a list of separate, labelled NL phrases. The task state phrase does not include any temporal references; it is implied that each phrase is true, eventually.

One Step Inference

The inference procedure from the *deadlock analysis tool* of chapter 5, observed (recorded) PSA whenever each task had reached a "waiting for rendezvous" state. For the COL interpreter, we require a finer granularity of observation; yet we would also like to reproduce the inference behaviour of the deadlock analyzer (since it generates all possible executions of the Ada program). We achieve these objectives with *one step inference*: derive new PSA until each task has progressed one implication, if possible; the last deduced PSA is the observed PSA.

The inference procedure is user controlled; that is, once an observed PSA is deduced, the inference engine waits for the user to authorize the next one step inference. This granularity of observation does not allow the inference engine to skip execution steps without the user's knowledge. The deadlock analyzer inference procedure for generating all possible executions of the specified program; stipulates that the inference engine must generate deductions until no deductions are possible (at which point expansion is tried, or a system-wide deadlock is declared). This requirement is satisfied by the user control of the inference procedure.

We illustrate the one step inference in the example below; note: in this example, the COL Ada language implication statements are used for inference.

Starting PSA:

```
[
  (a.*<>(task_call(a,a!1,b!e) & ts(a,[prm([])].[]))),
  (b.*<>(start_accept(b,a!1,b!e) & ts(b,[prm([])].[]))),
  (c.*<>(proc_call(c,c!2,c!p1) & ts(c,[prm([])].[])))
]
```


Observed (deduced) PSA:

```
[
  (a.*<>(call_accepted(a,a!1,b!e) & ts(a.[prm({})].[]))),
  (b.*<>(start_rendez(b,a!1,b!e) & ts(b.[prm({})].[...]))),
  (c.*<>(start_proc(c,c!p1) & ts(c.[prm({})].[...])))
]
```

Expansion Process

The expansion process creates a set of deterministic, alternative PSA from a given PSA using *define predicates* (cref. § 4.2.7). We integrate the expansion process into the COL interpreter by allowing the user to invoke it whenever the inference engine pauses after a one step inference; thus, the user can elect to perform an inference, or an expansion, on the observed PSA. If there is a choice in alternatives, a manual selection is made.

The manner in which expansion is performed is left open-ended in the COL operational semantics (cref. 4.2.7); however, since it is our goal to be able to reproduce all possible program behaviours, we adopt the expansion procedure employed by the deadlock analyzer (which satisfies this goal). In the deadlock analyzer, the alternative PSA are produced from the current PSA such that only one define predicate is expanded in any alternative PSA; furthermore, the expansion process is invoked whenever the deduction process fails. We use the first part of the procedure to generate the alternative PSA from the observed PSA. We can choose to follow the second part of the procedure by invoking the expansion process only after the one step inference has failed. In summary, we can realize all possible execution behaviours of the COL-specified Ada program by sequencing properly the one step inference and expansion process.

Inference Backtracking

The inference backtracking function provides the user with the flexibility to choose an expansion alternative, follow its deduction path and then backtrack along the

deduction path to pursue one of the other alternatives. It can be used also to refresh the user about previous events. The backtracking operation is provided at the inference engine pause; that is, it can be selected instead of one step inference and expansion. Once executed, the observed PSA becomes the current observed PSA's parent PSA, regardless of its source (deduction or expansion); the inference engine remains paused. The backtracking facility is implemented by recording each observed PSA in the current deduction path.

8.1.3 COL Interpreter Algorithm

- The COL Interpreter algorithm that integrates all of the functions described earlier is presented below.

```
(1)      observed_PSA <== initial_PSA;
(2)      start COL interpreter;
(3)      get user request (one-step, expand, backtrack, quit);
(4)      if command = 'one-step' then
(5)          apply one step inference to observed_PSA;
(6)          if inference succeeds then
(7)              observed_PSA <== deduced_PSA;
(8)              record observed_PSA;
(9)          else report "deduction failed";
(10)     else if command = 'expand' then
(11)         expand observed_PSA;
(12)         if expansion created > 1 alternative then
(13)             get user request for alternative;
(14)             observed_PSA <== selected alternative PSA;
(15)             record observed_PSA;
(16)         else if expansion created 1 alternative then
(17)             observed_PSA <== selected alternative PSA;
(18)             record observed_PSA;
(19)         else
(20)             report "deduction failed"
(21)         end if
(22)     else if command = 'backtrack' then
(23)         observed_PSA <== parent_of observed_PSA;
(24)     else if command = 'quit' then
(25)         goto (28);
(26)     end if
(27)     go to (3);
(28) end COL interpreter;
```

8.2 Interpreter Implementation

The COL interpreter used the same basic structure as the deadlock analyzer; in fact, it shares some of the same subsystems. Recall from chapter 5 that the deadlock analyzer was composed of four subsystems: (1) the inference engine, (2) the PSA manager, (3) the COL language storage manager and (4) the task animator. The same subsystems are present for the COL interpreter. The distinguishing features of the interpreter are found in the inference engine. The remaining three subsystems are identical.

The COL interpreter inference uses a different strategy for deciding when to store (and observe) PSA and when to invoke the expansion process. The presence of a user interface to control the behaviour of the inference engine and the inference backtracking functions are completely new.

Chapter 9

Practical Considerations of Analysis

In this chapter, we assess the practical limitations of the deadlock, starvation and critical race analysis tools, that we have described in the previous chapters. In the discussion, we identify the constraints of computational complexity and the key parameters of complexity. Finally, we provide a set of heuristics for developing COL specifications that can be analyzed under practical conditions.

9.1 The Limitations of Analysis

Our description of the deadlock analyzer in chapter 5 included a discussion on the computational complexity of the analysis algorithm. In this discussion, we showed that the complexity of the algorithm was essentially the cost of producing the program state assertion tree, or PSA-tree. We concluded that this cost was exponential with the number of tasks in the system. We further demonstrated that the base of the exponent was approximately bounded by the worst case state space of a task.

Both the starvation analyzer and the non-deterministic rendezvous detector use the PSA-tree as input. In the case of starvation, the analysis effort was primarily the pre-processing cost, which was shown to be proportional to the size of the PSA-

tree. The detection of non-deterministic rendezvous involved a simple one-time search of the PSA-tree, the cost of which is proportional to the size of the PSA-tree. Given these complexity relationships, we shall use the cost of producing the PSA-tree to be representative of the complexity of the tool-set as a whole, with the exception of the COL Interpreter. In this case, issues of computational complexity are of no interest since the interpreter is not an algorithm to solve some specific problem.

From the analyses of the design examples that were presented throughout the chapters describing the three analyzer tools (and summarized in Appendix C), we observe that designs with a large number of tasks, coupled with a high degree of non-deterministic rendezvous are computationally expensive. We also find that designs with a large data space require a considerable analysis effort. The Remote Mailbox System (RMS) and Five Dining Philosophers are good examples of such designs. The Ten-Task design illustrates a contrast to the complexity of a large number of tasks, because it does not contain non-deterministic rendezvous; i.e. it can be analyzed at very low cost.

Our experience with analysis and our understanding of the analysis complexity suggests a collection of parameters that affect the practical, computational effectiveness of our tool-set.

Number of tasks — the computational complexity rises, possibly dramatically, with the number of tasks in the design.

Level of non-determinism — the greater the number of non-deterministic rendezvous, the more closely the computational complexity approaches the exponential worst case.

Size of data state space — if variables can take on a large number of different values, or, if there are a large number of variables, the state space of individual tasks will be large.

Size of control state space — if many changes of control occur within tasks, then the state space of individual tasks will be large.

If there is sufficient flexibility in the design under evaluation, a user can tailor the design with due consideration to these parameters. In the next section, we present a number of heuristics rules that assist the user in performing this task.

9.2 Analysis Heuristics

The analysis of all possible designs, even with limited numbers of tasks (e.g. 10), is not a practical objective within our tool-set. For example, the analysis of one RMS node is not complete when the actual system is a set of such nodes; i.e. to completely analyze such a system we would have to evaluate a system of considerably more tasks than the 10 in a single RMS node. Similarly, a complete evaluation of the Gas-Station example would include all customer tasks and all pump tasks, not simply two customers and one pump.

The practical application of our tool-set will often require that the COL specifications be limited such that they do not completely reflect the actual target system. This may be entirely acceptable if there are no losses due to simplifications, or, if the losses are deemed acceptable in view of the insight gained by analysis. Since it may be very difficult to prove either of these points quantifiably, we must rely on human experience and understanding to guide the simplification process.

The parameters of complexity that we identified in the previous section, provide us with a basis for simplification heuristics. We describe these heuristics below. They have been applied in the development of the design examples presented in Appendix C.

Heuristic 1 — Minimal Task Pools

Use the minimum number of tasks in a task pool, that reliably represent the behaviour of the system. This will reduce the number of tasks in the system and the level of non-deterministic rendezvous (since tasks of the same task type must call the same task entries, thus creating non-deterministic rendezvous). For example, this could be applied to the Readers and Writers design and the Gas-

Station design, but not the Five Dining Philosophers.

Heuristic 2 — No Excess Statements

Do not introduce calls to passive procedures if they do not affect temporal behaviour. Similarly, do not introduce variables that have no impact on temporal behaviour. This will reduce the size of the task state space. This rule was applied to all designs, however, it was heavily applied in the Dialogue example, where all references to a passive screen management package were removed from the CCI specification.

Heuristic 3 — Simulate External Events

Simulate external events instead of actually supplying full specifications of the "outside world"; this may be the only option in some designs. This heuristic should reduce the number of tasks to be analyzed. We used this technique in the RMS example where the user, mailbox and hardware tasks approximated the outside world for the three-layer system. A similar technique was used in the Alternating Bit and Dialogue examples.

Heuristic 4 — Divide and Conquer

Divide the design into subsystems and analyze each of these individually. The missing portions of the design should be simulated. This should substantially reduce the number of tasks to be analyzed. Divide and conquer was applied to the RMS example with considerable success.

Heuristic 5 — Minimal Buffer Size

When possible, buffers and like structures should be used at their minimum sizes. This will reduce the data state space by indirectly reducing the number of variables. This principle was applied to the message buffers of the RMS, Alternating Bit and Dialogue examples.

3 of/de 3

NEED

Chapter 10

Conclusions

10.1 A Research Summary

The complex nature of embedded, concurrent systems requires that system developers use particular care during design. Without automated design support, such ventures can be very costly at both ends of implementation: design and maintenance. In this thesis, we have addressed the problem of automated design support by developing a comprehensive temporal analysis facility for *CAEDE*, an iconic design environment for Ada-oriented concurrent systems.

The analysis system is composed of an operational specification language, *COL*, and four analysis tools: the *deadlock analyzer*, the *starvation analyzer*, a critical race analysis assistant, *ND-Analyzer* and the *COL Interpreter*. The design of the *COL* language and the analysis tool-set was influenced by the following goals:

- the tools should be readily comprehensible to a designer with a practical orientation, but little facility for advanced methods in computer science;
- design should represent an operational rather than a declarative specification of the system;
- the specification and evaluation methods should be based on formal techniques where possible;

- design analysis should be expressed in terms of CAEDE's Ada-oriented design elements.

Summaries of the individual research efforts are presented below.

CAEDE Operational Language (COL)

The CAEDE operational language (COL) provides a formal foundation for our temporal analysis tool-set. This language, based on *discrete linear time temporal logic* (LTL), specifies Ada program behaviour in a manner amenable to both axiomatic and operational analysis.

The COL language expresses Ada programs by: (1) temporal logic implication formulæ that describe the behaviour of tasks and procedures, (2) temporal logic implication formulæ that describe the Ada concurrency model and (3) an assertion specifying the initial state of the program. The implication formulæ define allowable sequences of program states by stating legal transformations of the program state from the perspective of a single task. The initial program state assertion (initial PSA) describes the state of the program relative to some arbitrary temporal origin, t_0 . The formulæ describing the Ada concurrency model describe the mechanics of the Ada rendezvous and procedure calls.

The operational semantics of COL are defined by a *bottom-up* inference engine. The inference engine applies the implication formulæ describing the program to a *current PSA* in order to deduce a new PSA; the first current PSA is the initial PSA. The inference engine uses a set of temporal logic inference rules to perform deductions. The basic description of the COL inference engine is open-ended; however, it is completed when the inference engine is used in the deadlock analyzer and COL Interpreter tools.

A practical version of the language uses a Prolog-based syntax and Prolog code fragments to express predicates and operations on Ada program variables. We have provided a complete design example expressed in the language.

Deadlock Analyzer

The deadlock analyzer uses temporal logic as the theoretical basis for automatically proving the absence of *system-wide deadlock* in Ada concurrent system designs. (We demonstrated that *local deadlock* could be detected by the starvation analyzer). In this tool, the design under review is expressed in COL and then automatically analyzed using a special purpose implementation of the COL inference engine. The inference engine generates a tree of deduced program state assertions (PSA-tree) that describe all possible behaviours of the design under analysis. If there exists any PSA from which further deductions are possible, then a system-wide deadlock is proven.

We showed that the method is *decidable* for designs in which the data space is finite and it is *acceptable* in general. We further proved that when the algorithm is decidable, it has a computational complexity that is exponential with the number of tasks in the program.

Starvation Analyzer

We have developed an efficient method for identifying starvation in Ada designs using an optimized set-theoretic starvation test. The input to the starvation analyzer is the PSA-tree generated by the deadlock analyzer. The deadlock analysis is assumed to have shown absence of system-wide deadlock.

The starvation analyzer uses the PSA-tree to generate a repeating sequence of successive program state assertions, σ_r , to which is applied the set-theoretic starvation test. The test determines if there exists some task that is blocked waiting for a rendezvous and if an infinite repetition of the sequence can represent a fair computation. A depth-first traversal of the PSA-tree is used to generate the σ_r to be tested. Any σ_r in which all tasks are active or that represent unfair computations are ineligible. Optimizations obtained by pre-processing the PSA-tree allows us to prune many ineligible σ_r .

We have shown that the method is decidable. We also showed that the cost of pre-processing is linearly proportional to the cost of the deadlock analysis and the cost of the analysis algorithm is exponential with the cost of the deadlock analysis.

Finally, we demonstrated the efficiency of the tool for several designs; in fact, for some designs, the cost of the analysis algorithm was negligible.

ND-Analyzer (Critical Race Analysis)

We have identified the two sources of critical races in Ada programs: the violation of a mutual exclusion constraint for some resource and an invalid assumption of rendezvous behaviour. By presuming that no errors due to violations of mutual exclusion exist, we have reduced the problem of detecting the source of critical races to: (1) identifying a non-deterministic rendezvous (a race) and (2) proving that the design has assumed that it was deterministic. Unfortunately there is no general algorithm for performing this task.

As a compromise solution, we have developed the ND-Analyzer which automatically detects non-deterministic rendezvous and aids the user in proving that it invalidates some design assumption. The set of all non-deterministic rendezvous is derived from the PSA-tree produced by the deadlock analyzer.

COL Interpreter

The COL interpreter we have developed provides the designer with the ability to interactively "execute" the design. This capacity supplements the analysis system where the other tools may fail, or, to investigate other temporal aspects of the design, such as the congestion of many callers at a task entry.

The interpreter is a specialized COL inference engine that presents an operational interpretation of the COL specification of a design. The user has some control over the inference process, specifically, the engine proceeds, backtracks and explores different deduction paths at the user's discretion. The behaviour of the interpreter has been illustrated by an example.

* * *

The analysis tools were evaluated with a suite of design examples expressed in COL. These examples were chosen to meet a variety of objectives:

- to demonstrate proper operation of the various tools;
- to provide some estimate of the computational performance;
- to illustrate the effects of those factors that most influence computational performance.

All tools correctly identified temporal errors in both well-known examples (e.g. the Five Dining Philosophers, Readers and Writers) and through inadvertent errors (e.g. deadlock in the Remote Mailbox System).

The basic measure of computational performance is the time required to produce the PSA-tree in the deadlock analyzer, since all analyzer tools used this structure as input. In this regard, we identify a need for a 20-fold increase in computing power to produce analysis within acceptable limits; e.g. the generation of 1500 task-PSA/minute¹. Under these conditions, more sophisticated designs, such as the RMS example, could be analyzed for deadlock within 10 minutes and some of the simpler designs would be analyzed in about 20 seconds (100 PSA, 4 tasks).

The key parameters affecting the computation were demonstrated to be: (1) number of tasks, (2) level of non-deterministic rendezvous, (3) size of data state space and (4) size of control state space. These parameters affect the computational performance as summarized below.

- In general, increasing the number of tasks will increase, perhaps substantially, the time and space required to generate the PSA-tree.
- A non-deterministic rendezvous produces extra branches in the PSA-tree, therefore, the greater the number of non-deterministic rendezvous, the larger the PSA-tree.
- The data and control state space define the number of unique PSA that are generated. This may have a profound affect if the number of tasks, N_t , is

¹The task-PSA/minute approximates the storage and computation costs of a PSA over time; e.g. 1500 task-PSA/min could analyze 10 tasks producing 1000 PSA in 6.6 minutes.

large because the size of the PSA-tree is proportional to the worst case state space of a single task to the N_i th power. For example, a doubling of state space would produce a 2^{N_i} -fold increase in the size of the PSA-tree.

Considerable savings could be gained by reducing the scale of the design along any of these parameters, particularly if the the tool-set is to be applied to significant designs. In this regard, we have identified a set of analysis heuristics that suggest methods for simplifying design specifications. If used properly, simplifications produced with the aid of these heuristics should substantially reduce execution time with only minor losses in analysis accuracy.

We identified five heuristics: (1) minimal task pools, (2) no excess statements, (3) simulate external events, (4) divide and conquer and (5) minimal buffer size. Divide and conquer is probably the most powerful and commonsense heuristic. We apply divide and conquer as follows: divide the design into subsystems and analyze a subsystem by simulating the remainder of the design. In general, this approach will effect, in a positive way, all of the complexity parameters, therefore it could substantially reduce the analysis time for the design as a whole. Furthermore, it is quite realistic to develop a significant design in subsystems and evaluate each subsystem separately.

We have applied the analysis heuristics to the development of our design examples. Of particular interest, was the use of the heuristics in the RMS example. This example, fairly representative of a real design, illustrates both the need for and power of analysis heuristics. In an analysis of the design as a whole, we used three heuristics: simulate external events, no excess statements and minimal buffer size. When the performance of this analysis was compared to a divide and conquer version of RMS, we achieved a 50-fold reduction in analysis time with no perceived loss of accuracy. This demonstrates the tremendous improvements in performance that can be attained with the heuristics.

With this brief review of our research, we can begin to assess the progress towards our stated goals and the general goal of practical engineering value.

Assessment of Stated Goals

We believe our tool-set is relatively easy to understand because of its temporal logic roots. Temporal logic allows use to state and study problems in natural terms; i.e. without the introduction of additional mathematical abstractions. Although some theoretical details of the tools may be complex, the basic operating principles are straightforward. Another attribute of the tool-set that simplifies its understanding is its integration: both the analyzer tools and the interpreter use the same operational principles (the COL inference engine), the COL language underlies the entire tool-set and the PSA-tree of the deadlock analyzer is central to all of the analyzer tools. In short, "the less there is to learn, the easier it is to understand".

The COL language conveniently supports operational and axiomatic interpretations, thus we can provide the designer with the operational specification view. Furthermore, the Ada flavour of the COL language and analysis tools allows the designer to perceive the operational interpretation of all aspects of the tool-set by reasoning their meaning in Ada terms. The COL Interpreter provides the essence in the operational view through execution of the COL specification.

The evaluation methods are based entirely on formal methods, with the exception of the critical race analysis tool. In this case, the detection of race conditions is formally presented, but the isolation of critical races is left to human insight.

Assessment of Engineering Value

We assess the engineering value of our analysis system by formulating a response to the following question: "*What is required to make this system run in a practical environment on real designs?*" We decompose our answer into several points below.

Getting Information In and Out — In a practical environment the entry and retrieval of information must be convenient activities. In CAEDE convenience translates to iconic. CAEDE's structure graphs provide part of the information required for a COL specification; other information would need to be supplemented

by new icons and text. The output of tools should be expressed in terms of the CAEDE icons, most likely through some form of animation. Animated timing diagrams would also be a useful output mechanism. Issues of input and output are identified later as areas of further research. ▲

Generating Target Software — An operational specification of a design loses value if it cannot be used to generate the target system. COL could be used, in conjunction with the information represented in CAEDE's structure icons, to generate Ada code; however, we first require a Prolog-to-Ada transformation system to map data state pre- and post-conditions (which are expressed in Prolog). We suggest this as an additional research topic.

Easy Application of Heuristics — In order to accommodate the analysis of "real" designs, we must be able to apply easily our analysis heuristics. To this end, we suggest the following approach. Suppose the user specifies the design (using some iconic/text notation) in considerable detail. We execute this specification using the COL interpreter without any difficulty; however, in order to analyze the design, we would want to partition it, remove sections and add simulated sections. A convenient "copy" or "perspective" feature should allow the user to modify a copy of the design; thus, modifications could be made without destroying the original work and additional analysis (employing different heuristics) could be produced easily.

Language Enhancements — Some enhancements to the basic language would be desirable in order to meet the expected demands. These include: some scheme for introducing non-determinism in the application (as a modelling aid), dynamic tasking, exceptions and Ada timeout structures. These issues are suggested for future research.

10.2 Further Research

A researcher's work is never done; that's what keeps them in business². This thesis, like most before it, has left a few stones unturned. In particular, we identify the following open problems:

- limitations of the COL language;
- a desire for greater performance in the deadlock and starvation analyzers;
- a lack of automated critical race analysis;
- a COL to Ada transformation process;
- integration of the analysis system within CAEDE's iconic framework.

We address each of these issues below.

The COL language paints a limited but useful picture of the Ada language. These limitations were introduced to simplify the axiomatic analysis of COL specifications; yet, it is still desirable to include many of the features we have overlooked. Some of the features worth considering are: dynamic task creation, task termination, selective accepts with delay and else alternatives, timed and conditional entry calls, exceptions and recursion in procedures. It may be possible to include task creation and destruction within our current model, however, structures such as exceptions and the time-out rendezvous facilities interfere with our ability to prove that specific events will take place and procedure recursion may introduce inefficiencies in the axiomatic analyses. Significant rethinking of the current temporal logic model may be required to include these additional features.

It is naturally desirable to extract maximum performance from the deadlock and starvation analyses, both in the class of programs that can be analyzed and their computational complexity. There is little promise for a general absence of deadlock prover since such a program would be able to solve the *halting problem* [67] which is not decidable; i.e. we cannot construct a program that can say

²G.M. Karam, September 25, 1986

whether or not another program will halt. The starvation analyzer is similarly limited in scope. Improvements in performance might be attained by resorting to problem-specific information; however, in general, it is unlikely that either algorithm can be improved much beyond the exponential complexity that currently exists. Razouk and Hirschberg [88] identify a similar problem in their P-NUT analysis system.

The lack of a completely automated critical race analysis method is rather perturbing. The first step in developing such a method for Ada programs would require a formal characterization of the assumption condition with the same rigor that has been applied to the mutual exclusion problem (as exemplified by [64,65]). Given suitable formal treatment, an analysis algorithm may be forthcoming.

The chief benefit of an operational specification language is that we can exercise a design before producing target software code; however, if we have developed a specification language that cannot be transformed in some provably correct fashion to a target language then we have accomplished little. In part, this is the case for COL. There is essentially a one-to-one mapping from the COL implication statements to Ada language code with the exception of the Prolog code used to express variable data structures and predicates or functions on variable data structures. We suggest that a provably correct, perhaps semi-automated transformation process could be constructed using automated programming techniques described by Balzer [6] or Smith et al. [100]. This approach may prove very fruitful because of the simplicity of Prolog (versus the wide spectrum languages that classically have been used for program specification [6,100,107]).

Finally, we leave open the question of introducing iconic specification and iconic presentation of analysis results to our analysis system. Of these two problems, iconic specification is quite tractable as demonstrated by similar efforts for state machine-based systems [58,80,106]. Iconic display of analysis results is essentially the process of displaying the activity of a hierarchical, concurrent system where the locus of control of any of the concurrent entities may move through the hierarchy. The most effective means of performing this task will successfully address the limits of human perception towards concurrent activities and hierarchy.

* * *

The *Software CAD* revolution is continuing. With the development of our temporal analysis system, Software CAD has moved forward a significant step — we now have the basis for a comprehensive analysis system. The outlook on concurrent system design is improving, yet, out of every discovery, a new set of questions are born.

References

- [1] B. Adelson and E. Soloway, "The Role of Domain Experience in Software Design," *IEEE Transactions on Software Engineering*, vol. SE-11, no. 11, November 1985, pp. 1351-1360.
- [2] T. Agerwala, "Putting Petri Nets to Work," *IEEE Computer*, vol. 12, no. 12, December 1979, pp. 85-94.
- [3] G.S. Avrunin and J.C. Wileden, "Describing and Analyzing Distributed Software System Designs," *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 3, July 1985, pp. 380-403.
- [4] G.S. Avrunin, L.K. Dillon, J.C. Wileden and W.E. Riddle, "Constrained Expressions: Adding Analysis Capabilities to Design Methods for Concurrent Software Systems," *IEEE Transactions on Software Engineering*, vol. SE-12, no. 2, February 1986, pp. 273-278.
- [5] P. Azema, G. Juanole, E. Sanchis and M. Montbernard, "Specification and Verification of Distributed Systems Using Prolog Interpreted Petri-Nets," *7th International Conference on Software Engineering*, Orlando, FLA, March 1984, pp. 510-518.
- [6] R. Balzer, "A 15 Year Perspective on Automatic Programming," *IEEE Transactions on Software Engineering*, vol. SE-11, no. 11, November 1985, pp. 1257-1267.
- [7] A. Barr and E.A. Feigenbaum, *The Handbook of Artificial Intelligence Volume 2*, Heuristech Press, Stanford, CA, 1982.
- [8] M. Ben-Ari, Z. Manna and A. Pnueli, "The Temporal Logic of Branching Time," *Eighth Annual ACM Symposium on Programming Languages*, Williamsburg, VA, January 1981.
- [9] G. Berthelot and R. Terrat, "Petri Nets Theory for the Correctness of Protocols," *IEEE Transactions on Communications*, vol. COM-30, no. 12, December 1982, pp. 2497-2505.

- [10] G.V. Bochmann, "Finite State Descriptions of Communication Protocols," *Computing Networks*, vol. 2, October 1978, pp. 361-372.
- [11] G.V. Bochmann, "Hardware Specification with Temporal Logic: An Example," *IEEE Transactions on Computers*, vol. C-31, no. 3, March 1982, pp. 223-231.
- [12] D. Brand and P. Zafropulo, "On Communicating Finite State Machines," *Journal of the ACM*, vol. 30, no. 2, April 1983, pp. 323-342.
- [13] G.P. Brown, R.T. Carling, C.F. Herot, D.A. Kramlich and P. Souza, "Program Visualization: Graphical Support For Software Development," *IEEE Computer*, vol. 18, no. 8, August 1985, pp. 27-35.
- [14] G. Bruno and G. Marchetto, "Process-Translatable Petri Nets for the Rapid Prototyping of Process Control Systems," *IEEE Transactions on Software Engineering*, vol. SE-12, no. 2, February 1986, pp. 346-357.
- [15] D. Budgen, "Combining MASCOT with Modula-2 to Aid the Engineering of Real-Time Systems," *Software — Practice and Experience*, vol. 15, no. 8, August 1985, pp. 767-793.
- [16] R.J.A. Buhr, C.J. Hayes, G.M. Karam, E. Neron and R. Casselman, *CAEDE 1.5 User's Guide*, Technical Report SCE-86-11, Dept. of Systems and Computer Engineering, Carleton University, Ottawa, Ontario, CAN, May 1986.
- [17] R.J.A. Buhr, *System Design with Ada*, Prentice-Hall, Englewood Cliffs, NJ.
- [18] R.J.A. Buhr and G.M. Karam, "An Informal Overview of CAEDE: A Design Environment for Ada," *Ada Letters*, vol. 4, no. 5, March/April 1985, pp. 49-58.
- [19] R.J.A. Buhr, G.M. Karam and C.M. Woodside, "An Overview and Example Application of CAEDE: A New Design Environment for Ada," *International Ada Conference*, Paris, FRA, May 1985.
- [20] R.J.A. Buhr, C.M. Woodside, G.M. Karam, K. Van Der Loo and D.G. Lewis, "Experimenting with Prolog Design Descriptions and Tools in CAEDE: An Iconic Environment for Multitasking Embedded Systems," *8th International Conference on Software Engineering*, London, ENG, August 1985.
- [21] R.J.A. Buhr, G.M. Karam, C.J. Hayes and C.M. Woodside, "Software CAD: A Revolutionary Approach," to appear in, *IEEE Transactions on Software Engineering*.

- [22] I.M. Campos and G. Estrin, "SARA Aided Design of Software for Concurrent Systems," *Proceedings of the National Computer Conference*, 1978, pp. 325-336.
- [23] P.M. Cashin, *Inter-Process Communication*, Bell-Northern Research, Ottawa, Ontario, CAN, May 1980.
- [24] A.R. Cavalli, "A Method of Automatic Proof for the Specification and Verification of Protocols," *ACM SIGCOMM Computer Communication Review (SIGCOMM'84)*, vol. 14, no. 2, pp. 100-106.
- [25] A.R. Cavalli and L. Farinas Del Cerro, "A Decision Method for Linear Temporal Logic," *7th International Conference on Automated Deduction*, Lecture Notes in Computer Science #170, Springer-Verlag, New York, NY, 1984, pp. 113-127.
- [26] 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*, vol. 8, no. 2, April 1986, pp. 244-263.
- [27] L.A. Clarke, J.C. Wileden and A.L. Wolf, "GRAPHITE: A Meta-Tool for Ada Environment Development," *2nd International Conference on Ada Applications and Environments*, Miami Beach, FLA, April 1986, pp. 81-92.
- [28] W.F. Clocksin and C.S. Mellish, *Programming in Prolog*, Springer-Verlag, New York, NY, 1984.
- [29] E.G. Coffman, M.J. Elphick and A. Shashani, "System Deadlocks," *Computing Surveys*, vol. 3, no. 2, June 1971, pp. 67-78.
- [30] P.J. Courtois, F. Heymans and D.L. Parnas, "Concurrent Control with Readers and Writers," *Communications of the ACM*, vol. 14, no. 2, October 1971, pp. 667-668.
- [31] P. De Bondeli, "Models for the Control of Concurrency in Ada Based on Predicate Transition Nets," C.R.S.A., 18 rue d'Arras, 92000 Nanterre, FRA, 1985.
- [32] H.M. Deitel, *An Introduction to Operating Systems*, Addison-Wesley, Reading, MA, 1984.
- [33] E.W. Dijkstra, "Hierarchical Ordering of Sequential Processes," *Acta Informatica*, vol. 1, no. 2, October 1971, pp. 115-138.

- [34] E.A. Emerson and J.Y. Halpern, " 'Sometimes' and 'Not Never' Revisited: On Branching Time Versus Linear Time," *Tenth Annual ACM Symposium on the Principles of Programming Languages*, Austin, TX, January 1983, pp. 127-140.
- [35] E.A. Emerson and J.Y. Halpern, " 'Sometimes' and 'Not Never' Revisited: On Branching Time Versus Linear Time Temporal Logic," *Journal of the ACM*, vol. 33, no. 1, January 1986, pp. 151-178.
- [36] G. Estrin, "A Methodology for Design of Digital Systems — Supported by SARA at the Age of One," *AFIPS Conference Proceedings*, National Computer Conference, 1978.
- [37] G. Estrin, R.S. Fenchel, R.R. Razouk and M.K. Vernon, "SARA (System ARchitects Apprentice): Modeling, Analysis and Simulation Support for Design of Concurrent Systems," *IEEE Transactions on Software Engineering*, vol. SE-12, no. 2, February 1986, pp. 293-311.
- [38] R. Feys, *Modal Logics*, Collection de Logique Mathematique, Louvain, FRA, 1965.
- [39] N. Francez, *Fairness*, Springer-Verlag, New York, NY, 1986.
- [40] F.C. Furtek, "Specification and Verification of Real-Time, Distributed Systems Using the Theory of Constraints," *5th International Conference on Automated Deduction*, Lecture Notes in Computer Science #87, Springer-Verlag, New York NY, 1980, pp. 110-125.
- [41] S.M. German, "Monitoring for Deadlock and Blocking in Ada Tasking," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 6, November 1984, pp. 764-776.
- [42] E.F. Girczyc, R.J.A. Buhr and J.P. Knight, "Analysis of a Subset of Ada as a High-Level Hardware Description Language for Graph-Based Compilation," *IEEE Transactions on Computer-Aided Design*, vol. CAD-4, no. 2, April 1985.
- [43] M.G. Gouda, "Closed Covers: To Verify Progress for Communicating Finite State Machines," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 6, November 1984, pp. 846-855.
- [44] M.G. Gouda and C.K. Chang, "Proving Liveness for Networks of Communicating Finite State Machines," *ACM Transactions on Programming Languages and Systems*, vol. 8, no. 1, January 1986, pp. 154-182.

- [45] M.G. Gouda and Y.T. Yu, "Protocol Validation by Maximal Progress State Exploration," *IEEE Transactions on Communications*, vol. COM-32, no. 1, January 1984, pp. 94-97.
- [46] F. Guterl, "In Pursuit of the One Month Chip," *IEEE Spectrum*, vol. 21, no. 9, September 1984, pp. 28-29.
- [47] B.T. Hailpern, *Verifying Concurrent Processes Using Temporal Logic*, Lecture Notes in Computer Science #129, Springer-Verlag, New York, NY, 1982.
- [48] S.L. Hantler and J.C. King, "An Introduction to Proving the Correctness of Programs," *ACM Computing Surveys*, vol. 8, no. 3, September 1983, pp. 331-353.
- [49] M.T. Harandi and M.D. Lubars, "A Knowledge-Based Design Aid for Software Systems," *2nd Conference on Software Development Tools, Techniques and Alternatives*, San Francisco, CA, December 1985, pp. 67-74.
- [50] A.C. Hartmann, "Software or Silicon? The Designer's Option," *Proceedings of the IEEE*, vol. 74, no. 6, June 1986, pp. 769-874.
- [51] J.W. Havender, "Avoiding Deadlock in Multitasking Systems," *IBM Systems Journal*, vol. 7, no. 2, 1968, pp. 74-84.
- [52] F. Hayes-Roth, D.A. Waterman and D.B. Lenat, *Building Expert Systems*, Addison-Wesley, Reading, MA, 1983.
- [53] D. Helmbold and D. Luckham, "Debugging Ada Tasking Programs," *IEEE Software*, vol. 2, no. 2, March 1985, pp. 47-57.
- [54] H.J. Hoffman, "Expert Knowledge To Check Design Decisions in a Design Support System For Interactive Programs," *Proceedings of the International Conference on Systems Research, Informatics & Cybernetics*, Baden-Baden, W. Germany, August 1984.
- [55] C.J. Hogger, *Introduction to Logic Programming*, Academic Press, New York, NY, 1984.
- [56] R.C. Holt, "Some Deadlock Properties of Computer Systems," *ACM Computing Surveys*, vol. 4, no. 3, September 1972, pp. 179-196.
- [57] R.C. Holt, E.D. Lazowska, G. Scrantom and M.A. Scott, *Structured Concurrent Design with Operating System Applications*, Addison-Wesley, Reading, MA, 1978.

- [58] R.J.K. Jacob, "A State Transition Diagram Language For Visual Programming," *IEEE Computer*, vol. 18, no. 8, August 1985, pp. 51-59.
- [59] G.M. Karam and R.J.A. Buhr, "Embedding Design Knowledge in CAEDE: Experiments on the Detection of Critical Races," *2nd International Conference on Ada Applications and Environments*, Miami Beach, FLA, April 1986, pp. 93-98.
- [60] R.A. Kowalski, "Algorithm = Logic + Control," *Communications of the ACM*, vol. 22, no. 7, July 1979.
- [61] R.A. Kowalski, *Logic for Problem Solving*, North-Holland, New York, N.Y., 1979.
- [62] Y.S. Kwong, "On the Absence of Livelocks in Parallel Programs," *Semantics of Concurrent Computation*, Lecture Notes in Computer Science #70, Springer-Verlag, New York, NY, 1979, pp. 172-190.
- [63] L. Lamport, " 'Sometime' is Sometimes 'Not Never'," *7th Annual ACM Symposium on the Principles of Programming Languages*, Las Vegas, NV, January 1980, pp. 174-185.
- [64] L. Lamport, "The Mutual Exclusion Problem: Part I — A Theory of Inter-process Communication," *Journal of the ACM*, vol. 33, no. 2, April 1986, pp. 313-326.
- [65] L. Lamport, "The Mutual Exclusion Problem: Part II — Statement and Solutions," *Journal of the ACM*, vol. 33, no. 2, April 1986, pp. 327-348.
- [66] K. Lautenbach and H.A. Schmid, "Use of Petri Nets for Proving Correctness of Concurrent Process Systems," *Proceedings of I.F.I.P. Congress*, Stockholm, Sweden, August 1974, pp. 187-191.
- [67] H.R. Lewis and C.H. Papadimitriou, *Elements of the Theory of Computation*, Prentice-Hall, Englewood Cliffs, NJ, 1981.
- [68] Logicware Inc., *MProlog Language Reference, Release 2.1*, Toronto, CAN, September, 1986.
- [69] R.L. London and R.A. Duisberg, "Animating Programs Using Smalltalk," *IEEE Computer*, vol. 18, no. 8, August 1985, pp. 61-71.
- [70] D.C. Luckham and F.W. Von Henke, "An Overview of ANNA, A Specification Language for Ada," *IEEE Software*, vol. 2, no. 2, March 1985, pp. 9-23.

- [71] S.E. Madnick and J.J. Donovan, *Operating Systems*, McGraw-Hill, 1974.
- [72] Z. Manna and A. Pnueli, "Verification of Concurrent Programs: The Temporal Framework," *The Correctness Problem in Computer Science*, ed. R.S. Boyer, J.S. Moore, Academic Press, New York, NY, 1981, pp. 215-273.
- [73] Z. Manna and A. Pnueli, "Verification of Concurrent Programs: Temporal Proof Principles," *Logics of Programs*, Lecture Notes in Computer Science #131, Springer-Verlag, New York, NY, 1981, pp. 200-252.
- [74] R. Milner, *A Calculus of Communication Systems*, Lecture Notes in Computer Science #92, Springer-Verlag, New York, NY, 1980.
- [75] J. Misra and K.M. Chandy, "Proofs of Networks of Processes," *IEEE Transactions on Software Engineering*, vol. SE-7, no. 4, July 1981, pp. 417-426.
- [76] M. Moriconi and D.F. Hare, "Visualizing Program Designs Through PegaSys," *IEEE Computer*, vol. 18, no. 8, August 1985, pp. 72-85.
- [77] V. Nguyen, D. Gries and S. Owicki, "A Model and Temporal Proof System for Networks of Processes," *12th Annual Symposium on the Principles of Programming Languages*, New Orleans, LA, January 1985, pp. 121-131.
- [78] L.J. Osterweil, "Toolpack — An Experimental Software Development Environment Research Project," *IEEE Transactions on Software Engineering*, vol. SE-9, no. 6, November 1983, pp. 673-685.
- [79] S. Owicki and L. Lamport, "Proving Liveness Properties of Concurrent Programs," *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 3, July 1982, pp. 455-495.
- [80] T. Pearce, *Adapting an Iconic Software Design Environment for Protocol Specification with Estelle*, M.Eng. Thesis, Dept. of Systems and Computer Engineering, Carleton University, Ottawa, Ontario, Canada, 1986.
- [81] J.L. Peterson, "Petri Nets," *ACM Computing Surveys*, vol. 9, no. 3, September 1977, pp. 223-252.
- [82] J.L. Peterson, *Petri Net Theory and the Modeling of Systems*, Prentice-Hall, Englewood Cliffs, NJ, 1981.
- [83] A. Pnueli, "The Temporal Logic of Programs," *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, Providence, RI, October 1977, pp. 46-57.

- [84] A. Pnueli, "The Temporal Semantics of Concurrent Programs," *Semantics of Concurrent Computation*, Lecture Notes in Computer Science #70, Springer-Verlag, New York, NY, 1979, pp. 1-21.
- [85] A. Pnueli and W.P. DeRoeover, "Rendezvous with Ada — A Proof Theoretic View," *Proceedings of the AdaTEC Conference on Ada*, Arlington, VA, October 1982, pp. 129-137.
- [86] R.R. Razouk and G. Estrin, "Modeling and Verification of Communication Protocols in SARA: The X.21 Interface," *IEEE Transactions on Computers*, vol. C-29, no. 12, December 1980, pp. 1038-1052.
- [87] R.R. Razouk and D.S. Hirschberg, "Tools for Efficient Analysis of Concurrent Software Systems," *2nd Conference on Software Development Tools, Techniques and Alternatives*, San Francisco, CA, December 1985, pp. 192-197.
- [88] R.R. Razouk, M. Vernon and G. Estrin, "Evaluation Methods in SARA — The Graph Model Simulator," *Conference on Simulation, Measurement and Modelling of Computer Systems*, 1979.
- [89] S.P. Reiss, "PECAN: Program Development Systems That Support Multiple Views," *IEEE Transactions on Software Engineering*, vol. SE-11, no. 3, March 1985, pp. 276-284.
- [90] N. Rescher and A. Urquhart, *Temporal Logic*, Springer-Verlag, New York, NY, 1971.
- [91] W.E. Riddle, "An Assessment of DREAM," *Software Engineering Environments*, North-Holland, New York, NY, 1980.
- [92] W.E. Riddle and J.C. Wileden, "Languages for Representing Software Specifications and Designs," *ACM SIGSOFT Software Engineering Notes*, October 1978.
- [93] W.E. Riddle, J.C. Wileden, J.H. Sayler, A.R. Segal and A.M. Stavely, "Behaviour Modelling During Software Design," *IEEE Transactions on Software Engineering*, vol. SE-5, no. 4, July 1978, pp. 283-292.
- [94] J.A. Robinson, "A Machine-Oriented Logic Based on the Resolution Principle," *Journal of the ACM*, vol. 12, no. 1, January 1965, pp. 23-41.
- [95] J. Rubin and C.H. West, "An Improved Protocol Validation Technique," *Computer Networks*, vol. 6, 1982, pp. 65-73.

- [96] D. Schwabe and A.R. Cavalli, "Temporal Logic Specification of a Virtual Ring LAN Access Protocol," *Proceedings of the Fourth Workshop on Protocol Specification, Verification and Testing*, Skytop, PA, June 1984.
 - [97] R.L. Schwartz and P.M. Melliar-Smith, "Temporal Logic Specifications of Distributed Systems," *Proceedings of the Second International Conference on Distributed Systems*, Paris, FRA, April 1981, pp. 446-454.
 - [98] R.L. Schwartz and P.M. Melliar-Smith, "From State Machines to Temporal Logic: Specification Methods for Protocol Standards," *IEEE Transactions on Communications*, vol. COM-30, no. 12, December 1982, pp. 2486-2496.
 - [99] C.U. Smith and R.R. Gross, "Technology Transfer Between VLSI Design and Software Engineering: CAD Tools and Design Methodologies," *Proceedings of the IEEE*, vol. 74, no. 6, June 1986, pp. 875-885.
 - [100] D.R. Smith, G.B. Kotik and S.J. Westfold, "Research on Knowledge-based Software Environments at Ketsrel Institute," *IEEE Transactions on Software Engineering*, vol. SE-11, no. 11, November 1985, pp. 1278-1295.
 - [101] R.N. Taylor, "A General Purpose Algorithm for Analyzing Concurrent Programs," *Communications of the ACM*, vol. 26, no. 5, May 1983, pp. 362-376.
 - [102] R.N. Taylor and T.A. Standish, "Steps to an Advanced Ada Programming Environment," *IEEE Transactions on Software Engineering*, vol. SE-11, no. 3, March 1985, pp. 302-313.
 - [103] R.N. Taylor, L.A. Clarke, L.J. Osterweil, J.C. Wileden and M. Young, "Arcadia: A Software Development Environment Research Project," *2nd International Conference on Ada Applications and Environments*, Miami Beach, FLA, April 1986, pp. 137-149.
 - [104] U.S. Department of Defence, *Reference Manual for the Ada Programming Language*, MIL-STD-1815a, 1983.
 - [105] P. Wallich, "The One Month Chip: Design," *IEEE Spectrum*, vol. 21, no. 9, September 1984, pp. 30-34.
 - [106] A.I. Wasserman, "Extending State Transition Diagrams For The Specification of Human-Computer Interactions," *IEEE Transactions on Software Engineering*, vol. SE-11, no. 8, August 1985, pp. 699-713.
-

- [107] R.C. Waters, "The Programmer's Apprentice: A Session With KBEmacs," *IEEE Transactions on Software Engineering*, vol. SE-11, no. 11, November 1985, pp. 1296-1320.
- [108] J.C. Wileden, "DREAM — An Approach to Designing Large Scale Concurrent Software Systems," *Proceedings of the ACM National Conference*, October 1979, pp. 88-94.
- [109] P.M. Winston, *Artificial Intelligence*, Addison-Wesley, Reading, MA, 1984.
- [110] N. Wirth, *Programming in Modula-2*, Springer-Verlag, New York, NY, 1982.
- [111] P. Wolper, *Synthesis of Communicating Processes from Temporal Logic Specifications*, PhD Thesis, Dept. of Computer Science, Stanford University, Palo Alto, CA, 1983.
- [112] S.S. Yau and M.U. Caglayan, "Distributed Software System Design Using Modified Petri Nets," *IEEE Transactions on Software Engineering*, vol. SE-9, no. 6, November 1983, pp. 733-745.
- [113] Y.T. Yu and M.G. Gouda, "Deadlock Detection for a Class of Communicating Finite State Machines," *IEEE Transactions on Communications*, vol. COM-30, no. 12, December 1982, pp. 2514-2518.
- [114] P. Zave, "The Operational Versus the Conventional Approach to Software Development," *Communications of the ACM*, vol. 27, no. 2, February 1984.
- [115] P. Zave, "A Distributed Alternative to Finite State Machine Specifications," *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 1, January 1985, pp. 10-36.
- [116] P. Zave and W. Schell, "Salient Features of an Executable Specification Language and Its Environment," *IEEE Transactions on Software Engineering*, vol. SE-12, no. 2, February 1986, pp. 312-325.
- [117] P. Zave and R.T. Yeh, "Executable Requirements for Embedded Systems," *Proceedings of the 5th International Conference on Software Engineering*, San Diego, CA, 1981.

Appendix A

CAEDE Icon Set

The complete set of CAEDE icons is illustrated in Figure A.1¹. Each icon is described briefly below.

task — an Ada task;

task template — corresponds approximately to an Ada task type specification and body; it uses the same icon as a task (see Note 1 below);

dynamic task — an Ada task instantiated from a task type by "new" (see Note 1 below);

task instance — an Ada task instantiated from a task type in a declaration; it uses the same icon as a task (see Note 1 below);

dynamic task pool — an array of Ada tasks instantiated from a single task type by "new" (see Note 1 below);

task pool — an array of Ada tasks instantiated from a single task type in a declaration (see Note 1 below);

package — an Ada package;

package template — corresponds approximately to an Ada generic package; it uses the same icon as the package (see Note 1 below);

package instance — an Ada package that is an instance of a generic package; it uses the same icon as a package (see Note 1 below);

package external procedure — an Ada procedure that is declared in the specification of a package;

¹The descriptions and figure in this appendix are reproduced from [21]

procedure — an Ada procedure that declared in the body of a task or package;

task entry — an Ada task entry;

timeout task entry — the timeout alternative of a selective accept;

mainline — represents the executable body of a task or package;

data area — a program variable local to a package or task;

visible data area — a program variable that is declared in the specification of a package;

labelled connection point — represents an imported or required service; this does not exist in Ada at the specification level;

data flow arrow — represents a parameter of a task entry, procedure or package external procedure;

selective accept — three icons that all represent the Ada selective accept: a one-bar icon, a three-bar icon and a four-bar icon;

guard — marks a potentially closed task entry with a guard condition;

procedure call, unconditional entry call — a call to an Ada procedure or an unconditional call to a task entry;

conditional entry call — a conditional entry call on a task entry;

timed entry call — a timed entry call on a task entry;

unconditional entry call (dynamic) — an unconditional call to a task entry through an access variable;

conditional entry call (dynamic) — a conditional entry call on a task entry through an access variable;

timed entry call (dynamic) — a timed entry call on a task entry through an access variable;

provider port — a icon that defines the call and data flow interface of a package; a provider port indicates a flow a data from the package through the provider port (see Note 2 below);

user port — a icon that defines the call and data flow interface of a package; a user port indicates a flow a data into the package through the user port (see Note 2 below);

channel — a icon that defines a group of program elements that implement data transfer between a provider port and a user port; this may be as simple as procedure or task entry calls and as complex as packages and tasks (see Note 2 below);

channel template — a icon that defines a channel type that may be used to generate channel instances; it uses the same icon as a channel (see Notes 1,2 below);

channel instance — a channel instantiated from a channel template; it uses the same symbol as a channel (see Notes 1,2 below);

receive port plug — a call connection point for a provider or user port; a receive port plug associates an incoming call to a port with a package external procedure (see Note 2 below);

transmit port plug — a call connection point for a provider or user port; a transmit port plug associates an outgoing call from a port with a labelled connection point. (see Note 2 below);

Note 1 — Operationally, any task, package or channel that is instantiated from a template does not have any accessible internal structure; i.e. all internal structure is associated with the original template. Therefore, any editing of the internal structure of an instance must be performed on the template, not the instance.

Note 2 — These icons are included to support communication protocol software design using Estelle structures [80]. They have no significance in Ada but are included here for the sake of completeness; however, suitable Ada code skeletons can be generated from designs that use these structures.

Figure A.1: Full CAEDE Icon Set

Appendix B

Temporal Logic Axioms and Rules

In this appendix, we present axioms and inference rules defined in the literature and derived from the literature that are used in proofs. Each axiom or rule is identified with its source (if not derived) and local label.

From Feys [38], pg. 12 eqn. a021:

$$\models (P \wedge Q) \Leftrightarrow (\neg P \vee \neg Q) \quad (B.1)$$

From Feys [38], pg. 12 eqn. a06:

$$\models P \vee \neg P \quad (B.2)$$

From Feys [38], pg. 12 eqn. a10:

$$\models (P \wedge Q) \Rightarrow P \quad (B.3)$$

From Feys [38], pg. 12 eqn. a13:

$$\models [P \wedge (Q \vee R)] \Leftrightarrow [(P \wedge Q) \vee (P \wedge R)] \quad (B.4)$$

From Feys [38], pg. 12 eqn. a20:

$$\models P \Rightarrow (P \vee Q) \quad (B.5)$$

From Feys [38], pg. 13 eqn. a501:

$$\models (P \Rightarrow Q) \Leftrightarrow (\neg P \vee Q) \quad (B.6)$$

From Feys [38], pg. 14 eqn. a62:

$$\models [(P \Rightarrow Q) \wedge (Q \Rightarrow R)] \Rightarrow (P \Rightarrow R) \quad (B.7)$$

From Feys [38], pg. 14 eqn. a651:

$$\models [(P \Rightarrow Q) \wedge (P \Rightarrow S)] \Leftrightarrow [P \Rightarrow (Q \vee S)] \quad (B.8)$$

From Feys [38], pg. 14 eqn. a661:

$$\models [(P \Rightarrow Q) \wedge (R \Rightarrow S)] \Leftrightarrow [(P \wedge R) \Rightarrow (Q \wedge S)] \quad (B.9)$$

From Feys [38], pg. 14 eqn. a665:

$$\models [(P \Rightarrow Q) \wedge (R \Rightarrow Q)] \Leftrightarrow [(P \vee R) \Rightarrow Q] \quad (B.10)$$

From Manna and Pnueli [72], pg. 228 eqn. 1:

$$\models \Box \neg P \Leftrightarrow \neg \Diamond P \quad (B.11)$$

From Manna and Pnueli [72], pg. 228 eqn. 2:

$$\models \Diamond \neg P \Leftrightarrow \neg \Box P \quad (B.12)$$

From Manna and Pnueli [72], pg. 228 eqn. 7':

$$\models \Box P \Rightarrow \Diamond P \quad (B.13)$$

From Manna and Pnueli [72], pg. 228 eqn. 8:

$$\models \Box P \Rightarrow \Box \Box P \quad (B.14)$$

From Manna and Pnueli [72], pg. 228 eqn. 9:

$$\models P \vee Q \Rightarrow \Diamond Q \quad (B.15)$$

From Manna and Pnueli [72], pg. 228 eqn. 10:

$$\models \Diamond \Box P \Rightarrow \Box \Diamond P \quad (B.16)$$

From Manna and Pnueli [72], pg. 229 eqn. 11:

$$\models \Box P \Leftrightarrow \Box \Box P \quad (B.17)$$

From Manna and Pnueli [72], pg. 229 eqn. 12:

$$\models \Diamond P \Leftrightarrow \Diamond \Diamond P \quad (B.18)$$

From Manna and Pnueli [72], pg. 229 eqn. 13:

$$\models \Box \Box P \Leftrightarrow \Box \Box P \quad (B.19)$$

From Manna and Pnueli [72], pg. 229 eqn. 14:

$$\models \Diamond \Diamond P \Leftrightarrow \Diamond \Diamond P \quad (B.20)$$

From Manna and Pnueli [72], pg. 229 eqn. 16:

$$\models \Box (P \wedge Q) \Leftrightarrow \Box P \wedge \Box Q \quad (B.21)$$

From Manna and Pnueli [72], pg. 229 eqn. 17:

$$\models \Diamond (P \vee Q) \Leftrightarrow (\Diamond P \vee \Diamond Q) \quad (B.22)$$

From Manna and Pnueli [72], pg. 229 eqn. 18:

$$\models \Box (P \wedge Q) \Leftrightarrow (\Box P \wedge \Box Q) \quad (B.23)$$

From Manna and Pnueli [72], pg. 229 eqn. 20:

$$\models \Box (P \Rightarrow Q) \Rightarrow (\Box P \Rightarrow \Box Q) \quad (B.24)$$

From Manna and Pnueli [72], pg. 230 eqn. 24:

$$\models \Diamond (P \wedge Q) \Rightarrow \Diamond P \wedge \Diamond Q \quad (B.25)$$

From Manna and Pnueli [72], pg. 230 eqn. 27:

$$\models \Box (P \Rightarrow Q) \Rightarrow (\Box P \Rightarrow \Box Q) \quad (B.26)$$

From Manna and Pnueli [72], pg. 231 eqn. 35:

$$\models \Box (P \Rightarrow \Box P) \Rightarrow (P \Rightarrow \Box P) \quad (B.27)$$

From Manna and Pnueli [72], pg. 231 eqn. 41:

$$\models (\Box P \wedge \Diamond Q) \Rightarrow (P \vee Q) \quad (B.28)$$

From Manna and Pnueli [72], pg. 233 eqn. 52:

$$\text{if } \models P \text{ then } \models \Box P \quad (B.29)$$

From Manna and Pnueli [72], pg. 233 eqn. 53:

$$\text{if } \models P \text{ then } \models \Diamond P \quad (B.30)$$

From Manna and Pnueli [72], pg. 233 eqn. 56:

$$\text{if } \models P \Rightarrow Q \text{ then } \models \Diamond P \Rightarrow \Diamond Q \quad (B.31)$$

From Manna and Pnueli [72], pg. 233 eqn. 57:

$$\text{if } \models P \Rightarrow Q \text{ then } \models \Box P \Rightarrow \Box Q \quad (B.32)$$

From Manna and Pnueli [72], pg. 233 eqn. 59:

$$\begin{aligned} \text{if } \models P \Rightarrow \Diamond Q \text{ and } \models Q \Rightarrow \Diamond R \text{ then} \\ \models P \Rightarrow \Diamond R \end{aligned} \quad (B.33)$$

From Manna and Pnueli [72], pg. 234 eqn. 61:

$$\begin{aligned} \text{if } \models P \Rightarrow Q \text{ and } \models Q \Rightarrow \Diamond R \text{ and } \models R \Rightarrow S \text{ then} \\ \models P \Rightarrow \Diamond S \end{aligned} \quad (B.34)$$

From Manna and Pnueli [72], pg. 234 eqn. 62:

$$\begin{aligned} \text{if } \models P \Rightarrow Q \text{ and } \models Q \Rightarrow \Box R \text{ and } \models R \Rightarrow S \text{ then} \\ \models P \Rightarrow \Box S \end{aligned} \quad (B.35)$$

From Hailpern [47], pg. 182 eqn. MP:

$$\text{if } \models P \text{ and } \models P \Rightarrow Q \text{ then } \models Q \quad (B.36)$$

From Hailpern [47], pg. 186 eqn. A6':

$$\models \Box \Diamond P \Rightarrow \Diamond P \quad (B.37)$$

From Hailpern [47], pg. 193 eqn. 18:

$$\models \Diamond \Box (P \wedge Q) \Rightarrow \Diamond \Box P \wedge \Diamond \Box Q \quad (B.38)$$

We prove below the following axiom:

$$\models (\Box P \wedge \Box Q) \Rightarrow P \cup Q \quad (B.39)$$

Proof:

- (1) $\models (\Box P \wedge \Box Q) \Rightarrow \Box P$ (B.3)
- (2) $\models \Box P \wedge \Box Q \Rightarrow \Diamond Q$ (B.12, B.31)
- (3) $\models (\Box P \vee \Box Q) \Rightarrow (\Box P \vee \Diamond Q)$ ((1, 2), B.36, B.7)
- (4) $\models (\Box P \vee \Box Q) \Rightarrow P \cup Q$ (B.22, B.31)

Q.E.D.

We prove below the rule of conjunction:

$$\text{if } \models P \text{ and } \models Q \text{ then } \models P \wedge Q \quad (B.40)$$

Proof:

- (1) $\models \neg(P \wedge Q) \vee (P \wedge Q)$ (B.2)
- (2) $\models \neg P \vee \neg Q \vee (P \wedge Q)$ (B.1)
- (3) $\models P \Rightarrow [Q \Rightarrow (P \wedge Q)]$ (B.6)
- (4) $\models Q \Rightarrow (P \wedge Q)$ (B.31, $\models P$)
- (3) $\models P \wedge Q$ (B.31, $\models Q$)

Q.E.D.

We prove below the rule of syllogism:

$$\text{if } \models P \Rightarrow Q \text{ and } \models Q \Rightarrow R \text{ then } \models P \Rightarrow R \quad (B.41)$$

Proof:

- (1) $\models P \Rightarrow Q$ (given)
- (2) $\models Q \Rightarrow R$ (given)
- (3) $\models (P \Rightarrow Q) \wedge (Q \Rightarrow R)$ ((1, 2), B.40)
- (4) $\models [(P \Rightarrow Q) \wedge (Q \Rightarrow R)] \Rightarrow (P \Rightarrow R)$ (B.7)
- (5) $\models P \Rightarrow R$ ((3, 4), B.36)

Q.E.D.

We prove below the rule of next-eventual:

$$\text{if } \models P \Rightarrow \bigcirc \Diamond Q \text{ and } \models Q \Rightarrow \bigcirc \Diamond R \text{ then } \models P \Rightarrow \bigcirc \Diamond R \quad (B.42)$$

Proof:

- (1) $\models P \Rightarrow \bigcirc \Diamond Q$ (given)
- (2) $\models P \Rightarrow \Diamond \bigcirc Q$ (1, B.20)
- (3) $\models Q \Rightarrow \bigcirc \Diamond R$ (given)
- (4) $\models \bigcirc Q \Rightarrow \bigcirc \bigcirc \Diamond R$ (3, B.14, B.19, B.24)
- (5) $\models \bigcirc Q \Rightarrow \bigcirc \Diamond R$ (4, B.37, B.35)
- (6) $\models P \Rightarrow \Diamond \bigcirc \Diamond R$ (2, 5, B.34)
- (7) $\models P \Rightarrow \bigcirc \Diamond R$ (7, B.20, B.18)

Q.E.D.

We prove below the rule of eventual premise:

$$\text{if } \models P \Rightarrow \bigcirc \Diamond Q \text{ then } \models \underline{\Diamond P} \Rightarrow \bigcirc \Diamond Q \quad (B.43)$$

Proof:

- (1) $\models P \Rightarrow \bigcirc \Diamond Q$ (given)
- (2) $\models \Diamond P \Rightarrow \Diamond \bigcirc \Diamond Q$ (1, B.31, B.29)
- (3) $\models \Diamond P \Rightarrow \bigcirc \Diamond \Diamond Q$ (2, B.20)
- (4) $\models \Diamond P \Rightarrow \bigcirc \Diamond Q$ (3, B.18)

Q.E.D.

We prove below the COL inference rule 4.29:

if $\models P \wedge \Diamond Q$ and $\models Q \Rightarrow \bigcirc \Diamond R$ then $\models P \wedge \bigcirc \Diamond R$

Proof:

- (1) $\models P \wedge \Diamond Q$ (given)
- (2) $\models P$ (1, B.3)
- (3) $\models \Diamond Q$ (2, B.3)
- (4) $\models Q \Rightarrow \bigcirc \Diamond R$ (given)
- (5) $\models \Diamond Q \Rightarrow \Diamond \bigcirc \Diamond R$ (4, B.31)
- (6) $\models \Diamond \bigcirc \Diamond R$ (5, B.36)
- (7) $\models \bigcirc \Diamond R$ (6, B.18, B.18)
- (8) $\models P \wedge \bigcirc \Diamond R$ (2, 7, B.40)

Q.E.D.

We prove below the COL inference rule 4.30:

if $\models P \wedge \bigcirc \Diamond Q$ and $\models Q \Rightarrow \bigcirc \Diamond R$ then $\models P \wedge \bigcirc \Diamond R$

Proof:

- (1) $\models P \wedge \bigcirc \Diamond Q$ (given)
- (2) $\models P$ (1, B.3)
- (3) $\models \bigcirc \Diamond Q$ (2, B.3)
- (4) $\models Q \Rightarrow \bigcirc \Diamond R$ (given)
- (5) $\models \bigcirc \Diamond R$ (4, B.42)
- (6) $\models P \wedge \bigcirc \Diamond R$ (2, 5, B.40)

Q.E.D.

We prove below the COL inference rule 4.31:

if $\models P \wedge \bigcirc Q$ and $\models Q \Rightarrow \bigcirc \Diamond R$ then $\models P \wedge \bigcirc \Diamond R$

Proof:

- (1) $\models P \wedge \bigcirc Q$ (given)
- (2) $\models P$ (1, B.3)
- (3) $\models \bigcirc Q$ (2, B.3)
- (4) $\models Q \Rightarrow \bigcirc \Diamond R$ (given)
- (5) $\models Q \Rightarrow \Diamond R$ (4, B.37, B.41)
- (6) $\models \bigcirc Q \Rightarrow \bigcirc \Diamond R$ (5, B.32)
- (7) $\models \bigcirc \Diamond R$ (3, 6, B.36)
- (8) $\models P \wedge \bigcirc \Diamond R$ (2, 7, B.40)

Q.E.D.

Appendix C

Design Examples

In this appendix, we describe the design examples used to illustrate the capabilities of our analysis tool-set. The examples provide some insight to performance for both well-known problems and typical designs. We summarize the design examples we have selected and the selection criteria below.

- 9 *Readers and Writers* — the Readers and Writers problem [17,30] shows the ability of the analysis system to detect starvation; we present two versions from [17], one with starvation (Read-S) and one without (Read). The starvation example will also serve to illustrate the COL Interpreter in Appendix D.
- Five Dining Philosophers* — the Five Dining Philosophers problem [57] demonstrates starvation and deadlock analysis on a design which exhibits considerable non-deterministic behaviour; three designs are presented: one with deadlock (Diner-D), one with starvation (Diner-S) and one that is starvation and deadlock-free (Diner).
- Gas-Station* — the Gas-Station problem [53] illustrates the critical race analysis facility; we provide two designs, one with a critical race (Gas-CR) and a corrected implementation (Gas).
- Dialogue* — the Dialogue example [17] shows the analysis of a relatively simple design (5 tasks) with limited non-deterministic behaviour. It is free of temporal errors.
- Remote Mailbox System* — the Remote Mailbox System (RMS) [80] is an example of a three layer communications system (service, link and driver layers). It demonstrates the ability of the analysis system to handle complex designs. We examine 4 variations of a simplified version¹ of the system: the basic

¹We do not include all of the functionality of the RMS problem.

RMS system (RMS), the basic RMS system with deadlock due to a critical race (RMS-D), analysis of the service and higher layers (RMS-S) and analysis of the link and lower layers (RMS-L). The latter two designs illustrate the analysis of subsystems in order to simplify of the evaluation of complex designs.

Alternating Bit Protocol — An implementation of the Alternating Bit Protocol [47] in the link layer of the RMS problem shows the analysis of a protocol implementation.

Ten Task — Ten Task illustrates the power of the inference engine to handle task deterministic programs, even with a large number of tasks (e.g. ten tasks).

Detailed descriptions of the various designs are presented in this appendix. These designs were developed in consideration of the practical analysis heuristics presented in chapter 9. For example, the decision to split the RMS design into two pieces was an application of the *divide and conquer* heuristic. The heuristics used in the designs are indicated.

The performance characteristics in which we are interested include: the number of PSA generated and recorded during deadlock analysis, the time required for deadlock analysis, the time required for pre-processing in starvation analysis, the time required for search in starvation analysis and the time required to identify non-deterministic rendezvous for critical race analysis. The implementation environment was a DEC AI VAXStation under MicroVMS V4.3, with 9 MBytes of memory using compiled MProlog V2.1. A summary of the analysis results is presented in Table C.1. A more powerful work station and a more powerful Prolog compiler could yield a ten-fold increase in speed.

C.1 Readers and Writers

The basic readers and writers example with one writer has been described in detail in chapter 3. We use this same example employing two readers to illustrate the starvation analysis. The structure graph for this design is shown in Figure C.1. The deadlock analyzer proved that this design was system-wide deadlock-free. The starvation analysis revealed that the two readers could alternately lock out the writer from the resource. A variation on this design which solves the starvation problem is suggested by Buhr [17]. In the starvation-free design, the writer is always accepted; however, if there are still readers in operation then the `End_acc` entry is accepted continually (while the writer is held) until no readers remain. This is a solution using a *nested accept*; that is, one accept statement is nested within another. The ND-Analyzer detected non-deterministic rendezvous in the Scheduler task in both designs (as would be expected).

Name	#Task	#PSA	D_t	$S1_t$	$S2_t$	ND_t
Read	4	36	29	6	< 1	2
Read-S	4	22	19	3	1	1
Diner	6	1137	7589	1767	164	381
Diner-D	6	439	1937	—	—	—
Diner-S	6	62	82	18	24	5
Gas	4	60	58	11	< 1	3
Gas-CR	4	98	108	24	< 1	3
Dialog	5	70	183	19	< 1	4
RMS	10	1356	11048	2769	1454	1041
RMS-D	10	1978	17736	—	—	—
RMS-L	6	66	158	16	< 1	3
RMS-S	5	38	75	7	< 1	2
Alt-Bit	7	150	360	71	10	15
10 Task	10	4	11	1	1	< 1

D_t — deadlock analysis time

$S1_t$ — starvation analysis, pre-processing time

$S2_t$ — starvation analysis, searching time

ND_t — critical race analysis, non-determinism search time

All times quoted in seconds

Table C.1: Summary of Analysis Results for All Design Examples

Figure C.1: Structure Graph: Readers and Writers

We have employed two analysis heuristics to simplify the specification of the design: (1) the *minimal task pool* heuristic is used to reduce the number of readers to two and the number of writers to one and (2) the *no excess statements* heuristic is used to eliminate all activity other than access request and release.

C.2 Five Dining Philosophers

The five dining philosophers problem illustrates the notions of deadlock and starvation. The problem is as follows: five philosophers sit around a table ready to dine, however there is but one fork per person and two forks are required to eat. Each fork is placed between the philosophers, thus there is a left-hand fork and a right-hand fork. Each diner must execute the same algorithm for acquiring and releasing the forks; furthermore, once a pair of forks are acquired, the diners will return them, eventually (since they must periodically stop and think).

We model this problem in Ada by 6 tasks, five diners and a fork manager. In our first attempt at a solution, we construct a fork manager with 6 entries: 5 representing each fork at the table and a fork return entry. The fork request entries are numbered according to the philosopher on the left. Each diner calls the correct fork request entry to acquire a fork, thus, two such calls must be made. Each fork request entry is guarded such that only one diner can acquire the requested fork at a time. Forks are returned two at a time to the fork manager.

The algorithm used by the diners is simple: get the left fork, get the right fork, eat, release forks, think, start over. The structure graph for this system is illustrated in Figure C.2. This rather simple implementation can deadlock if all

Figure C.2: Structure Graph: Five Dining Philosophers, w/deadlock

diners acquire the left fork simultaneously; a fact that is verified by the deadlock analysis system.

We improve upon this system by the design of Figure C.3 in which each request entry returns the fork pair of the calling diner task. If the fork pair is not currently available then the fork pair request entry is closed. We have eliminated deadlock from this system since no task can hold indefinitely one of the resources; i.e. resources are handed over collectively, not individually. Unfortunately, we have introduced starvation into the design: two diners on either side of a third diner can lock out the middle diner from eating by cooperatively acquiring one of the middle diners forks. The fork manager may dole out forks two at a time, but it does so rather indiscriminately; that is, it is not necessarily fair.

In our third design, shown in Figure C.4, we simplify the interface of the fork manager task to a single request entry and a return entry and we add to the diners a "forks assigned" entry. The system functions as follows:

- each diner places a request for a fork pair by calling the request entry;
- once the request is registered, the diner accepts the forks assigned entry;

Figure C.3: Structure Graph: Five Dining Philosophers_w/starvation

Figure C.4: Structure Graph: Five Dining Philosophers, corrected

- the fork manager task maintains a FIFO queue of diners for each fork; whenever the same diner percolates to the top of two queues then its forks are assigned by a call to the diner's assign entry.

The FIFO fork assignment scheme implemented by the fork manager guarantees that any fork request is eventually serviced; thus, there is no deadlock since resources are assigned in pairs and no starvation since fork requests are guaranteed to be serviced. Indeed, this was shown to be true by the deadlock and starvation analyzers.

In all of these designs we have applied the *no excess statements* analysis heuristic to remove any activity other than fork acquisition and return from the specification. The *minimal task pool* heuristic could not be applied because all of the diner tasks affect each other.

Figure C.5: Structure Graph: Gas-Station, w/critical race

C.3 Gas-Station

- The gas-station problem has been described in detail in chapter 7. This design (shown in Figure C.5), consists of a set of gas pumps, a group of customers and a gas pump operator. In essence, the operator allows customers to prepay gas for a particular pump and then activates the pump for use. Change is returned to the customer when the actual gas consumption is known.

The critical race occurs between customer tasks trying to reach the start entry of a given pump. The operator queues up prepayment of gas in FIFO order, but does not enforce the same arrival order at the pump. The end result is that customers may receive incorrect change. The critical race analyzer demonstrated that, for a two customer, one pump system, two customers could arrive at the start entry of the same pump non-deterministically, which violates the intended behaviour of the system.

In a corrected version of this design, illustrated in Figure C.6, we add a start entry to each customer. We modify the behaviour of the operator so that it calls the start entry of each customer when that customer has reached the top of the operators FIFO queue of prepaid customers of a given pump. In this way the customers do not queue themselves, rather they are indirectly queued by the operator. Given this correction, the ND-Analyzer indicated that all rendezvous with the pump were deterministic.

We have used the *minimal task pools* analysis heuristic to reduce the specifica-

Figure C.6: Structure Graph: Gas-Station, corrected

tion of the design to one pump and two customers. We used one pump because all pumps are completely autonomous and we used two customers in order to study the effect of customers competing for the same pump.

C.4 Dialogue

The dialogue example implements a two-way message-based conversation system between two work stations. The concept is relatively simple: a user sends a message from a local terminal to a remote terminal by pressing a *reserve* key, the message text and a *send* key. The reserve and send keys guarantee that the screen is not corrupted by incoming messages. When the send key is pressed, the message currently displayed on the screen is sent to the remote terminal through some underlying communications system. Incoming messages are displayed on the screen whenever it is not reserved.

The high level structure graph for this system is illustrated in Figure C.7. The four packages, event decoder, keyboard messenger and screen services implement all operations. For analysis purposes, we simulate the keyboard by an unending stream of reserve and send keystrokes with a nul message; furthermore, we simulate the messenger as an infinite source and sink of messages. The screen services package is a set of passive procedures for manipulating the screen image. The event decoder coordinates keystroke and message events to produce correct screen images.

Figure C.7: Structure Graph: Dialogue Example

Internally, the event decoder package, (shown in Figure C.8), has three tasks: the event decoder task, a get keystroke transport task and a get message transport task. The two transport tasks bring messages and keystrokes to the event decoder task. When the screen is reserved, the put message entry of the decoder task is closed so that any incoming messages cannot be accepted. The put keystroke entry is always open. The event decoder task linearizes the concurrent message and keystroke events. The analysis tool-set (with appropriate human assistance) demonstrated that this design was deadlock, starvation and critical race-free.

The development of this design saw the use of three analysis heuristics: (1) *no excess statements* — the screen services package was ignored, (2) *simulate external events* — the keyboard and message system were simulated and (3) *minimal buffer size* — nul messages were transmitted.

C.5 Remote Mailbox System

The remote mailbox system is a layered communications system for transmitting mail messages between RMS nodes in a network. Each node implements the same software structure. The basic layered system is illustrated in Figure C.9. The user end is represented by a user task (a source of messages) and a mailbox task (a sink of messages). The transmission end is defined by the hardware task which provides character transmission between sender and receiver. We simulate the presence of another node by allowing the same node to act as both sender and receiver.

The service layer (see Figure C.10) provides service messages to be sent by the link layer to a particular node. It identifies the appropriate link by a routing table. Service messages are acknowledged by the receiving service layer. Outgoing mail messages are passed to the service layer by the send remote mailbox procedure. These messages are converted into service messages and passed to the link layer. Every outgoing message is recorded with an acknowledge status task. This task will match up incoming acknowledgements with outgoing messages and place an acknowledgement in the mailbox. If any unexpected acknowledgements arrive, they are ignored. Incoming messages arrive through the link to service layer procedure. These messages are propagated to the mailbox task; acknowledgement service messages are generated and passed down to the link layer.

The link layer (see Figure C.11) provides link messages to be sent by the driver layer over a specific link. The link manager task controls the flow of all incoming and outgoing link messages. Two transport tasks are used to: (1) transfer incoming service messages to the service layer and (2) transfer outgoing link messages to the driver layer. The link manager entries for these transport tasks are closed when there are no messages. Outgoing service messages arrive at the link layer

Figure C.8: Structure Graph: Dialogue Event Decoder Package

Figure C.9: Structure Graph: Remote Mailbox System

Figure C.10: Structure Graph: Remote Mailbox System, Service Layer

through the service to link procedure and incoming link messages enter through the driver to link procedure. The link layer acknowledges every link message that is sent. The acknowledgement must arrive before another message is sent.

The driver layer (see Figure C.12) consists primarily of two interrupt driven tasks, one for incoming characters and one for outgoing characters. The TX interrupt task buffers one link message and sends it one character at a time through the hardware. The hardware signals an interrupt whenever it is ready to accept a character. The TX interrupt task then transmits the character by a call to the hardware. Outgoing link messages wait at the TX message entry until the currently buffered message is sent. The RX interrupt task assembles one incoming link message from characters arriving from the hardware. Every time a character has arrived in the hardware, an interrupt is generated; the RX interrupt task then calls the hardware for the character. Complete incoming link messages are passed by transport task to the link layer. The transport task waits at the RX message entry of the RX interrupt task whenever there are no incoming link messages ready.

We model this system by imposing a few simplifications: (1) the user task will not transmit a new mail message until its last one is acknowledged, (2) there is only one self-link, thus, the system is both sender and receiver, (3) the hardware transmission sends all characters at once instead of one at a time.

In the first design of this system, we include a design flaw: when sending a service message, the user task (through the send remote mailbox procedure) passes the service message to the link layer before registering the transmission with the acknowledge status task. Under these circumstances, a critical race would ensue between the arrival of the acknowledgement and the registration of the transmission. If the acknowledgement arrives first, then it is discarded and a deadlock occurs because the user task will never receive a mail acknowledgement (hence it will not generate any new messages). This design is described in more detail in chapter 2. We rectify this situation in a second design by reversing the order of the transmission activities; that is, we register the expected acknowledgement first and then forward the service message.

The deadlock analysis associated with both designs is rather long because of their complexity (10 tasks with many non-deterministic interactions). Although better hardware and software could improve the speed of analysis by nearly a factor of 10, there is perhaps a better route. As an experiment, we partitioned the RMS system in half and analyzed each subsystem separately; to maintain correctness, we simulated the behaviour of the missing half. One analysis considered the service layer and above, while the other studied the link layer and below. The same conclusions were obtained at a fraction of the analysis cost, as summarized in Table C.2.

The partitioning experiment is an example of the *divide and conquer* analysis

Figure C.11: Structure Graph: Remote Mailbox System, Link Layer

Figure C.12: Structure Graph: Remote Mailbox System, Driver Layer

Name	#Task	#PSA	D_t	$S1_t$	$S2_t$	ND_t
RMS	10	1356	11048	2769	1454	1041
RMS-Dead	10	1978	17736	—	—	—
RMS-Link	6	66	158	16	< 1	3
RMS-Service	5	38	75	7	< 1	2

D_t — deadlock analysis time

$S1_t$ — starvation analysis, pre-processing time

$S2_t$ — starvation analysis, searching time

ND_t — critical race analysis, non-determinism search time

All times quoted in seconds

Table C.2: Summary of Analysis Results for RMS Design Examples

heuristic; i.e. divide a design into subsystems to reduce the total number of tasks to be analyzed. Two other analysis heuristics were used: (1) *simulate external events* — to simplify the modelling of the user, mailbox, communication hardware and another node and (2) *minimal buffer size* — no more than one mail, service, link or driver message could be outstanding.

C.6 Alternating Bit Protocol

The alternating bit protocol is implemented for the link layer of the RMS example. The design used for analysis, illustrated in Figure C.13 employs a simulated driver layer, infinite sources and sinks of service messages and feedback of transmitted messages; i.e. the system is both sender and receiver. The alternating bit protocol is a relatively simple communications protocol. In this technique, a bit concatenated to outgoing messages, is used by the receiving system to decide if the received message is expected or appropriate. For example, If an unexpected bit arrives with the incoming message, the receiver assumes that the acknowledgement of the last received message was lost and then retransmits the acknowledgement. We do not include timer facilities for retransmission time-outs that are normally required; all retransmission is based on unexpected responses. Retransmission by time-out suggests that the application can behave non-deterministically, a facility that is not currently supported. Using our simpler version, the analysis tools show that the system is free of temporal errors.

We have used two analysis heuristics in development of this example: (1) *simulate external events* — to simplify the modelling of the user, mailbox and driver layer and (2) *minimal buffer size* — no more than one mail, or link message

could be buffered ahead.

C.7 Ten Task

The ten task example, illustrated in Figure C.14, consists of five even numbered tasks and five odd numbered tasks that produce a completely task deterministic system. The tasks form a chain of callers in which odd tasks call successor even tasks and then accept predecessor even tasks; e.g. task 1 calls task 2 and then accepts task 10. Consequently, even tasks accept predecessor odd tasks and then call successor odd tasks; e.g. task 2 accepts task 1 and then calls task 3. The deadlock analyzer proves that this system is system-wide deadlock-free in just 4 recorded PSA. This demonstrates the power of the deadlock analyzer to analyze quickly task deterministic systems without any pre-processing.

Figure C.13: Structure Graph: Alternating Bit Design

Figure C.14: Structure Graph: Ten Task Design

Appendix D

COL Language and Interpreter Example

In this appendix, we provide an example of the use of COL to specify an Ada design and we illustrate the behaviour of the COL Interpreter using this same example. The design of interest is the Readers and Writers problem, with starvation described in § C.1.

D.1 Readers and Writers COL Specification

The readers and writers example with two readers and one writer appears below as a COL specification of the Ada design. The description consists of a reader(N) task, representing the reader task type, writer(N) representing the writer task type and a scheduler task. Besides the basic parameter list variable (prm([])), present in all tasks, the scheduler is the only task with other variables; these are: nr(N) — the number of readers with read permission, read(FLAG) — a boolean variable guarding the read_req entry and write(FLAG) — a boolean variable guarding the write_req entry. The boolean flags must have the value t in order for the entries to be open.

In the text below, comments are preceded by percent symbols (%).

```
% Initial PSA definition—
%
% initially, no readers have permission and both request
% entries are open

init_set([
  (reader(1).<>(init(reader(1)) & ts(reader(1),[prm([])].[[]])).
```

```

    (reader(2).<>(init(reader(2)) & ts(reader(2).[prz([])].[[]])).
    (writer(1).<>(init(writer(1)) & ts(writer(1).[prz([])].[[]])).
    (scheduler.<>(init(scheduler) &
    ts(scheduler.[prz([]).nr(0).read(t).write(t)].[[]]))
  ])].

```

```

%   reader(N) task

```

```

%   call request entry

```

```

init(reader(N)) & ts(reader(N).=[].Q) =>
  *<>(task_call(reader(N),reader(N)!11,scheduler!read_req)
  & ts(reader(N).=[].Q)).

```

```

%   call return entry (intermediate activity omitted)

```

```

return_task_call(reader(N),reader(N)!11,scheduler!read_req)
& ts(reader(N).=[].Q) =>
  *<>(task_call(reader(N),reader(N)!12,scheduler!fin)
  & ts(reader(N).=[].Q)).

```

```

%   go to top of loop

```

```

return_task_call(reader(N),reader(N)!12,scheduler!fin)
& ts(reader(N).=[].Q) =>
  *<>(init(reader(N)) & ts(reader(N).=[].Q)).

```

```

%   writer(N) task

```

```

%   call request entry

```

```

init(writer(N)) & ts(writer(N).=[].Q) =>
  *<>(task_call(writer(N),writer(N)!11,scheduler!write_req)
  & ts(writer(N).=[].Q)).

```

```

%   call return entry (intermediate activity omitted)

```

```

return_task_call(writer(N),writer(N)!11,scheduler!write_req)
& ts(writer(N).=[].Q) =>
  *<>(task_call(writer(N),writer(N)!12,scheduler!fin)

```



```

    & ts(writer(N).=[].Q)).

%   go to top of loop

return_task_call(writer(N).writer(N)!12.scheduler!fin)
& ts(writer(N).=[].Q) =>
    *<>(init(writer(N)) & ts(writer(N).=[].Q)).

%   scheduler task

%   enter selective accept

init(scheduler) & ts(scheduler.=[].Q) =>
    *<>(sel_accept(scheduler.scheduler!1) & ts(scheduler.=[].Q)).

    *<>(sel_accept(scheduler.scheduler!1) & ts(scheduler.V.Q)) ::= [
        (=[read(t)].
            *<>(accept(scheduler.scheduler!read_req)
                & ts(scheduler.V.Q))).
        (=[].
            *<>(accept(scheduler.scheduler!fin) & ts(scheduler.V.Q))).
        (=[write(t)].
            *<>(accept(scheduler.scheduler!write_req)
                & ts(scheduler.V.Q)))
    ].

%   leave selective accept and go to top of loop

end_sel_accept(scheduler.scheduler!1) & ts(scheduler.=[].Q) =>
    *<>(init(scheduler) & ts(scheduler.=[].Q)).

%   accept writer. then close read request entry

start_rendez(scheduler.scheduler!write_req)
& ts(scheduler.=[].Q) =>
    *<>(end_rendez(scheduler.scheduler!write_req)
        & ts(scheduler.=[read(f).write(f)].Q)).

%   accept writer to finish (since reader entry closed).
%   open reader entry

```

```

start_rendez(scheduler, scheduler!fin)
& ts(scheduler,=[read(f)].Q) =>
    *<>(end_rendez(scheduler, scheduler!fin)
    & ts(scheduler,=[read(t).write(t)].Q)).

%   accept last reader to finish, open writer entry

start_rendez(scheduler, scheduler!fin)
& ts(scheduler,=[nr(1)].Q) =>
    *<>(end_rendez(scheduler, scheduler!fin)
    & ts(scheduler,=[nr(0).write(t)].Q)).

%   accept non-last reader to finish, decrement number
%   of readers

start_rendez(scheduler, scheduler!fin)
& ts(scheduler,=[nr(X), @ (X>1)].Q) =>
    *<>(end_rendez(scheduler, scheduler!fin)
    & ts(scheduler,=[@ (X2 is X-1), nr(X2)].Q)).

%   accept reader request, increment number of readers
%   and close writer request entry

start_rendez(scheduler, scheduler!read_req)
& ts(scheduler,=[nr(X)].Q) =>
    *<>(end_rendez(scheduler, scheduler!read_req)
    & ts(scheduler,=[@ (X2 is X+1), nr(X2).write(f)].Q)).

%   no non-critical sections for any entries

end_accept(scheduler, scheduler!read_req) & ts(scheduler,=[]).Q) =>
    *<>(end_sel_accept(scheduler, scheduler!1)
    & ts(scheduler,=[]).Q)..

end_accept(scheduler, scheduler!fin) & ts(scheduler,=[]).Q) =>
    *<>(end_sel_accept(scheduler, scheduler!1)
    & ts(scheduler,=[]).Q)).

end_accept(scheduler, scheduler!write_req) & ts(scheduler,=[]).Q) =>
    *<>(end_sel_accept(scheduler, scheduler!1)

```

```
& ts(scheduler,=[].Q)).
```

D.2 Readers and Writers Interpreter Transcript

In this COL interpreter transcript we examine the execution behaviour that produces starvation in the readers and writers problem. The example illustrates the interpreter's main features: one-step inference (infer command), expansion (expand command) and execution backtracking (backtrack command). We do not show the entire transcript, only highlights. Comments are preceded by percent symbols (%).

```
% we join the interpreter in progress when both readers have
% gained read permission; they are now attempting to
% relinquish permission; the writer is cannot be accepted
% because the the write_req entry is closed (i.e. write(f));
%, the number of readers with permission is 2 (i.e. nr(2));
```

Next action?

Enter one of infer expand backtrack quit help {infer}

> infer

```
reader(1) : task call by reader(1)!12 on entry scheduler!fin.
           vars: [prm({})]
reader(2) : task call by reader(2)!12 on entry scheduler!fin.
           vars: [prm({})]
writer(1)  : task call by writer(1)!11 on entry scheduler!write_req.
           vars: [prm({})]
scheduler : selective accept 1.
           vars: [prm({}).nr(2).read(t).write(f)]
```

Next action?

Enter one of infer expand backtrack quit help {infer}

> expand

```
reader(1) : task call by reader(1)!12 on entry scheduler!fin.
           vars: [prm({})]
reader(2) : task call by reader(2)!12 on entry scheduler!fin.
           vars: [prm({})]
```

```

writer(1) : task call by writer(1)!11 on entry scheduler!write_req.
            vars: [prm([])]
scheduler : start accept of call from reader(1)!12 on entry
            scheduler!fin. vars: [prm([]).nr(2).read(t).write(f)]

```

% we allow reader(1) to relinquish permission

Expansion 1. use this?

Enter one of use next {next}

> use

```

reader(1) : task call by reader(1)!12 on entry scheduler!fin.
            vars: [prm([])]
reader(2) : task call by reader(2)!12 on entry scheduler!fin.
            vars: [prm([])]
writer(1) : task call by writer(1)!11 on entry scheduler!write_req.
            vars: [prm([])]
scheduler : start accept of call from reader(1)!12 on entry
            scheduler!fin. vars: [prm([]).nr(2).read(t).write(f)]

```

Next action?

Enter one of infer expand backtrack quit help {infer}

> infer

```

reader(1) : call reader(1)!12 has been accepted on scheduler!fin.
            vars: [prm([])]
reader(2) : task call by reader(2)!12 on entry scheduler!fin.
            vars: [prm([])]
writer(1) : task call by writer(1)!11 on entry scheduler!write_req.
            vars: [prm([])]
scheduler : start rendezvous with call from reader(1)!12 on entry
            scheduler!fin. vars: [prm([]).nr(2).read(t).write(f)]

```

Next action?

Enter one of infer expand backtrack quit help {infer}

> infer

```

reader(1) : call reader(1)!12 has been accepted on scheduler!fin.
            vars: [prm([])]
reader(2) : task call by reader(2)!12 on entry scheduler!fin.
            vars: [prm([])]

```

```

writer(1) : task call by writer(1)!11 on entry scheduler!write_req.
            vars: [prm([])]
scheduler : ending rendezvous on entry scheduler!fin.
            vars: [prm([]).nr(1).read(t).write(f)]

```

Next action?

Enter one of infer expand backtrack quit help {infer}

> infer

```

reader(1) : return from task call by reader(1)!12 on entry
            scheduler!fin. vars: [prm([])]
reader(2) : task call by reader(2)!12 on entry scheduler!fin.
            vars: [prm([])]
writer(1) : task call by writer(1)!11 on entry scheduler!write_req.
            vars: [prm([])]
scheduler : ending accept of entry scheduler!fin.
            vars: [prm([]).nr(1).read(t).write(f)]

```

Next action?

Enter one of infer expand backtrack quit help {infer}

> infer

```

reader(1) : loop point.
            vars: [prm([])]
reader(2) : task call by reader(2)!12 on entry scheduler!fin.
            vars: [prm([])]
writer(1) : task call by writer(1)!11 on entry scheduler!write_req.
            vars: [prm([])]
scheduler : end of selective accept 1.
            vars: [prm([]).nr(1).read(t).write(f)]

```

Next action?

Enter one of infer expand backtrack quit help {infer}

> infer

```

reader(1) : task call by reader(1)!11 on entry scheduler!read_req.
            vars: [prm([])]
reader(2) : task call by reader(2)!12 on entry scheduler!fin.
            vars: [prm([])]
writer(1) : task call by writer(1)!11 on entry scheduler!write_req.
            vars: [prm([])]

```

```
scheduler : loop point,
           vars: [prm([]).nr(1),read(t),write(f)]
```

Next action?

Enter one of infer expand backtrack quit help {infer}

> infer

```
reader(1) : task call by reader(1)!l1 on entry scheduler!read_req,
           vars: [prm([])]
```

```
reader(2) : task call by reader(2)!l2 on entry scheduler!fin,
           vars: [prm([])]
```

```
writer(1) : task call by writer(1)!l1 on entry scheduler!write_req,
           vars: [prm([])]
```

```
scheduler : selective accept 1,
           vars: [prm([]).nr(1),read(t),write(f)]
```

Next action?

Enter one of infer expand backtrack quit help {infer}

> expand

```
% at this point we omit several steps which allow
% reader(1) to be accepted for read_req and reader(2) to be
% accepted for fin; in effect they reverse their roles
% as shown below
```

Next action?

Enter one of infer expand backtrack quit help {infer}

> expand

```
reader(1) : task call by reader(1)!l2 on entry scheduler!fin,
           vars: [prm([])]
```

```
reader(2) : task call by reader(2)!l1 on entry scheduler!read_req,
           vars: [prm([])]
```

```
writer(1) : task call by writer(1)!l1 on entry scheduler!write_req,
           vars: [prm([])]
```

```
scheduler : start accept of call from reader(1)!l2 on entry
           scheduler!fin, vars: [prm([]).nr(1),read(t),write(f)]
```

Expansion 1. use this?

Enter one of use next {next}

> next

```

reader(1) : task call by reader(1)!l2 on entry scheduler!fin.
           vars: [prm([])]
reader(2) : task call by reader(2)!l1 on entry scheduler!read_req.
           vars: [prm([])]
writer(1) : task call by writer(1)!l1 on entry scheduler!write_req.
           vars: [prm([])]
scheduler : start accept of call from reader(2)!l1 on entry
            scheduler!read_req.
            vars: [prm([]),nr(1),read(t),write(f)]

```

Expansion 2. use this?

Enter one of use next {next}

> use

```

reader(1) : task call by reader(1)!l2 on entry scheduler!fin.
           vars: [prm([])]
reader(2) : task call by reader(2)!l1 on entry scheduler!read_req.
           vars: [prm([])]
writer(1) : task call by writer(1)!l1 on entry scheduler!write_req.
           vars: [prm([])]
scheduler : start accept of call from reader(2)!l1 on entry
            scheduler!read_req.
            vars: [prm([]),nr(1),read(t),write(f)]

```

```

% selecting this alternative continues along the starving
% computation by eventually returning us to the starting
% point of this transcript; (i.e. when both readers have
% read permission); thus we decide to backtrack and use the
% other expansion alternative to return us to a point in
% which the writer can gain access

```

Next action?

Enter one of infer expand backtrack quit help {infer}

> backtrack

```

reader(1) : task call by reader(1)!l2 on entry scheduler!fin.
           vars: [prm([])]
reader(2) : task call by reader(2)!l1 on entry scheduler!read_req.
           vars: [prm([])]
writer(1) : task call by writer(1)!l1 on entry scheduler!write_req.

```

```

vars: [prm([])]
scheduler : selective accept 1.
vars: [prm([]),nr(1),read(t),write(f)]

```

Next action?

Enter one of infer expand backtrack quit help {infer}

> expand

```

reader(1) : task call by reader(1)!12 on entry scheduler!fin.
vars: [prm([])]
reader(2) : task call by reader(2)!11 on entry scheduler!read_req.
vars: [prm([])]
writer(1) : task call by writer(1)!11 on entry scheduler!write_req.
vars: [prm([])]
scheduler : start accept of call from reader(1)!12 on entry
scheduler!fin. vars: [prm([]),nr(1),read(t),write(f)]

```

% select alternative in which all readers relinquish control

Expansion 1, use this?

Enter one of use next {next}

> use

```

reader(1) : task call by reader(1)!12 on entry scheduler!fin.
vars: [prm([])]
reader(2) : task call by reader(2)!11 on entry scheduler!read_req.
vars: [prm([])]
writer(1) : task call by writer(1)!11 on entry scheduler!write_req.
vars: [prm([])]
scheduler : start accept of call from reader(1)!12 on entry
scheduler!fin. vars: [prm([]),nr(1),read(t),write(f)]

```

% we omit some of the intermediate steps and rejoin the
 % interpreter when all readers have left the scheduler; i.e.
 % the write flag variable guarding the write_req entry is
 % now true, write(t) and the number of readers is zero, nr(0).

Next action?

Enter one of infer expand backtrack quit help {infer}

> infer

```

reader(1) : task call by reader(1)!11 on entry scheduler!read_req.

```



```
vars: [prm([])]
reader(2) : task call by reader(2)!11 on entry scheduler!read_req.
vars: [prm([])]
writer(1) : task call by writer(1)!11 on entry scheduler!write_req.
vars: [prm([])]
scheduler : selective accept 1.
vars: [prm([]),nr(0),read(t),write(t)]
```

END

3|0.1 2. |87

FIN