DESIGN AND VERIFICATION OF DISTRIBUTED TASKING SUPERVISORS FOR CONCURRENT PROGRAMMING LANGUAGES

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By
David Samuel Rosenblum
March 1988

Abstract

A *tasking supervisor* implements the concurrency constructs of a concurrent programming language. This thesis addresses two fundamental issues in constructing *distributed* implementations of a concurrent language: (1) Principles for designing a tasking supervisor for the language, and (2) Practical techniques for verifying that the supervisor correctly implements the semantics of the language. Previous research in concurrent languages has focused on the design of constructs for expressing concurrency, while ignoring these two important implementation issues.

First, the thesis describes the design of a tasking supervisor for the Ada programming language. The Supervisor implements the full Ada tasking language, and it performs distributed program execution on multiple CPUs. The Supervisor is a portable, modular, distributed software system written in Ada. The interface between the Supervisor and application programs forms the topmost layer of the Supervisor and is formally specified in *Anna* (ANNotated Ada). All machine dependences are encapsulated in the bottom layer of the Supervisor; this layer is an implementation of an *abstract virtual loosely coupled multiprocessor*. The principles used to design the Supervisor may be used to design a distributed supervisor for *any* concurrent language.

Second, the thesis presents new and practical techniques for automatically verifying the behavior of a distributed supervisor; these techniques are illustrated by the verification of the Distributed Ada Supervisor. An event-based formalization of the Ada tasking semantics is expressed as a collection of machine-processable specifications written in *TSL* (Task Sequencing Language). Correctness of the Supervisor is established by automatically checking executions of test programs for consistency with the TSL specifications. Since the specifications are derived solely from the Ada semantics, the specifications can be used to test *any* implementation of Ada tasking. In addition, *every* Ada tasking program may be used as test input.

The theory and practice of concurrent programming is in its infancy. The research described in this thesis represents a major step toward the development of a theory of constructing multiprocessor implementations of concurrent programming languages.

Acknowledgments

Many people have given their time and talent in helping me to complete this dissertation. First and foremost among these people is my principal advisor, David Luckham, who for five years has been a constant source of inspiration and sage advice, teaching me to hold my research to only the highest and most rigorous of intellectual standards.

My associate advisor, Susan Owicki, whose thoroughness was especially helpful in shaping the more theoretical aspects of this thesis, provided countless insightful comments and observations.

Doug Bryan and Geoff Mendal read portions of this thesis in a different form and contributed many comments that helped clarify my understanding of Ada tasking. In addition, Sriram Sankar was very helpful in developing the presentation of various material related to formal specifications. The comments and suggestions of my third reader, Jim Angell, were also helpful.

My colleagues Dave Helmbold, Sigurd Meldal, Manu Thapar, Will Tracz and Neel Madhav provided many astute suggestions during various public presentations of this research. Dave Olien of Sequent was a great source of knowledge on the more subtle aspects of the Balance shared memory architecture, and Steve Deller of VERDIX helped shed light on a few of the dark corners of the VADS Environment.

The research described in this dissertation was supported by DARPA contract N00039-84-C0211 and AFOSR contract 87-0150.

My parents instilled in me the desire to strive for excellence in my education, and their constant support over the years is greatly appreciated.

I dedicate this dissertation to my wife Sarah. The completion of this dissertation would never have been possible without her patience, endurance and loving support.

Table of Contents

List of Tables xv

List of Figures xvii

1 Introduction 1

2 Background and Related Work 5

2.1 Concurrent Programming Languages . . . 5

2.1.1 Concurrent Pascal . . . 7

2.1.2 CSP . . . 8

2.1.3 Ada . . . 10

Features of the Ada Tasking Language . . . 10

Implementations of Ada . . . 12

Towards Distributed Ada Tasking . . . 14

2.2 Distributed Operating Systems . . . 15

2.2.1 Special Features . . . 16

2.2.2 The V System . . . 17

2.3 Testing and Verifying Concurrent Software . . . 18

2.3.1 Parallel Debuggers . . . 18

2.3.2 Monitoring Systems . . . 19

2.3.3 Formal Verification of Parallel Programs . . . 21

2.3.4 Machine-Processable Specification Languages . . . 24

Anna—ANNotated Ada . . . 26

TSL—Task Sequencing Language . . . 27

2.4 Summary . . . 35

3 Distributed Supervisor Design Principles 37

3.1 The Virtual Machine Interface 40

3.1.1 Interface Requirements 41

3.2 The Supervisor Kernel 42

3.3 Supervisor Subprograms 43

3.4 Summary of Design Principles 44

4 Distributed Ada Supervisor Design 45

4.1 Overview of the Distributed Ada Supervisor Design 45

4.2 Ada Supervisor Structure 47

4.3 The Virtual Machine Interface 50

4.4 The Supervisor Kernel 52

4.4.1 The Local Task Map 53

4.4.2 The Global Task Map 55

4.4.3 The Message-Sending Algorithm 57

4.4.4 The Message-Receiving/Routing Algorithm 58

4.4.5 Analysis of the Kernel Algorithms 59

4.4.6 Elimination of Kernel Race Conditions 63

4.5 The Supervisor Message-Passing Protocol 63

4.5.1 Minor Enhancements to the Weatherly Protocol 65

4.5.2 Messages for Scheduling Newly Activated Tasks 65

4.5.3 Messages for Remote Exception Propagation 66

4.5.4 Messages for Evaluation of Task Attributes 67

4.5.5 Messages for Remote Activation of Dependents 68

4.5.6 Messages for the Terminate Alternative 70

4.5.7 Messages for the Abort Statement 75

4.6 The Supervisor Interface 77

4.6.1 The Supervisor Data Types 79

4.6.2 The Vocabulary of the Anna Specifications 81

4.6.3 The Scope Control Subprograms 82

4.6.4 Synchronization Points 83

4.6.5 The CALLABLE_ATTR Function 83

4.6.6 The TERMINATED_ATTR Function 84

4.6.7 The COUNT_ATTR Function 84
4.6.8 The ENTRY_CALL Procedure 84
4.6.9 The ACCEPT_BEGIN Procedure 85
4.6.10 The ACCEPT_END Procedure 89
4.6.11 The CHILD_TASK Function 89
4.6.12 The ACTIVATE_TASK Procedure 91
4.6.13 The ELABORATE_TASK Function 92
4.6.14 The TERMINATE_TASK Procedure 93
4.6.15 The ABORT_TASKS Procedure 93
4.6.16 The DELAY_TASK Procedure 94
4.7 Design Summary 95

5 Distributed Ada Supervisor Implementation 97
5.1 The Uniprocessor Supervisor Simulator 98
5.2 The Distributed Supervisor Implementation 102
5.2.1 Task Scheduling with DOMINO 103
5.2.2 Synchronization with Sequent Locks 106
5.2.3 Parallel Execution with Unix FORK 107
5.3 Communication Overhead 108
5.4 Diagnostic Supervisor Output 109

6 Distributed Supervisor Verification 111
6.1 The Verification Approach 111
6.2 Formalization of the Ada Tasking Semantics 113
6.2.1 Choosing the Tasking Events and Predicates of the Rendezvous . . . 114
6.2.2 Formalizing the Semantics of the Accept Statement 118
TSL Definition of the Tasking Events and Predicates 119
Construction of the TSL Specifications 122
6.3 The NET_SERVICES Connectedness Assumptions 124
6.4 Runtime Checking Experiments 127
6.5 Comparison With Other Approaches 130

7 Conclusions 133
7.1 Contributions of the Dissertation 133

7.2 Future Work 135
7.2.1 Remaining Ada Supervisor Design Details 135
7.2.2 New Models of Concurrency 137

A The Virtual Machine Interface 139

B Ada Supervisor Kernel Algorithms 145
B.1 The SEND Algorithm 145
B.2 The Routing and Receiving Algorithm 146

C Ada Supervisor Messages 149
C.1 The Supervisor Message Type 149
C.2 Task Execution Status Types 152

D Terminate Alternative Algorithm 153

E The Supervisor Preprocessor 157
9.1 Task Specifications and Task Bodies 158
9.2 Task Types and Task Objects 162
9.3 Task Execution—Task Activation 165
9.4 Task Dependence—Termination of Tasks 165
9.5 Entries, Entry Calls, and Accept Statements 169
9.6 Delay Statements, Duration and Time 169
9.7 Select Statements 170
9.7.1 Selective Waits 170
9.7.2 Conditional Entry Calls 173
9.7.3 Timed Entry Calls 173
9.8 Priorities 175
9.9 Task and Entry Attributes 175
9.10 Abort Statements 176
9.11 Shared Variables 176

F Ada Tasking Events 177
F.1 Action Declarations for the Tasking Events 177
F.2 TSL Specification of the Tasking Events 181

9.1 Task Specifications and Task Bodies 181
9.2 Task Types and Task Objects 182
9.3 Task Execution—Task Activation 182
9.4 Task Dependence—Termination of Tasks 183
9.5 Entries, Entry Calls, and Accept Statements 183
9.6 Delay Statements, Duration and Time 184
9.7 Select Statements 184
9.7.1 Selective Waits 184
9.7.2 Conditional Entry Calls 185
9.7.3 Timed Entry Calls 185
9.8 Priorities 186
9.9 Task and Entry Attributes 186
9.10 Abort Statements 186
9.11 Shared Variables 187

G TSL Formalization of Ada Tasking **189**
G.1 Task Activation 189
G.1.1 Property Declarations 189
G.1.2 Functional Specification 191
G.1.3 Safety Specifications 192
G.2 Task Termination 193
G.2.1 Property Declarations 194
G.2.2 Functional Specification 194
G.2.3 Safety Specifications 195
G.3 Task Execution 196
G.3.1 Property Declarations 196
G.3.2 Functional Specification 196
G.3.3 Safety Specifications 197
G.4 The Entry Call Statement 198
G.4.1 Property Declarations 198
G.4.2 Functional Specification 199
G.4.3 Safety Specifications 201
G.5 The Delay Statement 202

G.5.1 Property Declarations 202
G.5.2 Functional Specification 203
G.5.3 Safety Specifications 203
G.6 Other Specifications 203

Bibliography **205**

List of Tables

1 Semantics of TSL Specifications. 30
2 Locks, Semaphores and Condition Variables. 107

List of Figures

1 Abstract View of Ada Program Execution. 2
2 An Ada Implementation of Dining Philosophers. 13
3 The TSL View of Concurrent Program Execution. 28
4 Properties for TSL Specification of Dining Philosophers. 34
5 TSL Specification of Dining Philosophers. 35
6 A Supervisor Supporting Execution of Concurrent Programs. 38
7 Partitioning the Tasking Supervisor Design. 39
8 Structure of the Distributed Tasking Supervisor Design. 40
9 Computational Environment of the Distributed Ada Supervisor. 46
10 Structure of the Distributed Ada Supervisor Design. 48
11 Correspondence with the Generic Supervisor Design. 49
12 Interaction of the Supervisor Packages During an Entry Call. 50
13 Outline of the Virtual Machine Interface. 51
14 The Supervisor Message Type. 53
15 Contents of a Local Task Map Entry. 54
16 Contents of a Global Task Map Entry. 55
17 Global Map Updates Resulting from Activation of a Dependent. 56
18 Task Dependency and Visibility Relationships. 60
19 Task Visibility Relationships in a Sample Ada Fragment. 61
20 Message Transaction for Scheduling a Newly Activated Dependent. 67
21 Message Transaction for Remote Propagation of TASKING_ERROR. 68
22 Message Transaction for Evaluation of Attribute CALLABLE. 69
23 Remote Activation of Dependent Tasks. 70
24 Message Transaction for Remote Activation of a Dependent. 71
25 Message Transaction for Execution of a **Terminate** Alternative (I). 73

26 Message Transaction for Execution of a **Terminate** Alternative (II). 74
27 Message Transaction for Abandonment of a **Terminate** Alternative. 75
28 Message Transaction for Execution of the **Abort** Statement. 78
29 Outline of the Supervisor Interface. 79
30 Supervisor Data Types. 80
31 The Supervisor Virtual Functions. 82
32 The Supervisor ENTER_NEW_SCOPE Procedure. 82
33 The Supervisor CALLABLE_ATTR Function. 83
34 The Supervisor TERMINATED_ATTR Function. 84
35 The Supervisor COUNT_ATTR Function. 85
36 The Supervisor ENTRY_CALL Procedure. 86
37 The Supervisor ACCEPT_BEGIN Procedure. 87
38 The Supervisor ACCEPT_END Procedure. 89
39 The Supervisor CHILD_TASK Function. 90
40 The Supervisor ACTIVATE_TASK Procedure. 92
41 The Supervisor ELABORATE_TASK Function. 92
42 The Supervisor TERMINATE_TASK Procedure. 93
43 The Supervisor ABORT_TASKS Procedure. 94
44 The Supervisor DELAY_TASK Procedure. 95
45 Sample Tasking Program Before Preprocessing. 98
46 Sample Program After Full Preprocessing. 99
47 Sample Program After Partial Preprocessing for the Simulator. 100
48 Implementation of Semaphores and Condition Variables as Tasks. 101
49 Observing Tasking Events Outside the Supervisor. 112
50 Action Declarations for the Tasking Events of the **Accept** Statement. . . . 119
51 The S_ACTIVATED and S_COMPLETED Properties. 120
52 The S_QUEUE_SIZE Property. 121
53 The S_ACCEPTING and S_IN_RENDEZVOUS Properties. 121
54 Generic TSL Functional Specification. 122
55 Functional Specification of the **Accept** Statement Semantics. 123
56 Safety Specifications of the **Accept** Statement Semantics. 124
57 Transformation of Test Programs for the Distributed Ada Supervisor. . . . 128
58 Transformation of Test Programs for Any Tasking Supervisor. 129

Chapter 1

Introduction

Many software systems are designed to exploit the resources of a distributed computing system for added execution speed and/or fault-tolerance. A *concurrent software system* is a software system in which multiple *logical* threads of control are explicitly declared. A *distributed software system* is a concurrent software system that is executed on multiple CPUs with two or more logical threads of control executing simultaneously.

A *concurrent programming language* is a high-level programming language that contains features for expressing concurrency in a software system; these features collectively comprise a subset of the language called its *tasking language*. A *runtime tasking supervisor*, or simply a *tasking supervisor*, is a collection of subprograms, each of which provides execution support for a separate feature of a tasking language [Fal82,BR85]. This thesis addresses two fundamental issues in the construction of *distributed* implementations of a concurrent language:

1. Development of principles for designing a tasking supervisor for the language.

2. Development of practical techniques for verifying that the supervisor correctly implements the semantics of the language.

The fundamental principles, techniques and conclusions set forth in this thesis are based on the design, implementation and verification of a distributed tasking supervisor for the Ada[1] programming language [Ada83].

Figure 1 depicts an abstract view of the execution environment of an Ada application

[1]Ada is a registered trademark of the US Government (Ada Joint Program Office).

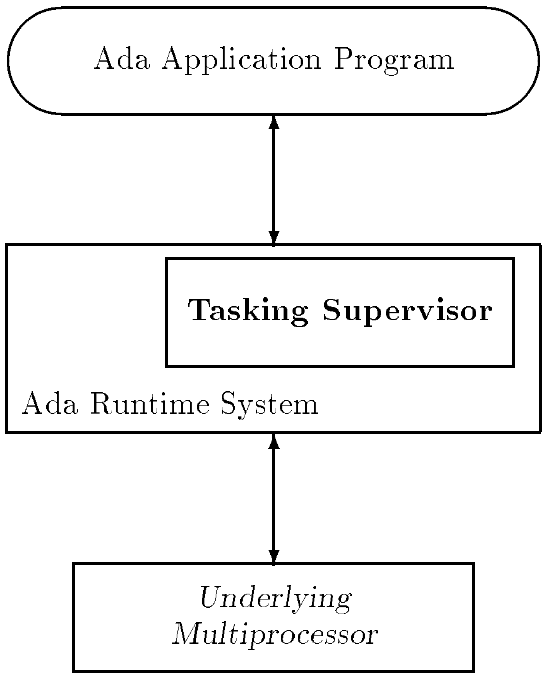

Figure 1: Abstract View of Ada Program Execution.

program. As shown in the figure, the program uses the facilities of an *Ada runtime system* (RTS) during its execution. The tasking supervisor is one component of the RTS; other components provide runtime support for exception handling, memory management, and attribute evaluation. The Ada compiler is responsible for translating all source-level tasking statements in the application program into calls to appropriate subprograms of the tasking supervisor.

The Distributed Ada Supervisor implements the full Ada tasking language, and it performs distributed program execution on multiple CPUs. The Supervisor is a modular, portable, distributed software system written in Ada. In addition, two high-level formal specification languages and their associated checking tools are used for formally describing and verifying the behavior of the Supervisor. *Anna* (ANNotated Ada) [LvHKO87,LvH85] is used to specify to the compiler writer constraints on the parameters of each subprogram that is declared in the top-level Supervisor interface; *TSL* (Task Sequencing Language) [HL85b] is used to specify and verify the overall distributed behavior patterns of the Supervisor.

Chapter 2 describes previous research efforts relevant to the work described in this thesis. This thesis relies on results from a variety of disciplines, including high-level concurrent programming languages, distributed operating systems, program verification theory and high-level formal specification languages.

Chapter 3 summarizes the principles that are used to design distributed tasking supervisors; these principles were used to design the Distributed Ada Supervisor. A distributed supervisor is constructed in layers, much in the same way that computer networks are layered [Zim80,Tan81]. The chapter summarizes the functionality of each design layer.

Chapter 4 describes in detail the design of the Distributed Ada Supervisor. The chapter also presents the formal specification of the Supervisor interface in Anna. Some of the specifications are *promises* about the Supervisor that may be assumed for the purposes of compiling source-level tasking constructs into supervisor subprogram calls; the remaining specifications are *constraints* that must be satisfied by the compiled source programs when calling the Supervisor.

Chapter 5 describes in detail the implementation of the Distributed Ada Supervisor on both uniprocessor and multiprocessor hardware. The uniprocessor simulator developed on a Data General *MV10000/Eclipse*[2] provided a framework for testing Supervisor algorithms without the need for debugging an actual distributed software system. The implementation of the Supervisor on the Sequent *Balance 21000*[3] provided the experience of creating a working distributed Ada tasking supervisor. The successful implementation of the Supervisor on the Sequent demonstrates the feasibility of the design principles presented in Chapter 3.

Chapter 6 describes new and practical techniques for verifying a tasking supervisor for consistency with the semantics of the language it implements. The techniques comprise a methodology based on *automatic runtime checking* of supervisor behavior for consistency with a formalization of the tasking semantics written in TSL. The methodology is illustrated by the formalization of a subset of the Ada tasking semantics. The chapter describes the mechanics of performing checking experiments on the Distributed Ada Supervisor using TSL. The major advantages of these consistency checking techniques over other testing methods are that

1. The TSL formalization of the Ada tasking semantics is a formal standard against which the behavior of *every* supervisor implementation may be automatically compared.

2. When a specification is violated during runtime checking, the violated specification

[2] *Eclipse* is a registered trademark of Data General Corporation.
[3] *Balance* is a registered trademark of Sequent Computer Systems, Inc.

provides an explicit characterization of an error in the supervisor.

3. Supervisor debugging is carried out using high-level concepts from the tasking language such as task dependency and rendezvous, not low-level concepts such as stack frames, network messages and program counters.

4. Spare processors, if available for runtime specification checking, can be used to make a supervisor permanently self-checking.

5. The specifications can also be used to construct a formal proof of supervisor consistency, once the appropriate proof rules and proving tools have been developed.

Chapter 7 concludes the thesis by first summarizing the contributions of the thesis, and then by suggesting other research endeavors which may aid future work in the design and verification of distributed tasking supervisors.

The theory and practice of concurrent programming is in its infancy. The research described in this thesis represents a major step toward the development of a theory of constructing multiprocessor implementations of concurrent programming languages. It is hoped that the tools for constructing and testing distributed tasking supervisors will eventually be based on a technology as sophisticated as that which is available for constructing compilers.

Chapter 2

Background and Related Work

This chapter surveys the literature describing the technology of concurrent programming. The previous work in this broad subject area can be subdivided into three categories. First, the need for special language facilities has led to the development of language notations for expressing concurrency and the design of concurrent programming languages. Second, operating systems have been designed explicitly to create a suitable environment for execution of distributed software systems. Third, the extreme difficulty of testing and debugging concurrent software has led to a large body of work, running the gamut from parallel debuggers to fully automated verification systems, most of which is too immature to be successfully applied to real software.

2.1 Concurrent Programming Languages

The most prevalent paradigm for concurrent programming languages is the *procedural* or *imperative* model, typified by a family of sequential programming languages that were originally based on ALGOL [Nau63]. However, the computational requirements of many artificial intelligence applications such as rule-based expert systems has led to the development of concurrent *functional* programming languages, such as QLISP [GM84]. This section focuses on the features of those "ALGOL-like" concurrent programming languages.

The designers of the concurrent language Linda name four desirable criteria of a concurrent programming language [ACG86]:

1. A machine-independent and (potentially) portable programming vehicle.

2. A programming tool that absolves [the programmer] as fully as possible from dealing with spatial and temporal relationships among parallel processes.
3. A programming tool that allows dynamic distribution of tasks at runtime.
4. A programming tool that can be implemented efficiently on existing hardware. [ACG86, p. 27]

As the designers of Linda admit, these needs have not been completely accommodated by existing languages. The NIL [PS83,SY83] language seems to be one of the better attempts at attaining this ideal.

Brinch Hansen has shown that most concurrent programming constructs are special cases of a notation he calls *distributed processes* [BH78]. This section will describe some of the better known special cases. In their survey of notations for concurrent programming, Andrews and Schneider [AS83] subdivide the facilities of concurrent programming languages into two basic categories: (1) Constructs for declaring multiple threads of control in a program and (2) Constructs for synchronizing the execution of the threads. The second class of facilities can be subdivided further into synchronization primitives based on shared variable manipulation and synchronization primitives based on message-passing. As Wegner and Smolka note [WS83], the first class of construct is typified by the CSP *process* and the Ada *task* while the second class is typified by Hoare's *monitor* construct (shared variable manipulation), CSP input/output commands (message-passing) and the Ada rendezvous (message-passing).

Before looking at the facilities of specific concurrent languages, it should also be mentioned that there is a school of thought which says that no special facilities are required for expressing parallelism in software, and that what is needed instead are software tools which detect and extract parallelism from sequential programs. Two approaches in this area have shown some amount of success. First, there are "parallelizing" FORTRAN compilers such as the Parafrase system [PKL80], which break apart loops in FORTRAN programs for parallel execution on vector machines. Such systems seem best suited to parallelization of "scientific" programs having a simple structure. Second, there are dataflow languages [Ack79] for graphically depicting data flow and data dependencies within a computation; the program graph is the "machine code" for the program, which is executed on special dataflow machines that detect and exploit the parallelism inherent in the program graph. However, the elegance of the ideal dataflow model has yet to be satisfactorily realized in a working machine.

2.1.1 Concurrent Pascal

One of the earliest concurrent languages to gain widespread attention is Concurrent Pascal [BH75], which was designed by Per Brinch Hansen as an extension of Pascal [Wir71] for structured programming of operating systems. The extensions to Pascal are in the form of three language constructs, the *process*, the *monitor* and the *class*. Each of these constructs is declared as a type, which must be explicitly initialized.

A separately executing thread of control is represented by a *process*, which is a sequence of Pascal statements that has its own private data. The *monitor* construct, first described by Hoare [Hoa74], is a means of providing synchronized access to a shared resource; shared objects in Concurrent Pascal must be monitors. The *class* construct of Simula 67 [DH72] is simply an unprotected monitor, which may not be shared, but which may be used in place of a monitor to increase implementation efficiency. The use of monitors and classes by processes is controlled in Concurrent Pascal through explicit, statically checkable access rights. Access rights take the form of *bindings* of existing monitors and classes to parameters of newly initialized processes.

The first operating system that was implemented in Concurrent Pascal was the Solo operating system [BH77]. The Solo kernel performs the low-level synchronization that is needed for the implementation of processes and monitors, and thus serves as the "tasking supervisor" for Concurrent Pascal. In truth Concurrent Pascal has no real tasking supervisor, in the sense of a separate software module which is called at runtime to execute a high-level concurrency construct. Instead, the Concurrent Pascal compiler generates a complete in-line code translation of each concurrency construct that is encountered in a source program; only the machine-level synchronization primitives are called by the generated code. This rather cumbersome approach to implementing concurrency has the disadvantages of hiding the functionality of the supervisor inside the compiler code generator and generating object files that contain large amounts of redundant code.

The major contributions of Concurrent Pascal were its incorporation of monitors as a language feature and its emphasis on compile-time checking of program semantics, especially the semantics of process creation and access to shared objects. Many of the ideas used in the development of Concurrent Pascal were later used in the design of Modula-2 [Wir82].

2.1.2 CSP

Hoare's landmark paper on *communicating sequential processes* [Hoa78] introduced a notation for describing concurrent computations that has become a standard language for formal specification and verification of concurrent software; this notation has come to be known as CSP.

The language is based on the premises that input and output are fundamental concurrent programming primitives and that parallel composition of sequential processes is a fundamental paradigm for structuring concurrent programs. Thus, a CSP *process* is simply a sequential body of code, and CSP processes communicate with each other in pairs with one process executing an *input command* and the other process executing an *output command*. An output command passes either a value or a signal to a matching input command; the input command stores the value in a variable or accepts the signal. In addition to these basic communication mechanisms, Hoare used Dijkstra's *guarded command* statement [Dij75] to incorporate nondeterminism into the language. The result is a language of great flexibility and expressive power which was eventually expanded into a full programming language for the INMOS *Transputer* called *Occam* [1] [Jon85]. In fact, Occam was intended by its designers to be the "assembly language" of the Transputer, thereby removing the need for a tasking supervisor for Occam.

To illustrate the features of CSP, a version of Hoare's specification of the Dining Philosopher's problem [Hoa78] will be presented. Recall that in the Dining Philosophers problem five philosophers are seated at a circular table, with a fork between each philosopher. Each philosopher repeatedly thinks and eats; in order to eat, a philosopher must first pick up both the fork to his left and the fork to his right. This problem will be specified by a CSP program in which each philosopher and each fork is represented by a process. The following CSP statement serves to declare the ten processes, which execute in parallel with one another:

```
[ a_phil (i : 0 . . 4) :: PHIL || a_fork (i : 0 . . 4) :: FORK ]
```

The philosophers and forks are differentiated by an index value between 0 and 4 and have a body that is specified respectively by the programs PHIL and FORK. Inside these program bodies, the variable i may be used to identify the particular process that is executing the body.

[1]Occam and Transputer are a registered trademark of the INMOS Group of Companies.

The body of the philosophers is simply a loop in which thinking alternates with eating; the required communications with the forks occur between these two activities. Philosopher number i will use forks number i and $(i + 1)\, mod\, 5$ for eating. The acquisition and release of a fork will be accomplished respectively by sending a *pickup()* or *putdown()* signal to the fork. Prior to eating, both forks must be picked up; after eating, the same two forks must be put down. Thus, the specification of PHILOSOPHER can be written as follows:

```
PHIL = *[ Think;
          a_fork(i) ! pickup();
          a_fork((i+1) mod 5) ! pickup();
          Eat;
          a_fork(i) ! putdown();
          a_fork((i+1) mod 5) ! putdown()
        ]
```

Repetition is indicated by the asterisk outside the bracketed sequence of statements, which is thus called a *repetitive command*. The statements with exclamation points are output commands. The left side of the exclamation point names the fork process that is the intended recipient of the *pickup()* or *putdown()* signal. The output command will be executed only when a matching input command is executed in the named fork process (and *vice versa*).

The basic outline of the fork is again a repetitive command, but this time one that nondeterministically chooses between two possible sequences of input commands, namely a sequence for the philosopher to its left (number $(i - 1)\, mod\, 5$) and a sequence for the philosopher to its right (number i). The fork must "decide" which philosopher to give the fork to when both eligible philosophers are competing for the fork. In order to guarantee that a hungry philosopher has immediate access to a fork when his neighbor is thinking, a Boolean-valued *guard* will be used to control how the choice is made. The evaluation of all of the guards in a so-called *guarded command* is performed prior to choosing the sequence that will be executed; a sequence is chosen only if its guard is true. Since there is a binary possibility of executing an input command at any point in time (i.e., either the named process is ready or not ready to execute a matching output command), the first input command in each sequence will serve as the guard for the sequence. Thus, the specification of FORK may be written as follows:

```
FORK = *[ a_phil((i−1) mod 5) ? pickup() →
```

```
    a_phil((i−1) mod 5) ? putdown()
[]a_phil(i) ? pickup() →
    a_phil(i) ? putdown()
]
```

The box specifies that only one of the two sequences of statements is to be executed, and that the choice of sequence is nondeterministic.

This simple example demonstrates the ability of CSP to describe nontrivial concurrent computation in a few lines of code. As will be seen throughout the rest of this chapter, CSP is one of the most significant developments in the theory and practice of concurrent programming.

2.1.3 Ada

The need of the US Department of Defense for a common, high-level programming language for "mission-critical" embedded software systems [Fis78,Gro78] led ultimately to the controversial development and standardization of the programming language Ada [Ada83]. Ada is probably the most rigorously defined programming language in use. Not only is the language a US Military Standard, an ANSI Standard and an ISO Standard, but it also has a widely used common intermediate representation for compilers and programming environments called DIANA [EBGW83].

The Ada language is a very rich and, some would say, complex collection of features that have an extremely diverse lineage, incorporating ideas from Algol 68, CLU, Pascal, Simula 67 and other languages. Woodger [Woo87] has surveyed the sources of the features of Ada that were incorporated into the language throughout its ten-year development history.

Features of the Ada Tasking Language

One of the most controversial features of Ada is its tasking language. The tasking model of Ada is based on CSP. An Ada task is a sequential program in the same way that a CSP process is. It may have multiple, nested declarative regions, or *scopes*, including the scopes of subprograms it calls.

A *task declaration* declares either a *single task* or a *task type*; an associated *task body* must be declared for each task declaration. A *task object* (or simply a *task*) is either (1) A single task, (2) An object, or a subcomponent of an object, that is declared by an Ada *object declaration*, or (3) A task designated by a value of an access type (i.e., pointer type) whose designated subtype is a task type. The type of a single task is anonymous; tasks of the other two kinds are objects of explicitly named task types.

The execution of a task object begins with its *activation*. Tasks of the first two kinds are activated when program execution reaches the **begin** of the declarative region in which they are declared. Execution of an *allocator* for an appropriate access type causes activation of a task of the third kind; the allocator returns an access value (pointer) designating the newly activated task. Activation through execution of an allocator is sometimes referred to in the literature as “dynamic activation”.

Every task has a *direct master*, of which it is a *direct dependent*. An *indirect master* is some master of a direct master. A direct master is a particular scope of some task; thus, the task which owns this master scope is a master of the dependent. In the case of single tasks and tasks declared by object declarations, the direct master is the scope in which the declaration appears. The direct master of a task designated by an access value is the scope in which the corresponding access type definition appears. All tasks declared in library packages are dependents of the *environment task*, which can be thought of as the operating system process in which a program is executed; the main program is a procedure that is called by the environment task once all library packages have been elaborated. Thus, the tasks of a program form a *dependency tree* whose root is the environment task.

A scope is *completed* once the end of its sequence of statements is reached; a scope is *terminated* once it is completed and once all of its dependents have terminated. A task is completed once its outermost scope is completed; a task is terminated once its outermost scope is terminated.

The preferred form of communication between tasks is the *rendezvous*, a synchronous mechanism for communication between pairs of tasks similar to communication in CSP. For a rendezvous to take place, a *caller* task must call an *entry* (communication port) of a specific task, and the task called by the caller must execute an **accept** statement for the called entry. Once both of these statements have been reached (assuming that the caller is the first task in the *queue* of the called entry), then the two tasks are *in rendezvous*. During a rendezvous, the caller is suspended while the called task executes a sequence of statements associated with the **accept** statement.

Added to the basic rendezvous statements is the **select** statement, which resembles Dijkstra’s guarded command. The **select** statement is a general facility for introducing nondeterminism into the execution of multiple accept statements, for placing time constraints on rendezvous statements, and for achieving simultaneous termination of a family of tasks. A dependency tree of tasks may be simultaneously terminated if all tasks in the tree are either terminated or waiting at a **terminate** alternative of a *selective wait* (a **select** statement with one or more **accept** alternatives). The root task of such a tree must itself be completed at the time termination is carried out.

Of the remaining features of the Ada tasking language, the **delay** statement allows a task to delay itself for a specified duration; the semantics of the various uses of Ada's **delay** statement are particularly weak and are considered by some to be of limited utility [VM87]. The **abort** statement is used by a task to abort the execution of one or more other tasks (possibly including itself); an aborted task is said to be *abnormal*, and it remains so until it reaches a *synchronization point* (e.g., an entry call), at which time it becomes completed.

As an example of an Ada tasking program, an Ada version of the CSP Dining Philosophers program of Section 2.1.2 is shown in Figure 2. Each philosopher and fork is represented by a task; the tasks are activated when the main program reaches its **begin** statement. Note that because an Ada **accept** statement does not name the task that is to be accepted for rendezvous, the FORK task may be programmed with a single pair of **accept** statements. The remainder of the operation of the program is analogous to the CSP version.

Implementations of Ada

Ada was adopted as an ANSI standard in February 1983. As of August 17, 1987, there were 101 Ada compilers that were validated as conforming to the Ada83 standard; all of these compilers are *uniprocessor implementations*. And with one exception, the various experimental implementations of Ada that have been reported over the past seven years are also all *uniprocessor implementations*. The only publicly reported distributed implementation of Ada is an unvalidated, proprietary, shared memory implementation developed by Sequent Computer Systems [Oli87] [2]. The tasking supervisor for this implementation was customized solely for parallel execution of Ada tasking programs on the Sequent *Balance 21000* multiprocessor.

As Volz et al. have pointed out [VMNM85], the development of a distributed implementation of Ada requires the consideration and solution of several problems that are not

[2] In addition, in Autumn 1987 Alliant Computer Systems announced the FX/Ada runtime system, which supports parallel Ada tasking on the Alliant FX/8 machine.

```
with THINK, EAT;    -- Make these visible.
procedure DINING_PHILOSOPHERS is
    task type PHILOSOPHER is
        entry GET_ID (ID : in NATURAL);
    end PHILOSOPHER;

    task type FORK is
        entry GET_ID (ID : in NATURAL);
        entry PICKUP;
        entry PUTDOWN;
    end FORK;

    MAX_PHILOSOPHERS : constant POSITIVE := 5;
    subtype TABLE_RANGE is NATURAL range 0 .. MAX_PHILOSOPHERS - 1;
    A_PHILOSOPHER : array (TABLE_RANGE) of PHILOSOPHER;
    A_FORK        : array (TABLE_RANGE) of FORK;

    task body PHILOSOPHER is
        I : NATURAL;
    begin
        accept GET_ID (ID : in NATURAL) do  I := ID;  end GET_ID;
        loop
            THINK; A_FORK (I).PICKUP; A_FORK ((I + 1) mod 5).PICKUP;
            EAT; A_FORK (I).PUTDOWN; A_FORK ((I + 1) mod 5).PUTDOWN;
        end loop;
    end PHILOSOPHER;

    task body FORK is
        I : NATURAL;
    begin
        accept GET_ID (ID : in NATURAL) do  I := ID;  end GET_ID;
        loop
            accept PICKUP; accept PUTDOWN;
        end loop;
    end FORK;
begin
    for I in TABLE_RANGE loop      -- Pass IDs to the tasks:
        A_PHILOSOPHER (I).GET_ID (I);  A_FORK (I).GET_ID (I);
    end loop;
end DINING_PHILOSOPHERS;
```

Figure 2: An Ada Implementation of Dining Philosophers.

addressed in the development of single-CPU implementations, such as how the semantics of Ada's real-time constructs are to be implemented in an execution environment that has no global sense of time. Thus, it would be nearly impossible to use currently available uniprocessor implementations for distributed execution of Ada tasking programs because the issues involved in achieving distributed execution have never been addressed by their developers.

One of the earliest uniprocessor implementations of Ada was actually a superset of the 1980 draft standard version of Ada, called *Adam* [LLSvH83]. Adam contains extensions to Ada that were designed for multiprocessing research (which explains the origin of the early name for the language, *Ada-M*). These extensions comprise an intermediate tasking language that is at a lower level of abstraction than the Ada tasking constructs. The implementation of Ada tasking was originally specified by translating each Ada tasking construct to an equivalent sequence of Adam statements [Ste80].

The tasking supervisor for Adam is built as a package of subprograms, each of which implements a different tasking construct [Fal82]. Thus, the Adam compiler translates each tasking construct that appears in a source program into an appropriate sequence of supervisor subprogram calls. This organization enhances the portability of the Adam runtime system and simplifies the translations required of the Adam compiler. This method of compilation seems to be the universal paradigm for the implementation of Ada tasking.

Since the development of Adam, there have been numerous other research efforts in the design and analysis of Ada runtime systems and tasking supervisors. Leathrum [Lea84] gives an overview of an Ada runtime system, focusing on mechanisms for task activation, termination and abortion, and task stack allocation. Baker and Riccardi [RB84,RB85,BR85] discuss highlights of their tasking supervisor design, which supports task priorities. In addition, Baker has proposed a standard interface for an Ada runtime system [Bak86]. The authors of all of the above papers explicitly state that their designs are suitable only for single-CPU, single shared address space implementations of Ada tasking.

Towards Distributed Ada Tasking

Except for the Sequent implementation described above, all other research on distributed Ada tasking has either been preliminary analysis (such as the paper by Volz et al.) or paper design. Furthermore, none of these efforts has satisfactorily addressed the problem of testing a distributed implementation of Ada for consistency with the language semantics.

The most notable previous work on distributed Ada tasking is the supervisor interface and message-passing protocol described by Weatherly [Wea84a,Wea84b]. Weatherly's protocol supports distributed execution of a subset of the Ada tasking constructs in a loosely coupled computing environment, and his interface provides most of the services required for distributed execution of compiled Ada tasking programs. In Weatherly's design, an identical copy of the tasking supervisor executes at each system processor. Weatherly tested his design by simulation, but he did not implement it on on parallel hardware. Another of his contributions was the reduction of Ada tasking to its "essential syntax", noting for example that a selective wait with a **delay** alternative is a general construct from which the simple **accept** statement and all forms of selective wait are derived. A paper by Fisher and Weatherly [FW86] describes a continuation of Weatherly's work.

An interesting approach to the development of a distributed Ada tasking supervisor was described by Clemmensen [Cle82]. He began with an abstract formal operational model of a distributed tasking supervisor for a subset of Ada tasking; this model was designed to be free of any implementation dependencies. He then proposed deriving an implementation from this model by stepwise refinement and functional decomposition, gradually incorporating necessary implementation dependencies. He claimed that the resulting implementation could be formally verified for consistency with the formal model by verifying each level of refinement for consistency with the previous level. Although the model was apparently refined a few steps toward an implementation, no results were given to support his claims of verifiability, and no consideration is given to the verification of implementation dependencies as they are introduced.

2.2 Distributed Operating Systems

An *operating system* provides users and programs with simple, controlled access to and allocation of system resources [Tan87]. Part of the function of a tasking supervisor is to provide a concurrent program with access to system resources in a way that conforms to the semantics of the programming language. Thus, a tasking supervisor must either use the services of an available operating system or implement such services itself in order to support the execution of a concurrent program. A distributed tasking supervisor uses the services of a *distributed operating system*, including an interface to the resources of a *computer network* [Tan81,Sta84]. This section briefly discusses some of the common features of distributed operating systems that are used in the implementation of a distributed tasking supervisor.

2.2.1 Special Features

In their excellent survey of distributed operating systems, Tanenbaum and van Renesse describe the characteristics of distributed operating systems which differentiate them from conventional operating systems [TvR85]. They define a distributed operating system as follows:

> A *distributed* operating system is one that looks to its users like an ordinary centralized operating system but runs on multiple, independent central processing units (CPUs). ... In other words, the use of multiple processors should be invisible (transparent) to the user. [TvR85, p. 419]

They note that few operating systems have been built which fulfill this principle, and they use the term *network operating system* to refer to systems which provide a minimal set of facilities for accessing the resources of a distributed system. They draw the distinction between the two types of system by noting the following characteristics of a network operating system:

- Each computer [in the distributed system] has its own private operating system ...
- Each user normally works on his or her own machine ...
- Users are typically aware of where each of their files are kept and must move files between machines ...
- The system has little or no fault tolerance ... [TvR85, p.420]

A network operating system is simply a traditional operating system with a few extra facilities for accessing a computer network.

Tanenbaum and van Renesse describe several services provided by a distributed operating system, including communication primitives, naming service, file service, process service, scheduling, print service, terminal service, mail service, time service and network gateway service. The most important of these services to the designer of a distributed tasking supervisor are communication and process management services. In addition, naming service is also important (e.g., program tasks are typically named with globally unique integer identifiers), but the naming services provided by most distributed operating systems are incompatible with the needs of a distributed tasking supervisor.

Although a distributed operating system is intended to provide a specialized set of services to concurrent programs, many operating system-level services are ill-suited for use as primitives of a programming language library. Indeed, Cheriton concludes his description of the V Kernel with the following words:

> A fundamental issue is whether the V model of processes and messages, which was designed to support concurrency, is suitable for structuring parallel programs where the objective is to make the program run faster. [Che84, p. 42]

As an example of this problem, Ousterhout demonstrated that the usual global multiprogram scheduling mechanisms of operating systems are incompatible with the scheduling requirements of individual concurrent programs [Ous82].

Many of the distributed operating systems that have been described in the literature are experimental research projects; the more notable of these include LOCUS [PWC*81], CHORUS [ZGMB82], Mach [ABB*86], Charlotte [ACF87], and V [BBC*83]. To illustrate the way in which the facilities of a distributed operating system are used by a distributed tasking supervisor, some of the facilities of the V System are examined briefly below.

2.2.2 The V System

The V System is based to some extent on earlier experiences with the Thoth portable real-time operating system [CMMS79]. The V System has been designed with a keen eye on performance [CZ83], but with little fault-tolerance [TvR85]. The heart of the V System is the V Kernel [Che84], which provides process, communication and naming services, including multicasting services [CZ84,CZ85], remote execution and process migration services [TLC85], and a decentralized naming facility [CM86]. In addition, the V System has a highly flexible internetworking capability [Che83].

The finest grain of execution in the V System is a "lightweight" *process*, which shares the address space of a process *team*. All processes of a team all execute at the same system node. This hierarchy is ideal for a distributed tasking supervisor. Using the distribution paradigm in which an identical copy of the supervisor is assigned to each system node, concurrent program execution can proceed in V with each copy of the supervisor comprising a separate team, one per system node. Then each program task is scheduled as a V process inside the team of its node, sharing the address space with other co-resident program tasks.

Interprocess communication in V is based on the *transaction*, or Send-Receive-Reply, model of message-passing. Since the V IPC protocol provides for reliable delivery of single messages, it is a suitable basis on which to build the communication kernel of a tasking supervisor.

However, V lacks certain facilities that would be useful to the supervisor implementor. For example, programs written in languages such as Ada must execute within a single address space, since one task may access global variables that are declared in another task; thus, because V teams cannot share an address space, the partitioning of an Ada program onto multiple processors would be difficult using V. Furthermore, although V supports priorities at the team level, it does not support priorities at the process level; thus, since Ada allows the association of priorities with tasks, priority-based scheduling of tasks must be implemented on top of V.

2.3 Testing and Verifying Concurrent Software

As mentioned at the beginning of this chapter, the methods available for testing and verifying concurrent software covers an extremely wide spectrum; this section describes some of the specific tools and systems that are available. The current state-of-the-art is dominated by systems which require the programmer to do most of the work and supply most of the intuition in discovering and finding program bugs; although the theory of parallel program verification has reached a respectable level of maturity, a useful system for automatic verification of real concurrent programs is still a thing of the future.

2.3.1 Parallel Debuggers

The most obvious way to test concurrent software is to use something similar to a symbolic debugger for sequential programs. Such "parallel debuggers" still require the programmer to first discover bugs on his own, and then gradually pinpoint the cause of the bug in the original software by examining program variables and controlling the progress of execution. This method of debugging is usually performed in an *ad hoc* manner—the software is always assumed to be correct while it is being written, and the occurrence of bugs usually mystifies the programmer. Once the bug is eventually found, the programmer berates himself for missing something so obvious.

Despite this rather simplistic picture of debugging concurrent software, parallel debugging is inherently much more complex than sequential debugging, especially when the execution of the software is distributed. Like sequential software, correct execution of concurrent software depends to some extent on the correct computation of data values. However, unlike sequential software, correct execution of concurrent software depends heavily on the order in which computations are performed and the order in which synchronizations are carried out by the multiple threads of control in the program. It is the existence, complexity and erroneous implementation of such event orderings which make the chore of debugging concurrent software so difficult. In addition, the mere act of interactively debugging concurrent software can influence and alter the event orderings of the program.

The *Pdbx*[3] debugger, developed by Sequent Computer Systems, is representative of most parallel debuggers [Pdb86]. Pdbx is an extension of the sequential *Dbx* debugger for Berkeley Unix[4] 4.3BSD environments. Pdbx views a parallel program as a collection of Unix processes. Pdbx has facilities for running processes individually, suspending specific processes, terminating specific processes, halting the program at the occurrence of process forks and signals (interrupts), delivering signals to specific processes, and tracing the statements that are executed by specified processes. These features are supplied in addition to the usual features of sequential debuggers. However, like all sequential debuggers, Pdbx and other parallel debuggers are unable to discover erroneous program behavior. They simply provide the programmer with a primitive collection of facilities for trying to make these discoveries on his own.

2.3.2 Monitoring Systems

Monitoring systems are passive observers of program behavior which add to the power of parallel debuggers by performing a great deal of the information gathering that is part of the debugging process. However, most such systems are incapable of detecting erroneous execution behavior on their own.

The simplest monitoring systems simply maintain a history log of the significant events that occur during the execution of a concurrent program; the log can then be used to perform a post-mortem analysis or simulation of the execution that was logged. *BugNet* [CW82] is

[3]Pdbx is a registered trademark of Sequent Computer Systems, Inc.
[4]Unix is a registered trademark of AT&T Bell Laboratories.

an example of such systems. The user must specify which events are to be logged during execution; the occurrence of an event is stamped with the local process time of occurrence before being placed in the log. In addition to providing traditional debugger facilities such as interactive execution control, BugNet gives the programmer the capability of simulating a logged execution using the event log as a script. Similar to BugNet is the *Radar* system [LR85], which supports interactive graphical simulation of an event log. During simulation the programmer can force the debugger to ignore events which are irrelevant to the execution that is being examined.

The most notable characteristic of both BugNet and Radar is that only the event history chosen by the programmer is "executed" during debugging, not the actual program. The main problem with executing concurrent software over and over again for debugging is that the inherent nondeterminism of concurrent software may make it impossible to reproduce erroneous execution patterns once they have been observed.

Tai and Obid have overcome this problem to some extent with their notion of "reproducible testing" [TO86]. To perform reproducible testing, a concurrent program is instrumented to execute in two modes; it may either execute in "normal mode", during which a history of synchronization events is logged, or it may execute in "replay mode", during which the program completely reproduces an execution that is contained in a history log. Thus, unlike debugging in the BugNet and Radar systems, reproducible testing takes place by re-executing the code itself. Tai and Obid have applied their method to Ada tasking programs with some success, although they experienced difficulty deriving a way of instrumenting some of the more esoteric constructions of Ada tasking statements. However, the Tai and Obid system provides no interactive debugging capability during replay mode.

LeBlanc and Mellor-Crummey added interactive debugging to reproducible testing and came up with the *Instant Replay* system [LM87]. Using Instant Replay, a program is instrumented to execute in two modes as in the Tai and Obid system, but replay mode takes place interactively for debugging, so that the usual facilities of parallel debuggers can be applied.

Like parallel debuggers, the monitoring systems described so far are unable to automatically detect erroneous execution behavior on their own. As a step toward automatic detection of erroneous behavior, Helmbold and Luckham developed a system for automatic monitoring and detection of deadness errors in (instrumented) Ada tasking programs [GHL82]. The monitoring algorithms can detect both *circular deadlock* and *global blocking*. Although their monitor was able to detect only a limited class of tasking errors, the work of Helmbold and Luckham nevertheless represents the first major step toward automating the detection of erroneous behavior in concurrent software. Natarajan used some of the ideas of Helmbold and Luckham in the development of a *distributed* deadlock detection algorithm [Nat86].

2.3.3 Formal Verification of Parallel Programs

Formal program verification, or mathematical proof of program correctness, was looked upon for many years as the ultimate solution to the problem of developing software that meets its specifications. Today it is considered a powerful complement to program testing. The testing methodology to be described later in this thesis is based on a mixture of both verification and testing, using automatic runtime checking of formal specifications as a viable compromise between the two extremes. Since the programmer must have a thorough conception of the purpose and intended behavior of a software system, it is reasonable to require the development of a formal model of what is considered correct behavior so that techniques from verification theory may be applied in testing the software.

Program verification involves reasoning about the behavior of programs based on a formal semantics of the source programming language. The development of a formal syntax for ALGOL60 represents one of the most important developments in the formal description of programming languages [Nau63]. Since then various models have been suggested for formally describing the semantics of programming languages. For example, a *denotational semantics* gives a mathematical *interpretation* of each program construct according to its effect on the global program state [SS71,Ten76]. In a similar vein, the Vienna Definition Language (VDL) is used to define a language semantics by formally describing an abstract interpreter for the language [Weg72b]. An *attribute grammar* is a syntax-directed method of specifying language semantics; an attribute grammar associates semantic attributes with nodes in an abstract syntax tree (AST) representation of a program [ASU86]. An *operational semantics* defines a language in terms of an abstract implementation [Weg72a].

By far, however, the most popular and most easily understood semantic model for verification is an *axiomatic semantics*, as first described in Hoare's landmark paper [Hoa69] and as first illustrated successfully by the full axiomatic definition of Pascal [HW73]. An axiomatic semantics defines a programming language by a set of axioms and rules of inference (proof rules) that are satisfied by the execution of all programs written in the language; the axioms and proof rules are usually specified using first-order predicate calculus [MW85]. Such a description is naturally suited to the theorem-proving orientation of program verification as it has developed over the past two decades.

The origins of axiomatic program verification are usually credited to the papers of Floyd [Flo67] and Hoare [Hoa69], who described similar approaches to proving program correctness. Floyd's approach is based on dividing a correctness proof into smaller proofs of each linear path in the program, which is usually represented by a flowchart. Hoare further formalized this proof process by defining an axiom for each class of language statement, based on a *precondition* and *postcondition* for the statement. For example, the axiomatic semantics of an assignment statement is given by the following axiom:

$$\vdash \ \{P^x_e\}\ x := e;\ \{P\}$$

This axiom says that in order for the above assignment statement to terminate in a state satisfying some arbitrary postcondition P, it must begin execution in a state satisfying a precondition which is the predicate that is formed when all free occurrences of the variable x in P are replaced with the expression e.

Given a desired precondition and postcondition for a full program, a correctness proof is carried out using the full set of axioms and proof rules for the programming language. More accurately, a program is proven to be *consistent* with its *given* precondition and postcondition; from this consistency proof, one may decide that the program is *correct.* If correctness can be proven within the given axiomatic system, the resulting proof is said to be a proof of *partial correctness.* That is, the program is correct *if it terminates.* However, since knowledge about termination is not built into the proof system, one must perform reasoning outside the proof system to conclude that termination takes place. Floyd's paper included techniques for proving termination of possibly non-terminating program constructs (e.g., infinite loops) within a partial correctness framework. This is done by introducing counter variables for loops which must monotonically decrease to zero in order to conclude that termination will take place. A system that includes explicit axioms and rules for proving termination is said to support proof of *total correctness.* Apt has surveyed the large amount of research in verification that was generated by Hoare's initial paper [Apt81].

The theory of *automatic* program verification was pioneered by Luckham et al. [ILL75]. The approach they describe requires the programmer to provide *assertions* at certain places in the program text (e.g., an invariant assertion for each loop). A program enhanced in this way is fed to a *verification condition generator* (or VCG) along with the desired precondition and postcondition for the program. The VCG uses an axiomatic semantics for the programming language to construct a set of *verification conditions*, which are first-order formulae that must be satisfied in order to prove correctness of the program. The verification conditions are then fed to a theorem prover which attempts to prove the verification conditions using a first order reasoning system that may be enhanced with proof rules supplied by the programmer. The various concepts developed in their theory of automatic verification ultimately resulted in the development of the Stanford Pascal Verifier [LGvH*79].

The work that has been described so far is suitable for the verification of sequential programs. Owicki and Gries pioneered the theory of verification of parallel programs [OG76], using the basic Hoare-style axiomatic approach. A proof of partial correctness of a parallel program is begun by constructing *local proofs* of each program task. In order to demonstrate that the program is consistent with its postcondition, it is necessary to introduce *auxiliary variables* in order to demonstrate *non-interference* of the local proofs. That is, although each individual task may be proven consistent with the precondition and postcondition, the computation of one task may interfere with the computation of another task (e.g., in reading and writing shared variables). This interference is not taken into account in the local proofs, so the local proofs may be invalid without a demonstration of non-interference. The Owicki-Gries proof system includes axioms for reasoning about auxiliary variables and non-interference. In addition, they described techniques for proving total correctness properties based on a proof of partial correctness.

Since the development of the Owicki-Gries proof theory there have been many other approaches to axiomatic verification of parallel programs. The survey by Barringer is a good summary of the major contributions in this area [Bar85]. Apt, Francez and de Roever describe a proof system for CSP which is again based on local proofs of correctness [AFdR80]. *Coöperation* of the local proofs is demonstrated by proving the satisfaction of a *global invariant* for the program. Other notable approaches to parallel verification include the CSP proof system of Levin [Lev80], and the proof system of Flon and Suzuki [FS81], which includes axioms and inference rules for proofs of total correctness properties.

The work of Apt, Francez and de Roever has inspired the development of proof systems for Ada tasking. For example, Gerth describes an axiomatization of the Ada rendezvous [Ger82]. Gerth and de Roever [GdR84], and Barringer and Mearns [BM82], both define a proof system for the "CSP subset" of Ada tasking. All of these systems deal only with programs which activate tasks through CSP-style "splitting". Meldal has developed an axiomatic basis for "spawning" of tasks, such as through execution of allocators to activate tasks [Mel87].

A somewhat different approach to verification of parallel programs is based on *temporal logic* specifications of program behavior [MP81a]. Temporal logic is a first-order logic extended with modal operators for describing temporal, or time-based, properties; these operators are $\Box$ ("always"), $\diamond$ ("eventually"), $\bigcirc$ ("next") and $\mathcal{U}$ ("until"). For example, the temporal formula

$$\Box(PRODUCE \supset \diamond CONSUME)$$

says that it is always the case that whenever a PRODUCE operation is performed, a corresponding CONSUME operation will eventually be performed. Temporal logic is well suited to the specification of total correctness properties of programs. Hailpern developed heuristics for temporal-based specification and verification of parallel programs [Hai80], and other temporal proof principles have been described by Manna and Pnueli [MP81b]. Nguyen et al. described a temporal proof system for a computation model in which interprocess communication takes place solely through special communication ports [NGO85]. Pnueli and de Roever described a temporal-based approach to reasoning about a small subset of Ada tasking [PdR82], based on an operational semantics of the rendezvous.

Many ideas from program verification have been borrowed for the development of practical systems for *automated testing* of software, often based on *automatic runtime checking* of formal specifications of the intended behavior of the software. Such ideas have been applied to checking the correctness of language implementations. For example, Bird and Muñoz described a system for automatically generating large, random test programs for testing the correctness of a PL/I compiler [BM83]. In addition, the validation of Ada compilers is carried out using the Ada Compiler Validation Capability (ACVC), a huge suite of programs for testing the correctness of an Ada implementation [Ada87]. Klarund described an extended temporal logic called Temporal Rule Logic (TRL) [Kla85] for the specification of Ada tasking programs; TRL was developed from a perceived weakness in the power of the ACVC in checking the implementation of Ada tasking. The power and usefulness of the ACVC test suite will be discussed in detail in Chapter 6.

2.3.4 Machine-Processable Specification Languages

In applying formal methods to verification and testing of concurrent programs, formal specifications are expressed in a machine-processable *specification language*. The specifications are then automatically processed for performing verification or runtime checking. While purely mathematical formalisms such as first-order logic are suitable as a specification language, it is often desirable to design a specification language to meet other practical criteria in order to make the language attractive to non-mathematically inclined programmers. The most important criterion is that the syntax and semantics of a specification language should complement the features of a chosen implementation language.

Larch is one of the newest of the current breed of specification languages. The Larch system contains a variety of software tools and design techniques for the application of formal specifications to software development. Specifications in Larch are written in two parts. First, language-independent algebraic specifications of the behavior of a program module are written in the Larch Shared Language [GHW85]; then, language-dependent specifications of the interfaces between program modules are written in the Larch Interface Language for the chosen source language [Win87]. The features of the Larch Shared Language provide for the design and hierarchical composition of abstract module specifications and algebraic theories that are amenable to incremental verification.

One interesting application of Larch described by Birrell et al. is the formal specification of operating system interfaces [BGHL87]. Such specifications require taking into account the atomicity of the primitive operations being specified, as well as their concurrent interleaving on a multiprocessor. A primitive is viewed as the composition of its atomic actions, each of which is specified separately from the others. The authors' experience demonstrates both the advantages and the drawbacks of a purely algebraic approach to specifying concurrent software. While such specifications provide a fairly precise and understandable description of the interfaces to the user, it is difficult to strongly state various safety properties that may be guaranteed by an actual implementation. For example, the authors found it difficult to state that an atomic wakeup operation causes at most one task to resume execution, or that raising an exception during a primitive takes precedence over normal termination of the primitive under certain conditions.

GYPSY [AGB*77] and GEM [LO83] are two languages that were designed explicitly for specifying parallel programs. GYPSY is a high-level specification language for writing verifiable programs whose threads of control communicate through *mailboxes*, a construct similar to monitors. The GYPSY environment includes an interactive system for design, formal verification and runtime validation of parallel programs. The GEM language, an event-oriented specification language for describing concurrency problems, foreshadowed the design of TSL (described below). Each GEM computation is viewed as a partially ordered sequence of events, which is characterized mathematically based on properties of intertask synchronization.

Anna—ANNotated Ada

Anna was designed as the high-level formal specification language for an Ada software development environment [LvHKO87,LvH85,LNR87]. An Anna annotation is a first-order formula specifying a constraint that must hold over a scope; the scope of an annotation is determined both by its kind (e.g., a subprogram annotation) and by its position in the source text. Anna also supports declaration of *virtual Ada code* for defining concepts used in annotations. Anna constructs are introduced into program text as *formal comments* which are ignored by conventional Ada compilers, but which can be manipulated by special Anna software tools. An annotation is indicated by the −−| indicator at the beginning of each line, while virtual code is indicated by the −−: indicator.

In addition to the usual constructs of other specification languages, Anna has special facilities for specifying packages [LP80b,vHLKO85] and exceptions [LP80a]. However, Anna does not support the specification of the concurrency in Ada programs. The designers of Anna envisioned the language as serving a multitude of needs. It is suitable as a formal documentation language, it can be used for automatic runtime consistency checking, it is a suitable basis for a high-level program design language, and it can be used as the assertion language for an Ada verification system.

Research with Anna has focused mainly on its use for automatic runtime checking of Ada programs [SRN85]. A large suite of software tools has been developed for this purpose, the most important one being the Anna Transformer. The Anna Transformer transforms the annotations in an annotated program into executable Ada checking code [Kri83,SR86]. The resulting "self-checking program" then invokes the Anna Debugger each time a specification is violated. The Anna Transformer currently transforms a large subset of Anna, including subtype annotations, object annotations, subprogram annotations, (exception) propagation annotations, (function) result annotations and statement annotations.

Transformation of more advanced constructs, such as *state expressions* and *axiomatic* annotations for algebraic specification of packages, is currently under investigation; runtime checking of these constructs will require the use of a reasoning tool such as a Prolog engine [CM84]. Also under investigation is a methodology for *parallel runtime checking* of annotations [RSL86]. Runtime checking of certain annotations (e.g., annotations on access type collections) requires large computational power; parallel checking is viewed as a way of reducing the interference with the underlying program while such checking is carried out. Parallel checking would also be useful for creating *permanently* self-checking Ada software.

In the work described in this thesis, Anna is used to formally specify the top-level interface of the Distributed Ada Supervisor. The annotations that will be presented should be fairly easy to understand; therefore, the reader is directed to the excellent descriptions that are readily available in the literature, especially [LvH85].

TSL—Task Sequencing Language

The specification language *TSL* represents the current state of the technology of automated testing of concurrent software [HL85b,LHM*87]. TSL is a language for formally specifying constraints on the event sequencing behavior of concurrent programs and is the main specification language that is used in this thesis for the formal description of the Ada tasking semantics.

The development of TSL was inspired by preliminary work on event-based concepts for debugging Ada tasking programs, and by previous work on automated deadlock detection in Ada tasking programs [HL85a]. However, TSL is defined to a great extent independently of Ada, making it possible to reconfigure TSL for formal specification and testing of programs written in other concurrent languages. The notation and semantics of TSL are based to some extent on the interval logic of Schwartz et al. [SMV83] and the Event Description Language (EDL) of Bates and Wileden [BW82]. Baiardi et al. later described a similar system for automated testing of CSP programs for consistency with high-level formal behavior specifications [BdFV86].

Figure 3 depicts the TSL view of concurrent program execution; in the figure, the T_i represent program tasks, while the E_i represent tasking events. The *Merger* and *Monitor* shown in the figure together form the *TSL Runtime System*.

Each task generates a *local stream* of events characterizing its execution. The local event streams are then merged by the Merger into a single *global stream* of events before being fed to the Monitor. A computation may be characterized by many different global streams, depending on how the local streams are merged during a particular observation. A pair of events is said to be *connected* if the events are causally related; thus, connected events must occur in the same order in all global streams representing a computation. Every pair of

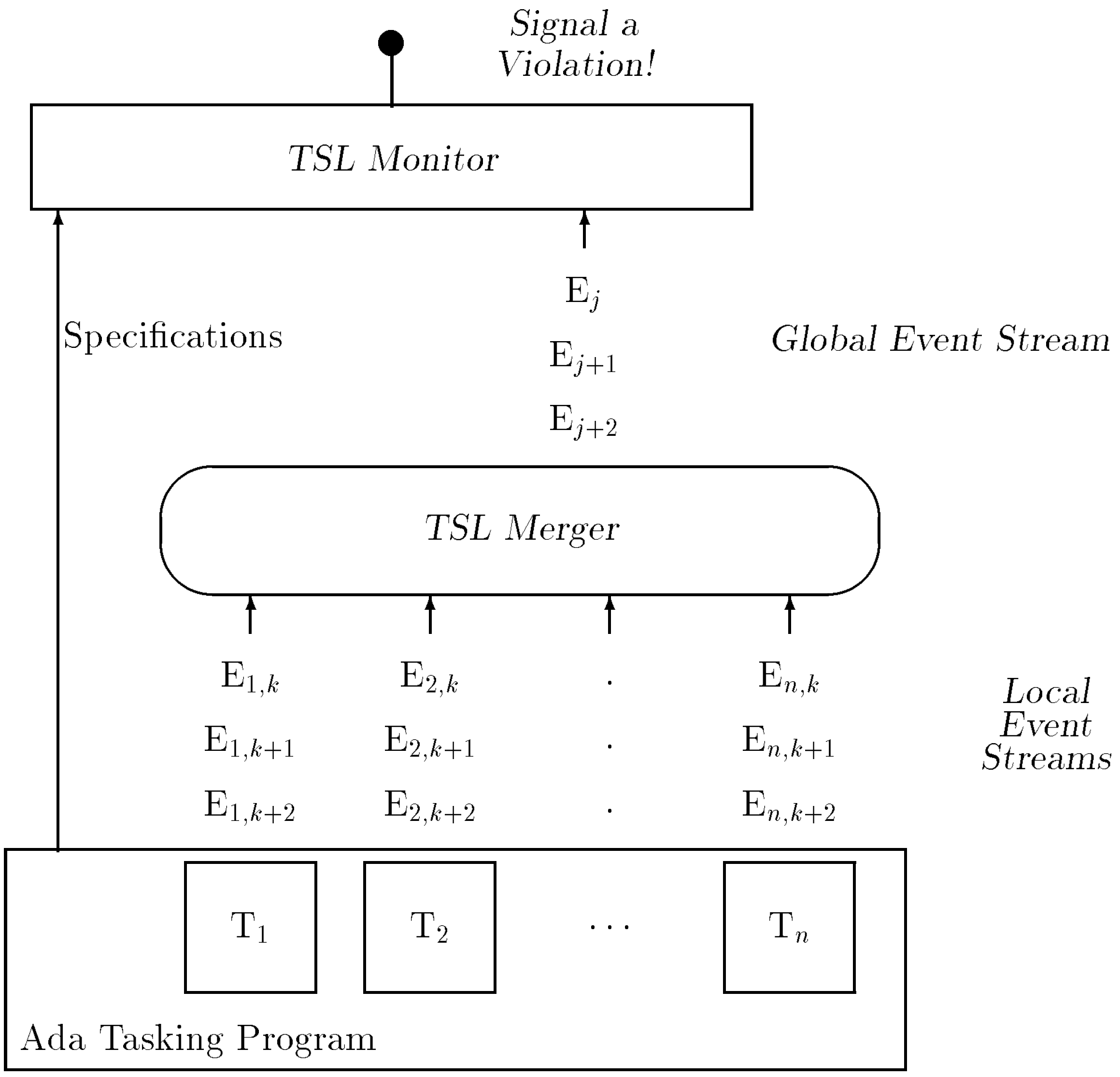

Figure 3: The TSL View of Concurrent Program Execution.

events generated by a single task is connected, while pairs of events generated by different tasks need not be connected. A fundamental assumption in using TSL is that connected events from the local streams will appear in the correct order in the global stream; it is up to the TSL Merger to guarantee the validity of this assumption.

TSL constructs are introduced in program text as formal comments using the $--+$ formal comment indicator. The basic construct of TSL is the *specification*, which is a special formation of a set of *compound events*. The TSL Monitor decides whether a specification is *satisfied* or *violated* by matching the constituent compound events against the global event stream. The basic format of the specification is as follows, with optional parts shown in square brackets:

[**when** E_a
then] [**not**] E_b
[**before** E_t];

The compound event E_a is referred to as the *activator* of the specification, the compound event E_b is referred to as the *body*, and the compound event E_t is referred to as the *terminator*. Additionally, the terminator may be specified using the keyword **until** instead of the keyword **before**. The presence of the keyword **not** signifies a *negative specification*; its absence signifies a *positive specification*.

Each time the TSL runtime system matches the activator of a specification, it creates an *instance* of the specification. Each instance of a specification is satisfied or violated depending on three criteria:

1. Whether the body is matched first, the terminator is matched first, or the two are matched simultaneously.
2. Whether the specification is positive or negative.
3. Whether the terminator is specified with **before** or **until**.

Table 1 summarizes the semantics of a TSL specification based on these three criteria. Essentially, the syntax of the specification makes its meaning intuitive; for example, the specification "**when** A **then** B **before** C" simply means "whenever A is matched, then B must be matched before C is matched". If the activator of a specification is missing, the activator is implicitly "**when** the scope of this specification is entered". Similarly, if the

	Positive Specifications		Negative Specifications	
	Before	**Until**	**Before**	**Until**
Body Matched First	*Satisfied*	*Satisfied*	*Violated*	*Violated*
Terminator Matched First	*Violated*	*Violated*	*Satisfied*	*Satisfied*
Body and Terminator Matched Simultaneously	*Violated*	*Satisfied*	*Satisfied*	*Violated*

Table 1: Semantics of TSL Specifications.

terminator is missing, the terminator is implicitly "**before** the scope of this specification is exited".

A compound event is a pattern of *basic events*. There are two kinds of basic events in TSL, *predefined basic events* and user-defined *actions*. The *TSL Compiler* instruments a user tasking program with calls to the TSL Runtime System; these calls generate the events that are sent to the TSL Monitor for pattern-matching and specification checking. The TSL Compiler causes the program to generate the predefined events at all places in the program where the corresponding source construct is executed; however, the user must insert **perform** statements at each place a user-defined action is to be generated.

There are six TSL predefined events for Ada:

- C **calls** T **at** E—generated by C immediately prior to each simple, timed or conditional call to entry E of task T.
- T **accepting** E—generated by T immediately prior to each simple **accept** statement for entry E or a selective wait with an open **accept** alternative for entry E.
- T **accepts** C **at** E—generated by T at the beginning of the body of each **accept** statement for entry E; note that this corresponds to the beginning of a rendezvous, the calling task in this case being task C.
- T **releases** C **from** E—generated by T at the end of the body of each **accept** statement for entry E; note that this corresponds to the end of a rendezvous, the calling task in this case being task C.

- *T* **activates** *D*—generated by *T* immediately following the activation of task *D*.
- *T* **terminates**—generated by *T* at the end of its body; note that this corresponds to the completion of *T*, *not* its termination.

A user defined action is declared by an **action** declaration, which declares the name of the action along with an optional set of parameters; it is matched by a basic **performs** event. For example, if the **calls** event were not predefined, it could be declared by the following **action** declaration:

```
--+ action CALLS (CALLEE : task; CALLEE_ENTRY : entry);
```

The CALLS event could then be generated by *C* prior to each entry call, as in the following:

```
--+ perform CALLS (T, E);
T.E;        -- Call to entry E of task T.
```

This occurrence of CALLS would be matched by the following event:

```
--+ C performs CALLS (CALLEE => T, CALLEE_ENTRY => E)
```

Note that the predefined TSL type names **task** and **entry** have been used in the above declaration.

Compound events are defined inductively as follows. A *basic event* is a compound event that is matched by a single event in the global stream. A *sequence* of compound events is matched when the constituent events are matched in the order specified; the sequence "*A* then *B* then *C*" is written in TSL as "*A* => *B* => *C*". A *disjunction* of compound events is matched when any one of the constituent events is matched; the disjunction of the events *A*, *B* and *C* is written in TSL as "*A* **or** *B* **or** *C*". A *conjunction* of compound events is matched when any complete permutation of the constituent events is matched; a conjunction may be defined in terms of sequences and disjunctions. For example, the conjunction

```
A and B and C
```

is equivalent to the compound event

```
(A => B => C) or (A => C => B) or
(B => A => C) or (B => C => A) or
(C => A => B) or (C => B => A)
```

An *iteration* of a compound event is matched if the compound event is matched the specified number of times; the iteration "3 occurrences of A" is written in TSL as "$(A) \uparrow 3$".

Constituents of events and actions (e.g., task names and entry names) may be specified either by a name from the program, by the special name **any**, or by a *placeholder*; a placeholder is a variable name preceded by a question mark (e.g., *?A*). A placeholder appearing in a specification is initially unbound and is at various times bound to values appearing in the global event stream. A placeholder that is bound during matching of the activator of a specification remains bound during matching of the rest of the activator and during the matching of the created instance. If a placeholder remains unbound after creation of an instance, the placeholder may be bound to different values in the body and in the terminator. As an example, suppose that the TSL Monitor is attempting to match an instance of the TSL event "*?X* **calls** *?Y* **at** *?Z*", and suppose that *?X* is bound to task C. Then the event will match the first occurrence in the global event stream of an entry call by task C. Suppose the first matching occurrence is the event "C **calls** T **at** E". The match will cause the unbound placeholders *?Y* and *?Z* to be bound to T and E, respectively.

In addition to this basic event language, TSL provides a *guard* facility for qualifying which instances of basic events are to be matched; a basic event E guarded by guard G is written as "E **where** G" and is matched by an occurrence of E only when G is true. A guard is a Boolean-valued expression that is written using the Ada expression language and *property* evaluations.

A property is a function of the global event stream, but it is often convenient to think of a property as defining an array of values, indexed by other values. There are predefined properties, and the user can define his own with a **property** declaration. A property declaration gives the name of a property, its type, the types of the values it is indexed by, the initial value for all "components", and the events which cause the value of the property to change. Value changes are specified by *update* statements within the property declaration. An update statement specifies an activator and optional body that causes an assignment to the property to be executed each time the specified events are matched. For example, the predefined property CALLING, indicating whether or not one task is calling another task, is defined as follows:

```
property CALLING (task) : BOOLEAN := FALSE
is
  when ?C calls any at any then
```

```
            set CALLING (?C) := TRUE;
        when any releases ?C from any then
            set CALLING (?C) := FALSE;
    end CALLING;
```

As a larger example of TSL specifications, the formal specification of the Dining Philosophers program of Figure 2 will be presented. There are several requirements of this program that can be formally specified in TSL, including the following:

- The actions of picking up and putting down a fork must alternate.
- Each philosopher uses only the forks to his immediate right and left.
- No fork is picked up by a philosopher when he is eating.
- No fork is put down by a philosopher when he is not eating.
- It is never the case that all philosophers are simultaneously holding one fork while waiting to pick up the other.

This last specification, a specification of *deadlock-freedom*, would not be satisfied by the program of Figure 2, but it may nevertheless be specified in TSL.

Since the tasks in the program are components of arrays, it is necessary to define an ID_OF property, which keeps track of the ID of each task. A FORK_COUNT property is used to keep track of how many forks each philosopher is holding. It is also necessary to define an EATING property, which states that a philosopher is eating once he has picked up two forks and is no longer eating once he has put down two forks. Finally, a property WAITING_TO_EAT_COUNT is used to count the number of philosophers which have picked up the first fork for eating but have not yet picked up the second. These four properties are defined in TSL in Figure 4. It will be assumed for the sake of simplicity that (1) Only philosophers pick up and put down forks, (2) If an eating philosopher puts down one fork then he will put down the second before picking the first up again, and (3) A non-eating philosopher that has picked up one fork will pick up the second fork before he puts the first fork down. All of these assumptions can be easily stated in TSL. Figure 5 gives the TSL specification of the program.

```
--+ property ID_OF (task) : NATURAL := 0
--+ is
--+   when ?T accepts any at GET_ID (ID => ?I) then
--+     set ID_OF (?T) := ?I;
--+ end ID_OF;

--+ property FORK_COUNT (task) : NATURAL := 0
--+ is
--+   when ?F accepts ?P at PICKUP then
--+     set FORK_COUNT (?P) := FORK_COUNT (?P) + 1;
--+   when ?F accepts ?P at PUTDOWN then
--+     set FORK_COUNT (?P) := FORK_COUNT (?P) - 1;
--+ end FORK_COUNT;

--+ property EATING (task) : BOOLEAN := FALSE
--+ is
--+   when ?F1 accepts ?P at PICKUP where FORK_COUNT (?P) = 0
--+   then ?F2 accepts ?P at PICKUP where ?F2 /= ?F1
--+           set EATING (?P) := TRUE;
--+
--+   when ?F1 accepts ?P at PUTDOWN where FORK_COUNT (?P) = 2
--+   then ?F2 accepts ?P at PUTDOWN where ?F2 /= ?F1
--+           set EATING (?P) := FALSE;
--+ end EATING;

--+ property WAITING_TO_EAT_COUNT : NATURAL := 0
--+ is
--+   when ?F accepts ?P at PICKUP where FORK_COUNT (?P) = 0 then
--+           set WAITING_TO_EAT_COUNT := WAITING_TO_EAT_COUNT + 1;
--+   when ?F accepts ?P at PICKUP where FORK_COUNT (?P) = 1 then
--+           set WAITING_TO_EAT_COUNT := WAITING_TO_EAT_COUNT - 1;
--+ end WAITING_TO_EAT_COUNT;
```

Figure 4: Properties for TSL Specification of Dining Philosophers.

```
--   PICKUP alternates with PUTDOWN:
--+ when ?F accepts ?P1 at PICKUP
--+ then ?F accepts ?P1 at PUTDOWN
--+ before ?F accepts ?P2 at PICKUP;
--+
--+ when ?F accepts ?P1 at PUTDOWN
--+ then ?F accepts ?P1 at PICKUP
--+ before ?F accepts ?P2 at PUTDOWN;

--   A philosopher uses only the forks to his immediate right and left:
--+ not ?P calls ?F where ID_OF (?F) /= ID_OF (?P) and
--+                        ID_OF (?F) /= (ID_OF (?P) + 1) mod 5;


--   A philosopher does not pick up a fork while he's eating:
--+ not ?P calls any at PICKUP where EATING (?P);

--   A philosopher does not put down a fork while he's not eating:
--+ not ?P calls any at PUTDOWN where not EATING (?P);

--   Deadlock never occurs:
--+ not any calls any at PICKUP
--+          where WAITING_TO_EAT_COUNT = MAX_PHILOSOPHERS;
```

Figure 5: TSL Specification of Dining Philosophers.

2.4 Summary

As the survey of this chapter demonstrates, the theory and practice of concurrent programming is at a stage at which it is feasible to develop large distributed software systems for execution on parallel hardware. However, several deficiencies and unsolved problems are apparent in this technology as it currently exists:

1. The technology underlying the construction of distributed implementations of concurrent programming languages is deficient for the following reasons:

   (a) Descriptions of concurrent languages in general focus on the design of constructs for expressing concurrency. Implementations, if they exist, are usually mentioned only in passing for the purpose of claiming the successful implementation of a language.

   (b) Concurrent programming languages other than Ada which have been successfully implemented on multiprocessors are in general small, experimental languages.

   (c) No successful distributed implementation of Ada has been described to a satisfactory extent in the literature; previous descriptions have dealt exclusively with the design of uniprocessor implementations. Thus, no serious consideration has been given to the important issues in distributed supervisor design—the overall software architecture, the interface to application programs, the interface to the underlying computer system, the separation of machine-independent components from machine-dependent components, and the separation of components of unrelated functionality.

2. The technology available for verifying that a distributed implementation of a concurrent language is consistent with the language semantics is deficient for the following reasons:

   (a) No suitable system exists for automatic verification of concurrent software systems, such as a distributed supervisor. Furthermore, the systems which do exist require a great deal of reasoning on the part of the programmer to introduce auxiliary variables and construct a global invariant that is strong enough for proving non-interference or coöperation of the proofs of individual tasks.

   (b) Other testing methods—parallel debuggers, history logging, reproducible testing, interactive replays—are unable to automatically differentiate incorrect program behavior from correct program behavior. Such determination is left entirely up to the programmer.

   (c) As will be further discussed in Chapter 6, methods for verifying a language implementation based on automated testing suffer from the fact that each test program must be instrumented individually, with machine-processable specifications or with executable checking code, so that it can check some part of the language implementation.

This thesis describes new approaches to solving the problems which have caused the continued existence of the above deficiencies.

Chapter 3

Distributed Supervisor Design Principles

This chapter presents in a language-independent manner a set of principles for designing distributed tasking supervisors for concurrent programming languages. The next chapter shows how these principles were applied to the design of the Distributed Ada Supervisor.

A distributed implementation of a concurrent programming language requires a distributed tasking supervisor, since a centralized supervisor is a highly inefficient bottleneck—a localized site of relatively intense communication activity. The distribution is best achieved by assigning an identical copy of the supervisor code to each node (processor) of the target system. Each supervisor copy serves as the execution *agent* for locally executing program tasks and acts as a gateway for communication with remotely executing program tasks.

The design of a distributed supervisor is influenced by both the tasking features of the language and the various computer hardware systems it is targeted for. As a starting point for the discussion, consider Figure 6, which shows abstractly how a tasking supervisor achieves distributed execution of concurrent programs on a multiprocessor system. In this figure, the interface between the tasking program and the supervisor is considered to be the *top-level interface* to the supervisor; the interface between the supervisor and the underlying machine is the *bottom-level interface.*

On the one hand, since the compilation of source-level tasking constructs is nothing more than calls to supervisor subprograms, it is reasonable to expect the top-level supervisor interface to be portable, machine-independent and standardized. Thus, there are components

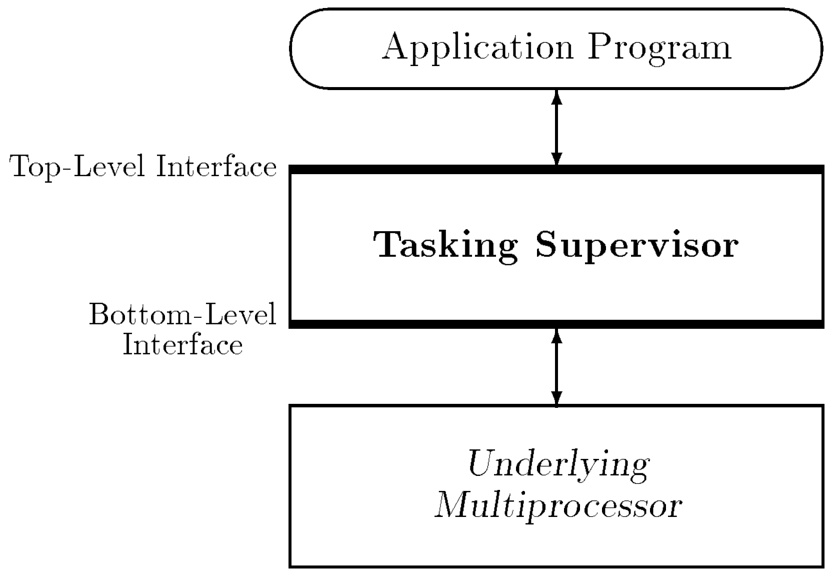

Figure 6: A Supervisor Supporting Execution of Concurrent Programs.

of the tasking supervisor design that remain unchanged as the supervisor is targeted for different machines.

On the other hand, the bottom-level interface between the supervisor and the underlying system is by necessity machine-dependent. Multiprocessor hardware configurations vary widely, depending on the availability of shared memory, the reliability of processor interconnections and the type of communication subsystem available. If the supervisor is to be built on top of, or as part of, an available operating system, it must be recognized that different operating systems provide widely varying collections of primitives for accessing the resources of the target system. If the supervisor is to be implemented on a bare machine, one is confronted with a wide variety of available processor architectures. Thus, there are components of the tasking supervisor design that differ radically as the supervisor is ported between different distributed target systems.

The portability of the supervisor of Figure 6 can be increased dramatically by constructing a layered design which separates and isolates into a single module the machine-independent components from the machine-dependent components. To port the supervisor from computer system A to computer system B, the machine-dependent module for system A is replaced by an appropriate module for system B, leaving the machine-independent modules untouched.

To attain maximum portability, the upper layers of the supervisor must be provided with

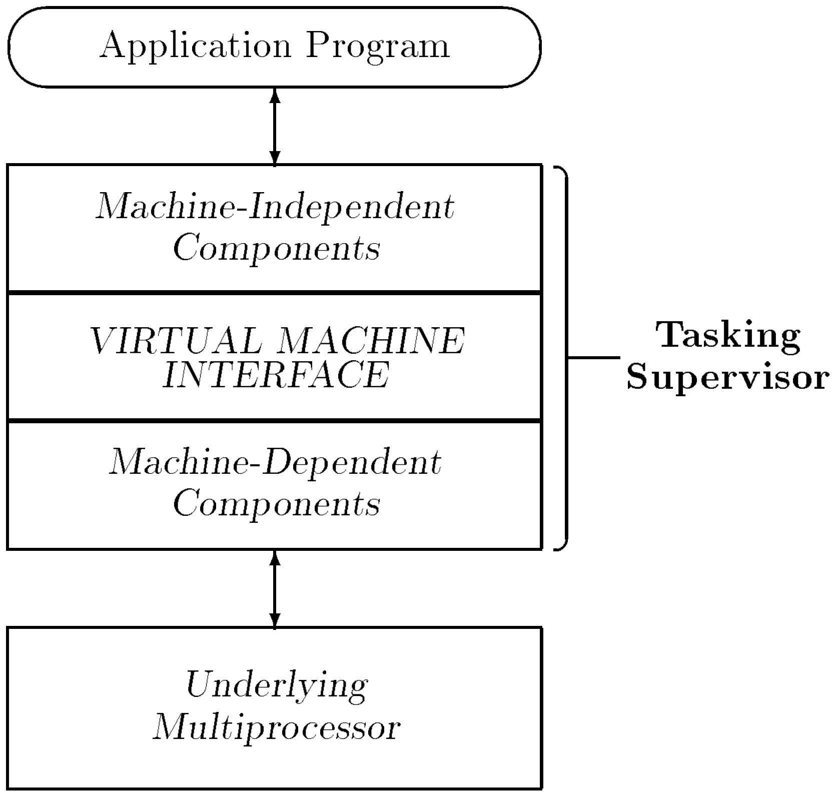
Application Program
Machine-Independent Components
VIRTUAL MACHINE INTERFACE
Machine-Dependent Components
Underlying Multiprocessor
Tasking Supervisor

Figure 7: Partitioning the Tasking Supervisor Design.

a machine-independent view of the underlying computational resources. Brinch Hansen demonstrates the utility of such an *abstract virtual machine* in his description of the semantics of Concurrent Pascal [BH75]. To achieve the desired separation of supervisor components, the virtual machine must support a "universal" distributed computational model which can effectively serve the needs of the machine-independent parts of the supervisor and also encapsulate the relevant features most likely to be encountered on any chosen target system. This computational model is represented as a *virtual machine interface* which is the boundary between the machine-independent and machine-dependent components of the supervisor. It provides an interface to relevant data structures and services that are typical of distributed operating systems (see Section 2.2). Figure 7 depicts the binding of machine-dependent supervisor components to machine-independent supervisor components through the virtual machine interface. Figure 8 depicts a final refinement of Figure 7, showing the individual layers of the supervisor design structure, each of which is described below.

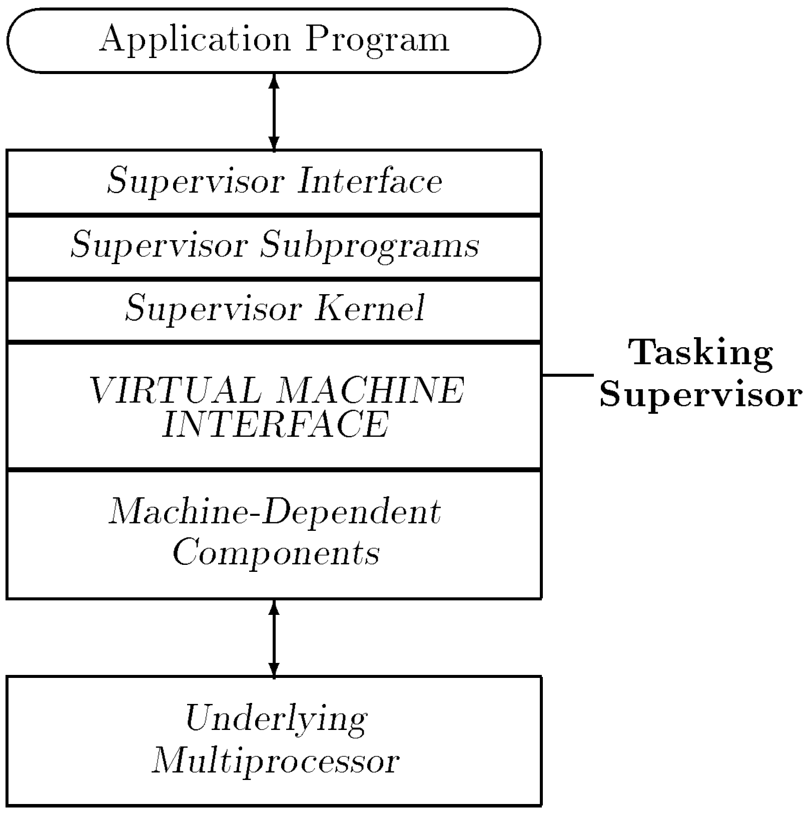

Figure 8: Structure of the Distributed Tasking Supervisor Design.

3.1 The Virtual Machine Interface

In order for the supervisor to be portable between as many different computer systems as possible, the underlying computational model of the supervisor design must be a *loosely coupled multiprocessor* [HB84], i.e. a multiprocessor in which the processors (nodes) communicate over memoryless internode links and for which no global shared memory is available. A restriction of this model is the *homogeneous loosely coupled multiprocessor*, i.e. a loosely coupled multiprocessor in which all processor architectures are identical. This restricted model represents a practical limit on the types of target system which can support a distributed implementation of a concurrent programming language. It also greatly simplifies the design requirements of the supervisor, since homogeneity of the nodes removes the need for a common external data representation and the need for managing multiple sets of object code. The virtual machine interface of the supervisor must accurately present this fundamental computational model.

Some examples of loosely coupled multiprocessors include the *FLEX/32* [Mat85], the *Cm** [FFS77] and a local network of workstations. All other computational models are special cases of the homogeneous loosely coupled multiprocessor. A *tightly coupled multiprocessor* is a loosely coupled multiprocessor in which the set of internode communication links is a single, global shared memory space. A *uniprocessor* is a tightly coupled multiprocessor containing a single processing element.

Because the underlying computational model of the supervisor is a loosely coupled multiprocessor, communication between nodes must be in the form of *message-passing* [Tan81] or one of its variants, such as *remote procedure call* [BN84]. The most representative form of message-passing is point-to-point message-passing, that is, communication of messages between a single source node and a single destination node. Other special forms of message-passing, such as *broadcasting* [Tan81], are suited only to certain system topologies which can be exploited for communication efficiency. Since message-passing is used for communication between supervisor copies, a language-dependent message-passing protocol is required which correctly implements the semantics of the concurrency constructs available in the language.

3.1.1 Interface Requirements

The upper layers of the supervisor require an interface to the following low-level services:

- **Execution management facilities**, for the creation, activation, suspension and termination of multiple threads of execution. Such facilities are needed for two different types of processes:
    - ◦ *OS Processes*, large-grained threads of control which execute on different processors. These are scheduled by the underlying operating system and correspond to the multiple executing copies of the distributed supervisor.
    - ◦ *Local Processes*, fine-grained threads of execution which are scheduled inside a single OS process. These correspond to the tasks in a source program.

  Both types of process may be implemented by a *lightweight process* if the operating system provides such a construct. As mentioned in Section 2.2, if the *V System* were to be used for the underlying operating system support, a local process could be implemented by a V *process*, and an OS process could be implemented by a V *team* [Che84].

- **Synchronization facilities**, such as semaphores, for protecting critical sections.
- **Communication facilities**, to support reliable, sequenced message-passing [Tan81] between OS processes.
- **ID generators**, which return globally unique numerical identifiers for distinguishing program tasks and for identifying other tasking transactions, such as rendezvous.
- **Configuration predicates**, which provide fundamental information about the configuration of a chosen target system, such as the number of processors available.

The implementation of a service may simply be a call to an operating system primitive if such exists; otherwise the service must be implemented from scratch. If the supervisor is to be targeted to a bare machine, then all of the above services must be implemented from scratch.

3.2 The Supervisor Kernel

Each copy of the supervisor contains a *kernel component* which logically resides on top of the virtual machine interface; the set of kernel copies collectively forms the *supervisor kernel*. Within the kernel, and within all layers which logically reside on top of the kernel, each program task is referred to by a globally unique integer identifier called its *task name*. In addition, each system processor (or equivalently, each supervisor copy) is referred to by a globally unique integer identifier called its *node address*. Supervisor subprograms use the kernel to send and receive messages to and from other copies of the supervisor.

The supervisor kernel has the following responsibilities:

1. The supervisor kernel serves as the *transport* communication layer [Zim80] of the supervisor in the sense that it presents a logically fully connected network of program tasks to the higher layers of the supervisor. For example, if execution of a supervisor subprogram requires sending a message to a remote task, the kernel is responsible for determining which system node to send this message to so that the message reaches the destination in an efficient manner. The algorithm for the subprogram can then be designed without regard to node addresses or system topology.

2. The supervisor kernel is responsible for maintaining a consistent global picture of the locations, dependencies and execution states of tasks during program execution. This

global picture represents the dynamically changing state of an executing program.

The first responsibility is satisfied by special language-dependent SEND and RECEIVE routines; the second responsibility is satisfied by a database which stores information about program tasks. For the sake of efficiency, each copy of the kernel stores and maintains in its database only that subset of the global picture relevant to locally executing tasks. This subset is comprised of a *local task map* and a *global task map*. The local task map contains a variety of state information about locally executing tasks. The global task map serves as an "address book" listing the name and node address of all tasks whose location is "known" to the supervisor copy which owns the map.

3.3 Supervisor Subprograms

The top-level interface between the supervisor and a compiled application program is a set of subprogram declarations; the bodies of these subprograms implement the concurrency constructs of the source language. The subprograms logically reside on top of the kernel layer of the supervisor. A good understanding of the language semantics should make clear what subprograms are needed and how they should be parameterized. Roughly speaking, a single subprogram must be provided for each of the following kinds of language construct:

- The creation of a program task.
- The termination of a program task.
- Each language construct which allows one program task to interact with another (for example, the Ada entry call statement).
- Each language construct whose execution requires the use of some low-level service such as an operating system primitive (for example, the Ada **delay** statement).
- Each language construct whose execution requires information maintained by the supervisor (for example, the Ada COUNT attribute).
- Each language construct whose execution affects the execution of one of the other kinds of language construct (for example, the beginning of a nested scope of execution in Ada).

Once the subprograms have been chosen, the design of the supervisor interface is completed by formally specifying the subprograms. These specifications are derived from the language semantics and are constraints on both the compilation of application programs and on the implementation of the subprograms.

3.4 Summary of Design Principles

The design of a distributed tasking supervisor can be summarized as follows:

1. The supervisor is distributed, since a centralized supervisor would be a bottleneck.
2. Distribution is achieved by placing an identical copy of the supervisor code on each node of the target system.
3. The supervisor is constructed in layers to ease enhancement of the design; modification of one module should have little impact on other modules.
4. To maximize portability, the underlying computational model of the supervisor is a loosely coupled multiprocessor, the most general model of distributed computation.
5. Supervisor copies communicate with each other using a language-dependent message-passing protocol.
6. Machine dependencies are encapsulated into a single module, which comprises the lowest layer of the supervisor.
7. The supervisor is formally specified, for documentation, automated testing and/or formal verification.

A well-designed supervisor will manifest itself in both the ease with which portability is achieved and the ease with which an implementation is tested and maintained.

Chapter 4

Design of the Distributed Ada Supervisor

This chapter presents the design of the Distributed Ada Supervisor; this chapter provides the first known detailed description of such a design. In designing the Supervisor, the principles presented in the previous chapter were adhered to rigorously.

4.1 Overview of the Distributed Ada Supervisor Design

The Supervisor is written in Ada to take advantage of Ada's features for strong typing and modularity. In addition, The Supervisor interface is specified in Anna in order to formally describe the interface to the compiler implementor. Figure 9 depicts the Distributed Ada Supervisor executing an application program on a two-node multiprocessor. The Supervisor is an almost full implementation of the Ada tasking language; only the optional interrupt, task priority and shared variable features of Ada are unaccounted for in the design. In particular, each copy of the Supervisor is able to perform the following operations:

1. Perform message-passing with other supervisor copies to support execution of simple **accept** statements, selective waits, simple entry calls, conditional entry calls and timed entry calls. Thus, full support for the various rendezvous mechanisms is provided.

2. Perform message-passing with other supervisor copies to support execution of **abort** statements.

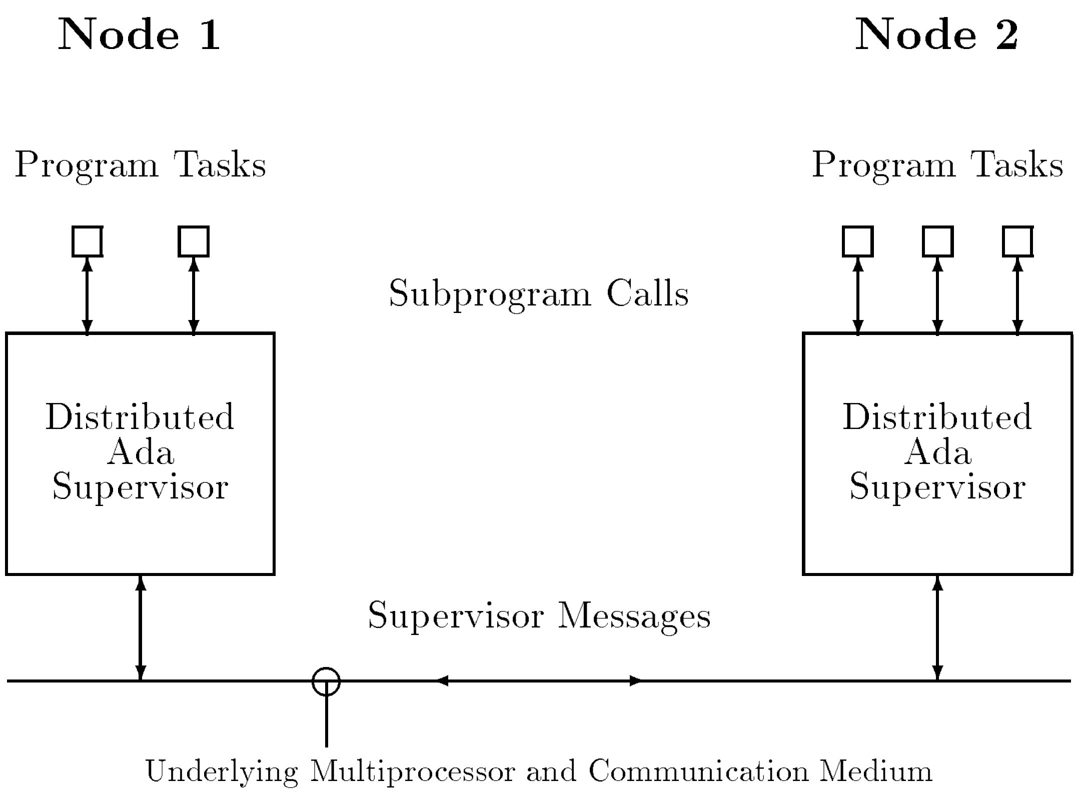
Node 1
Node 2
Program Tasks
Program Tasks
Subprogram Calls
Distributed Ada Supervisor
Distributed Ada Supervisor
Supervisor Messages
Underlying Multiprocessor and Communication Medium

Figure 9: Computational Environment of the Distributed Ada Supervisor.

3. Perform message-passing with other supervisor copies to evaluate the task attributes CALLABLE and TERMINATED.

4. Perform message-passing with other supervisor copies to achieve remote scheduling of program tasks.

5. Perform message-passing with other supervisor copies to correctly synchronize the activation and termination of dependent tasks with the execution of their masters.

6. Perform message-passing with other supervisor copies to perform distributed execution of the **terminate** alternative.

7. Evaluate the entry attribute COUNT. It is a consequence of the Ada tasking semantics [Ada83, § 9.9, ¶ 5–6] that a task may only evaluate the COUNT attribute of its own entries; thus, this service is always a local operation—it requires no message-passing between copies of the supervisor.

For the purposes of discussion, the first six of the above operations are all situations in which one program task *communicates* with another.

4.2 Ada Supervisor Structure

Figure 10 depicts the set of packages which comprise each copy of the Distributed Ada Supervisor; packages are shown with a solid border, while parts of a package are separated by dashed lines. Figure 11 depicts the correspondence between this figure and Figure 8 in Chapter 3.

Several characteristics of the Supervisor are similar to the model described by Clemmensen [Cle82]; these similarities will be noted throughout the description of the Supervisor design. The packages shown in the diagram have the following functions:

- NET_SERVICES—This package is the *abstract virtual loosely coupled multiprocessor* upon which the rest of the Supervisor is built. Its visible part is the virtual machine interface; the interface declares in a language-independent and machine-independent manner the set of low-level services needed by the rest of the Supervisor. Its body contains the implementation of the virtual machine using the services available on the chosen target system.

- MESSAGES—This package declares the Supervisor message type and is the interface between the Supervisor kernel (see below) and the low-level message-passing routines of the virtual machine.

- TASK_INFO_MANAGER—This is a subunit of TASKING_SUPERVISOR and is responsible for maintaining a consistent picture of the execution state of the application program at runtime.

- TASKING_SUPERVISOR—This is the main Supervisor package. Its visible part is the interface to the subprograms which are called by the application program at runtime. Its body contains the implementation of these subprograms plus the special SEND and RECEIVE communication subprograms. The SEND and RECEIVE subprograms, together with the TASK_INFO_MANAGER, comprise the *Supervisor kernel*.

Figure 12 depicts an example of how these packages interact. The figure depicts part of the

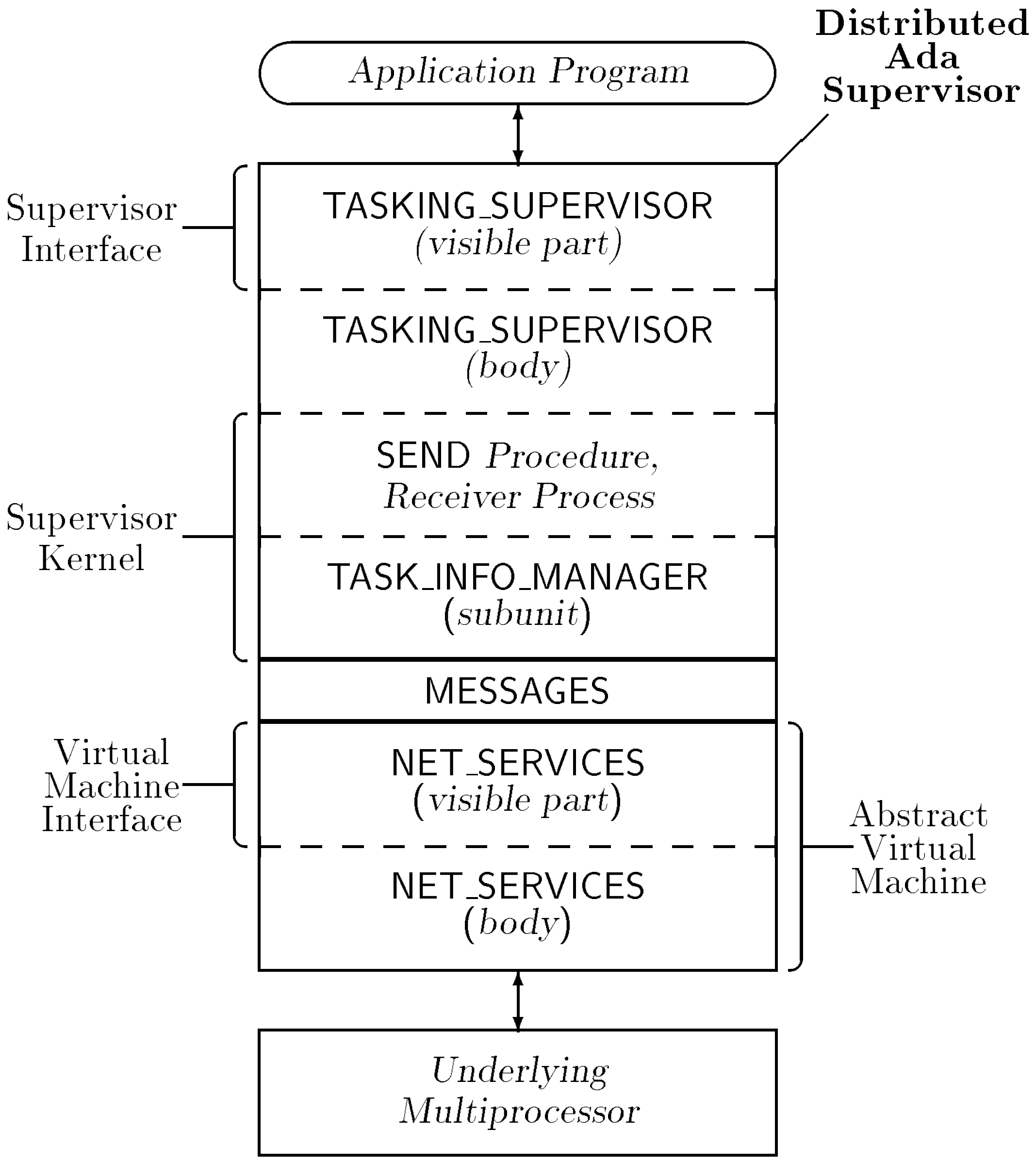
Distributed Ada Supervisor
Application Program
Supervisor Interface
TASKING_SUPERVISOR (visible part)
TASKING_SUPERVISOR (body)
Supervisor Kernel
SEND Procedure, Receiver Process
TASK_INFO_MANAGER (subunit)
MESSAGES
Virtual Machine Interface
NET_SERVICES (visible part)
NET_SERVICES (body)
Abstract Virtual Machine
Underlying Multiprocessor

Figure 10: Structure of the Distributed Ada Supervisor Design.

"Generic" Distributed Supervisor	Distributed Ada Supervisor
Supervisor Interface	TASKING_SUPERVISOR (*visible part*)
Supervisor Subprograms	TASKING_SUPERVISOR (*body*)
Supervisor Kernel	SEND *Procedure, Receiver Process*
	TASK_INFO_MANAGER (*subunit*)
	MESSAGES
VIRTUAL MACHINE INTERFACE	NET_SERVICES (*visible part*)
Machine-Dependent Components	NET_SERVICES (*body*)

Figure 11: Correspondence with the Generic Supervisor Design.

execution of an entry call statement, with the arrows representing subprogram calls. The portion between the two horizontal lines represents execution inside the Supervisor. At the application program level, the entry call has been compiled into a call to the ENTRY_CALL procedure of the TASKING_SUPERVISOR package (described in Section 4.6.8). To call task T, ENTRY_CALL calls the kernel to SEND a CALL_MSG to T. The kernel uses the TASK_INFO_MANAGER to determine which node to send the message to, and it calls the MESSAGES package to SEND the message to the node. The MESSAGES package flattens the message into an array of bytes and calls NET_SERVICES to perform the SEND. Finally, NET_SERVICES calls a routine in the underlying operating system to SEND the message

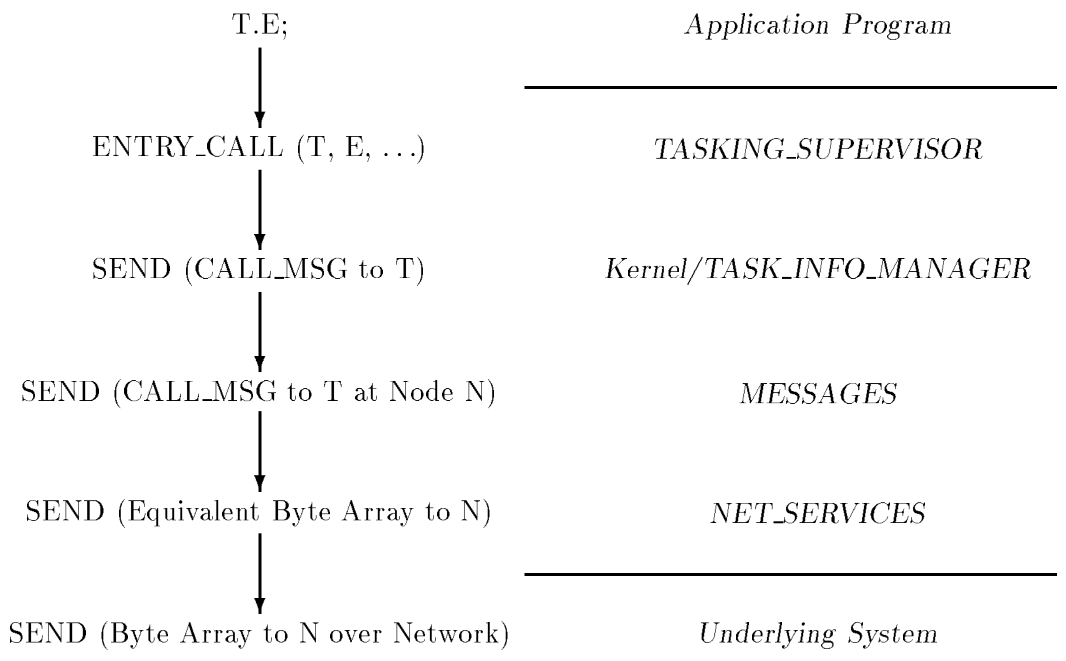

Figure 12: Interaction of the Supervisor Packages During an Entry Call.

over the network. The receipt of the message and the placement of the call on the queue for T's entry E are carried out in the reverse sequence of operations at node N.

The design of the various layers of the Supervisor are now presented in detail, in order from bottom to top.

4.3 The Virtual Machine Interface

Figure 13 is an outline Ada specification of the NET_SERVICES package specification; the complete specification is given in Appendix A. NET_SERVICES contains subprogram interfaces to each of the services described in Section 3.1. Chapter 5 describes the implementation of the body of NET_SERVICES for two different machines.

The facilities of NET_SERVICES have been chosen so as to closely represent the facilities available in a number of existing distributed operating systems, such as the *V System*. NET_SERVICES provides for the naming of system processors by a contiguous sequence of *logical node addresses*, which the package converts to system physical node addresses or process identifiers. It also provides an interface to low-level byte-oriented

```
package NET_SERVICES is
    Byte types—BYTE, BYTE_ARRAY, BYTE_ARRAY_REF;
    Constant—MAX_NODES;        -- Maximum number of CPUs available.

    ID types—NODE_ID_TYPE, PROCESS_ID_TYPE, UNIQUE_ID_TYPE;

    Unique ID generator—GET_UNIQUE_ID;

    Timer—TIME_OUT;

    -- SEND does not block its caller while the message is being sent;
    -- RECEIVE blocks its caller until a message has arrived:
    Message-passing subprograms—SEND, RECEIVE;

    Concurrency subprograms—CREATE_OS_PROCESS, DESTROY_OS_PROCESS,
              CREATE_LOCAL_PROCESS, ACTIVATE_LOCAL_PROCESS,
              SUSPEND_LOCAL_PROCESS, TERMINATE_LOCAL_PROCESS;

    Semaphore type—LOCK;
    Semaphore exception—LOCK_IS_UNINITIALIZED;
    Semaphore subprograms—INITIALIZE, FINALIZE, ACQUIRE, RELEASE,
             CONDITIONAL_ACQUIRE, IS_LOCKED, IS_INITIALIZED;

--| Formal specification of NET_SERVICES;
private
    Implementation of LOCK;
end NET_SERVICES;
```

Figure 13: Outline of the Virtual Machine Interface.

message-passing services, task switching and scheduling services, timing services and process management services. Most of the entities in the package should be self-explanatory. The semaphore type is called LOCK; the ACQUIRE subprogram is equivalent to the P operation, and the RELEASE subprogram is equivalent to the V operation. Additionally, CONDITIONAL_ACQUIRE is a non-blocking ACQUIRE attempt that returns an indication of success of failure. The SEND subprogram is called by program tasks during execution of a tasking statement; the RECEIVE subprogram is called by the RECEIVE process of the Supervisor kernel (see below). The behavior of the interface has been formally specified in Anna (see Appendix A).

4.4 The Supervisor Kernel

As described in the previous chapter, each copy of the Distributed Ada Supervisor contains a *kernel* component which logically resides on top of the virtual machine interface (see Figure 10). In addition to the responsibilities listed in the previous chapter, the kernel can be enhanced to support migration of tasks between system nodes during program execution; the necessary enhancements are described in [Ros87]. The TASK_INFO_MANAGER implements the local task map and global task map. These data structures are similar in functionality to the *monitortables* of Clemmensen's model and are described separately in Sections 4.4.1 and 4.4.2 below, following a discussion of their similarities. The SEND and RECEIVE routines of the kernel are described respectively in Sections 4.4.3 and 4.4.4.

Each entry of the local and global task maps contains information about a specific program task. Thus, searching within both maps is performed using a *task name* (i.e., unique integer identifier) as the key value. Since the task maps are shared by multiple threads of control, updates to the maps are implemented as critical sections. Construction and maintenance of the local and global task maps relies on information obtained from the tasking messages that are passed between supervisor copies. Each tasking message contains the following information:

- The name of the task which *initiated* the tasking operation that caused the message to be sent. This task is called the MSG_SOURCE.
- The name of the intended *recipient* task. This task is called the MSG_DEST.
- The *node address* of MSG_SOURCE, called MSG_SOURCE_NODE.

```
type SUPERVISOR_MESSAGE_TYPE (CLASS : MSG_CLASS) is
    record
        MSG_SOURCE                      : TASK_NAME;
        MSG_DEST                        : TASK_NAME;
        MSG_SOURCE_NODE                 : NODE_ADDRESS;
        CURRENT_SENDERS_MASTER          : TASK_NAME;
        case CLASS is
            . . .      -- Other Message Components;
        end case;
    end record;
```

Figure 14: The Supervisor Message Type.

- The name of an *ancestor* of the task currently serving as the *routing agent* for the message. This ancestor task is called the CURRENT_SENDERS_MASTER, and its function as a routing agent is described more fully below.

An outline of the Supervisor message type is shown in Figure 14. The CLASS discriminant indicates the function of the message and determines what remaining components are supplied with the message. The full declaration of SUPERVISOR_MESSAGE_TYPE is given in Appendix C.

4.4.1 The Local Task Map

The local task map maintained by each copy of the kernel contains a descriptor entry for each task executing at the node of the copy; this supervisor copy is the execution *agent* of each locally executing task. Figure 15 gives an Ada declaration of the local map entry type. The entry for task MY_NAME keeps track of the number of entries it declared (the ENTRY_COUNT discriminant), the name of its master (MASTER_NAME), its execution status (CURRENT_STATUS and MY_STATUS) and its current scope level (CURRENT_SCOPE). As mentioned in Section 2.1.3, tasks are dependents not only of a master task, but of a particular scope within that master task, and a scope may not be exited until all dependents of the scope have terminated. Thus, the CURRENT_SCOPE component keeps track of the nesting level a task is executing so that when waiting to exit the current scope, the Supervisor can differentiate between dependents of the scope being exited and dependents of enclosing scopes.

```
type LOCAL_MAP_ENTRY (ENTRY_COUNT : NATURAL := 0) is
   record
      MY_NAME                      : TASK_NAME;
      MASTER_NAME                  : TASK_NAME;
      CURRENT_STATUS               : TASK_STATUS := UNBORN;
      CURRENT_SCOPE                : SCOPE_NUMBER_TYPE := 0;
      MY_STATUS                    : STACK_TYPE_1;
      CURRENT_CALLS                : STACK_TYPE_2;
      CALL_MSG_QUEUES              : QUEUE_ARRAY_1
                                        (1 .. ENTRY_COUNT);
      CONFIRM_ABORT_MSG_QUEUES     : QUEUE_ARRAY_2
                                        (1 .. ENTRY_COUNT);
      ACKNOWLEDGMENT_MSG           : SUPERVISOR_MESSAGE_TYPE;
      ABORTED_TASKS                : AVL_TREE_TYPE_1;
      DEPENDENTS                   : AVL_TREE_TYPE_2;
      C_VARIABLES                  : C_VARS_TYPE;
   end record;
```

Figure 15: Contents of a Local Task Map Entry.

Arriving entry calls are placed on the CALL_MSG_QUEUES; there is one queue per declared entry. Messages indicating commitment to or abandonment of a rendezvous by a caller are placed on the CONFIRM_ABORT_MSG_QUEUES.

CURRENT_CALLS contains the names of the tasks which are currently engaged in rendezvous with task MY_NAME. CURRENT_CALLS is a stack because a task may be executing multiple nested **accept** statements simultaneously, and because an inner **accept** statement must complete before the **accept** statement it is contained within may proceed. It is a consequence of Ada syntax and semantics that nested **accept** statements are executed in a LIFO (last-in-first-out) manner.

ABORTED_TASKS is used during execution of an **abort** statement by task MY_NAME; it is an AVL-tree containing the names of aborted tasks which have yet to acknowledge their abnormality. ACKNOWLEDGMENT_MSG is used to hold various acknowledgment and reply messages. DEPENDENTS is an AVL-tree which stores the names of all currently active dependents of MY_NAME. Associated with each dependent is an execution status value and a scope number, indicating the nesting level within MY_NAME that is the direct master scope of the dependent.

```
type GLOBAL_MAP_ENTRY is
    record
        T_NAME : TASK_NAME;
        T_NODE : NODE_ADDRESS;
    end record;
```

Figure 16: Contents of a Global Task Map Entry.

Finally, C_VARIABLES is an array of Hoare-style condition variables [Hoa74] which are used to eliminate busy-wait spins from supervisor synchronization activities. For example, inside the Supervisor, a task beginning the execution of its body performs a WAIT on a condition variable that is used to signify activation of all new dependents; when the last of the new dependents is activated, the Supervisor performs a SIGNAL on the condition variable, allowing the program task to proceed with its execution.

Each time a task is scheduled for activation at a node, the copy of the Supervisor at the node creates an entry for that task in its local map. The name of the master of the task is inserted into the local map entry at this time. Each time the task declares a dependent, it calls its agent at the place of the declaration so that the agent may register the name and master scope level of the dependent in the local map entry for the task. When a dependent of the task terminates, the name of the dependent is removed from the local map entry. Thus, the local map entry for a task always contains the names of its master and currently active dependents.

When the task itself terminates, its local map entry is still kept in the local task map since it is possible for other tasks to attempt communication with the terminated task.

4.4.2 The Global Task Map

The global task map maintained by each copy of the kernel contains an entry for each task whose node location is "known" to the kernel copy, including all locally executing tasks; thus, as shown in Figure 16 each entry in the global map is simply a task name along with its node address. A copy of the Supervisor can learn the location of a remote task in one of two ways—whenever it receives a message from a remote task, and whenever it schedules a dependent of a locally executing task for remote execution. The global task map corresponds to the *WhereTable* of Clemmensen's model.

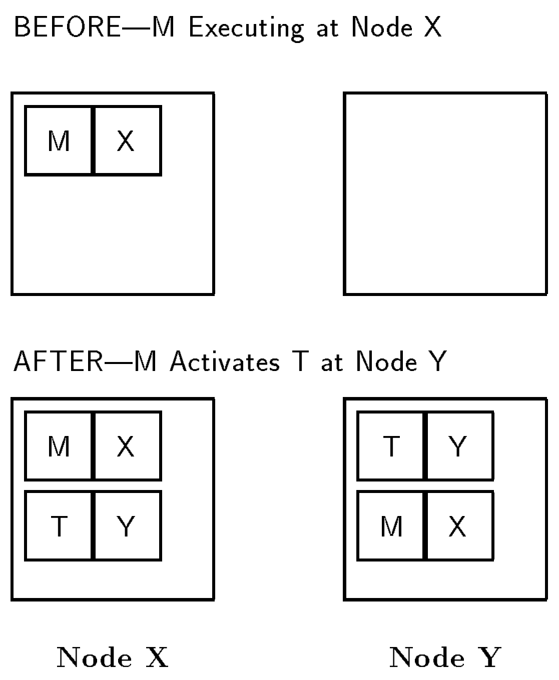

Figure 17: Global Map Updates Resulting from Activation of a Dependent.

Each time a task T is scheduled for activation at a node Y, three global map entries are created; assume that M is the master of T and that M is executing at node X:

1. An entry for T in the global map of node Y.

2. An entry for T in the global map of node X.

3. An entry for M in the global map of node Y.

These global map updates are depicted in Figure 17. Note that if the new task is assigned to the same node as its master, entry number 3 will already exist, and entry number 2 will be taken care of by entry number 1. Thus, the global map at a node contains entries for all locally executing tasks as well as entries for the direct masters and direct dependents of these tasks.

The continued existence of global map entries for terminated tasks is required since it is possible for other tasks to attempt communication with the terminated task.

4.4.3 The Message-Sending Algorithm

When a task communicates with another task, it is ultimately the agents serving the two tasks which must perform message-passing to carry out the tasking operation. If the two communicating tasks reside on the same node, then no distributed message-passing is required; messages are simply queued in appropriate fields of the local task map entries for the two tasks. On the other hand, if the two communicating tasks are executing on different nodes, the node address of one task may not be immediately available to the agent serving the other task, and *vice versa*. This is due to the fact that the agent serving a task initially has available from its global map only the node addresses of local tasks, their direct masters and their direct dependents; the node addresses of other tasks visible inside the body of a task may not be initially available.

Messages destined for non-local tasks are sent to one of two places:

1. If there is an entry in the global map for the destination task, the message is sent to the node location of that task.

2. If there is *not* an entry in the global map for the destination task, the message is sent to the node location of the "closest" ancestor of the source task which is not executing locally. The name of this ancestor becomes the value of the CURRENT_SENDERS_MASTER component of the message and is the *routing agent* for the first hop of the message.

At each node, all locally executing tasks and their dependents are represented in the global map at the node; thus, the second case listed above is required only when a task attempts communication with a remote task that is not one of its dependents. Therefore, it is guaranteed in this case that there is an ancestor of the task that is not executing locally; otherwise, the destination task would be a dependent of one of these ancestors and would thus be represented by an entry in the global map. This ancestor will be the direct master of the task, or the master of the master, or the master of the master of the master, etc.; this ancestor and node address may be determined locally because each intervening local ancestor has a local map entry containing the name of *its* master.

Once the message has been sent, the algorithm described in the next section for receiving and forwarding messages takes over at all subsequent nodes in the path of the message. The Ada implementation of the message-sending algorithm is given in Appendix B.1.

4.4.4 The Message-Receiving/Routing Algorithm

Message sending is an action performed by a program task executing inside the Supervisor; on the other hand, message receiving is an asynchronous operation that requires a single, continually executing and separate thread of control to ensure reliable receipt and processing of all messages arriving at a node. Thus, each copy of the Supervisor must have a dedicated message-receiving process, a separate thread of control responsible for receiving all messages sent to the copy and forwarding all messages not destined for locally executing tasks. This process can be scheduled in the same way that program tasks are scheduled, possibly with higher priority. Such a process serves as a "helper process" in the terminology of Cheriton [Che82], and it corresponds to the *COMMUNICATION* meta-process of Clemmensen's model.

Each time the receiver process receives a message, it adds or updates the global map entry for the MSG_SOURCE using the value of MSG_SOURCE_NODE. These update operations contribute to the frequent enlargement and updating of all global maps so that expensive forwarding of messages through intermediate nodes occurs less and less frequently during program execution.

After the receiver process updates the global map, it checks the global map to see if the MSG_DEST designates a local task. If it does, the message is placed in an appropriate field or queue of the local map entry for MSG_DEST; otherwise, the receiver process forwards the message to one of two places:

1. If there is an entry for MSG_DEST in the global map, the message is forwarded to the node address given in the entry.

2. If there is not an entry for MSG_DEST in the global map, the message is forwarded to the "closest" ancestor of CURRENT_SENDERS_MASTER which is not executing locally. This ancestor then becomes the new value of CURRENT_SENDERS_MASTER and is the *routing agent* for the next hop of the message.

Again, the specification of the local and global map contents ensures that in the second case above such an ancestor and node address can be found. The Ada implementation of the receiver process is given in Appendix B.2 as an Ada task.

4.4.5 Analysis of the Kernel Algorithms

The Ada compiler is responsible for ensuring the semantic legality of all task communications appearing in a source program. The compiler translates legal communication statements to supervisor subprogram calls; each call must pass enough information to the Supervisor so that the Supervisor can implement the requested communication. As will be seen in Chapter 6, other means have been developed for checking the *correctness* of the Supervisor. But because the Supervisor kernel plays such an important rôle in implementing communication between tasks at runtime, it is worth analyzing the kernel algorithms to ensure that all possible communications are implemented.

At each point in the body of a task, communication may be attempted with any other task visible at that point. As a consequence of the Ada scoping and visibility rules, each of these visible tasks must fit into one of the following mutually exclusive classes:

1. A single task or a task declared by an object declaration, which is declared in some declarative part *within* the body in question; or a task designated by an access value, such that the corresponding access type is declared in some declarative part *within* the body in question.

2. A single task or a task declared by an object declaration, which is declared in the *same* declarative part containing the declaration of the body in question; or a task designated by an access value, such that the corresponding access type is declared in the *same* declarative part containing the body in question.

3. A single task or a task declared by an object declaration, which is declared in some declarative region which *encloses* the declarative part containing the declaration of the body in question; or a task designated by an access value, such that the corresponding access type is declared in some declarative region which *encloses* the declarative part containing the declaration of the body in question.

4. A task which is passed as a *parameter* to the body in question and which does not fit one of the above classes.

Figure 18 is a pictorial representation of the relationships between these classes in a hypothetical dependency tree from the point of view of a single task T. Figure 19 is a sample Ada source fragment which also depicts the above visibility relationships from the point of view of a single task T. Note that a task may communicate with any other locally executing

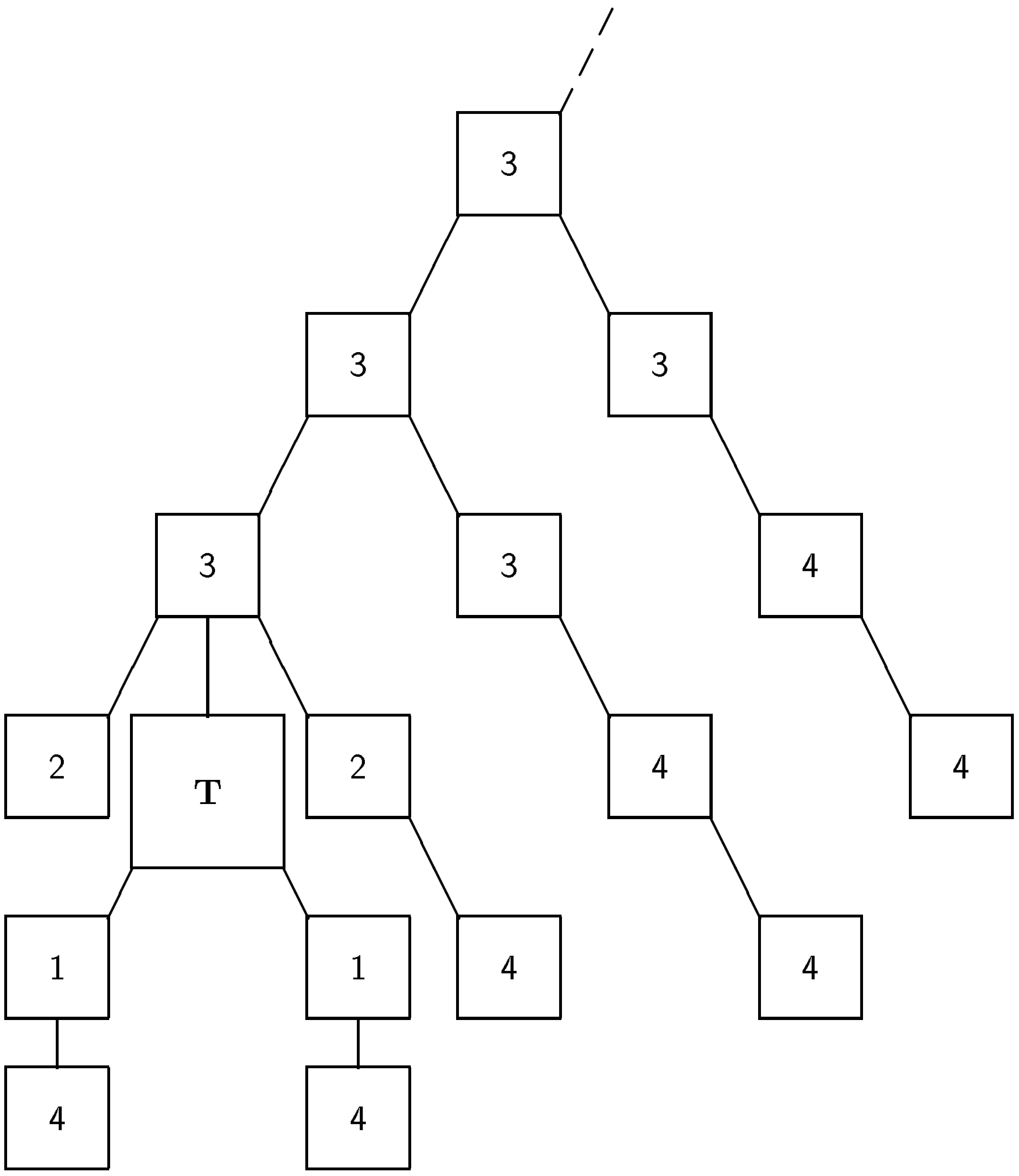

Figure 18: Task Dependency and Visibility Relationships.

```
declare
    task M;
    task body M is
        task type S;
        type S_REF is access S;
        S1 : S;
        S2 : S_REF;

        task T is
            entry E (SP : in out S);
        end T;

        task body S is ... ;

        task body T is
            S3 : S;
            S4 : S_REF := new S;    -- M, not T, is the master of S4.all.
        begin
            -- S3 is visible here as a Class 1 task.
            -- S4.all, S1 and S2.all are visible here as Class 2 tasks.
            -- M is visible here as a Class 3 task.
            accept E (SP : in S) do
                -- Additionally, S5 is visible here as a Class 4 task during
                -- the entry call executed below.
                null;
            end E;
        end T;
    begin      -- M.
        S2 := new S;
        declare
            S5 : S;
        begin
            T.E (S5);       -- S5 is not directly visible to T,
                            -- but is passed as a parameter to T.
        end;
    end M;
begin      -- outer block.
    null;
end;
```

Figure 19: Task Visibility Relationships in a Sample Ada Fragment.

task, since the SEND algorithm of Section 4.4.3 first checks the co-resident global map for the location of the destination of each tasking message, and since Section 4.4.2 specifies that the global map contains entries for all locally executing tasks.

Class 1 tasks are direct dependents of an activation of the body in question. As specified in Section 4.4.1, the location of each dependent is stored in the global map of the agent serving the activation in question. Furthermore, the SEND algorithm first checks the co-resident global map for the location of the destination of each tasking message. Thus, the kernel algorithms allow a task to communicate with all visible Class 1 tasks.

Class 2 and Class 3 tasks are dependents of some (direct or indirect, respectively) master of an activation of the body in question. As specified in Section 4.4.1, the *name* of the direct master of a task is available in the local map entry for the task. In addition, Section 4.4.2 specifies that the *location* of the direct master is available in the co-resident global map. Thus, if a task T attempts communication with any task of these two classes, the tasking message involved can first be sent to the *direct* master M of T. If the destination task is a dependent of M, then the arguments used above for Class 1 apply for this case. Otherwise, the destination task is a dependent of some *indirect* master of T, and the immediately preceding arguments used for T apply inductively to M. Since the algorithms of Section 4.4.3 and Section 4.4.4 perform exactly this routing through masters when the location of a destination task is not immediately available, the kernel algorithms allow a task to communicate with all visible Class 2 and Class 3 tasks.

Class 4 tasks are all objects of named task types; the declarations of the types must be visible to an activation of the body in question. In general, a Class 4 task can be a dependent of any scope in which the declaration of the corresponding task type is visible. Thus, some Class 4 tasks will *not* be direct dependents of an activation of the body in question or direct dependents of some master of such an activation. This fact implies that the kernel algorithms as presented so far do *not* allow communication between an activation of the body in question and an "unrelated" Class 4 task. A simple remedy for this situation is that each time an object of a named task type is activated, a message is sent to the task which declared the task type; this message allows the new task to be registered in the global map at the node of the task which declared the type. Since this latter task is some master of an activation of the body in question, the location of the new task would be available during routing of messages from the body in question. Thus, using these simple enhancements to the message-passing protocol used by the Supervisor, the kernel algorithms allow a task to communicate with all Class 4 tasks. (The details of the message-passing enhancement are described later in Section 4.5.)

4.4.6 Elimination of Kernel Race Conditions

Even though a correct implementation of the task maps and message-passing algorithms should guarantee 100% reliability of all distributed task communication (assuming the use of a reliable lower-level communication system), it is conceivable that certain race conditions could cause the algorithms to fail. Many potential race conditions are overcome by the requirement that map updates be implemented as critical sections. However, more serious race conditions are handled by placing a few simple additional constraints on the Supervisor design.

One serious race condition that could arise during parallel activation of a set of tasks declared in the same declarative part is that one of the activated tasks attempts communication with one of the unactivated tasks. This is legal Ada semantics, and one might suppose that the unactivated task may not be fully accounted for in the appropriate task maps. To overcome this situation, the Supervisor is required to schedule each task at a specific system node prior to the end of the elaboration of the declarative part in which the task is declared. Since scheduling a task includes updating appropriate tasks maps as specified in Sections 4.4.1 and 4.4.2, the location of a task will be known to all other tasks which could communicate with it prior to its activation.

One abnormal system condition that must be considered is failure of a system node on which program tasks are executing. To gracefully recover from such a failure, some amount of distributed redundancy of program state information is required. Such recovery is ignored by the Ada language and is beyond the scope of the research described in this thesis; further research along the lines described by Knight and Urquhart [KU87] will be necessary before full fault-tolerance may be built into a distributed Ada supervisor.

4.5 The Supervisor Message-Passing Protocol

The Distributed Ada Supervisor uses a special protocol for passing messages between participating agents in order to correctly synchronize distributed execution of tasking operations according to the Ada semantics. Weatherly [Wea84a,Wea84b] described such a protocol for a subset of Ada tasking. His protocol also specifies the transitions in execution status that a task undergoes while message-passing is performed on its behalf. These status values are stored on a stack; the top of the stack may be changed, or values may be pushed on and popped off the stack.

The protocol described by Weatherly is an adequate starting point for the protocol required by the Distributed Ada Supervisor; however, the Weatherly protocol lacks the following functionality that is needed for a full implementation of Ada tasking:

1. A message-passing sequence which blocks a task executing an **abort** statement until all of the aborted tasks have become abnormal.

2. A message-passing sequence for scheduling a newly activated task on a remote node.

3. A message-passing sequence for evaluation of the task attributes CALLABLE and TERMINATED.

4. A message-passing sequence for remote propagation of an Ada exception.

5. A message-passing sequence for notifying a remote master of the activation of a dependent, when the activation results from execution of an allocator.

6. Message-passing sequences for distributed execution of the **terminate** alternative.

An additional message-passing sequence is required to complete the implementation of the Supervisor kernel algorithms. As described previously in Section 4.4.5, this sequence enables an object of a named task type to be registered in the global map at the node of the task which declared the type.

This section describes enhancements to the Weatherly protocol which were designed to take care of the above deficiencies; the enhancements are described both textually and by Weatherly-style diagrammatic specifications. Appendix C contains the Ada declaration of the Supervisor message type. Also given in Appendix C is the declaration and description of the task execution status values used by the enhanced protocol. As in the Weatherly protocol, the message type is discriminated by a CLASS component indicating the purpose of the message. The Weatherly message has been enhanced with a BYTE_COUNT discriminant for those variants of the type containing a variable-size byte array component.

4.5.1 Minor Enhancements to the Weatherly Protocol

A few minor enhancements to the existing Weatherly protocol were made for purposes of Supervisor execution efficiency.

In the Weatherly version, **in**-mode parameters to the **accept** statement of a rendezvous are passed to the accepting task in the CALL_MSG of the rendezvous message sequence. In the protocol for the Distributed Ada Supervisor, **in**-mode parameters are passed in the CONFIRM_MSG of the rendezvous sequence instead. It makes no sense to pass a potentially large aggregate of parameters in a CALL_MSG since the call may be subsequently canceled; this would incur a waste of precious communication resources. Instead, the CONFIRM_MSG, which commits the caller to the rendezvous, is better-suited to the purpose of parameter passing. In addition to this major change, the ABORT_MSG of the rendezvous sequence has been renamed to ABORT_CALL_MSG to further distinguish it from the ABORT_TASK_MSG.

To solve the minor kernel communication problem described in Section 4.4.5, a message with CLASS value TYPED_TASK_MSG is sent each time an object of a named task type is activated, including tasks activated by execution of an an allocator. The MSG_SOURCE is the task which activated the typed task, and the MSG_DEST is the task which declared the corresponding task type. The variant for this message has two components, TYPED_TASK_NAME (the name of the new task) and TYPED_TASK_NODE (the node address of the new task). The recipient agent adds these last two pieces of information to its global map

4.5.2 Messages for Scheduling Newly Activated Tasks

Whenever a task declares a new dependent, the agent serving the task must schedule the dependent for execution on some system node. The most efficient partitioning, scheduling and migration strategies, such as those considered in [BF81,Cor84,SH86], require coöperation between supervisor copies to determine which node the task should be placed at so that parallel execution proceeds most efficiently. The simplest scheduling strategy, and the one adopted for use in the Distributed Ada Supervisor, is for the agent serving the activator of the new dependent to arbitrarily choose a node on its own. Even with this simple strategy, however, the agent serving the activator must send a message to another agent if the dependent is to be scheduled on a remote node.

For this purpose, the Supervisor uses a NEW_TASK_MSG. The message is sent during execution of the CHILD_TASK function, which informs the Supervisor of the existence of a new dependent task. The MSG_SOURCE and MSG_SOURCE_NODE components are set, respectively, to the name and node location of the task that is activating the new dependent. The MSG_DEST component is set to the name of the new dependent. The CURRENT_SENDERS_MASTER component is unused during this transaction, since the message is always sent to a particular supervisor copy rather than to a particular task.

The variant for the NEW_TASK_MSG has seven components. The NEW_TASK_ENTRY_ADDRESS component is initialized with the entry address for the code of the new dependent. The NEW_TASK_FRAME_PTR is initialized with the address of the activation record for the declarative part that contains the declaration of the body of the dependent. As was mentioned at the beginning of this chapter, the semantics of shared global variables is unaccounted for in the design of the Distributed Ada Supervisor. However, each task must be given some artifact which it can use to refer to its enclosing scopes. In the current design, this artifact is simply the address of an activation record for the innermost enclosing scope. With a design that incorporates a full implementation of global variables, this artifact would include additional information, such as a logical node address.

The NEW_TASK_STACK_SIZE component contains the number of bytes that are to be allocated for the execution stack of the dependent. The NEW_TASK_ENTRY_COUNT component is initialized with the number of entries that are declared for the dependent. The NEW_TASK_PRIORITY component is initialized with the priority of the dependent; this component is currently ignored. Finally, the NEW_TASK_MASTER component and NEW_TASK_MASTER_NODE component are set respectively to the name and node location of the master of the dependent; these will differ from MSG_SOURCE and MSG_SOURCE_NODE when the dependent is activated by a task that is not its master. Figure 20 depicts the use of the NEW_TASK_MSG in a scheduling transaction. The new dependent begins its activation once its agent receives an ELABORATE_MSG for the dependent.

4.5.3 Messages for Remote Exception Propagation

An exception may be propagated from one task to another in two cases—either during a rendezvous between two tasks, or when one task calls an entry of another task and the latter is abnormal, completed or terminated. In both cases the recipient of the propagated

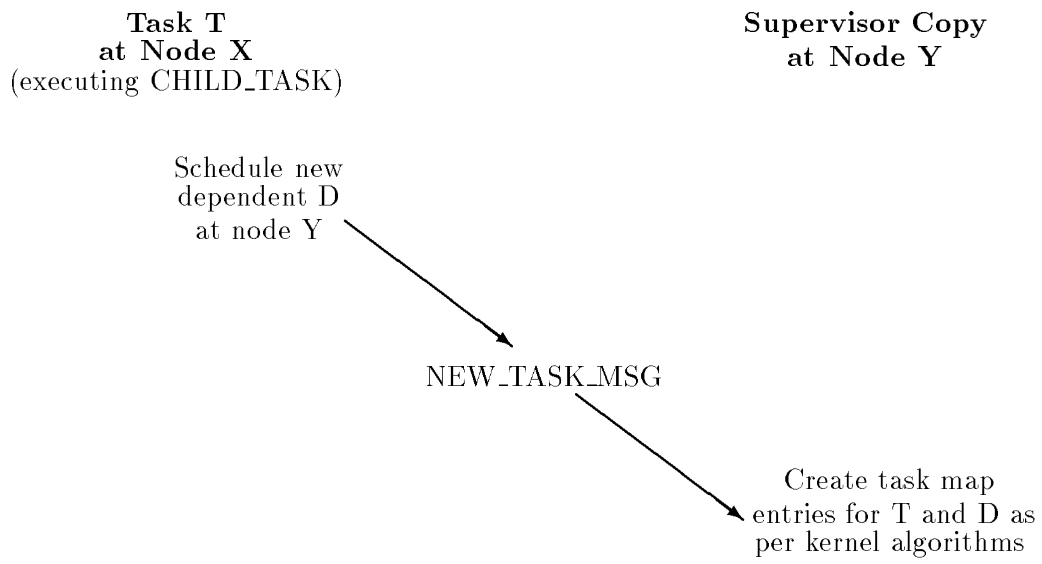
Task T
at Node X
(executing CHILD_TASK)
Supervisor Copy
at Node Y
Schedule new dependent D at node Y
NEW_TASK_MSG
Create task map entries for T and D as per kernel algorithms

Figure 20: Message Transaction for Scheduling a Newly Activated Dependent.

exception will be executing inside the Supervisor.

A single message is used by the Supervisor for remote propagation of an exception. Its CLASS value is EXCEPTION_MSG, and its associated message variant contains the single component EXCEPTION_NAME, a variable length string bounded by the value of the BYTE_COUNT discriminant. Whenever an exception is propagated during a rendezvous, an EXCEPTION_MSG is sent to the caller with the name of the exception placed in the EXCEPTION_NAME component. Whenever the execution status of a task becomes ABNORMAL, COMPLETED or TERMINATED, an EXCEPTION_MSG is sent to all tasks with calls on the entry queues of the first task and to all subsequent callers; in this case EXCEPTION_NAME is set to TASKING_ERROR. In all cases, the task which propagates the exception is the MSG_SOURCE and the recipient of the exception is the MSG_DEST. Figure 21 is a Weatherly-style diagrammatic depiction of remote exception propagation during a rendezvous.

4.5.4 Messages for Evaluation of Task Attributes

A task *T1* may evaluate the attributes *T2*'CALLABLE and *T2*'TERMINATED of any task *T2* visible inside the body of *T1*. Two new functions, CALLABLE_ATTR and TERMINATED_ATTR, have been added to the Supervisor for runtime evaluation of these attributes; they are described in Sections 4.6.5 and 4.6.6, respectively. Two messages are required

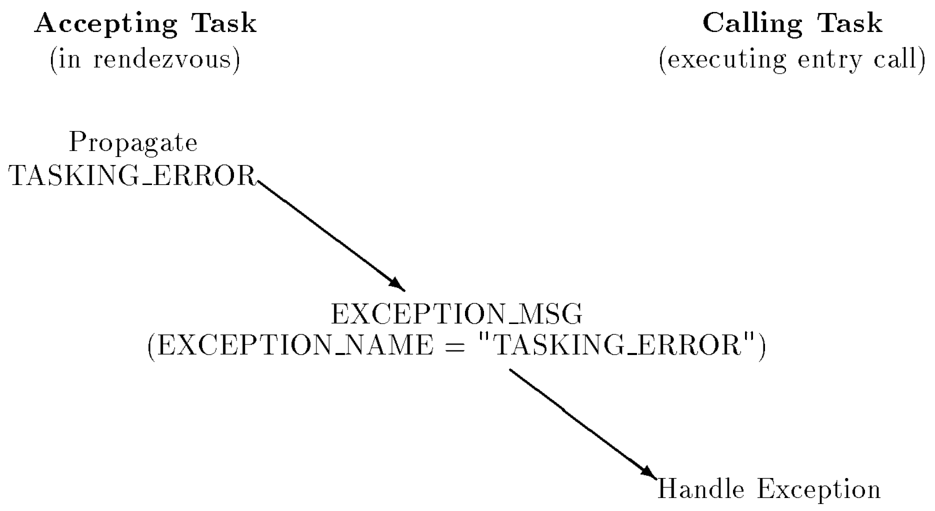

Figure 21: Message Transaction for Remote Propagation of TASKING_ERROR.

for distributed implementation of these attributes, a request and a reply. For the request message, the MSG_SOURCE component is set to *T1* and the MSG_DEST component is set to *T2*; for the reply message, the opposite settings are used.

The request message is indicated by a CLASS value of ATTR_REQ_MSG. Its variant has a single BOOLEAN component named CALLABLE_MSG. This component is set to TRUE if the CALLABLE attribute is requested evaluated or to FALSE if the TERMINATED attribute is requested.

The reply message is indicated by a CLASS value of ATTR_REPLY_MSG. The variant of this message has a single BOOLEAN component called ATTR_VALUE, which contains the result of evaluating the requested attribute. Figure 22 is a Weatherly-style diagrammatic depiction of the message transaction required for evaluation of the task attribute CALLABLE.

4.5.5 Messages for Remote Activation of Dependents

Consider the program fragment of Figure 23 which shows a task that is activated by execution of an allocator. The task *P* which executes the allocator is *not* the master of *T*.**all**; *T*.**all** is a direct dependent of the task that declared the corresponding access type definition, which in this case is *M*. Whenever a dependent of *M* is activated, the agent serving *M* must be informed of this fact so that other aspects of the Ada tasking semantics (such as ensuring satisfaction of task termination conditions) may be carried out correctly. Since

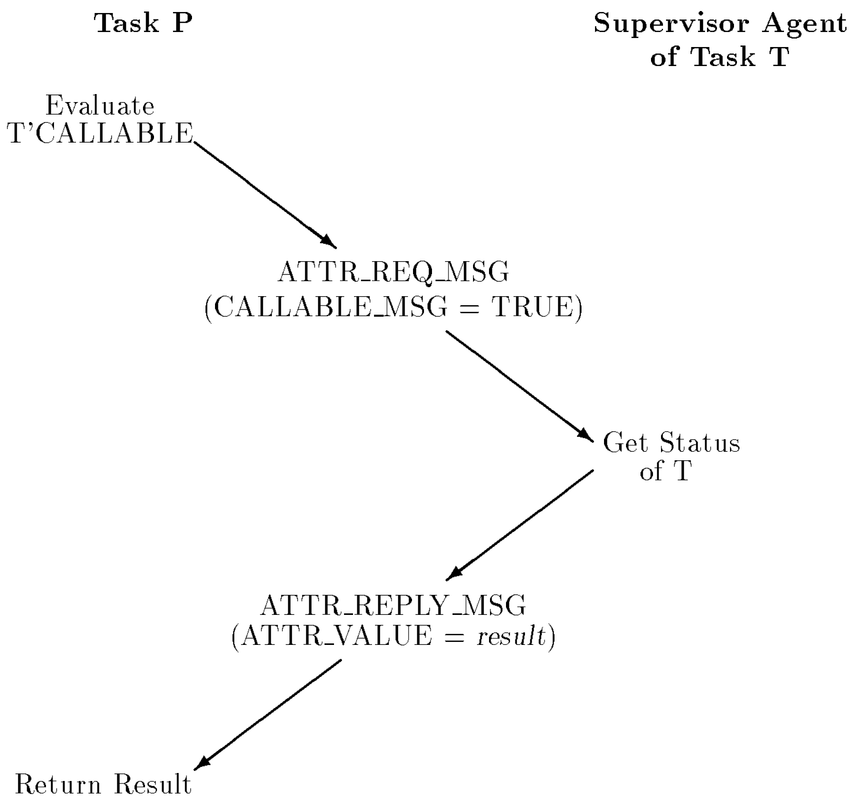
Task P
Supervisor Agent of Task T
Evaluate T'CALLABLE
ATTR_REQ_MSG (CALLABLE_MSG = TRUE)
Get Status of T
ATTR_REPLY_MSG (ATTR_VALUE = *result*)
Return Result

Figure 22: Message Transaction for Evaluation of Attribute CALLABLE.

M and P are different tasks in the example, it is necessary for P to send a message to M notifying M of the activation of a dependent; M must then send an acknowledgment to P.

In the first message, MSG_SOURCE is set to the name of the task which activated the dependent (e.g., task P in the above example), while the MSG_DEST is set to the name of the master of the activated dependent (e.g., task M in the above example). The CLASS value of this message is DEPENDENT_MSG, and its variant has three components. The DEPENDENT_NAME component is the name of the activated dependent; the DEPENDENT_NODE component is the node location to which the activated dependent was assigned. The MASTER_SCOPE component indicates the nesting level in the master task which contains the corresponding access type definition.

The reply message has a CLASS value of DEPENDENT_REPLY_MSG and no variant; the MSG_SOURCE and MSG_DEST are set opposite to the settings in the DEPENDENT_MSG. Figure 24 is a Weatherly-style diagrammatic depiction of the message transaction described above, using the tasks of Figure 23.

```
declare
    task M;
    task body M is
        task type T_TYPE;
        type T_REF is access T_TYPE;

        task body T_TYPE is ··· ;

        task P;
        task body P is
            T : T_REF
        begin
            T := new T_TYPE;
            -- T.all is a direct dependent of M, not P.
            ...
        end P;
    begin
        ...
    end M;
begin
    ...
end;
```

Figure 23: Remote Activation of Dependent Tasks.

4.5.6 Messages for the Terminate Alternative

The major defect of the Weatherly protocol is its lack of support for the **terminate** alternative. Consider the execution of a selective wait with a **terminate** alternative by some task T. Assume that either T has no dependents, or else that the dependents of T are either terminated or are themselves waiting at a **terminate** alternative. It is a consequence of the termination semantics defined in Section 9.4 of the Ada Language Reference Manual (LRM) that T must communicate with its master M in order to determine whether or not it is able to execute its **terminate** alternative. Three new messages (AT_TERM_MSG, TERM_REQ_MSG and TERM_CONFIRM_MSG) and two new execution status values (WAIT_FOR_ENTRY_TERMINATE and WAIT_FOR_TERMINATE_CONFIRMATION) have been added to the Weatherly protocol to implement the **terminate** alternative. Some of the terminology used for this implementation is taken from Jha and Kafura [JK85,Kaf85].

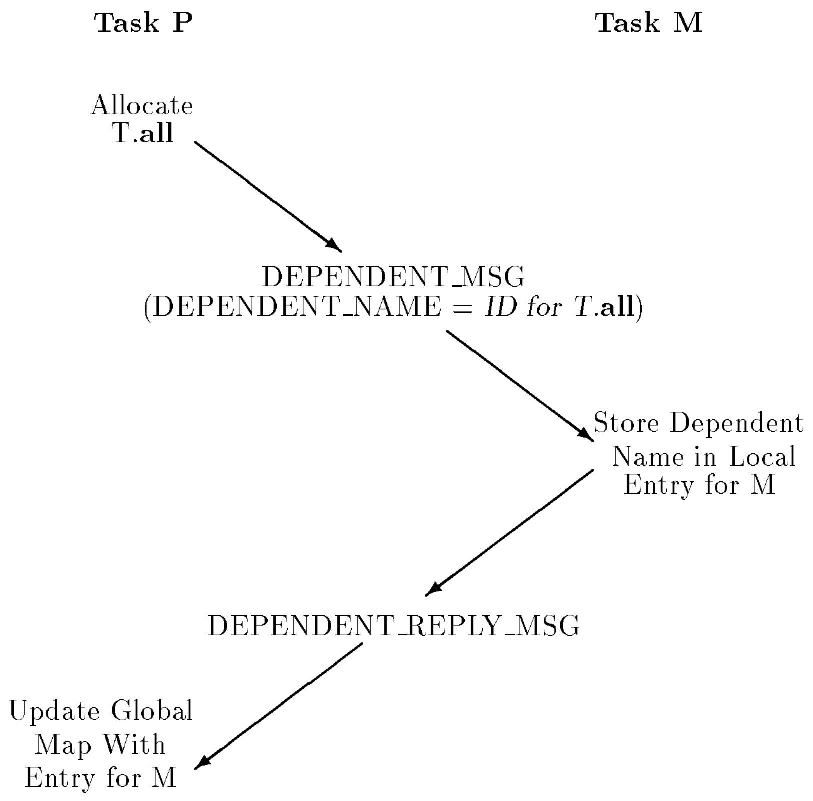
Task P
Task M
Allocate T.**all**
DEPENDENT_MSG
(DEPENDENT_NAME = *ID for T*.**all**)
Store Dependent Name in Local Entry for M
DEPENDENT_REPLY_MSG
Update Global Map With Entry for M

Figure 24: Message Transaction for Remote Activation of a Dependent.

The **terminate** alternative is complicated because of its nondeterministic nature, since it is possible for some task T waiting at a **terminate** alternative to be called by a task located "exterior" to the task dependency tree that is being terminated.

The functions of the new messages are as follows:

1. AT_TERM_MSG: This message informs T's master M that T is waiting at an open **terminate** alternative. The status of T changes to WAIT_FOR_ENTRY_TERMINATE after the message is sent.

2. TERM_REQ_MSG: This message requests an immediate indication from M as to whether or not all conditions have been satisfied for T to execute its **terminate** alternative. The status of T changes to WAIT_FOR_TERMINATE_CONFIRMATION after the message is sent.

3. TERM_CONFIRM_MSG: The reply to a TERM_REQ_MSG or AT_TERM_MSG. The BOOLEAN component CONFIRMED of the variant for this message indicates

whether or not permission to terminate has been granted by M.

The TERM_CONFIRM_MSG is sent in reply to an AT_TERM_MSG only when a termination decision is made by M based on some *eventual* change in the status of the dependency tree of which it is the root. On the other hand, the TERM_CONFIRM_MSG is *immediately* sent in reply to a TERM_REQ_MSG, because the TERM_REQ_MSG forces M to make a termination decision based on the *current* status of its dependency tree.

To enable a task to correctly determine when termination conditions have been satisfied, the execution status of the dependents of each such M is recorded in the DEPENDENTS component of the local map entry for M (see Figure 15). The following execution status values are defined for each dependent status record:

1. AWAITING_ACTIVATION: The dependent has not yet been activated, or else it is executing its declarative part.

2. ACTIVATED: The dependent is executing its body.

3. AWAITING_TERMINATION: The dependent is waiting at a **terminate** alternative, indicated by M's receipt of an AT_TERM_MSG from the dependent.

4. AWAITING_TERMINATION_CONFIRMATION: M has received from the dependent a TERM_REQ_MSG, indicating that the dependent received an entry call after reaching a **terminate** alternative. At this point only M has the power to allow or disallow the dependent to accept the entry call, since M may have already decided to carry out termination of its dependency tree.

5. PENDING_TERMINATION: The dependent has been told to execute its **terminate** alternative; equivalently, M has sent the dependent a TERM_CONFIRM_MSG with the CONFIRMED component set to TRUE.

6. TERMINATED: The dependent has terminated.

The need for both an AWAITING_TERMINATION dependent status value and an AWAITING_TERMINATION_CONFIRMATION status value is related to the issuance of a termination decision by M with a TERM_CONFIRM_MSG. If the decision is to *confirm* termination to dependents, both AWAITING_TERMINATION and AWAITING_TERMINATION_CONFIRMATION dependents must be notified of this decision. On the other

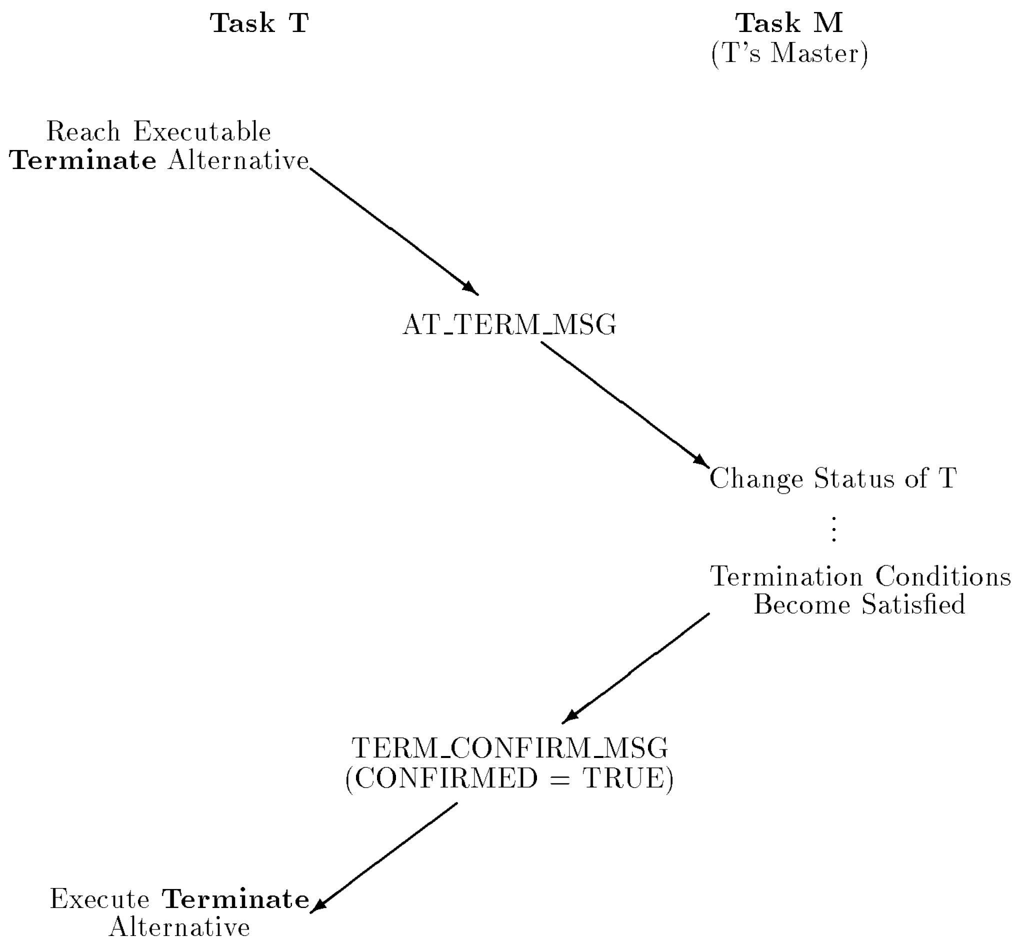
Task T
Task M
(T's Master)
Reach Executable **Terminate** Alternative
AT_TERM_MSG
Change Status of T
Termination Conditions Become Satisfied
TERM_CONFIRM_MSG (CONFIRMED = TRUE)
Execute **Terminate** Alternative

Figure 25: Message Transaction for Execution of a **Terminate** Alternative.

hand, if the decision is to *deny* termination to dependents, only AWAITING_TERMINATION_CONFIRMATION dependents must be notified of this decision, since AWAITING_TERMINATION dependents are still passively suspended waiting at a **terminate** alternative.

In every case, the MSG_SOURCE and MSG_DEST component settings correspond in an obvious way to the two communicants of each message transaction. The complete algorithm for distributed execution of the **terminate** alternative has been given in detail in Appendix D. Figures 25, 26 and 27 are Weatherly-style diagrammatic depictions of the possible message transactions; each figure depicts the message transactions between a task T waiting at a **terminate** alternative and T's master M. Figure 25 depicts the case of a **terminate** alternative that is executed after termination conditions become satisfied and

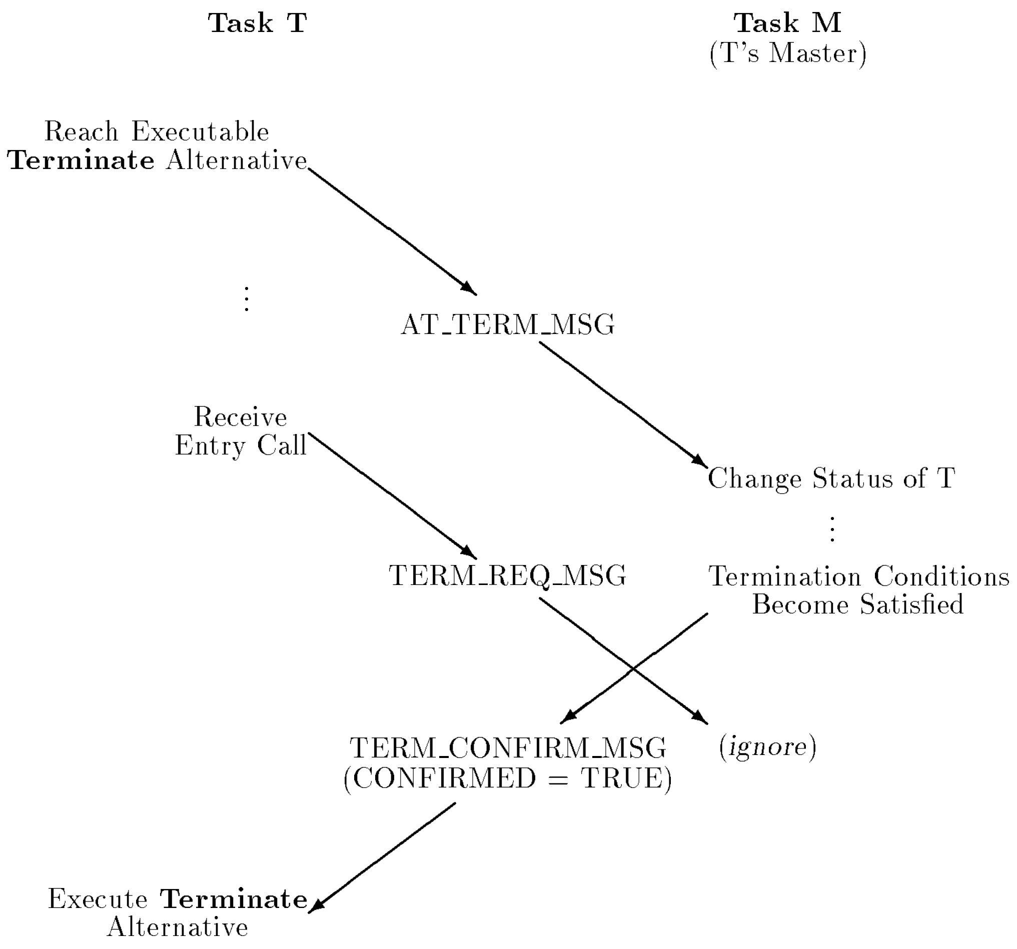

Figure 26: Message Transaction for Execution of a **Terminate** Alternative After Arrival of an Entry Call.

before any entry calls arrive at T. Figure 26 depicts the case of a **terminate** alternative whose execution results from a forced confirmation after arrival of an entry call. Finally, Figure 27 depicts the case of a **terminate** alternative that is abandoned in favor of accepting an entry call. Note that in the last two of these figures it is assumed that M is able to decide on its own whether or to confirm a request to execute a **terminate** alternative; however, it may be necessary in the general case for M to propagate the TERM_REQ_MSG to *its* master before making a termination decision.

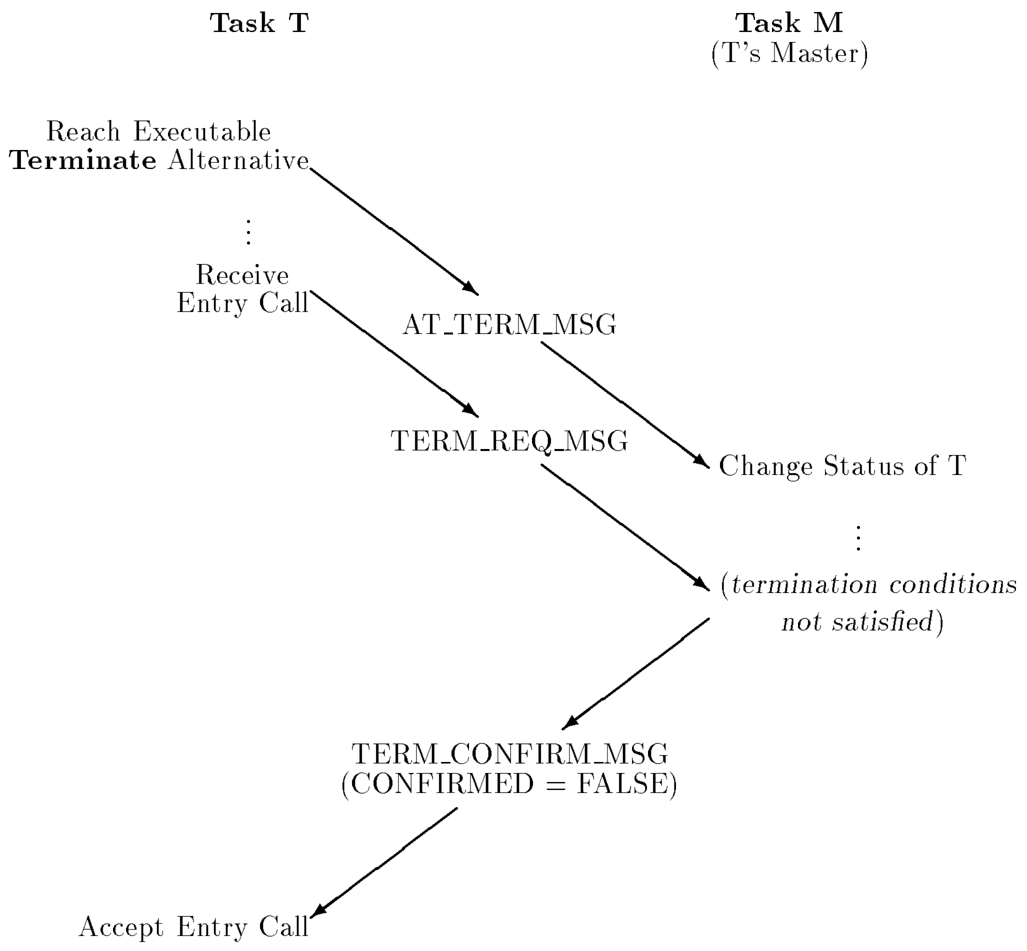

Figure 27: Message Transaction for Abandonment of a **Terminate** Alternative.

4.5.7 Messages for the Abort Statement

In the Weatherly protocol an ABORT_TASK_MSG is sent to tasks that are aborted by the execution of an **abort** statement. However, the semantics of Section 9.10 of the Ada LRM requires the tasks named in the **abort** statement to become abnormal *before* completion of the **abort** statement. Thus, to ensure correct implementation of these semantics, a task executing an **abort** statement must await a reply from *all* aborted tasks before continuing execution. Requiring the task to await these replies prevents race conditions which can lead to erroneous execution; for example, this improved protocol ensures that in the following sequence of statements

```
abort A;
A.E;        -- Call to an aborted task.
```

TASKING_ERROR will *always* be propagated during the entry call.

The reply message has a CLASS value of ABORT_REPLY_MSG. Each time a task T executes an **abort** statement, the following algorithm is executed by the Supervisor:

1. T's agent sends an ABORT_TASK_MSG to each task A named in the **abort** statement. The MSG_SOURCE is T and the MSG_DEST is A.

2. For each aborted task A the agent serving A checks to see if A has dependents; if so, A's agent sends an ABORT_TASK_MSG to each dependent D—MSG_SOURCE is A and MSG_DEST is D—and an ABORT_REPLY_MSG is awaited from each such D. This step is performed recursively for each level of aborted dependents.

3. For each aborted task A that has received an ABORT_REPLY_MSG from all of its dependents, A's agent sends an ABORT_REPLY_MSG to task T, with MSG_SOURCE set to A and MSG_DEST set to T.

4. An ABORT_REPLY_MSG must be received by T's agent from each aborted task A before T is allowed to continue execution.

Thus, the agents of all aborted tasks are able to correctly register the abnormality of these tasks before task T is allowed to continue execution. The ABORTED_TASKS component of the local map entry for T (see Figure 15) is used to keep track of the names of all tasks aborted by T. Each time a reply is received from one of these tasks, its name is deleted from ABORTED_TASKS; once all names have been deleted from ABORTED_TASKS, the execution of the **abort** statement is considered complete. In the case when T aborts itself, the above algorithm is applied to the dependents of T as well as to all other aborted tasks.

The execution status changes of the aborted tasks also require a more rigorous algorithm than that described by Weatherly, which simply makes all aborted tasks ABNORMAL. The new execution status of each aborted task A is determined directly according to the Ada semantics as follows:

1. If the execution status of A is WAIT_FOR_ENTRY, WAIT_FOR_TERMINATE_CONFIRMATION, WAIT_FOR_ENTRY_TERMINATE, DELAYED or WAIT_FOR_ACCEPT, then the status of A is changed to COMPLETED.

2. If the execution status of A is UNBORN or RUN_DCL then

    (a) If A is executing in its outermost scope, its status is changed to TERMINATED.

    (b) If A is *not* executing in its outermost scope, its status is changed to COMPLETED.

3. The status of a task aborting itself is changed to COMPLETED at the end of the **abort** statement.

4. In all other cases, the status of A is changed to ABNORMAL.

Figure 28 is a Weatherly-style diagrammatic depiction of the message transactions described in this section.

4.6 The Supervisor Interface

The interface to the Distributed Ada Supervisor is the visible part of the TASKING_SUPERVISOR package. Each subprogram in the package is responsible for executing a different tasking operation. An Ada compiler translates the source-level tasking constructs of application programs into calls to appropriate subprograms of this package.

A crucial component of the interface design is its documentation, since the compiler writer must understand the specification of the interface in order to implement correct compilation of source level tasking statements. Thus, a formal specification of the package interface was written in Anna. The Anna specifications are useful not only as formal documentation, but they can also be used to automatically check the correct use of the interface during execution of application programs. In this form, the formal package specification can serve as a standardized interface to the tasking supervisor of an Ada runtime system.

Figure 29 is an outline of the Supervisor interface. The Anna specifications describe constraints on the parameters of each Supervisor subprogram; the TSL specifications that are described in Chapter 6 serve as as the axiomatic annotation of the interface. The first section below describes the Supervisor data types in detail. The second section describes the "vocabulary" of the Anna specifications; the vocabulary includes the two virtual functions defined in this section. The remaining sections present each Supervisor subprogram, informally in text and formally in Anna. Type names in the subprogram declarations are

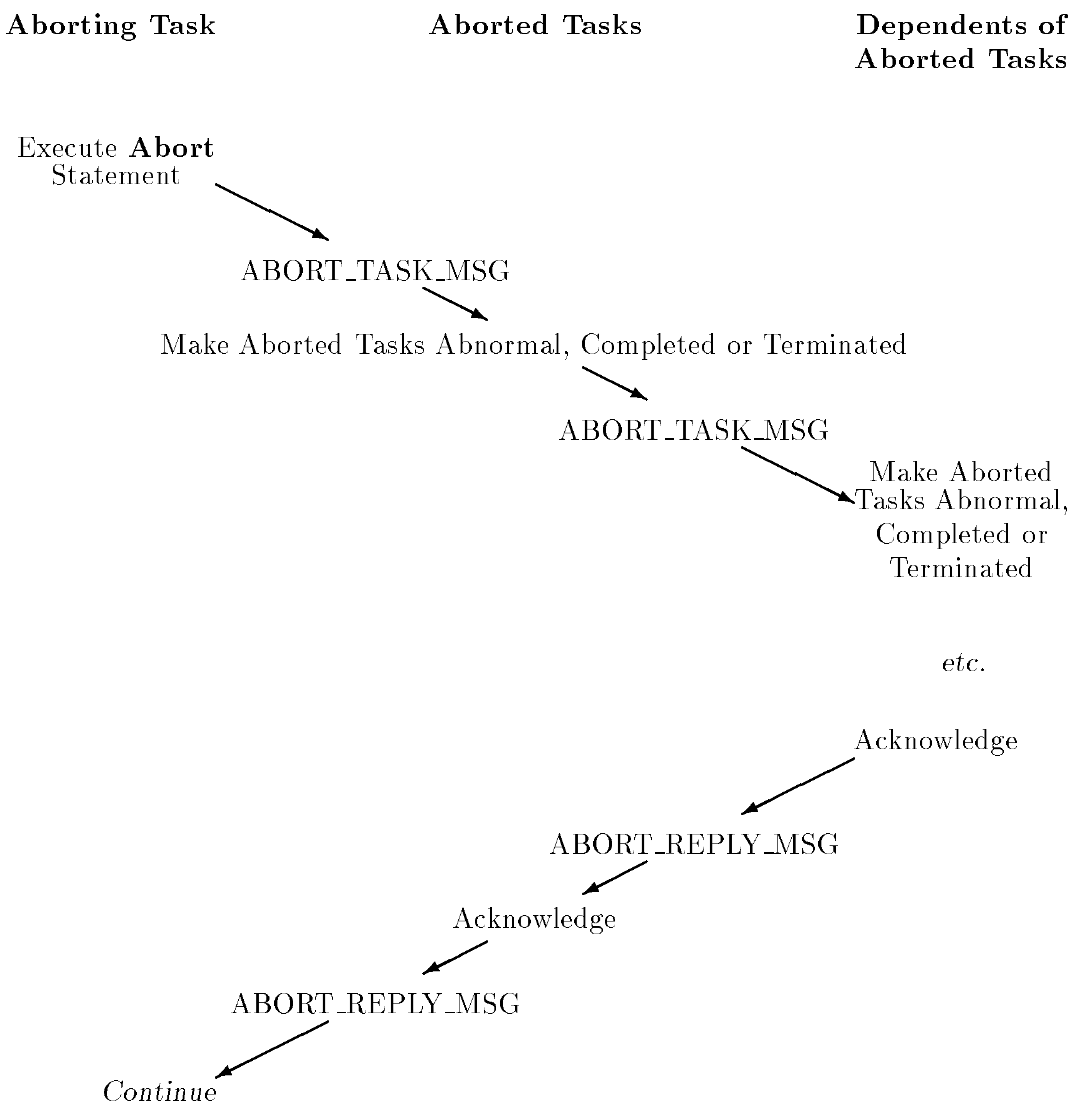

Figure 28: Message Transaction for Execution of the **Abort** Statement.

```
package TASKING_SUPERVISOR is

        -- Described in Section 4.6.1:
        Declarations of Supervisor Data Types;
    --| Specification of Supervisor Data Types;

        -- Described in Section 4.6.2:
    --: Declarations of Virtual Functions—CURRENT_SCOPE, ENTRY_COUNT;

        -- Described in Sections 4.6.3 through 4.6.16:
        Declarations of Supervisor Subprograms—ENTER_NEW_SCOPE,
                    CALLABLE_ATTR, TERMINATED_ATTR, COUNT_ATTR,
                    ENTRY_CALL, ACCEPT_BEGIN, ACCEPT_END, CHILD_TASK,
                    ACTIVATE_TASK, ELABORATE_TASK, TERMINATE_TASK,
                    ABORT_TASKS, DELAY_TASK;
    --| Specification of Supervisor Subprograms;

end TASKING_SUPERVISOR;
```

Figure 29: Outline of the Supervisor Interface.

in most cases self-explanatory and are described in the text when necessary. The subprogram declarations are based on those of Weatherly [Wea84b], with major enhancements and additions as described.

4.6.1 The Supervisor Data Types

The Supervisor uses the set of data types shown in Figure 30. Most of these types are adaptations and enhancements of the types described by Weatherly. Although some of the types are actually declared in the MESSAGES and NET_SERVICES packages, they are listed here for clarity.

NODE_ID_TYPE provides a logical numbering of the CPUs which are available for program execution. Each Supervisor subprogram has a parameter of type NODE_ID_TYPE called MY_NODE, which is set to the logical CPU address of the task calling the subprogram. This parameter is used to make the means of accessing the Supervisor data structures more efficient and portable.

```
      -- Type for logical CPU addresses:
      type NODE_ID_TYPE is new NATURAL range 0 .. Available_CPUs - 1;
      -- Types and constants for task and entry identifiers:
      subtype TASK_NAME is NATURAL;
      subtype ENTRY_NAME is NATURAL;

      MAIN_ID     : constant TASK_NAME := TASK_NAME'FIRST;
      MAIN_NODE   : NODE_ID_TYPE := NODE_ID_TYPE'FIRST;

      -- Type for numbering nested declarative regions:
      subtype SCOPE_NUMBER_TYPE is NATURAL;


      -- Supervisor array types:
      type GUARD_ARRAY is array (POSITIVE range <>) of BOOLEAN;
--|   where GA : GUARD_ARRAY => GA'FIRST = 1;

      type INDEX_ARRAY is array (POSITIVE range <>) of INTEGER;
--|   where IA : INDEX_ARRAY => IA'FIRST = 1;

      type TASK_NAME_ARRAY is array (POSITIVE range <>) of TASK_NAME;
--|   where TA : TASK_NAME_ARRAY =>
--    Component values of a TASK_NAME_ARRAY are unique:
--|           TA'FIRST = 1 and
--|           (for all I1, I2 : TA'RANGE => (I1 /= I2 -> TA (I1) /= TA (I2)));

      -- Array type for in-mode parameter value lists:
      type PARAM_LIST is array_of_bytes;

      -- Access type for out-mode parameter value lists:
      type PARAM_LIST_REF is access PARAM_LIST;
```

Figure 30: Supervisor Data Types.

The SCOPE_NUMBER_TYPE is used for numbering nested declarative regions of a task; its use is described in the next section. The TASK_NAME and ENTRY_NAME types are used for assigning unique integer identifiers to tasks and to the entries within a task, respectively. By convention, the task identifier 0 (the value of MAIN_ID in Figure 30) is reserved for the thread of control which activates the main program; this thread of control is the "environment task" referred to in the Ada LRM:

> ...Each main program acts as if called by some environment task; the means by which this execution is initiated are not prescribed by the language definition. ... [Ada83, § 10.1, ¶ 8]

It is convenient to think of this task as the Ada package STANDARD; Clemmensen refers to it as the *SYSTEM* meta-process. In numbering the entries of a task, entry families are "flattened out" to obtain a simple linear numbering scheme.

A GUARD_ARRAY holds the set of guard values for a selective wait, which is executed by the procedure ACCEPT_BEGIN (see Section 4.6.9). The INDEX_ARRAY type, also used for selective waits, assigns an integer value to the additional sequence of statements associated with each selective wait alternative. The ACCEPT_BEGIN procedure returns the index value associated with the alternative that was selected.

A TASK_NAME_ARRAY holds the identifiers of tasks named in an **abort** statement. Parameter values for rendezvous are passed to subprograms and in messages as variable-length arrays of bytes called PARAM_LISTs.

4.6.2 The Vocabulary of the Anna Specifications

Figure 31 presents the virtual functions that are used to specify the Supervisor interface. These virtual functions are part of the vocabulary that is used to formally describe the Supervisor interface in Anna. The vocabulary is further comprised of three "actual" subprograms—the attribute evaluation functions CALLABLE_ATTR and TERMINATED_ATTR of the TASKING_SUPERVISOR package (described below), and the CLOCK function of the predefined Ada package CALENDAR. This vocabulary, along with the predefined Ada expression operators, is sufficient for expressing all of the Supervisor interface constraints written in Anna.

The virtual function CURRENT_SCOPE provides the current scope level value of the task named in the parameter; the value of this function changes after execution of scope

```
    -- Function returning the current scope level of TASK_NAME:
--: function CURRENT_SCOPE (T : in TASK_NAME) return SCOPE_NUMBER_TYPE;

    -- Function returning the number of entries declared for TASK_NAME:
--: function ENTRY_COUNT (T : in TASK_NAME) return NATURAL;
```

Figure 31: The Supervisor Virtual Functions.

```
    function ENTER_NEW_SCOPE (MY_NAME : in TASK_NAME;
                              MY_NODE : in NODE_ID_TYPE)
                             return SCOPE_NUMBER_TYPE;
--| where
--  The current scope value of MY_NAME is incremented:
--|   return in CURRENT_SCOPE (MY_NAME) + 1,
--|   out (CURRENT_SCOPE (MY_NAME) =
--|            in CURRENT_SCOPE (MY_NAME) + 1);
```

Figure 32: The Supervisor ENTER_NEW_SCOPE Procedure.

control subprograms (see the next section). The virtual function ENTRY_COUNT returns the total number of entries declared by the input task; the Supervisor first receives this value during a call to the Supervisor function CHILD_TASK, described below in Section 4.6.11.

4.6.3 The Scope Control Subprograms

One of the deficiencies of Weatherly's work is its neglect of the semantics of nested scope execution. Correct implementation of the task dependency and termination semantics of Ada requires that the Supervisor keep track of when a task enters and leaves nested declarative regions. These actions are communicated to the Supervisor respectively by the ENTER_NEW_SCOPE and TERMINATE_TASK procedures. The first of these is declared in Figure 32. The latter procedure is used both to exit inner scopes and terminate the outer scope of each task; it is described more fully in Section 4.6.14. ENTER_NEW_SCOPE is called by a task immediately upon entering a nested declarative region; it returns the new nesting level value of the calling task, whose name is passed as the parameter. This value is

```
    function CALLABLE_ATTR (MY_NAME : in     TASK_NAME;
                            C_NAME  : in     TASK_NAME;
                            MY_NODE : in out NODE_ID_TYPE)
                                      return BOOLEAN;
--| where
-- MY_NAME's scope is unchanged:
--|   CURRENT_SCOPE (MY_NAME) = in CURRENT_SCOPE (MY_NAME);
```

Figure 33: The Supervisor CALLABLE_ATTR Function.

then used when dependents are declared to the Supervisor during a call to CHILD_TASK (see Section 4.6.11). Scope number zero is associated with the outermost scope of each task; each level of nesting adds one to the scope number.

4.6.4 Synchronization Points

A Boolean-valued **out**-mode ABORTED parameter is provided for those subprograms which implement synchronization points named in the Ada LRM:

> The completion of any other abnormal task need not happen before completion of the abort statement. It must happen no later than when the abnormal task reaches a synchronization point that is one of the following: the end of its activation; a point where it causes the activation of another task; an entry call; the start or the end of an accept statement; a select statement; a delay statement; an exception handler; or an abort statement. If a task that calls an entry becomes abnormal while in a rendezvous, its termination does not take place before the completion of the rendezvous (see 11.5). [Ada83, § 9.10, ¶ 6]

Upon termination of such a subprogram, ABORTED will be TRUE if the task MY_NAME is abnormal. The compiled application program then uses this value to force completion of the abnormal task.

4.6.5 The CALLABLE_ATTR Function

A task calls the CALLABLE_ATTR function, shown in Figure 33, to evaluate the CALLABLE attribute of a task. For task *T* executing *C*'CALLABLE, MY_NAME is set to the

```
    function TERMINATED_ATTR (MY_NAME : in      TASK_NAME;
                              T_NAME  : in      TASK_NAME;
                              MY_NODE : in out  NODE_ID_TYPE)
                                        return BOOLEAN;
--| where
--  MY_NAME's scope is unchanged:
--|   CURRENT_SCOPE (MY_NAME) = in CURRENT_SCOPE (MY_NAME),
--  If MY_NAME evaluates TERMINATED on itself, then it must not
--  be terminated:
--|   return B1 : BOOLEAN => MY_NAME = T_NAME -> not B1;
```

Figure 34: The Supervisor TERMINATED_ATTR Function.

identifier for *T*, and C_NAME is set to the identifier for *C*; the function returns the result of the evaluation. The CALLABLE_ATTR function initiates the the attribute message sequence described in Section 4.5.4.

4.6.6 The TERMINATED_ATTR Function

A task calls the TERMINATED_ATTR function, shown in Figure 34, to evaluate the TERMINATED attribute of a task; except for the difference in attribute that is evaluated, it behaves in exactly the same way as CALLABLE_ATTR.

4.6.7 The COUNT_ATTR Function

A task calls the COUNT_ATTR function, shown in Figure 35, to evaluate the COUNT attribute of one of its entries. The name of the calling task is passed in MY_NAME and the name of the entry is passed in MY_ENTRY; the function returns the result of the evaluation.

4.6.8 The ENTRY_CALL Procedure

Simple entry calls, conditional entry calls and timed entry calls are compiled as calls to the ENTRY_CALL procedure whose declaration is shown in Figure 36. The TIME_OUT parameter is used to differentiate between the three kinds of call. For a simple call, TIME_OUT is set to DURATION'LAST (the largest positive value of the predefined type DURATION), which the Supervisor uses by convention as an infinite time value. For a conditional call,

```
    function COUNT_ATTR (MY_NAME     : in      TASK_NAME;
                         MY_ENTRY    : in      ENTRY_NAME;
                         MY_NODE     : in out  NODE_ID_TYPE)
                                     return BOOLEAN;
--| where
--  MY_NAME's scope is unchanged:
--|   CURRENT_SCOPE (MY_NAME) = in CURRENT_SCOPE (MY_NAME),
--  MY_ENTRY must be in the range of entries declared by MY_NAME:
--|   1 <= MY_ENTRY <= ENTRY_COUNT (MY_NAME);
```

Figure 35: The Supervisor COUNT_ATTR Function.

TIME_OUT is set to zero. For a timed call, TIME_OUT is set to the value specified in the **delay** statement associated with the call.

MY_NAME is set to the name of the calling task, CALLEE_T_NAME is set to the name of the called task, and CALLEE_E_NAME is set to the name of the entry being called. **In**-mode parameters to the entry call are passed in IN_PARAMS, while **out**-mode parameters are returned in OUT_PARAMS. For conditional and timed entry calls, ACCEPTED indicates whether or not a rendezvous actually took place; for a simple call, ACCEPTED must be TRUE upon normal termination of the procedure.

Since an entry call and rendezvous are named in Section 9.10 of the Ada LRM as synchronization points at which an abnormal task must become completed, the Supervisor communicates the fact of abnormality to an abnormal caller through the ABNORMAL parameter so that the caller may complete its execution after ENTRY_CALL terminates.

ENTRY_CALL initiates the rendezvous message sequence of the Weatherly protocol and returns when either the call has been accepted, TASKING_ERROR has been propagated to the caller, or the specified TIME_OUT has elapsed. Characteristics of ENTRY_CALL which are a consequence of the semantics of Ada are formally specified in Anna in Figure 36.

4.6.9 The ACCEPT_BEGIN Procedure

Selective waits and simple **accept** statements are compiled into either one or two procedure calls. The first procedure is ACCEPT_BEGIN, whose declaration is shown in Figure 37. The second procedure, which is called if an **accept** alternative is selected, is ACCEPT_END,

```
procedure ENTRY_CALL (MY_NAME         : in     TASK_NAME;
                      CALLEE_T_NAME   : in     TASK_NAME;
                      CALLEE_E_NAME   : in     ENTRY_NAME;
                      IN_PARAMS       : in     PARAM_LIST;
                      TIME_OUT        : in     DURATION;
                      OUT_PARAMS      : out    PARAM_LIST_REF;
                      ACCEPTED        : out    BOOLEAN;
                      ABORTED         : out    BOOLEAN;
                      MY_NODE         : in out NODE_ID_TYPE);
--| where
--  MY_NAME's scope is unchanged:
--|   CURRENT_SCOPE (MY_NAME) = in CURRENT_SCOPE (MY_NAME),
--  A task cannot rendezvous with itself:
--|   out (ACCEPTED -> MY_NAME /= CALLEE_T_NAME),
--  For a simple entry call by a caller that is not abnormal, ENTRY_CALL
--  will not terminate until the call is accepted:
--|   out (TIME_OUT = DURATION'LAST and not ABORTED ->
--|            ACCEPTED),
--  The called entry must have been declared by CALLEE_T_NAME:
--|   1 <= CALLEE_E_NAME <= ENTRY_COUNT (CALLEE_T_NAME),
--  An unaccepted timed entry call by a caller that is not abnormal
--  lasts at least as long as TIME_OUT:
--|   out (not ABORTED and not ACCEPTED ->
--|          CALENDAR.CLOCK - in CALENDAR.CLOCK >= TIME_OUT),
--  An aborted task is no longer CALLABLE; i.e., any evaluation of the CALLABLE
--  attribute on MY_NAME should return FALSE if ABORTED is TRUE:
--|   out (ABORTED ->
--|          (for all T1 : TASK_NAME; N1 : NODE_ID_TYPE =>
--|             not CALLABLE_ATTR (T1, MY_NAME, N1))),
--  Execution of an entry call can cause TASKING_ERROR to be
--  propagated to the caller:
--|   raise TASKING_ERROR;
```

Figure 36: The Supervisor ENTRY_CALL Procedure.

```
      procedure ACCEPT_BEGIN (MY_NAME   : in     TASK_NAME;
                              GUARDS    : in     GUARD_ARRAY;
                              INDICES   : in     INDEX_ARRAY;
                              TIME_OUT  : in     DURATION;
                              TERM_ALT  : in     BOOLEAN;
                              IN_PARAMS : out    PARAM_LIST_REF;
                              INDEX     : out    NATURAL;
                              ABORTED   : out    BOOLEAN;
                              MY_NODE   : in out NODE_ID_TYPE);
--| where
--  MY_NAME's scope is unchanged:
--|   CURRENT_SCOPE (MY_NAME) = in CURRENT_SCOPE (MY_NAME),
--  GUARDS and INDICES have one component per entry of MY_NAME
--  plus an extra one:
--|   GUARDS'LAST = INDICES'LAST = ENTRY_COUNT (MY_NAME) + 1,
--  Execution of the delay alternative of a selective wait must occur
--  after TIME_OUT has elapsed:
--|   out (not ABORTED and not TERM_ALT and
--|        in TIME_OUT < DURATION'LAST and
--|        GUARDS (GUARDS'LAST) and
--|        INDEX = INDICES (INDICES'LAST) ->
--|          CALENDAR.CLOCK - in CALENDAR.CLOCK >= TIME_OUT),
--  Execution of a selective wait with all alternatives closed results in the
--  propagation of PROGRAM_ERROR to the accepting task:
--|   (for all I1 : GUARDS'RANGE => not GUARDS (I1)) =>
--|           raise PROGRAM_ERROR,
--  The value returned in INDEX must be a component of INDICES:
--|   out (not ABORTED ->
--|            exist I1 : INDICES'RANGE => INDEX = INDICES (I1)),
--  An aborted task is no longer CALLABLE:
--|   out (ABORTED ->
--|          (for all T1 : TASK_NAME; N1 : NODE_ID_TYPE =>
--|             not CALLABLE_ATTR (T1, MY_NAME, N1)));
```

Figure 37: The Supervisor ACCEPT_BEGIN Procedure.

described in the next section.

MY_NAME is set to the name of the task executing the simple **accept** or selective wait. The possible forms of **accept** statement or selective wait are differentiated by the values of the parameters TERM_ALT, TIME_OUT and GUARDS. GUARDS and INDICES are arrays of $N + 1$ components—the first N components for the N entries of MY_NAME, and the $N + 1th$ component for an **else** part, **delay** alternative or **terminate** alternative. TERM_ALT is set to TRUE only when executing a selective wait with a **terminate** alternative.

A simple **accept** statement for entry number K will have GUARDS (K) set to TRUE and all other GUARDS set to FALSE. If an entry is named as an **accept** alternative in a selective wait, the GUARDS component for the entry is set to the value of the guard on the alternative, or to TRUE if the alternative is unguarded. If an entry is *not* named in a selective wait, its GUARDS component is set to FALSE. If an entry is named in multiple **accept** alternatives of a selective wait, a single open alternative for the entry (if one exists) is chosen at runtime in an unspecified manner.

For a selective wait with an **else** part, GUARDS ($N+1$) is set to TRUE and TIME_OUT is set to zero. For a selective wait with a **terminate** alternative, TERM_ALT is set to TRUE, TIME_OUT is set to DURATION'LAST ("infinity") and GUARDS ($N + 1$) is set to the value of the guard on the **terminate** alternative (or TRUE if there is no guard). For a selective wait with one or more **delay** alternatives, a single open alternative with the smallest delay value (if one exists) is chosen at runtime in an unspecified manner; TIME_OUT is set to the chosen delay value. For a selective wait that contains only **accept** alternatives, GUARDS ($N + 1$) is set to FALSE.

The function of INDICES was explained in Section 4.6.1; the component of INDICES corresponding to the chosen alternative is returned in INDEX. The values of **in**-mode parameters to a selected **accept** are available in IN_PARAMS upon termination of the procedure. Since an **accept** statement and selective wait are both synchronization points for completion of an abnormal task, an ABNORMAL parameter is provided.

ACCEPT_BEGIN performs the beginning of the rendezvous message sequence of the Weatherly protocol. Characteristics of ACCEPT_BEGIN which are a consequence of the semantics of Ada are formally specified in Anna in Figure 37.

```
    procedure ACCEPT_END (MY_NAME      : in     TASK_NAME;
                          OUT_PARAMS   : in     PARAM_LIST_LIST;
                          ABORTED      : out    BOOLEAN;
                          MY_NODE      : in out NODE_ID_TYPE);
--| where
--  MY_NAME's scope is unchanged:
--|   CURRENT_SCOPE (MY_NAME) = in CURRENT_SCOPE (MY_NAME),
--  An aborted task is no longer CALLABLE:
--|   out (ABORTED ->
--|          (for all T1 : TASK_NAME; N1 : NODE_ID_TYPE =>
--|             not CALLABLE_ATTR (T1, MY_NAME, N1)));
```

Figure 38: The Supervisor ACCEPT_END Procedure.

4.6.10 The ACCEPT_END Procedure

The ACCEPT_END procedure, shown in Figure 38, is called at the end of the sequence of statements associated with every **accept** statement. MY_NAME is set to the name of the task executing the **accept** statement. The values of **out**-mode parameters computed during the rendezvous are passed in OUT_PARAMS. Again, an ABORTED parameter is provided since an **accept** statement is a synchronization point for completion of an abnormal task.

ACCEPT_END completes the rendezvous message sequence of the Weatherly protocol.

4.6.11 The CHILD_TASK Function

Declarations of tasks are compiled into calls to the CHILD_TASK function. It is called each place a single task is declared, each place a task is declared by an object declaration, and each place a task is activated by execution of an allocator. The declaration of this function is shown in Figure 39. Since the task which activates a new dependent may not be the master of the dependent (see Section 4.5.5), the names of both the activator (MY_NAME) and the master (MASTER_NAME) are passed to CHILD_TASK. CHILD_TASK generates a task identifier for the dependent and passes the generated value back to the caller (MY_NAME) as the return value. The number of entries declared for the dependent is passed as ENTRY_COUNT. MASTER_SCOPE indicates the nested scope level of the master on which the dependent directly depends.

```
function CHILD_TASK (MY_NAME            : in TASK_NAME;
                     MASTER_NAME        : in TASK_NAME;
                     ENTRY_COUNT        : in ENTRY_NAME;
                     MASTER_SCOPE       : in SCOPE_NUMBER_TYPE;
                     DEP_ENTRY_ADDR     : in SYSTEM.ADDRESS;
                     GLOBAL_FRAME_PTR   : in SYSTEM.ADDRESS;
                     MY_NODE            : in NODE_ID_TYPE;
                     DEP_STACK_SIZE     : in POSITIVE := 16384;
                     IMMEDIATE          : in BOOLEAN := FALSE;
                     PRE_BODY           : in BOOLEAN := TRUE;
                     DEP_PRIORITY       : in SYSTEM.PRIORITY :=
                                 SYSTEM.PRIORITY'FIRST)
        return TASK_NAME;
--| where
--  MY_NAME's scope is unchanged:
--|   CURRENT_SCOPE (MY_NAME) = in CURRENT_SCOPE (MY_NAME),
--  Definition of virtual function ENTRY_COUNT:
--|   return N : TASK_NAME =>
--|           ENTRY_COUNT (N) = ENTRY_COUNT and
--  The initial scope level of the new dependent is 0:
--|           CURRENT_SCOPE (N) = 0 and
--  A task cannot activate itself:
--|           MY_NAME /= N and MASTER /= N,
--  Dynamic activation of a task prior to elaboration of its body causes
--  PROGRAM_ERROR to be raised:
--|   PRE_BODY -> IMMEDIATE,
--|   IMMEDIATE and PRE_BODY => raise PROGRAM_ERROR;
```

Figure 39: The Supervisor CHILD_TASK Function.

Parameters are also passed that are used by CHILD_TASK to start up the new task as a separate thread of control. The execution entry point of the new dependent is passed in DEP_ENTRY_ADDR. GLOBAL_FRAME_PTR is the address of the activation record for the declarative part which contains the declaration of the body of the new dependent. The number of bytes to be allocated for the execution stack of the new dependent is passed in DEP_STACK_SIZE. The priority of the task is passed in DEP_PRIORITY; the Supervisor currently ignores this information. If the task is a single task or a task declared by an object declaration, then the IMMEDIATE parameter is set to FALSE. Otherwise, the IMMEDIATE parameter is set to TRUE, indicating that the task is to begin activation immediately; in this case CHILD_TASK subsumes the functionality of the ACTIVATE_TASK procedure described below. In addition, the PRE_BODY parameter indicates whether or not the task is being activated before its body has been elaborated; this can occur only in the situation when a task is activated through execution of an allocator within the same declarative part that contains the declaration of its body.

CHILD_TASK updates the task maps as described in Section 4.4 and assigns the new dependent to a system node, sending a NEW_TASK_MSG to a remote supervisor copy if necessary (see Section 4.5.2). The task activation semantics of Ada are enforced by the ACTIVATE_TASK procedure and ELABORATE_TASK function, described in the next two sections. Characteristics of CHILD_TASK which are a consequence of the semantics of Ada are formally specified in Anna in Figure 39.

An overloading of CHILD_TASK that returns a TASK_NAME_ARRAY is also supplied; this form of CHILD_TASK is used to activate an array of tasks. The number of new tasks to be activated is given by an extra **in**-mode parameter called DEPENDENT_COUNT.

4.6.12 The ACTIVATE_TASK Procedure

The ACTIVATE_TASK procedure is executed at the end of every declarative part (i.e, as the first action following the reserved word **begin**). Its declaration is shown in Figure 40. Its purpose is to enforce the task activation semantics of Ada, which requires that all dependents of a scope finish their activation before execution of the body of the scope may proceed. The procedure initiates the dependent activation message sequence of the Weatherly protocol; the ELABORATE_TASK function described below carries out the remaining communication of this sequence. MY_NAME is the identifier of the task which is activating new dependents.

```
    procedure ACTIVATE_TASK (MY_NAME : in   TASK_NAME;
                             ABORTED : out BOOLEAN;
                             MY_NODE : in   NODE_ID_TYPE);
--| where
--  MY_NAME's scope is unchanged:
--|   CURRENT_SCOPE (MY_NAME) = in CURRENT_SCOPE (MY_NAME),
--  An aborted task is no longer CALLABLE:
--|   out (ABORTED ->
--|          (for all T1 : TASK_NAME; N1 : NODE_ID_TYPE =>
--|             not CALLABLE_ATTR (T1, MY_NAME, N1))),
--  Failed activation of a dependent causes propagation of TASKING_ERROR:
--|   raise TASKING_ERROR;
```

Figure 40: The Supervisor ACTIVATE_TASK Procedure.

```
    function ELABORATE_TASK (MY_NAME : in TASK_NAME;
                             MY_NODE : in NODE_ID_TYPE)
                           return NODE_ID_TYPE;
--| where
--  MY_NAME is executing its outermost scope (scope 0):
--|        CURRENT_SCOPE (MY_NAME) = 0,
--|        return MY_NODE;
```

Figure 41: The Supervisor ELABORATE_TASK Function.

4.6.13 The ELABORATE_TASK Function

The ELABORATE_TASK function is executed at the beginning of the outermost declarative part of each task; the function returns the value of the MY_NODE parameter. The declaration of this function is shown in Figure 41. Its purpose is to block execution of the calling task, MY_NAME, until its master reaches the appropriate point for its activation. ELABORATE_TASK carries out the remaining communication of the task activation message sequence of the Weatherly protocol.

```
    procedure TERMINATE_TASK (MY_NAME : in TASK_NAME;
                              ABORTED : in BOOLEAN);
                              MY_NODE : in NODE_ID_TYPE);
--| where
--  MY_NAME's scope level number is decremented if
--  the current scope is not the outermost scope:
--|   out (in CURRENT_SCOPE (MY_NAME) > 0 ->
--|           CURRENT_SCOPE (MY_NAME) =
--|             in CURRENT_SCOPE (MY_NAME) - 1),
--  If the current scope is the outermost scope, then evaluation of
--  CALLABLE and TERMINATED on MY_NAME must yield
--  the values FALSE and TRUE, respectively:
--|   out (in CURRENT_SCOPE (MY_NAME) = 0 ->
--|         for all T1 : TASK_NAME; N1: NODE_ID_TYPE =>
--|           not CALLABLE_ATTR (T1, MY_NAME, N1) and
--|           TERMINATED_ATTR (T1, MY_NAME, N1)));
```

Figure 42: The Supervisor TERMINATE_TASK Procedure.

4.6.14 The TERMINATE_TASK Procedure

The TERMINATE_TASK procedure, shown in Figure 42, is executed at the end of every declarative region (i.e, as the last action preceding the reserved word **end**). The main purpose of TERMINATE_TASK is to block the calling task (MY_NAME) at the end of a scope until all dependents of the scope have terminated. If the scope level of MY_NAME is 0, the outermost scope level, then MY_NAME is also terminated; otherwise, the scope level number of MY_NAME is decremented.

4.6.15 The ABORT_TASKS Procedure

The **abort** statement is compiled into a call to the ABORT_TASK procedure, whose declaration is shown in Figure 43. The ABORT_TASK procedure of Weatherly's supervisor has been enhanced to accept in an array the names of all tasks named in an **abort** statement. Because a reply must be received from each aborted task, compilation of an **abort** of N tasks into N sequential calls to Weatherly's ABORT_TASK procedure would be extremely inefficient, since after sending an ABORT_TASK_MSG to a task, an ABORT_REPLY_MSG must be received before another ABORT_TASK_MSG can be sent. With the modification,

```
     procedure ABORT_TASKS (MY_NAME : in      TASK_NAME;
                            VICTIMS : in      TASK_NAME_ARRAY;
                            ABORTED : out     BOOLEAN;
                            MY_NODE : in out  NODE_ID_TYPE);
--| where
--  MY_NAME's scope is unchanged:
--|   CURRENT_SCOPE (MY_NAME) = in CURRENT_SCOPE (MY_NAME),
--  If MY_NAME is aborting itself, then ABORTED must be TRUE:
--|   out ((exist I1 : VICTIMS'RANGE => MY_NAME = VICTIMS (I1)) ->
--|              ABORTED),
--  Aborted tasks are no longer CALLABLE:
--|   out (ABORTED ->
--|          (for all T1 : TASK_NAME; N1 : NODE_ID_TYPE =>
--|             not CALLABLE_ATTR (T1, MY_NAME, N1))),
--|   out (not ABORTED ->
--|          (for all T1, T2 : TASK_NAME; N1 : NODE_ID_TYPE =>
--|               (exist I1 : VICTIMS'RANGE => T2 = VICTIMS (I1)) ->
--|                 not CALLABLE_ATTR (T1, T2, N1)));
```

Figure 43: The Supervisor ABORT_TASKS Procedure.

all ABORT_TASK_MSGs can be sent simultaneously, allowing the responses to the messages to be received in parallel.

The identifier of the task executing the **abort** statement is passed in MY_NAME; the identifiers of the aborted tasks are passed in the VICTIMS array. Since an **abort** statement is a synchronization point for completion of an abnormal task, an ABORTED parameter is provided. This will be set to TRUE if either MY_NAME is ABNORMAL upon input to the procedure, if MY_NAME is ABNORMAL upon exit from the procedure, or if MY_NAME is a component of VICTIMS. In the first case, the **abort** statement is *not* carried out; in the latter two cases, all VICTIMS are aborted. ABORT_TASK initiates the abort message sequence described in Section 4.5.7. Characteristics of ABORT_TASK which are a consequence of the semantics of Ada are formally specified in Anna in Figure 43.

4.6.16 The DELAY_TASK Procedure

Delay statements which do not appear as the first statement of a **delay** alternative in

```
      procedure DELAY_TASK (MY_NAME : in     TASK_NAME;
                            A_WHILE  : in     DURATION;
                            ABORTED : out    BOOLEAN;
                            MY_NODE : in out NODE_ID_TYPE);
--| where
--  MY_NAME's scope is unchanged:
--|   CURRENT_SCOPE (MY_NAME) = in CURRENT_SCOPE (MY_NAME),
--  An aborted task is no longer CALLABLE:
--|   out (ABORTED ->
--|          (for all T1 : TASK_NAME; N1 : NODE_ID_TYPE =>
--|             not CALLABLE_ATTR (T1, MY_NAME, N1))),
--  If MY_NAME is not abnormal, then it must delay for at least A_WHILE:
--|   out (not ABORTED ->
--|          CALENDAR.CLOCK - in CALENDAR.CLOCK >=
--|                                   A_WHILE);
```

Figure 44: The Supervisor DELAY_TASK Procedure.

a **select** statement are compiled into a call to the DELAY_TASK procedure, shown in Figure 44. The identifier of the delaying task is passed in MY_NAME, and the delay value is passed in A_WHILE. Since a **delay** statement is a synchronization point for completion of an abnormal task, an ABORTED parameter is provided. The semantics of the **delay** statement are formally specified in Anna in Figure 44.

4.7 Design Summary

During development of the Distributed Ada Supervisor, it was necessary to debug Supervisor components and port them to different machines. The ease with which these endeavors were accomplished is a result of the way in which the Supervisor is designed. The success of the design can be attributed to the following factors:

1. The encapsulation of all machine-dependencies inside a single package body results in a design that is highly portable.

2. The modularization of the design means that modification of one component takes place without disturbing unrelated components.

3. The kernel algorithms exploit the visibility and scoping rules of Ada, resulting in the dynamic shortening of message paths and reduction of overall message-passing activity at runtime.

4. The interface to the Supervisor is formally specified in Anna, making it suitable for standardization.

The next chapter presents a detailed description of two particular implementations of the Distributed Ada Supervisor.

Chapter 5

Implementation of the Distributed Ada Supervisor

This chapter presents the first known detailed description of the configuration, operation and execution of a distributed Ada supervisor on a specific multiprocessor architecture—an implementation of the Distributed Ada Supervisor on the Sequent *Balance 21000*, a shared memory multiprocessor. An earlier implementation, a simulator that was implemented on the Data General *MV10000/Eclipse*, is also described; the simulator simulates the execution of an Ada tasking program on a virtual distributed multiprocessor. In addition, the communication overhead of a distributed program execution is discussed. Finally, the use of diagnostic output for preliminary testing of both implementations is described at the end of the chapter.

Both implementations were tested with programs that test all of the Supervisor subprograms. Since both the Supervisor and the test programs are compiled using the commercial Ada compiler available on each target system, the test programs are manually preprocessed to replace all source-level tasking constructs with calls to the Distributed Ada Supervisor; this preprocessing is necessary to ensure that (1) The use of the tasking supervisor of the commercial compiler is completely eliminated, and (2) The remaining functionality of the runtime system of the commercial compiler is preserved for use by the test programs. The source preprocessing is described in detail in Appendix E in the form of *Before/After* examples such as those found in [HL83].

```
procedure MAIN is
    task T;
    task body T is
    begin
        ...     -- Code of body.
    end T;
begin-- Activation of T occurs here.
    ...         -- Code of body.
end MAIN;
```

Figure 45: Sample Tasking Program Before Preprocessing.

5.1 The Uniprocessor Supervisor Simulator

The Supervisor simulator was implemented on the Data General using the ROLM Ada compiler. All of the Supervisor subprograms were implemented; however, the Supervisor relied on the ROLM runtime system to perform task activation and scheduling. This dependence on the ROLM runtime system simplified the implementation of the simulator as well as the preprocessing of test programs. All other aspects of Ada tasking were performed by the Supervisor simulator. The reliance on the ROLM runtime system can be illustrated by an example. Consider the source program of Figure 45, which depicts a main program with a single dependent task. Figure 46 shows the program of Figure 45 after it has undergone the *complete* set of preprocessing transformations for testing the Distributed Ada Supervisor on the Sequent—the program no longer contains any source-level tasking constructs, the task bodies have become Ada procedures (which are "called" by the Supervisor to activate the bodies as tasks), and calls to Supervisor subprograms have been inserted appropriately. In the figure, italicized names stand for Supervisor-generated task identifiers. On the other hand, Figure 47 depicts the program of Figure 45 as it would be transformed for testing with the simulator. Calls to subprograms have been inserted, but the tasks remain in the program, which means that the task is activated by the ROLM runtime system. Due to this reliance on the ROLM supervisor it was necessary to split the functionality of the ACTIVATE_TASK procedure. ACTIVATE_TASK_BEGIN causes ELABORATE_MSGs to be sent to all dependents that are to be activated, and it is then exited to allow the activation to proceed. Once the activation has finished, the remaining functionality of ACTIVATE_TASK, the awaiting of ACTIVATE_MSGs from the new

```
with TASKING_SUPERVISOR;
procedure MAIN is
    procedure T (T : in TASK_NAME) is
    -- Task T is activated when the Supervisor "calls" this procedure.
        DUMMY1 : BOOLEAN :=
                TASKING_SUPERVISOR.ELABORATE_TASK (T, ...);
    begin
        TASKING_SUPERVISOR.ACTIVATE_TASK (T, ...);
        ...     -- Code of body.
        TASKING_SUPERVISOR.TERMINATE_TASK (T, ...);
    end T;
begin
    -- Activation of T occurs during this call:
    TASKING_SUPERVISOR.ACTIVATE_TASK (MAIN, ...);
    ...     -- Code of body.
    TASKING_SUPERVISOR.TERMINATE_TASK (MAIN, ...);
end MAIN;
```

Figure 46: Sample Program After Full Preprocessing.

dependent, is taken care of by the ACTIVATE_TASK_END procedure.

Tasks were also used in the implementation of the Supervisor simulator itself. Supervisor tasks served four purposes:

1. The set of simulated processors were represented by tasks.

2. The kernel message receiver/router was implemented as a task (see Section 4.4.4).

3. Tasks were used to implement semaphores [Dij68] for protecting critical sections of the Supervisor.

4. Tasks were used to implement monitor condition variables [Hoa74], which are used to avoid inefficient busy-waits in the Supervisor.

Associated with each "hardware task" was a queue for incoming messages; each queue was visible to all other hardware tasks. Message-passing between these hardware tasks occurs as follows. Messages destined for remote nodes are passed to the "local" hardware task by the kernel SEND procedure (see Section 4.4.3). The local hardware task then places the message at the tail of the queue of the destination hardware task. The receiver/router

```
with TASKING_SUPERVISOR;
procedure MAIN is
     task T;

     DUMMY2 : BOOLEAN :=
                 TASKING_SUPERVISOR.ACTIVATE_TASK_BEGIN (MAIN, ...);

     task body T is
          DUMMY1 : BOOLEAN :=
                      TASKING_SUPERVISOR.ELABORATE_TASK (T, ...);
          DUMMY2 : BOOLEAN :=
                      TASKING_SUPERVISOR.ACTIVATE_TASK_BEGIN (T, ...);
     begin
          TASKING_SUPERVISOR.ACTIVATE_TASK_END (T, ...);
          ...      -- Code of body.
          TASKING_SUPERVISOR.TERMINATE_TASK (T, ...);
     end T;
begin     -- Activation of T occurs here.
     TASKING_SUPERVISOR.ACTIVATE_TASK_END (MAIN, ...);
     ...           -- Code of body.
     TASKING_SUPERVISOR.TERMINATE_TASK (MAIN, ...);
end MAIN;
```

Figure 47: Sample Program After Partial Preprocessing for the Simulator.

at the destination node continually requests messages from its local hardware task, which takes messages off the head of its own queue.

The implementation of the receiver/router of the kernel as a task is an obvious manifestation of its behavior as an asynchronous thread of control inside the Supervisor. However, the implementation of semaphores and condition variables requires a bit more explanation. In reality, semaphores and condition variables are coded in the same way; each has two associated operations which are always performed in an alternating order. The difference lies in the manner in which semaphores and condition variables are used in software. To use a semaphore, a task must execute both operations to bracket a critical section. To use a condition variable, one task must execute one of the operations to wait for a condition; the occurrence of the condition is signaled by another task executing the second operation. Figure 48 depicts the implementation of semaphores and condition variables as tasks and

```
task SEMAPHORE is
    entry P;
    entry V;
end SEMAPHORE;
task body SEMAPHORE is
begin
    loop
        select
            accept P; accept V;
        or
            terminate;
        end select;
    end loop;
end SEMAPHORE;

task CONDITION_X is
    entry SIGNAL;
    entry WAIT;
end CONDITION_X;
task body CONDITION_X is
begin
    loop
        select
            accept SIGNAL; accept WAIT;
        or
            terminate;
        end select;
    end loop;
end CONDITION_X;

task body A is
begin
    SEMAPHORE.P;
    ...     -- Critical region.
    SEMAPHORE.V;
    CONDITION_X.WAIT;    -- Wait for condition X to occur.
end A;

task body B is
begin
    SEMAPHORE.P;
    ...     -- Critical region.
    SEMAPHORE.V;
    CONDITION_X.SIGNAL;    -- Signal that condition X has occurred.
end B;
```

Figure 48: Implementation of Semaphores and Condition Variables as Tasks.

illustrates their use.

5.2 The Distributed Supervisor Implementation

The uniprocessor Supervisor simulator was converted into a full-fledged distributed tasking supervisor on the Sequent. This conversion was aided by the modular, layered structure of the Supervisor software. The Sequent implementation of the Distributed Ada Supervisor contains 7377 lines of code (2253 Ada statements, 2224 comments) in 28 files; 93 of these statements are Ada machine code insertions. In addition, the Supervisor calls 3 functions from the Sequent C library and 5 functions from the Sequent Parallel Programming Library. It takes approximately 30 minutes to compile the Supervisor on the Sequent using Version 5.41 of the Verdix Ada compiler.

In converting the simulator for parallelization, it was necessary only to provide replacements for certain software modules; no massive global modifications to the Supervisor were necessary. The conversions that were required fall into three categories:

1. Implementation of a task scheduler.
2. Removal of Ada tasks from the Supervisor software.
3. Achieving synchronized parallel execution of multiple copies of the Supervisor software.

These conversions were accomplished respectively by three software tools:

1. The public-domain DOMINO multitasking library [OSvdG86].
2. The Sequent Parallel Programming Library (PPL) [PPL86].
3. The `fork()` library function of DYNIX[1], the Sequent version of Unix.

The basic task switching algorithms and data structures of DOMINO were adapted to the needs of the Distributed Ada Supervisor, and the resulting modifications were implemented from scratch in Ada, with an unavoidable smattering of C and NS32032[2] assembly language. The interfaces to the DOMINO routines are the NET_SERVICES procedures CREATE_LOCAL_PROCESS (task creation), ACTIVATE_LOCAL_PROCESS (task activation), SUSPEND_LOCAL_PROCESS (task suspension) and TERMINATE_LOCAL_PROCESS (task

[1]DYNIX is a registered trademark of Sequent Computer Systems, Inc.
[2]Series 32000 is a registered trademark of National Semiconductor Corporation.

termination). The interface to the DYNIX fork primitive is the CREATE_OS_PROCESS procedure of NET_SERVICES, and the LOCK subprograms of NET_SERVICES provide the interface to the lock facilities of the Sequent PPL.

The VERDIX compiler (VADS[3] Version 5.41) was used on the Sequent to compile both the Supervisor and preprocessed test programs. As a result of the above conversions, the VERDIX tasking supervisor is completely ignored during compilation and execution of preprocessed application programs. The Sequent implementation of the Distributed Ada Supervisor was instrumented to accept from the command line the number of processors to be used for program execution. To execute application program *foo* on N processors, the user executes the command "`foo N`". This form causes the diagnostic output described in Section 5.4 to be turned on; the diagnostic output is turned off by executing "`foo -N`" instead.

Once the conversion from Ada tasks to equivalent sequential constructs was completed in the Sequent implementation of the Supervisor, the Supervisor was ported from the Sequent to a Sun-3 workstation. It was configured to run in simulator mode and was compiled using a VERDIX compiler. This porting operation took only an hour to achieve.

5.2.1 Task Scheduling with DOMINO

DOMINO contains a collection of routines for scheduling and switching between tasks in a single program. A new task is scheduled by allocating stack space for its execution and creating a descriptor in a global list. The descriptor contains the entry address for the code of the task, a unique identifier for the particular instance of the task, pointers to the top and bottom of its stack, and a status variable; the status may be either "READY" (waiting to begin execution), "ACTIVE" (executing) or "DETACHED" (finished execution).

Execution in the DOMINO environment takes place with the DOMINO *control* routine (the task switching loop) acting as main program and the user tasks (including the user's main program) executed as subservient coroutines. DOMINO performs simple round-robin scheduling of tasks without time-slicing; tasks must explicitly suspend themselves in favor of the *control* routine, which places the suspended task on the tail of the descriptor list and then awakens the first READY or ACTIVE task at the head of the list.

DOMINO is ideally suited to parallel programs containing a linear, non-hierarchical collection of execution threads whose synchronization and scheduling are programmed by

[3] *VADS* is a registered trademark of the VERDIX Corporation.

the user. In adapting DOMINO to the needs of the Distributed Ada Supervisor, Ada tasks were quite easily implemented as DOMINO tasks; however, several enhancements were necessary:

1. The descriptor for each program task was expanded to contain the node address of the task. Instances of the scheduler then schedule tasks whose node address matches that of the scheduler instance.

2. The descriptor for each program task was expanded to contain a pointer to the activation record for the innermost scope enclosing the declaration of the task body. This allows the building of "cactus stacks", whereby multiple threads of control execute on branches of a tree of execution stack frames.

3. Instead of having the *control* program serve as the main program, it is called as a subroutine which early in the elaboration of the Ada main program. During this initialization, the *control* routine transparently usurps the rôle of main program, and the Ada main program thread is transparently transformed into a DOMINO task.

4. A fourth status value, NOT_READY, is used so that each new task remains unactivated until it is explicitly activated by the Supervisor.

In addition to the above enhancements, implementation of Ada task priorities requires the ability to time-slice and preëmpt tasks; this particular functionality was not added.

With these enhancements to DOMINO, scheduling of Ada program tasks is accomplished at runtime as follows. As explained in Section 4.6.11, each declaration of a task is translated into a call to the Supervisor function CHILD_TASK. If CHILD_TASK schedules the new task for local execution, it calls the CREATE_LOCAL_PROCESS procedure, which creates a descriptor for the new task. On the other hand, if CHILD_TASK schedules the new task for execution at a remote processor, it sends a NEW_TASK_MSG message to the destination copy of the Supervisor (see Section 4.5.2); this message provides the remote agent with the information it needs to make the call to CREATE_LOCAL_PROCESS. In both cases, the status of the new task is initially NOT_READY. When the master of the new task is ready for the new task to begin its activation, a call is made to the ACTIVATE_LOCAL_PROCESS procedure, which changes the status of the new task from NOT_READY to READY. In addition to program tasks, the receiver/router of each supervisor copy is scheduled as a DOMINO task.

Since no time-slicing or asynchronous preëmption is used by the scheduler to perform task switching, task switching occurs at predetermined places inside the Supervisor corresponding to execution of "blocking" statements in the application program. The following list describes these blocking constructs:

- The activation of new dependents causes the master to block until the activations are completed. The blocking occurs in both the ACTIVATE_TASK procedure and the CHILD_TASK function.
- A task whose agent is waiting for a DEPENDENT_REPLY_MSG (see Section 4.5.5) is blocked from scheduling new dependents until the message has been received and processed. The blocking occurs in the CHILD_TASK function.
- A single task or a task declared by an object declaration is blocked from beginning its activation until the master of the new task passes the **begin** in its body. The blocking occurs in the ELABORATE_TASK function.
- A master is blocked from terminating a scope until all dependents of the scope have terminated. The blocking occurs in the TERMINATE_TASK procedure.
- A task executing an **accept** statement or selective wait becomes blocked if the queues of the accepted entries are empty and the associated timeout has not yet expired. The blocking occurs in the ACCEPT_BEGIN procedure.
- A task that is executing a selective wait with a **terminate** alternative is blocked when waiting for notification from its master (see Section 4.5.6). The blocking occurs in the ACCEPT_BEGIN procedure.
- A task that has accepted a particular entry call (with an ACCEPT_MSG) is blocked until a CONFIRM_MSG or ABORT_CALL_MSG is received. The blocking occurs in the ACCEPT_BEGIN procedure.
- A task that is executing an entry call is blocked until either a rendezvous completes, a timeout expires or an exception is propagated to the task. The blocking occurs in the ENTRY_CALL procedure.
- A task that is executing an **abort** statement is blocked until the aborted tasks (and their dependents) have become abnormal. The blocking occurs in the ABORT_TASKS procedure.

- A task is blocked while waiting for the result of the evaluation of a task attribute. The blocking occurs in the procedures CALLABLE_ATTR and TERMINATED_ATTR.

- A task is blocked while executing a simple **delay** statement. The blocking occurs in the DELAY_TASK procedure.

In addition to the above constructs, the receiver/router of the kernel is blocked during a RECEIVE whenever there is no message to be received. The SUSPEND_LOCAL_PROCESS procedure is the interface to the switching routine of DOMINO. A call to SUSPEND_LOCAL_PROCESS suspends the caller and passes control to the *control* routine, which then awakens the next task in its descriptor list. Eventually, a suspended task becomes reawakened, at which time it attempts to progress through its blocking construct.

The removal of a program task from the descriptor list is accomplished by a call to the TERMINATE_LOCAL_PROCESS procedure. This call is made at the very end of the TERMINATE_TASK procedure only when a task is terminating its outermost declarative region.

5.2.2 Synchronization with Sequent Locks

The Sequent PPL *lock* facility provides a suitable replacement for the semaphore and condition variable tasks. A lock is a bistable component of special memory hardware called *Atomic Lock Memory* (ALM). The PPL provides primitives for initializing, locking, unlocking and testing locks; locking is accomplished by an atomic test-and-set instruction. A lock is initially in its unlocked state. A lock that is locked remains so until it is unlocked; in particular, an attempt to lock a lock that is already locked is blocked until the lock becomes unlocked. However, since a lock may be repeatedly unlocked before being locked, the implementation of condition variables as locks requires explicit testing for the unlocked state so as to preserve strict alternation between waiting and signaling. The correspondence between locks, semaphores and condition variables is immediately obvious and is portrayed in Table 2. Note that when a lock is used as a condition variable, it must be set initially in its locked state, since the WAIT operation is implemented by locking the lock.

Operations		
Sequent Lock	Semaphore	Condition Variable
`lock()`	P	WAIT
`unlock()`	V	SIGNAL

Table 2: Correspondence Between Locks, Semaphores and Condition Variables.

5.2.3 Parallel Execution with Unix FORK

Parallel execution of application programs is achieved in the Sequent implementation by forking copies of the Supervisor for execution on multiple processors. The following algorithm achieves parallelization of the Supervisor at runtime. Suppose that the application program is to execute on N processors:

1. At the earliest possible point in the elaboration of the Supervisor packages, initialize the DOMINO *control* routine (the main task switching loop).

2. Elaborate the Supervisor packages.

3. At the latest possible point in the elaboration of the Supervisor packages, call the CREATE_LOCAL_PROCESS procedure to start up a locally executing copy of the kernel message receiver/router. The node address of this instance of the Supervisor is zero.

4. Parallelize the Supervisor.

   (a) In each iteration of a loop from 1 to $N-1$ (since one copy of the Supervisor is already executing), fork a new DYNIX process by calling the CREATE_OS_PROCESS procedure. It is assumed that each new process is transparently assigned to a different Sequent processor by the DYNIX load-balancing mechanism.

   (b) Each new DYNIX process calls the CREATE_LOCAL_PROCESS procedure to create a new instance of the kernel message receiver/router. The node address of the new instance of the kernel is the number of the loop iteration which created the instance.

   (c) Each new process then starts up the DOMINO *control* routine, which performs task switching on local tasks. Initially, the set of local tasks will include only the above-scheduled instance of the receiver/router, which then accepts scheduling requests from remote agents in the form of NEW_TASK_MSGs.

This parallelization algorithm is almost completely portable; the only implementation dependencies are the actual forking primitive used to start up new copies of the Supervisor and the method of binding a new process to a particular system processor.

5.3 Communication Overhead

Although no comprehensive performance analysis was undertaken for either implementation, it is worthwhile to compute the minimum overhead in message-passing between supervisor copies that is required for distributed program execution. This overhead represents the cost of parallelizing the execution of an Ada tasking program.

To compute this overhead, assume that scalar-valued message components require four bytes of storage. Looking at the declaration of SUPERVISOR_MESSAGE_TYPE in Appendix C, every message contains two 4-byte discriminants and four 4-byte components, plus additional storage for variant components. Thus, every message contains at least 24 bytes.

Consider the distributed execution of an Ada tasking program on n processors. Assume that the program activates t tasks, t_p of which are activated through execution of an allocator. Assume further that all activations are remote scheduling operations requiring message-passing between two different supervisor copies; this is a worst-case assumption about the overhead required to assign each task to a processor.

Under these assumptions, the execution of each task is represented by a sequence of four messages, each of which is assumed to require one hop to reach its destination: (1) A NEW_TASK_MSG, for scheduling the task at a remote node, (2) An ELABORATE_MSG, for activating the task, (3) An ACTIVE_MSG, for signaling the end of activation, and (4) A COMPLETE_MSG, for signaling termination of the task. The last three of these messages have no variant components and thus require 24 bytes of storage each. The NEW_TASK_MSG contains seven 4-byte variant components requiring an additional 28 bytes of storage, for a total of 52.

Activation of a task designated by an access value requires the passing of two extra messages, a DEPENDENT_MSG and DEPENDENT_REPLY_MSG. The DEPENDENT_MSG has three 4-byte variant components and thus requires a total of 36 bytes of storage; the DEPENDENT_REPLY_MSG has no variant components and thus contains 24 bytes. Assume that the DEPENDENT_MSGs require an average of two hops to reach their destinations, and assume that each DEPENDENT_REPLY_MSG requires a single hop.

Finally, NEW_TASK_MSGs are used to create the receiver/router process of each supervisor copy; as described in the previous section, $n-1$ of these processes are activated remotely.

Using the above figures, the *minimum* number of bytes B_{min} that are passed over the interprocessor network during distributed execution is expressed as a function of n, t and t_p:

$$\begin{aligned} B_{min} &= t(52+24+24+24) && (1)\\ &\quad + t_p((2*36)+24) && (2)\\ &\quad + (n-1)(52) && (3)\\ &= 124t + 96t_p + 52n - 52 \end{aligned}$$

Term (1) is the communication overhead for activating t tasks. Term (2) is the additional overhead for activating t_p of those t tasks using an allocator. Term (3) is the overhead for starting up $n-1$ additional copies of the Distributed Ada Supervisor.

The amount of other communication occurring during distributed execution is highly dependent on the algorithm of the program. This amount is proportional to the number of rendezvous, task attribute evaluations, intertask exception propagations, **terminate** alternatives and task abortions that are executed.

5.4 Diagnostic Supervisor Output

In the early stages of implementation it became apparent that some form of diagnostic information displayed on the terminal would be useful for visually monitoring the execution of the Supervisor. To this end, the Supervisor was instrumented with an optional tracing capability which signals the occurrence of the sending and receiving of messages. During tracing, each occurrence of the following message-passing events is indicated on the terminal:

1. A message is processed locally at some system node X. This occurs when a message is passed between two tasks executing at the same node.

2. A message is sent from some node X to some other node Y.

3. A message is received at some node X and forwarded to some other node Y. This occurs when the indirect routing capabilities of the Supervisor kernel are invoked.

4. A message is received and processed at some node Y.

The message involved in each event is identified by its message class, source task name, destination task name, and any other relevant fields, such as the transaction number assigned to a rendezvous. Thus, messages passed between tasks executing on the same node are represented by a single tracing event, while messages passed between tasks executing on different nodes are represented by two or more tracing events. In addition to the message-passing events, execution of **delay** statements and evaluations of the COUNT attribute are also traced.

Chapter 6

Verification of the Distributed Ada Supervisor

This chapter presents a new and practical methodology for verifying a distributed supervisor for consistency with the semantics of the language it implements. The chapter first describes how TSL specifications are used in runtime checking experiments to detect errors in the Distributed Ada Supervisor. The chapter then discusses how runtime checking is related to fundamental assumptions about the behavior of the NET_SERVICES virtual machine that was described in Section 4.3; these assumptions are also formalized in TSL. As will be seen, these assumptions play an important rôle in interpreting the results of runtime checking, for the validity of the assumptions implies that the global event stream seen by the TSL Monitor during checking is an accurate representation of the actions performed by the Supervisor. The chapter ends with a summary of the advantages of the methodology over other approaches to testing a language implementation.

6.1 The Verification Approach

The behavior of the Distributed Ada Supervisor is considered to be correct if it is consistent with the semantics of Ada tasking. For the purpose of verifying the correctness of the Supervisor, the Supervisor is viewed as a "black box" whose behavior is manifested at the application program level during the execution of tasking statements. Chapter 9 of the Ada Language Reference Manual (LRM) informally presents the semantics of each tasking construct by specifying the sequence of tasking events which comprises its execution

```
        C          |        A
                   |   -- R
                   |   accept E do
      -- X         |      -- S
      A.E;         |      ...
      -- Y         |      -- T
                   |   end E;
                   |   -- U
```

Figure 49: Observing Tasking Events Outside the Supervisor.

behavior. Using the semantics of the LRM as a starting point, the following methodology is used to check the behavior of the Supervisor:

1. *Express the informal semantics of Chapter 9 of the Ada LRM as a set of TSL specifications.* These specifications are constraints on the patterns of Ada tasking events that may occur during the execution of an Ada program.

2. *Monitor Ada program executions for consistency with the TSL specifications.* An execution satisfying all specifications increases confidence in the correctness of the Supervisor, while a violated specification explicitly characterizes a bug in the Supervisor.

The specifications that are constructed in Step 1 are a formal definition of the Ada tasking semantics. In addition to using the specifications for automatic runtime consistency checking, the specifications may also be used to develop a formal proof of correctness, once TSL proof rules and proving tools become available.

To that observation of supervisor behavior can be performed at the application program level, consider the matching Ada entry call and **accept** statements shown in Figure 49. In this figure, the rôle of the Supervisor can be viewed as progressing task C from point X to point Y, and progressing task A from point R to point S and from point T to point U. Furthermore, if the Supervisor implements the rendezvous between tasks C and A correctly, then point S will not be reached in task A until point X has been reached in task C, and point Y will not be reached in task C until point T has been reached in task A. Thus, according to the Ada semantics, the six observation points must be reached in one of the following four orders:

- $X\ R\ S\ T\ U\ Y$

- $R\ X\ S\ T\ U\ Y$
- $X\ R\ S\ T\ Y\ U$
- $R\ X\ S\ T\ Y\ U$

By having each task generate an observable event each time an interesting point is reached in its body, the behavior of the Supervisor may be observed by observing the sequences of events that are thus formed.

In order to test the feasibility of the verification methodology, the following subset of the Ada tasking semantics was formalized in TSL:

- The activation and termination semantics of single tasks.
- The activation and termination semantics of tasks declared by object declarations.
- The semantics of the entry call statement, except for conditional or timed entry calls.
- The semantics of the **accept** statement, except for selective waits.
- The semantics of the **delay** statement.

The semantics of exception propagation and task abnormality as they apply to this subset were not considered. The semantics of the Ada **accept** statement are used in this chapter for the purposes of illustration; the complete formalization of the subset in TSL is presented in Appendix G.

6.2 Formalization of the Ada Tasking Semantics

The formalization of the Ada tasking semantics is carried out in two steps:

1. *Choose a set of events and Boolean-valued predicates for modeling the Ada tasking semantics.* These events and predicates are chosen based on an intuitive understanding of the informal English presentation of the Ada tasking semantics in Chapter 9 of the Ada LRM. The occurrence of a tasking event causes a change in the value of one or more predicates, reflecting a change in the computational state of the program.
2. *Formally define the events and predicates in TSL, and construct TSL specifications which formalize the semantics of each Ada tasking statement.* Each specification is

restricted to describing the execution of a *single tasking statement*, from the point of view of a *single participant task*.

The first step is illustrated by determining the events and predicates that are used to informally express the semantics of the rendezvous in Section 9.5 of the Ada LRM, which is entitled "Entries, Entry Calls and Accept Statements"; Appendix F gives a complete listing of the tasking events of Ada. The second step is illustrated by a TSL formalization of the semantics of the **accept** statement.

6.2.1 Choosing the Tasking Events and Predicates of the Rendezvous

The first nine paragraphs of Section 9.5 describe the syntax and static (i.e., compile-time) semantics of entry declarations, entry call statements and **accept** statements. One consequence of the static semantics is that any task participating in a rendezvous must be activated, but not yet completed. Thus, two predicates can be used to characterize the state of a task T executing an entry call or **accept** statement—*ACTIVATED (T)*, which is true if and only if T has been activated, and *COMPLETED (T)*, which is true if and only if T has completed its execution.

Paragraph 10 begins the description of the dynamic semantics:

10 Execution of an accept statement starts with the evaluation of the entry index (in the case of an entry of a family). Execution of an entry call statement starts with the evaluation of the entry name; this is followed by any evaluations required for actual parameters in the same manner as for a subprogram call (see 6.4). Further execution of an accept statement and of a corresponding entry call statement are synchronized.

Paragraph 10 suggests that it is useful to delimit the execution of each statement by a *Begin* event and an *End* event. Thus, the first four events that Section 9.5 mentions are

- Task C begins call to task T at entry E
- Task C ends entry call
- Task T begins **accept** statement for entry E
- Task T ends **accept** statement

The predicate *CALLING (C, T, E)* can be used to characterize the state of a task during execution of an entry call; the predicate is true between the occurrence of the *Begin* and *End* events and is false at all other times. The state of a task executing an **accept** statement can be similarly characterized using the predicate *ACCEPTING (T, E)*. The remaining events described in the paragraph—evaluation of the entry name and evaluation of actual parameters—are outside the domain of the responsibilities of an Ada tasking supervisor and will therefore remain unaccounted for in the set of events and predicates.

> If a given entry is called by only one task, there are two possibilities: 11
>
> - If the calling task issues an entry call statement before a corresponding accept statement is reached by the task owning the entry, the execution of the calling task is *suspended*. 12
> - If a task reaches an accept statement prior to any call of that entry, the execution of the task is suspended until such a call is received. 13

Paragraphs 11 through 13 describe certain conditions under which a task may become suspended. Thus, these paragraphs describe no events; in fact they specify that there must be *no* occurrence of tasking events!

> When an entry has been called and a corresponding accept statement has been reached, the sequence of statements, if any, of the accept statement is executed by the called task (while the calling task remains suspended). This interaction is called a *rendezvous*. Thereafter, the calling task and the task owning the entry continue their execution in parallel. 14

Paragraph 14 describes the rendezvous itself. Although the caller remains suspended, the called task resumes its execution if it was previously suspended; this suggests that the beginning of a rendezvous is an event. Furthermore, since the body of an **accept** statement can generate an arbitrary sequence of tasking events, it is necessary to use still another event to delimit the end of the rendezvous. Thus, the next two events of Section 9.5 are

- Task T begins rendezvous with task C at entry E
- Task T ends rendezvous with task C at entry E

The predicate *IN_RENDEZVOUS (C, T, E)* can be used to characterize the state of the tasks in rendezvous; the predicate is true during the rendezvous and false at all other times.

Since Paragraph 14 specifies that the sequence of statements associated with the rendezvous is executed by the called task, and since the paragraph also specifies that end of the **accept** statement always coincides with the end of the rendezvous, the *End Rendezvous* event will serve to indicate both the end of the rendezvous and the end of the execution of the **accept** statement. However, there is an observable period of time between the end of the rendezvous and the end of the entry call statement; the truth of the predicate *RENDEZVOUSED (C, T, E)* can be used to characterize the state of the caller during this period (true during the period, false at all other times).

> 15 If several tasks call the same entry before a corresponding accept statement is reached, the calls are queued; there is one queue associated with each entry. Each execution of an accept statement removes one call from the queue. The calls are processed in the order of arrival.

Paragraph 15 describes the mechanism by which a rendezvous is allowed to take place. Execution of an entry call generates an artifact that is placed on a queue controlled by the called task. Thus the event

- Call arrives from task C at entry E of task T

sufficiently describes the enqueuing of an entry call on behalf of the called task; the removal of the call from the queue coincides with the beginning of the rendezvous. The predicate *QUEUED (C, T, E)* can be used to characterize the state of the tasks in this interval; the predicate is true between the enqueuing of the call and the beginning of the rendezvous and false at all other times. The integer-valued function *QUEUE_SIZE (T, E)* can be used to indicate the number of calls on an entry queue.

> 16 An attempt to call an entry of a task that has completed its execution raises the exception TASKING_ERROR at the point of the call, in the calling task; similarly, this exception is raised at the point of the call if the called task completes its execution before accepting the call (see also 9.10 for the case when the called task becomes abnormal). The exception CONSTRAINT_ERROR is raised if the index of an entry of a family is not within the specified discrete range.

The first sentence of this paragraph mentions three related tasking events, namely

- Task T completes its execution
- Task T becomes abnormal
- TASKING_ERROR is raised in task T

These three events are also mentioned in other sections of Chapter 9 of the Ada LRM and will be discussed no further.

The remaining seven paragraphs of Section 9.5 mention no other tasking events; they give source examples, describe implications of the semantics and list cross-references to other sections of the Ada LRM. Thus, the execution of a rendezvous can be described by a sequence of the following nine events:

1. Task C begins call to task T at entry E
2. Task C ends entry call
3. Task T begins **accept** statement for entry E
4. Task T begins rendezvous with task C at entry E
5. Task T ends rendezvous with task C at entry E
6. Call arrives from task C at entry E of task T
7. Task T completes its execution
8. Task T becomes abnormal
9. TASKING_ERROR is raised in task T

Furthermore, the states of the participating tasks can be characterized by expressions involving the following eight predicates:

1. ACTIVATED (T)
2. COMPLETED (T)
3. CALLING (C, T, E)
4. ACCEPTING (T, E)
5. IN_RENDEZVOUS (C, T, E)

6. RENDEZVOUSED (C, T, E)

7. QUEUED (C, T, E)

8. QUEUE_SIZE (T, E)   (integer-valued)

A subset of these events and predicates will now be used to formalize the semantics of the **accept** statement in TSL; the remaining events and predicates are defined formally in Appendices F and G, respectively. The semantics of the entry call statement are formalized in TSL in Appendix G, as are the FCFS (first-come-first-served) semantics of entry queues and the semantics of suspension.

6.2.2 Formalizing the Semantics of the Accept Statement

In formalizing the semantics of a tasking construct, a collection of TSL specifications is constructed which describe the execution of the construct from the point of view of the task performing the execution. This task will be referred to as the *subject task* for purposes of discussion. Formalization of the semantics of the **accept** statement in TSL is comprised of the following four steps:

1. Formal definition of the tasking events as TSL *actions*.

2. Formal definition of the tasking predicates as TSL *properties*.

3. Construction of a *functional specification* in TSL. This is a positive specification that is activated each time a task begins execution of an **accept** statement. The activation must be satisfied in order to conclude correctness of the execution of the statement.

4. Construction of *safety specifications* in TSL. Each of these is a negative specification that is activated once at the beginning of program execution. In order to conclude correctness of the execution of *all* **accept** statements in the program, no activation may ever be violated.

The need for complementary functional and safety specifications is a consequence of the semantics of TSL. A TSL specification constrains *relevant subsequences* of the global event stream, treating all other subsequences as irrelevant. Thus, although the functional specification is needed to describe what event sequences *can* be observed during monitoring,

```
--   Begin Accept event:
--+ action SIMPLE_ACCEPT (ACCEPTED_E : entry);

--   Call Arrival event:
--+ action ENQUEUE_CALL (C_TASK : task; Q_TASK : task; Q_ENTRY : entry);

--   Begin Rendezvous event:
--+ action BEGIN_RENDEZVOUS (CALLER : task; CALLED_E : entry);

--   End Rendezvous event:
--+ action END_RENDEZVOUS (CALLER : task; CALLED_E : entry);
```

Figure 50: Action Declarations for the Tasking Events of the **Accept** Statement.

the safety specifications are also needed to describe when events *cannot* be observed. Construction of the safety specifications follows simply from the construction of the functional specification—for each guarded event of the form "E **where** P" appearing in the functional specification that is performed by the subject task of the specification, the safety specification "**not** E **where not** P;" is constructed.

TSL Definition of the Tasking Events and Predicates

The execution semantics of the **accept** statement may be specified using four of the tasking events and five of the predicates described in the previous section. The TSL action declarations for the four events are shown in Figure 50. (The ENQUEUE_CALL action formally defines the *Call Arrival* event, which is event number 6 on page 117). Except for the ENQUEUE_CALL action, each action is performed by the subject task, which therefore need not be a parameter to the declared actions since TSL automatically makes the performing task a constituent of each generated event. The subject task *is* passed as a parameter of the ENQUEUE_CALL action, since this action is performed by the Supervisor.

In defining the predicates as TSL properties, the property names are all prefixed by "S_" to distinguish them from any predefined TSL properties with the same name. The first two predicates that are defined are *ACTIVATED (T)* and *COMPLETED (T)*. As described in the previous section, *ACTIVATED (T)* is defined for all tasks T to be false initially and true after the activation of T is finished. Similarly, *COMPLETED (T)* is defined for all tasks

```
--   End Activation event:
--+ action END_ACTIVATION;
--+
--   ACTIVATED predicate:
--+ property S_ACTIVATED (task) : BOOLEAN := FALSE
--+ is
--+   when ?T performs END_ACTIVATION then
--+          set S_ACTIVATED (?T) := TRUE;
--+ end S_ACTIVATED;

--   Task Completion event:
--+ action COMPLETE;
--+
--   COMPLETED predicate:
--+ property S_COMPLETED (task) : BOOLEAN := FALSE
--+ is
--+   when ?T performs COMPLETE then
--+          set S_COMPLETED (?T) := TRUE;
--+ end S_COMPLETED;
```

Figure 51: The S_ACTIVATED and S_COMPLETED Properties.

T to be false initially and true after the completion of T. The TSL properties declared in Figure 51 formally define these predicates. The action declarations for the *End Activation* and *Task Completion* events are also given in the figure.

Next, the function *QUEUE_SIZE (T, E)* is defined. From the previous discussion, this function is initially zero for all T and E. It is incremented by the arrival of an entry call and is decremented by the beginning of a rendezvous. The TSL property of Figure 52 formally defines *QUEUE_SIZE (T, E)*; since *QUEUE_SIZE (T, E)* must always be greater than or equal to zero, the property is declared to be of type NATURAL.

Finally, the predicates *ACCEPTING (T, E)* and *IN_RENDEZVOUS (C, T, E)* are defined. These predicates are initially false for all C, T and E. *ACCEPTING (T, E)* is true between the beginning of an **accept** statement and the end of the corresponding rendezvous, while *IN_RENDEZVOUS (C, T, E)* is true only between the beginning and end of the rendezvous. The TSL properties declared in Figure 53 formally define these predicates.

```
--   QUEUE_SIZE function:
--+ property S_QUEUE_SIZE (task, entry) : NATURAL := 0
--+ is
--+    when any performs ENQUEUE_CALL (Q_TASK => ?T,
--+                                    Q_ENTRY => ?E) then
--+          set S_QUEUE_SIZE (?T, ?E) := S_QUEUE_SIZE (?T, ?E) + 1;
--+    when ?T performs BEGIN_RENDEZVOUS (CALLED_E => ?E) then
--+          set S_QUEUE_SIZE (?T, ?E) := S_QUEUE_SIZE (?T, ?E) - 1;
--+ end S_QUEUE_SIZE;
```

Figure 52: The S_QUEUE_SIZE Property.

```
--   ACCEPTING predicate:
--+ property S_ACCEPTING (task, entry) : BOOLEAN := FALSE
--+ is
--+    when ?T performs SIMPLE_ACCEPT (ACCEPTED_E => ?E) then
--+          set S_ACCEPTING (?T, ?E) := TRUE;
--+    when ?T performs END_RENDEZVOUS (CALLED_E => ?E) then
--+          set S_ACCEPTING (?T, ?E) := FALSE;
--+ end S_ACCEPTING;

--   IN_RENDEZVOUS predicate:
--+ property S_IN_RENDEZVOUS (task, task, entry) : BOOLEAN := FALSE
--+ is
--+    when ?T performs BEGIN_RENDEZVOUS (CALLER => ?C,
--+                                       CALLED_E => ?E) then
--+          set S_IN_RENDEZVOUS (?C, ?T, ?E) := TRUE;
--+    when ?T performs END_RENDEZVOUS (CALLER => ?C,
--+                                     CALLED_E => ?E) then
--+          set S_IN_RENDEZVOUS (?C, ?T, ?E) := FALSE;
--+ end S_IN_RENDEZVOUS;
```

Figure 53: The S_ACCEPTING and S_IN_RENDEZVOUS Properties.

```
--+ when Begin where Initial_State
--+ then (Intermediate Events => End where Final_State)
--+ until End;
```

Figure 54: Generic TSL Functional Specification.

Construction of the TSL Specifications

The execution of every Ada tasking construct is delimited by a *Begin* event and an *End* event. The functional specification for each statement can thus be constructed in the format of Figure 54. According to Table 1 in Chapter 2, the specification says that whenever the *Begin* event is observed in the *Initial_State* (as characterized by an expression involving one or more predicates), then by the time the *End* event is observed the remaining events of the sequence must have also been observed, including the *End* event in the *Final_State*.

The functional specification of the semantics of the **accept** statement is constructed by considering the two possible event sequences that can comprise its execution:

- Sequence 1—Empty Queue:
    1. Some task T begins execution of an **accept** statement for some entry E when the queue for E is *empty*.
    2. A call from some task C subsequently arrives at the queue for E.
    3. T begins a rendezvous with C at E.
    4. T ends the rendezvous with C at E.
- Sequence 2—Non-empty Queue:
    1. Some task T begins execution of an **accept** statement for some entry E when the queue for E is *not empty*.
    2. T begins a rendezvous at E with the first caller on the queue, C.
    3. T ends the rendezvous with C at E.

Each of these events must occur in an appropriate state. The *Begin Accept* event can occur only when *ACTIVATED (T)* is true and *COMPLETED (T)* is false. The *Begin Rendezvous* event can occur only when *ACCEPTING (T, E)* is true and *QUEUE_SIZE (T,*

```
--+ <<SIMPLE_ACCEPT_STATEMENT>>
--+ when ?T performs SIMPLE_ACCEPT (ACCEPTED_E => ?E)
--+          where S_ACTIVATED (?T) and not S_COMPLETED (?T)
--+ then ((any performs ENQUEUE_CALL (C_TASK => ?C, Q_TASK => ?T,
--+                                   Q_ENTRY => ?E)
--+           where S_ACCEPTING (?T, ?E) and S_QUEUE_SIZE (?T, ?E) = 0
--+         =>
--+         ?T performs BEGIN_RENDEZVOUS (CALLER => ?C,
--+                                       CALLED_E => ?E)
--+            where S_ACCEPTING (?T, ?E) and S_QUEUE_SIZE (?T, ?E) > 0
--+        )
--+        or
--+        ?T performs BEGIN_RENDEZVOUS (CALLER => ?C,
--+                                      CALLED_E => ?E)
--+           where S_ACCEPTING (?T, ?E) and S_QUEUE_SIZE (?T, ?E) > 0
--+       )
--+       =>
--+       ?T performs END_RENDEZVOUS (CALLER => ?C,
--+                                   CALLED_E => ?E)
--+          where S_IN_RENDEZVOUS (?C, ?T, ?E)
--+ until ?T performs END_RENDEZVOUS;
```

Figure 55: Functional Specification of the **Accept** Statement Semantics.

E) is greater than zero. The *End Rendezvous* event can occur only when *IN_RENDEZVOUS (C, T, E)* is true. The *Call Arrival* event can occur at any time from *T*'s point of view, but the occurrences relevant to the discussion are when *ACCEPTING (T, E)* is true and *QUEUE_SIZE (T, E)* is zero. Figure 55 gathers these facts together into the functional specification for the **accept** statement. The first line of the specification is its name. The specification is activated once for each occurrence of the *Begin Accept* event; matching on the activation is be terminated by the occurrence of the *End Rendezvous* event. The activation is satisfied if the complete sequencing behavior of the **accept** statement (as given in the specification body) is observed by the time the *End Rendezvous* event is matched. The placeholders *?C*, *?T* and *?E* are initially unbound. In each activation, *?T* and *?E* will be bound to the constituents of the *Begin Accept* event which created the activation. *?C* will be bound to a constituent from either the *Call Arrival* or *Begin Rendezvous* event, whichever is matched first.

```
--    The Begin Accept event must not occur in an incorrect state:
--+ not ?T performs SIMPLE_ACCEPT (ACCEPTED_E => ?E)
--+      where not (S_ACTIVATED (?T) and not S_COMPLETED (?T));

--    The Begin Rendezvous event must not occur in an incorrect state:
--+ not ?T performs BEGIN_RENDEZVOUS (CALLED_E => ?E)
--+      where not (S_ACCEPTING (?T, ?E) and S_QUEUE_SIZE (?T, ?E) > 0);

--    The End Rendezvous event must not occur in an incorrect state:
--+ not ?T performs END_RENDEZVOUS (CALLER => ?C, CALLED_E => ?E)
--+      where not (S_IN_RENDEZVOUS (?C, ?T, ?E));
```

Figure 56: Safety Specifications of the **Accept** Statement Semantics.

The safety specifications may now be derived from the functional specification. There are four different guarded events that appear in the functional specification of Figure 55. However, only three of these are performed by the subject task T; the *Call Arrival* event is performed by the Supervisor. Thus, a total of three safety specifications are required; they are presented in Figure 56. The safety specifications ensure that no tasking event will occur in an incorrect state. This fact implies that the guards appearing in the functional specification are redundant, and indeed that is the case, since satisfaction of the safety specifications logically implies the truth of the guards in the functional specification. Once the safety specifications have been constructed, the guards may be removed from the basic events appearing in the functional specification if desired. However, the results obtained from specification checking will be identical with or without guards in the functional specification.

Note that in specifying the semantics of a rendezvous, it is necessary to specify that task T cannot begin a rendezvous with task C until C calls T; however, this safety specification is derived from the functional specification for the semantics of the entry call statement.

6.3 The NET_SERVICES Connectedness Assumptions

When monitoring the Distributed Ada Supervisor for consistency with the TSL formalization of Ada tasking, the results of monitoring are interpreted *relative* to the assumption that the low-level communication facilities provided by NET_SERVICES satisfy certain *connectedness constraints*; the notion of connectedness of actions in TSL was discussed in Section 2.3.4.

The function of the Supervisor is essentially to *connect* the Ada tasking events generated by application programs; the TSL formalization of the Ada tasking semantics is thus a formal description of the connections the Supervisor must implement. The Supervisor connects the Ada tasking events using the SEND and RECEIVE message-passing subprograms declared in the NET_SERVICES package (see Figure 13).

When interpreting the results of consistency checking on the Supervisor, connectedness of the NET_SERVICES message-passing subprograms is assumed. If this assumption is satisfied by NET_SERVICES, then the global stream seen by the TSL Monitor will be consistent with the connectedness of the Ada tasking events performed by the application program. Thus, if the behavior of the Supervisor satisfies the specifications comprising the TSL formalization of Ada tasking, then the Supervisor is correctly connecting the tasking events; if a specification is violated, then a bug exists in the Supervisor.

Informally, the behavior of the NET_SERVICES actions is assumed to satisfy the following constraints:

Assumption 1 *The output of a RECEIVE is the input of a previous SEND.*

Assumption 2 *Each message is received only once.*

Assumption 3 *Messages sent from one thread to another are received in the order they are sent.*

Satisfaction of these assumptions by NET_SERVICES implies the pairwise connectedness of SEND and RECEIVE, for together they imply that all inductively corresponding pairs of SENDs and RECEIVEs are ordered so that the SEND precedes the RECEIVE.

To state these assumption formally in TSL, the following two actions are defined:

```
--+ action RECEIVE (M : MESSAGE_TYPE; BY : NODE_ID_TYPE);
--+ action SEND    (M : MESSAGE_TYPE; TO : NODE_ID_TYPE);
```

The RECEIVE action is performed immediately *after* a thread receives a message; the SEND action is performed immediately *before* a thread sends a message. Two properties are used to keep track of the number of times a message has been sent or received:

```
--+ property SEND_COUNT (MESSAGE_TYPE, NODE_ID_TYPE) : NATURAL := 0
--+ is
--+   when any performs SEND (M => ?M1, TO => ?S1) then
--+         set SEND_COUNT (?M1, ?S1) := SEND_COUNT (?M1, ?S1) + 1;
--+ end SEND_COUNT;

--+ property RECEIVE_COUNT (MESSAGE_TYPE, NODE_ID_TYPE) : NATURAL := 0
--+ is
--+   when any performs RECEIVE (M => ?M1, BY => ?S1) then
--+         set RECEIVE_COUNT (?M1, ?S1) := RECEIVE_COUNT (?M1, ?S1) + 1;
--+ end RECEIVE_COUNT;
```

A third property is used to indicate that a message has been sent but not yet received:

```
--+ property WAS_SENT (MESSAGE_TYPE, NODE_ID_TYPE) : BOOLEAN := 0
--+ is
--+   when any performs SEND (M => ?M1, TO => ?S1) then
--+         set WAS_SENT := TRUE;
--+   when any performs SEND (M => ?M1, TO => ?S1)
--+   then any performs RECEIVE (M => ?M1, BY => ?S1)
--+         set WAS_SENT := FALSE;
--+ end WAS_SENT;
```

The assumptions are then expressed formally by the following TSL specifications:

```
--+ <<ASSUMPTION_1>>
--+ not any performs RECEIVE (M => ?M1, BY => ?S1)
--+         where not WAS_SENT (?M1, ?S1);
--+ <<ASSUMPTION_2>>
--+ not any performs RECEIVE (M => ?M1, BY => ?S1)
--+         where RECEIVE_COUNT (?M1, ?S1) >= SEND_COUNT (?M1, ?S1);

--+ <<ASSUMPTION_3>>
--+ when ?S1 performs SEND (M => ?M1, TO => ?S2)
--+        =>
--+        ?S1 performs SEND (M => ?M2, TO => ?S2)
--+            where ?M2 /= ?M1
--+ then any performs RECEIVE (M => ?M1, BY => ?S2)
--+ before any performs RECEIVE (M => ?M2, BY => ?S2);
```

6.4 Runtime Checking Experiments

After TSL specifications have been constructed to formalize the Ada tasking semantics, *any* Ada tasking program may be used to test the behavior of the Supervisor for consistency with the specifications. However, each test program must be transformed for runtime checking. Figure 57 shows the transformations that are performed. The two *Compiler* transformations and the *Linker* transformation are currently automated; the others may easily be automated.

First, TSL **perform** statements are placed in the test program at each place that corresponds to the occurrence of a tasking event; they are placed according to the definitions given in Appendix F. In a few cases, events must be performed inside the Supervisor, such as placing an entry call on an entry queue; the Supervisor must be instrumented to perform these events in the appropriate places.

Second, the test program and the TSL specifications are fed together to the TSL Compiler, which translates the TSL statements into executable calls to the TSL Runtime System.

Third, for reasons mentioned at the beginning of Chapter 5, the test program is preprocessed so that its parallel execution will be controlled by the Distributed Ada Supervisor. Appendix E describes the necessary transformations, which convert the tasking statements of the test program into Supervisor subprogram calls.

Finally, the test program is compiled by the VERDIX compiler. The compiled code is linked with the Distributed Ada Supervisor and the TSL Runtime System to create a distributed self-checking test program.

Because the TSL specifications are derived from the Ada tasking semantics and do not describe internal Supervisor behavior, the specifications may be used to test *any* implementation of Ada; Figure 58 shows how test programs would be prepared in this case. As shown in the figure, a commercial Ada compiler is chosen for testing. The compiler translates the tasking statements of the test program, and the behavior of the supervisor that is shipped with the compiler is checked for correctness.

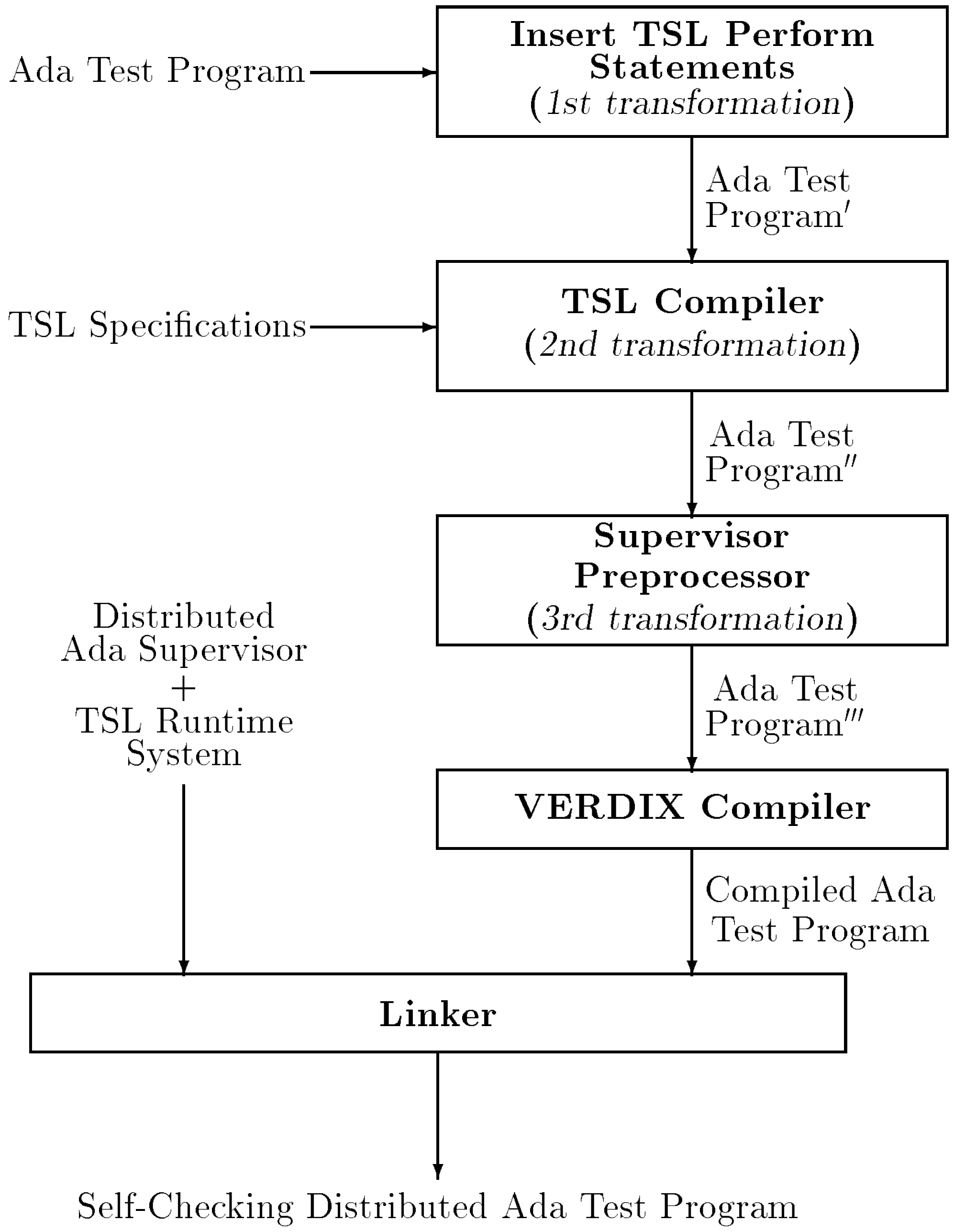
Ada Test Program
Insert TSL Perform Statements (1st transformation)
Ada Test Program′
TSL Specifications
TSL Compiler (2nd transformation)
Ada Test Program″
Supervisor Preprocessor (3rd transformation)
Distributed Ada Supervisor + TSL Runtime System
Ada Test Program‴
VERDIX Compiler
Compiled Ada Test Program
Linker
Self-Checking Distributed Ada Test Program

Figure 57: Transformation of Test Programs for the Distributed Ada Supervisor.

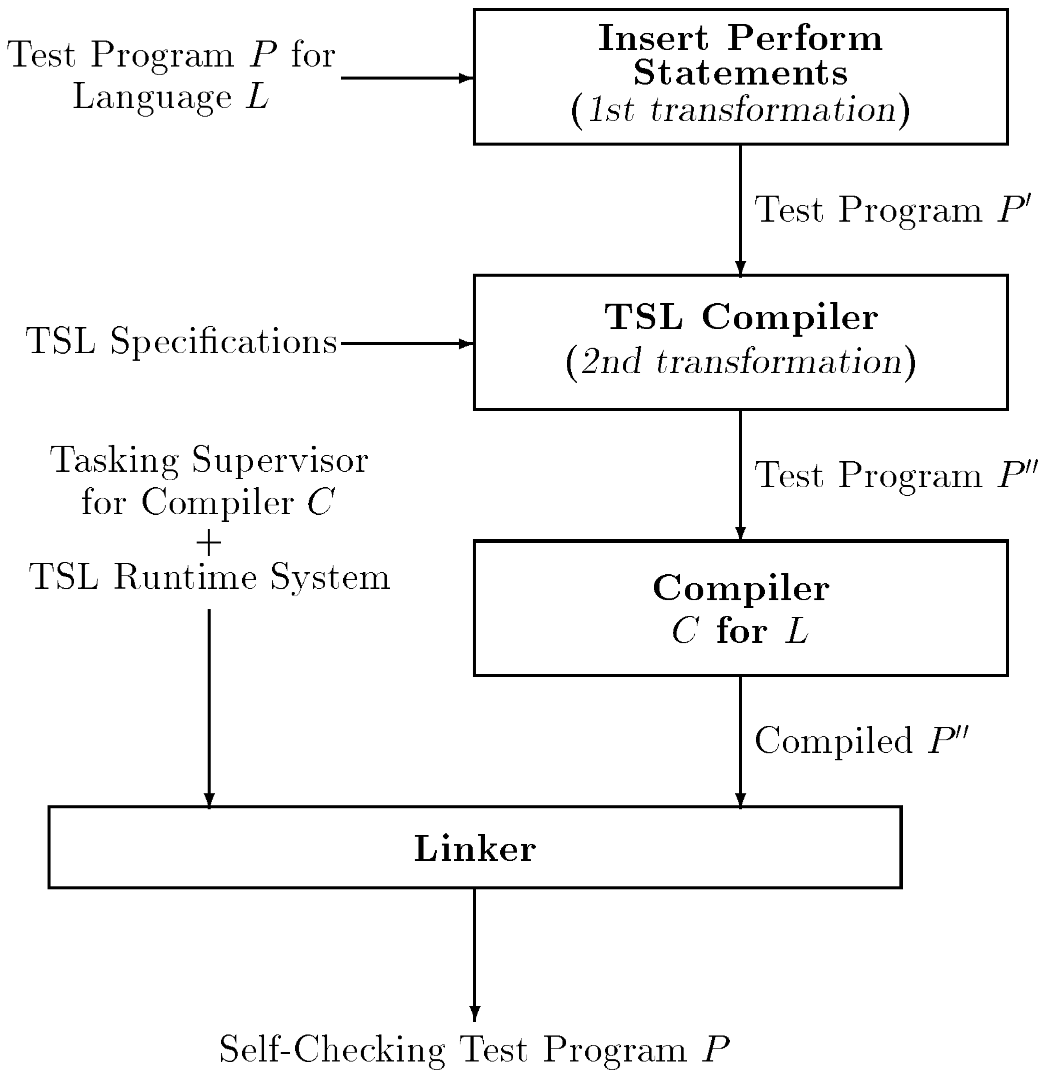
Test Program P for Language L
Insert Perform Statements (1st transformation)
Test Program P'
TSL Specifications
TSL Compiler (2nd transformation)
Test Program P''
Tasking Supervisor for Compiler C + TSL Runtime System
Compiler C for L
Compiled P''
Linker
Self-Checking Test Program P

Figure 58: Transformation of Test Programs for Any Tasking Supervisor.

Once the test program is transformed it may be executed, and the execution is then checked for consistency with the specifications. A violated specification indicates the existence of a bug in the Supervisor. Since each specification is associated with a single tasking construct, and since each Supervisor subprogram implements a single tasking construct, the violated specification indicates which subprogram most likely contains the bug. If more detailed information is desired, however, TSL specifications may be written and checked at the Supervisor code level.

6.5 Comparison With Other Approaches

Chapter 2 described several techniques and tools that are available for testing concurrent software, and the chapter demonstrated the superiority of automated specification-based methods over traditional debugger-based methods. In this section, the methodology described in this chapter is compared to two other approaches to testing a language implementation for consistency with the language semantics.

The Ada Joint Program Office (AJPO) requires the validation of every commercial Ada compiler, and it has created the Ada Compiler Validation Capability (ACVC) for this purpose [Ada87]. An Ada compiler is validated if it correctly compiles and/or executes a suite of test programs which currently[1] number over 3000 for the full language and around 300 for the tasking features. Each test program contains checking code that indicates its success or failure in executing its intended behavior. Although the ACVC has produced the strictest certification of compilers for any language, many of the validated Ada compilers are nevertheless of unsatisfactory quality.

As mentioned in Section 2.3.3, Klarund described a similar approach which was designed to overcome a major difficulty with the ACVC, namely the need to manually instrument each test program with checking code [Kla85]. In Klarund's approach, the behavior of each test program is formally specified in a variant of temporal logic; the specifications are then automatically transformed into checking code for runtime validation.

The methodology described in this chapter has the major advantage of the above two approaches, namely that it may be used to validate *any* implementation of Ada. However, there are many disadvantages of the above approaches that are overcome by the TSL-based methodology. The major flaw in both of the above approaches is that the behavior of both

[1]ACVC Version 1.9, August 1987.

the supervisor *and the test programs* is validated. If an error is detected, the checking code does not indicate which part of the system caused the error, the supervisor or the test program; in interpreting a report of incorrect behavior, there is a tacit *assumption* that the test program behaved correctly and that the error is inside the supervisor.

There are additional disadvantages to these other approaches:

- The validation suite is equivalent to the specification of a small subset of the semantically legal event sequences. Furthermore, each validation test is contrived to generate a legal sequence in a predictable and deterministic manner.
- It is impossible to automate the conversion of arbitrary Ada programs to a form suitable for testing; each program that is to be used as test input must be hand-instrumented either with checking code or with a formal specification of its behavior.
- The validation tests provide little information to directly relate a test failure to an implementation bug.

In contrast, the methodology of this chapter overcomes the above disadvantages:

- Because *only* the event sequencing of tasking statements is monitored during testing, a violated specification is equivalent to a bug in the supervisor, regardless of whether or not the test program behaved correctly.
- The TSL specifications completely describe *all* legal tasking event sequences.
- Every Ada program may be used as test input.
- A violated specification explicitly characterizes a bug in the supervisor.

By using formal specifications to describe the correct behavior of a tasking supervisor, and by automating the detection of incorrect behavior, a great deal of guesswork, intuition, inspiration and magic is removed from supervisor testing.

Chapter 7

Conclusions

This thesis has addressed two fundamental problems in the implementation of concurrent programming languages. First, the thesis described principles for designing a distributed tasking supervisor for a concurrent language. Second, the thesis described a methodology for verifying the consistency of a supervisor with the semantics of the language. The contributions of the thesis to the field of concurrent programming are detailed in the first section of this chapter. In addition, the second section of the this chapter suggests other avenues of exploration based on the results that were obtained.

7.1 Contributions of the Dissertation

The results described in this thesis are based on the design, implementation and verification of a distributed tasking supervisor for the Ada programming language. Many of the results can be generalized for application to other concurrent programming languages. This thesis has made the following contributions to the technology of concurrent programming languages.

First, the thesis presented principles for designing a distributed tasking supervisor. A tasking supervisor should be constructed in layers. The interface between the supervisor and application programs forms the topmost layer and is completely portable; a formal specification of the top-level interface provides a formal description of the correct use of the interface by the compiler. Portability is maximized by encapsulating machine dependencies within the lowest layer of the supervisor. The lowest layer is the implementation of an *abstract virtual machine* which provides an interface to a variety of low-level resources in a machine-independent manner. The easy porting of the Distributed Ada Supervisor from a Data General *Eclipse* to both a Sequent Balance *21000* and a Sun-3 workstation fully justifies the claims of portability.

Second, this thesis described a working distributed implementation of Ada tasking that supports parallel execution of Ada programs on a Sequent Balance *21000* multiprocessor. All previously described Ada supervisors and all currently validated Ada compilers are designed strictly as uniprocessor implementations of Ada. The only other known distributed supervisor is an unvalidated implementation that was custom-designed for the Sequent Balance *21000* architecture.

Third, this thesis described techniques for automatically checking a distributed supervisor for consistency with the semantics of the language it implements. Previous systems for testing and debugging concurrent software are unable to automatically distinguish incorrect execution behavior from correct execution behavior. Using the verification methodology described in this thesis, the tasking semantics of a concurrent language are formalized by specifying in TSL all legal sequences of tasking events that may occur during execution of application programs. By deriving the formal specifications from the tasking semantics of the language rather than specifying the internal behavior of the supervisor, the specifications are implementation-independent. Correctness of the supervisor is established by automatically checking the execution behavior of test programs for consistency with the TSL specifications. Correctness is implied by the satisfaction of all specifications during runtime checking, and a violated specification explicitly characterizes a bug in the supervisor. The specifications may also be used to construct a formal proof of correctness. The availability of spare processors dedicated to executing consistency checks makes feasible the construction of permanently self-checking supervisors.

Finally, several by-products of this thesis are suitable as a basis for standardization. In particular, the Anna specification of the interface to the Distributed Ada Supervisor should be of interest to the various committees and working groups involved in the definition of a standard Ada runtime environment. Additionally, the TSL specifications form the basis for a standard formal definition of Ada. A formal definition of the language would be a powerful tool for establishing even more rigorous standards of compiler quality than are possible with the current Ada Compiler Validation Capability (ACVC).

7.2 Future Work

The most immediately useful and important area for future research is the completion of the formal definition of Ada tasking in TSL. This can be accomplished through straightforward application of the principles described in Chapter 6.

There are also proposals for using the Distributed Ada Supervisor in the implementation of the formal specification languages being developed by the Program Analysis and Verification Group of the Computer Systems Laboratory at Stanford University. All of the proposals are directed toward reducing the interference of specification checking on the execution of application programs during runtime checking. In particular, the Supervisor will be used to implement automatic runtime checking of Anna specifications in parallel with the execution of the program whose behavior is being checked [RSL86]. It will also be used to build a distributed implementation of TSL; integration of the TSL Runtime System with the Supervisor would also simplify the generation of TSL events at runtime. Finally, *VAL* (VHDL Annotation Language) is being developed for the formal specification of VHDL hardware designs [AGH*87]; the Supervisor will be used for parallel design simulation and for checking the consistency of a simulated design with its VAL specification.

Other areas for future work in the short term include the completion of the Supervisor design to handle the unimplemented features of Ada tasking. A more general area for future work in the long term is the development of new paradigms for modeling concurrency. Both these areas are discussed in detail below.

7.2.1 Remaining Ada Supervisor Design Details

As was mentioned in Chapter 4, there are a few features of Ada tasking that are unaccounted for in the design of the Distributed Ada Supervisor. Implementation of these features would make the Supervisor more useful for real-time programs and other specialized concurrent software applications. Additionally, a realistic scheme must be incorporated into the Supervisor design for partitioning a program for efficient distributed execution. Ideally, the chosen scheme would divide the burden of partitioning between the compiler and the Supervisor. Several partitioning algorithms, both static and dynamic, have been proposed (e.g., Sarkar and Hennessy [SH86]). In addition, the Supervisor could be enhanced to support dynamic migration of tasks between processors; the enhancements described in [Ros87] would be required along with some atomic state update algorithms such as the ones described by

Theimer et al. [TLC85].

The implementation of task priorities requires that the Supervisor have the capability to preëmpt and time-slice task executions. The implementation of time-slicing requires the handling of asynchronous interrupts from timer hardware. As Ada compilers become more flexible, the incorporation of the necessary interrupt handlers into the Supervisor scheduler should be less of a problem than it was using the VERDIX compiler.

The compilation of rendezvous parameters for message-passing must efficiently handle all types of parameter values, including limited types. A more efficient scheme of parameter passing than the one described in Appendix E would use the facilities of an operating system that offers good distributed memory copying features, such as the V System.

Appendix E also outlined some algorithms for compiling some of the more difficult configurations of Ada selective waits, such as selective waits with multiple **accept** statements for the same entry or with multiple **delay** alternatives. Since the Supervisor allows at most one **accept** alternative per task entry and at most one **delay** alternative to be considered during execution of a selective wait, the compiled application program must choose a single alternative *before* calling the ACCEPT_BEGIN procedure. Thus, compilation of selective waits must take these special cases into account in a suitable fashion.

The message-passing protocol of the Supervisor supports the propagation of exceptions between tasks. However, no general transformations were given in Appendix E for catching an exception that is raised in the middle of a rendezvous and passing it to the calling task. The Supervisor currently only propagates TASKING_ERROR when an entry call is placed to a completed, terminated or abnormal task, and upon unsuccessful activation of a dependent task. Since Ada provides no general source-level facility for determining the name of an arbitrary propagated exception, exceptions and exception handlers must be compiled into simpler sequential constructs so that a general exception propagation facility can be incorporated into the Supervisor. Techniques described in [BR86] would be useful for this purpose.

Finally, the remaining unimplemented portion of the Ada tasking semantics is the management of shared global variables. Ada discourages the use of global variables for intertask communication, and indeed the semantics of shared variables in Ada are very weak. However, a minimum standard of behavior of shared variables is specified by the Ada Language Reference Manual and is required of all implementations. And despite Ada's discouragement of the use of shared global variables, programmers will nevertheless use them when more elegant solutions cannot be found.

Shared global variables can be accounted for in the Supervisor design in three ways. First, a protocol for remotely accessing shared variables must be incorporated into the Supervisor; a first implementation of this protocol could simply treat all shared global variables as tasks which have a READ entry and a WRITE entry. Second, the algorithms which partition a tasking program for distributed execution would consider data flow and data referencing patterns so as to assign tasks as often as possible to the same processor as the data they access. Third, the Supervisor could exploit new schemes for implementing *virtual shared memory* in distributed operating systems to make the management of shared global variables transparent.

7.2.2 New Models of Concurrency

This thesis has exploited the event-based nature of concurrent computation in the development of a methodology for automated consistency checking of just one class of concurrent software, distributed tasking supervisors. The event-based view of concurrent computation can be developed further by viewing the global event stream generated by the execution of a concurrent program as a string of a formal language.

For example, *path expressions* have been proposed as a formalism for expressing synchronization constraints of a concurrent program [CH74]; path expressions are simply a form of regular expression. It is highly probable that regular expressions are not powerful enough for expressing other general classes of concurrency constraints, and that a richer class of formal language would be needed, such as context-free languages [HU79]. But by viewing concurrent executions as strings of a formal language, a full array of automated techniques for language recognition (such as parsers) could be applied to consistency checking of a concurrent program.

Once concurrent programming becomes less of a black art and more of a rational science, it will be possible to apply many successful ideas from other areas of computer science to the development of concurrent programming languages. It is hoped that as the technology develops, concurrent programmers will recognize the value of formal specifications as an aid to both the design and testing of their software.

Appendix A

The Virtual Machine Interface

```
with GET_NODE_COUNT,    -- Function returning from command line the
                        -- number of nodes desired for execution; if
                        -- negative, no message tracing is performed.
     SYSTEM,            -- Predefined Ada package.
     CALENDAR;          -- Predefined Ada package.
package NET_SERVICES is
    type BYTE is range 0 .. 2 ** SYSTEM.STORAGE_UNIT - 1;
    for BYTE'SIZE use SYSTEM.STORAGE_UNIT;
    type BYTE_ARRAY is array (POSITIVE range <>) of BYTE;
--| where B1 : BYTE_ARRAY => B1'FIRST = 1;
    type BYTE_ARRAY_REF is access BYTE_ARRAY;

    TEMP_NODES        : constant INTEGER  := GET_NODE_COUNT;
    MAX_NODES         : constant NATURAL  := abs TEMP_NODES;
    MESSAGE_TRACING   : constant BOOLEAN  := TEMP_NODES > 0;

    -- Domains of numerical identifiers:
    type NODE_ID_TYPE is new NATURAL range 0 .. MAX_NODES - 1;
    type OS_PROCESS_ID_TYPE is new NATURAL;
    type UNIQUE_ID_TYPE is new NATURAL;

    -- Unique identifier generation:
```

```
    function  GET_UNIQUE_ID return UNIQUE_ID_TYPE;
    procedure GET_UNIQUE_ID (MY_ID : out UNIQUE_ID_TYPE);

    -- Returns pointer to activation record corresponding to the scope
    -- in which this function was called:
    function CURRENT_FRAME_POINTER return SYSTEM.ADDRESS;

    procedure TIME_OUT (DELAY_AMT : in DURATION);
--| where
--  Delay caller for DELAY_AMT:
--|      out (CALENDAR.CLOCK – in CALENDAR.CLOCK >= TIME_OUT);

    -- ASYNCHRONOUS SEND—caller not blocked. Send LENGTH bytes at
    -- address MSG from node SOURCE to node DEST:
    procedure SEND      (MSG     : in SYSTEM.ADDRESS;
                         LENGTH  : in NATURAL;
                         SOURCE  : in NODE_ID_TYPE;
                         DEST    : in NODE_ID_TYPE);

    -- SYNCHRONOUS RECEIVE—caller is blocked until message received.
    -- Receive message destined for node DEST. Sender is at node SOURCE;
    -- message is LENGTH bytes stored at address MSG:
    procedure RECEIVE   (MSG     : out SYSTEM.ADDRESS;
                         LENGTH  : out NATURAL;
                         SOURCE  : out NODE_ID_TYPE;
                         DEST    : in  NODE_ID_TYPE);



    -- Create an OS-level process for execution on a different CPU:
    function CREATE_OS_PROCESS
                    (PROC_ID        : in NATURAL;
                     PROC_NODE      : in NODE_ID_TYPE;
```

```
                  START_ADDR      : in SYSTEM.ADDRESS;
                  FRAME_PTR       : in SYSTEM.ADDRESS;
                  STACK_SIZE      : in POSITIVE;
                  PROC_PRIORITY   : in SYSTEM.PRIORITY :=
                                    SYSTEM.PRIORITY'FIRST)
                       return PROCESS_ID_TYPE;

   -- Terminate OS-level process number PID:
   procedure DESTROY_OS_PROCESS (PID : in PROCESS_ID_TYPE);

   -- Create a lightweight process for scheduling within the OS-level
   -- process of the caller:
   procedure CREATE_LOCAL_PROCESS
                   (PROC_ID         : in NATURAL;
                    PROC_NODE       : in NODE_ID_TYPE;
                    START_ADDR      : in SYSTEM.ADDRESS;
                    FRAME_PTR       : in SYSTEM.ADDRESS;
                    STACK_SIZE      : in POSITIVE;
                    PROC_PRIORITY   : in SYSTEM.PRIORITY :=
                                      SYSTEM.PRIORITY'FIRST);

   -- Suspend the lightweight process of the caller:
   procedure SUSPEND_LOCAL_PROCESS;

   -- Terminate the lightweight process of the caller:
   procedure TERMINATE_LOCAL_PROCESS;

   -- Declaration of LOCK type, similar to a semaphore;
   -- subprograms are self-explanatory:
   type LOCK is limited private;
   LOCK_IS_UNINITIALIZED : exception;

--: function IS_INITIALIZED (L : in LOCK) return BOOLEAN;
```

```
    function IS_LOCKED (L : in LOCK) return BOOLEAN;
--| where
--|     not IS_INITIALIZED (L) => raise LOCK_IS_UNINITIALIZED;

    procedure INITIALIZE (L : in out LOCK);
--| where
--|     out IS_INITIALIZED (L),
--|     out (not IS_LOCKED (L)),
--|     out (in IS_INITIALIZED (in L) -> L /= in L and not IS_INITIALIZED (in L));

    procedure FINALIZE (L : in out LOCK);
--| where
--|     out (not IS_INITIALIZED (L)),
--|     not IS_INITIALIZED (L) => raise LOCK_IS_UNINITIALIZED;

    procedure ACQUIRE (L : in out LOCK);
--| where
--|     out IS_LOCKED (L),
--|     not IS_INITIALIZED (L) => raise LOCK_IS_UNINITIALIZED;

    procedure CONDITIONAL_ACQUIRE (L : in out LOCK);
--| where
--|     out IS_LOCKED (L),
--|     out (SUCCESS <-> not in IS_LOCKED (in L)),
--|     not IS_INITIALIZED (L) => raise LOCK_IS_UNINITIALIZED;

    procedure RELEASE (L : in out LOCK);
--| where
--|     out (not IS_LOCKED (L)),
--|     not IS_INITIALIZED (L) => raise LOCK_IS_UNINITIALIZED;


--| axiom
--|     for all N1, N2 : NET_SERVICES'TYPE;
--|            U1     : UNIQUE_ID_TYPE =>
-- Function GET_UNIQUE_ID behaves exactly like procedure
-- GET_UNIQUE_ID, the former returning the out-value of
-- the latter's MY_ID parameter:
--|          N1.GET_UNIQUE_ID'OUT (MY_ID => U1).MY_ID =
--|              N1.GET_UNIQUE_ID,
-- GET_UNIQUE_ID returns the NATURAL numbers in a monotonically
-- increasing sequence:
--|          N1 [GET_UNIQUE_ID].GET_UNIQUE_ID = N1.GET_UNIQUE_ID + 1,
-- GET_UNIQUE_ID never returns the same value twice:
--|          N1 /= N2 <-> N1.GET_UNIQUE_ID /= N2.GET_UNIQUE_ID;

private
    -- Implementation of LOCK type:
    type LOCK_IMPL;   -- Defined in NET_SERVICES body.
    type LOCK is access LOCK_IMPL;
end NET_SERVICES;
```

Appendix B

Ada Supervisor Kernel Algorithms

B.1 The SEND Algorithm

```
procedure SEND_MESSAGE (MSG      : in out SUPERVISOR_MSG;
                        MY_NODE  : in     NODE_ID_TYPE) is
-- The MSG_SOURCE_NODE and CURRENT_SENDERS_MASTER
-- components of MSG are filled in by SEND_MESSAGE; all other
-- components of MSG must be initialized prior to calling
-- SEND_MESSAGE.
--
-- Assume visibility here of the following: low-level message SEND
-- procedure whose parameters are a message and a destination
-- node address; visibility of local and global task maps for this
-- supervisor copy; global task map subprogram NODE_ADDRESS, which
-- returns the node address of the input task; local task map subprogram
-- MASTER_TASK, which returns the name of the master of the input
-- (local) task.
begin
    MSG.MSG_SOURCE_NODE := MY_NODE;
    if MSG.MSG_DEST has an entry in the GLOBAL_TASK_MAP then
      if NODE_ADDRESS (MSG.MSG_DEST) = MY_NODE then
```

```
          -- MSG_DEST is a local task.
          Place MSG in an appropriate field of the local
                task map entry for MSG.MSG_DEST;
        else        -- MSG_DEST is not a local task.
          -- Forward message to MSG_DEST:
          SEND (MSG, NODE_ADDRESS (MSG.MSG_DEST));
        end if;
      else        -- MSG_DEST is not in the global task map.
        -- Set the CURRENT_SENDERS_MASTER field:
        MSG.CURRENT_SENDERS_MASTER :=
              MASTER_TASK (MSG.MSG_SOURCE);
        -- Find the closest remote ancestor of MSG_SOURCE:
        while NODE_ADDRESS (MSG.CURRENT_SENDERS_MASTER) =
                    MY_NODE loop
          MSG.CURRENT_SENDERS_MASTER :=
              MASTER_TASK (MSG.CURRENT_SENDERS_MASTER);
        end loop;
        SEND (MSG, NODE_ADDRESS (MSG.CURRENT_SENDERS_MASTER));
      end if;
end SEND_MESSAGE;
```

B.2 The Routing and Receiving Algorithm

```
task RECEIVE_AND_PROCESS_MESSAGES;

task body RECEIVE_AND_PROCESS_MESSAGES is
-- Assume visibility here of the following: low-level message SEND
-- procedure whose parameters are a message and a destination node
-- address; low-level message RECEIVE procedure whose parameter
-- is a pointer to a message; visibility of local and global task
-- maps for this supervisor copy; global task map subprogram
-- NODE_ADDRESS, which returns the node address of the
-- input task; local task map subprogram MASTER_TASK, which
-- returns the name of the master of the input (local) task; local
-- constant MY_NODE.
    TEMP_MSG : SUPERVISOR_MSG_REF;
begin
    loop
        -- RECEIVE blocks until a message arrives:
        RECEIVE (TEMP_MSG);
        Update the global map with an entry for TEMP_MSG.MSG_SOURCE
            using the value of TEMP_MSG.MSG_SOURCE_NODE;
        if TEMP_MSG.MSG_DEST has an entry in the global map then
          if (NODE_ADDRESS (TEMP_MSG.MSG_DEST)) =
                        MY_NODE then
             -- MSG_DEST is a local task.
             Place TEMP_MSG.all in an appropriate field of the
                 local task map entry for TEMP_MSG.MSG_DEST;
          else      -- MSG_DEST is not a local task.
             -- Forward the message to MSG_DEST:
             SEND (TEMP_MSG.all,
                    NODE_ADDRESS (TEMP_MSG.MSG_DEST));
          end if;
        else      -- MSG_DEST is not in the global task map.
          -- Update the CURRENT_SENDERS_MASTER field:
          TEMP_MSG.CURRENT_SENDERS_MASTER :=
               MASTER_TASK (TEMP_MSG.CURRENT_SENDERS_MASTER);
          -- Find the closest remote ancestor of MSG_SOURCE:
          while NODE_ADDRESS (TEMP_MSG.CURRENT_SENDERS_MASTER) =
                        MY_NODE loop
             TEMP_MSG.CURRENT_SENDERS_MASTER :=
               MASTER_TASK (TEMP_MSG.CURRENT_SENDERS_MASTER);
          end loop;
          SEND (TEMP_MSG.all,
                 NODE_ADDRESS (TEMP_MSG.CURRENT_SENDERS_MASTER));
        end if;
    end loop;
end RECEIVE_AND_PROCESS_MESSAGES;
```

Appendix C

Ada Supervisor Messages

C.1 The Supervisor Message Type

```
type MESSAGE_CLASS is
      (CALL_MSG,              -- Entry call.
       RETURN_MSG,            -- End rendezvous.
       ACCEPT_MSG,            -- Accept statement.
       CONFIRM_MSG,           -- Begin rendezvous.
       ABORT_CALL_MSG,        -- Cancel entry call.
       ATTR_REQ_MSG,          -- Request attribute value.
       ATTR_REPLY_MSG,        -- Returned attribute value.
       TERM_CONFIRM_MSG,      -- Permission to execute a terminate alternative.
       EXCEPTION_MSG,         -- Propagate exception.
       DEPENDENT_MSG,         -- Remote activation of dependent.
       NEW_TASK_MSG,          -- Remote scheduling of dependent.
       TYPED_TASK_MSG,        -- Declaration of an object of a named task type.
       ACTIVE_MSG,            -- End of task activation.
       COMPLETE_MSG,          -- Task termination.
       AT_TERM_MSG,           -- Waiting at a terminate alternative.
       ELABORATE_MSG,         -- Permission to begin elaboration.
       ABORT_TASK_MSG,        -- Abort a task.
       ABORT_REPLY_MSG,       -- Task was aborted.
       TERM_REQ_MSG,          -- Request permission to execute a
```

```
                              -- terminate alternative.
    DEPENDENT_REPLY_MSG -- Reply to DEPENDENT_MSG.
   );

type SUPERVISOR_MESSAGE_TYPE (CLASS : MESSAGE_CLASS := CALL_MSG;
                              BYTE_COUNT : NATURAL := 0) is
   record
      MSG_SOURCE                 : TASK_NAME;
      MSG_DEST                   : TASK_NAME;
      MSG_SOURCE_NODE            : NET_SERVICES.NODE_ID_TYPE;
      CURRENT_SENDERS_MASTER     : TASK_NAME;
      case CLASS is
         when CALL_MSG =>
            CALL_E_DEST : ENTRY_NAME;
            CALL_ID     : NET_SERVICES.UNIQUE_ID_TYPE;
         when ACCEPT_MSG =>
            ACCEPT_ID : NET_SERVICES.UNIQUE_ID_TYPE;
         when CONFIRM_MSG =>
            CONFIRM_E_DEST : ENTRY_NAME;
            CONFIRM_ID     : NET_SERVICES.UNIQUE_ID_TYPE;
            CONFIRM_DATA   : PARAM_LIST (1 .. BYTE_COUNT);
         when RETURN_MSG =>
            RETURN_E_SOURCE : ENTRY_NAME;
            RETURN_DATA     : PARAM_LIST (1 .. BYTE_COUNT);
         when ABORT_CALL_MSG =>
            ABORT_E_DEST : ENTRY_NAME;
            ABORT_ID     : NET_SERVICES.UNIQUE_ID_TYPE;
         when ATTR_REQ_MSG =>
            CALLABLE_MSG : BOOLEAN; -- TRUE  ≡ CALLABLE,
                                    -- FALSE ≡ TERMINATED.
         when ATTR_REPLY_MSG =>
            ATTR_VALUE : BOOLEAN;
         when TERM_CONFIRM_MSG =>
            CONFIRMED : BOOLEAN;
            when EXCEPTION_MSG =>
                EXCEPTION_NAME : STRING (1 .. BYTE_COUNT);
            when DEPENDENT_MSG =>
                DEPENDENT_NAME : TASK_NAME;
                DEPENDENT_NODE : NET_SERVICES.NODE_ID_TYPE;
                MASTER_SCOPE   : SCOPE_NUMBER_TYPE;

            when NEW_TASK_MSG =>
                NEW_TASK_ENTRY_ADDRESS : SYSTEM.ADDRESS;
                NEW_TASK_FRAME_PTR     : SYSTEM.ADDRESS;
                NEW_TASK_STACK_SIZE    : POSITIVE;
                NEW_TASK_ENTRY_COUNT   : ENTRY_NAME;
                NEW_TASK_PRIORITY      : SYSTEM.PRIORITY;
                NEW_TASK_MASTER        : TASK_NAME;
                NEW_TASK_MASTER_NODE   : NET_SERVICES.NODE_ID_TYPE;
            when TYPED_TASK_MSG =>
                TYPED_TASK_NAME : TASK_NAME;
                TYPED_TASK_NODE : NET_SERVICES.NODE_ID_TYPE;
            when others =>
                null;
        end case;
    end record;

--| where M : SUPERVISOR_MESSAGE_TYPE =>
--  Messages are never sent from a task to itself:
--|         M.MSG_SOURCE /= M.MSG_DEST and
--  A task can never be its own dependent:
--|         if M.CLASS = DEPENDENT_MSG then
--|            M.DEPENDENT_NAME /= M.MSG_SOURCE and
--|            M.DEPENDENT_NAME /= M.MSG_DEST
--  EXCEPTION_NAME is a legal Ada identifier:
--|         elsif M.CLASS = EXCEPTION_MSG then
--|            M.BYTE_COUNT > 0 and
--|            (M.EXCEPTION_NAME (1) in 'a' .. 'z' or
--|             M.EXCEPTION_NAME (1) in 'A' .. 'Z') and
--|           (for all I : 1 .. BYTE_COUNT =>
--|               M.EXCEPTION_NAME (I) = '_' or
--|               M.EXCEPTION_NAME (I) in '0' .. '9' or
--|               M.EXCEPTION_NAME (I) in 'a' .. 'z' or
--|               M.EXCEPTION_NAME (I) in 'A' .. 'Z') and
--|           (for all I : 1 .. BYTE_COUNT - 1 =>
--|               M.EXCEPTION_NAME (I .. I + 1) /= "__") and
--|           M.EXCEPTION_NAME (BYTE_COUNT) /= '_'
--|         else
--|           TRUE
--|         end if;
```

C.2 Task Execution Status Types

```
type TASK_STATUS is
    -- The meanings of these should be obvious except where noted:
    (UNBORN, RUN_DCL, WAIT_TO_ACTIVATE, RUN_BODY, COMPLETED,
     ABNORMAL, TERMINATED, DELAYED,
     RUN_CRITICAL,          -- i.e., executing an accept statement.
     WAIT_FOR_RETURN,       -- i.e., wait for end of rendezvous.
     WAIT_FOR_ENTRY, WAIT_FOR_ENTRY_OR_TERMINATE, WAIT_FOR_ACCEPT,
     WAIT_FOR_CONFIRMATION, WAIT_FOR_TERMINATE_CONFIRMATION);

type DEPENDENT_STATUS is    -- These values are described in Section 4.5:
    (AWAITING_ACTIVATION, ACTIVATED, AWAITING_TERMINATION, TERMINATED,
     AWAITING_TERMINATION_CONFIRMATION, PENDING_TERMINATION);
```

Appendix D

A Distributed Algorithm for the Terminate Alternative

The agent serving a task T that is executing a selective wait with an open **terminate** alternative executes the following algorithm; assume that M is the master of T.

1. If there are no waiting calls to the selected entries, and if no dependent of T is either AWAITING_ACTIVATION or ACTIVATED, then send an AT_TERM_MSG to M. Change the status of T to WAIT_FOR_ENTRY_TERMINATE. T is now awaiting CALL_MSGs from callers and a TERM_CONFIRM_MSG from M.

2. If the status of T is WAIT_FOR_ENTRY_TERMINATE, and if a CALL_MSG (i.e., an entry call) arrives at an entry which T is selecting, then send a TERM_REQ_MSG to M, change the status of T to WAIT_FOR_TERMINATE_CONFIRMATION, and await a TERM_CONFIRM_MSG from M.

3. Whenever a TERM_CONFIRM_MSG is received from M, then

   (a) If the CONFIRMED component is set to TRUE, change the status of T to COMPLETED and indicate permission to execute the **terminate** alternative; this indication is transmitted to T in **out**-mode parameters of the supervisor procedure ACCEPT_BEGIN (see Section 4.6.9).

   (b) If the CONFIRMED component is set to FALSE, then it is a consequence of the algorithm that the status of T must be WAIT_FOR_TERMINATE_CONFIRMATION. Thus, T is waiting to accept a call that arrived. Change the status of

T to WAIT_FOR_CONFIRMATION and send an ACCEPT_MSG to the calling task.

In addition, the supervisor copies perform the following algorithm at all times:

1. If a task M becomes COMPLETED, and if no dependent of M is AWAITING_ACTIVATION or ACTIVATED, then send a TERM_CONFIRM_MSG with CONFIRMED set to TRUE to all dependents that are either AWAITING_TERMINATION_CONFIRMATION or AWAITING_TERMINATION, and change the status of these dependents to PENDING_TERMINATION.

2. If a task M receives a COMPLETE_MSG from a dependent task T (indicating termination of T), and if the status of T in M's local map entry was previously ACTIVATED, and if no other dependent of M is AWAITING_ACTIVATION or ACTIVATED then

   (a) If the status of M is COMPLETED, send a TERM_CONFIRM_MSG to all dependents that are AWAITING_TERMINATION or AWAITING_TERMINATION_CONFIRMATION, and change the status of such dependents to PENDING_TERMINATION. In this message, CONFIRMED must be set to TRUE.

   (b) Otherwise, if the status of M is WAIT_FOR_ENTRY_TERMINATE, then send an AT_TERM_MSG to M's master.

3. If a task M receives an AT_TERM_MSG from a dependent task T, change the status of T in M's local map entry to AWAITING_TERMINATION. If no dependent of M is AWAITING_ACTIVATION or ACTIVATED then

   (a) If the status of M is COMPLETED, send a TERM_CONFIRM_MSG to all dependents that are AWAITING_TERMINATION or AWAITING_TERMINATION_CONFIRMATION, and change the status of such dependents to PENDING_TERMINATION. In this message, CONFIRMED must be set to TRUE.

   (b) Otherwise, if the status of M is WAIT_FOR_ENTRY_TERMINATE, then send an AT_TERM_MSG to M's master.

4. If a task M receives a TERM_REQ_MSG from a dependent task T then change the status of T in M's local map entry to AWAITING_TERMINATION_CONFIRMATION and

(a) If the status of M is WAIT_FOR_ENTRY_TERMINATE, and if no dependent of M is AWAITING_ACTIVATION or ACTIVATED, send a TERM_REQ_MSG to the master of M and await a TERM_CONFIRM_MSG in reply. Change the status of M to WAIT_FOR_TERMINATE_CONFIRMATION.

(b) Otherwise, if the status of M is WAIT_FOR_TERMINATE_CONFIRMATION, then await a TERM_CONFIRM_MSG reply from the master of M.

(c) Otherwise, if the status of M is COMPLETED, send a TERM_CONFIRM_MSG with CONFIRMED set to TRUE to all dependents that are AWAITING_TERMINATION_CONFIRMATION or AWAITING_TERMINATION, and change the status of all such dependents to PENDING_TERMINATION.

(d) Otherwise send a TERM_CONFIRM_MSG with CONFIRMED set to FALSE to all dependents that are AWAITING_TERMINATION_CONFIRMATION, and change the status of all such dependents to ACTIVATED.

5. If a TERM_CONFIRM_MSG is received from the master of a task M, then

(a) If the value of the CONFIRMED component is set to TRUE, then change the status of M to COMPLETED. Send a TERM_CONFIRM_MSG with CONFIRMED set to TRUE to each dependent that is either AWAITING_TERMINATION_CONFIRMATION or AWAITING_TERMINATION, and change the status of these dependents in M's local map entry to PENDING_TERMINATION.

(b) If the value of the CONFIRMED component is set to FALSE, then send a TERM_CONFIRM_MSG, with the CONFIRMED component set to FALSE, to each dependent that is AWAITING_TERMINATION_CONFIRMATION, and change the status of these dependents to ACTIVATED. Change the status of M to WAIT_FOR_CONFIRMATION if its status was WAIT_FOR_TERMINATE_CONFIRMATION, and continue with the rendezvous message sequence with M's caller.

Appendix E

The Supervisor Preprocessor

Although no automatic preprocessor was implemented for transforming application programs to use the Distributed Ada Supervisor, it can be implemented by straightforward application of the principles described in [Ros85]. This appendix describes the source transformations that were performed manually to enable testing the Supervisor.

The transformations are described as *Before/After* examples in the manner of Helmbold and Luckham [HL83]. They are described in order of syntax presentation in the Ada Language Reference Manual. The transformations are described as they would be applied to tasks and subprograms other than the environment task or main program. Each Supervisor subprogram has formal parameters called MY_NAME and MY_NODE. For transformations within the declarative regions of tasks and subprograms other than the environment task and main program, the values that are passed as actual parameters for these formal parameters are always MY_NAME and MY_NODE, respectively. For transformations directly within the declarative regions of the environment task (e.g., for the declaration of a library task) or main program (e.g., for the declaration of a dependent task of the main program), the actual parameters would instead be MAIN_ID and MAIN_NODE, respectively. In addition, italicized names are used in the transformations in some cases to informally describe values that are passed as actual parameters to Supervisor subprograms. Finally, named parameter association is used for all subprogram calls within the transformation descriptions as a reminder of the purpose of each parameter.

9.1 Task Specifications and Task Bodies

Single task declarations and task type declarations are transformed to procedure declarations; task bodies are transformed to procedure bodies. The Supervisor "calls" the procedures to activate them as tasks. First consider the transformation of single tasks. The task declaration is transformed to a procedure declaration followed by a call to the Supervisor function CHILD_TASK, which generates an identifier for the task; the value of this identifier is assigned to a variable that is used by other tasks to refer to the newly declared task:

```
-- Original source text—SINGLE TASK DECLARATION:
task T is
     ENTRY_LIST;
end T;

-- Transformed source text:
procedure T_BODY
  (MY_NAME        : in TASK_NAME;
   DUMMY_NODE   : in NET_SERVICES.NODE_ID_TYPE);

T : constant TASK_NAME :=
    CHILD_TASK
      (MY_NAME              => MY_NAME,
       MASTER_NAME          => MY_NAME,
       ENTRY_COUNT          => CARDINALITY_OF_ENTRY_LIST,
       MASTER_SCOPE         => CURRENT_SCOPE,
       DEP_ENTRY_ADDR       => T_BODY'ADDRESS,
       GLOBAL_FRAME_PTR => MY_FRAME_POINTER,
       MY_NODE              => MY_NODE);
```

As will always be the case, the original name of a task will be preserved by the transformations so that no "using" occurrences of a task name need be transformed. For this reason, the name of the body of a single task will have "*_BODY*" appended to the original task name.

In determining the cardinality of a list of entries, each single entry is counted once and each member of an entry family is counted once. For example, the cardinality of the entry list

```
entry A;
entry B (CHARACTER range 'a' .. 'z');
entry C (1 .. 10);
entry D;
```

can be computed by the expression

```
1 +                                            -- Entry A.
(BOOLEAN'POS ('z' > 'a') *                     -- Family B.
        ((CHARACTER'POS ('z') -
CHARACTER'POS ('a')) + 1)) +
(BOOLEAN'POS (10 > 1) * ((10 - 1) + 1)) +      -- Family C.
1                                              -- Entry D.
```

Note that if an entry family declaration contains an empty range of index values, the BOOLEAN'POS evaluation will cause the range size to be multiplied by zero, giving a cardinality of zero for the empty family.

The transformation of a task type declaration is similar to the transformation of a single task declaration, the difference being that a frame pointer variable is declared instead of a task identifier.

```
-- Original source text—TASK TYPE DECLARATION:
task type TT is
     ENTRY_LIST;
end TT;

-- Transformed source text:
procedure TT (MY_NAME       : in TASK_NAME;
              DUMMY_NODE    : in NET_SERVICES.NODE_ID_TYPE);

TT_FRAME_POINTER : SYSTEM.ADDRESS
                       renames MY_FRAME_POINTER;
```

The frame pointer is the address of the activation record for the scope enclosing the declaration of the body of the task type; this frame corresponds to the innermost scope that is visible inside the body.

Finally, a task body is transformed to a procedure body with several new variables and supervisor calls added:

```
-- Original source text—TASK BODY DECLARATION:
task body T is
     DECLARATIONS;
begin
     STATEMENTS;
exception
     when SOME_EXCEPTION =>
           SOME_EXCEPTION_HANDLER;
end T;

-- Transformed source text:
procedure T (MY_NAME        : in TASK_NAME;
             DUMMY_NODE     : in NET_SERVICES.NODE_ID_TYPE) is
  MY_NODE : NET_SERVICES.NODE_ID_TYPE :=
    ELABORATE_TASK (MY_NAME => MY_NAME,
                    MY_NODE => DUMMY_NODE);
  MY_FRAME_POINTER : SYSTEM.ADDRESS :=
      NET_SERVICES.CURRENT_FRAME_POINTER;
  CURRENT_SCOPE : constant SCOPE_NUMBER_TYPE := 0;

  ACCEPTED, ABORTED          : BOOLEAN;
  IN_PARAMS, OUT_PARAMS      : PARAM_LIST_REF;
  ACCEPT_INDEX               : NATURAL
      range 1 .. CARDINALITY_OF_ENTRY_LIST + 1;

  DECLARATIONS;
begin
  ACTIVATE_TASK (MY_NAME => MY_NAME,
                 ABORTED => ABORTED,
                 MY_NODE => MY_NODE);
  if ABORTED then
```

```
      goto END_OF_SCOPE_0;
    end if;
    STATEMENTS;
  <<END_OF_SCOPE_0>>
    TERMINATE_TASK (MY_NAME => MY_NAME,
                    ABORTED => ABORTED,
                    MY_NODE => MY_NODE);
exception
  when SOME_EXCEPTION =>
         SOME_EXCEPTION_HANDLER;
         TERMINATE_TASK (MY_NAME => MY_NAME,
                         ABORTED => ABORTED,
                         MY_NODE => MY_NODE);
  when others =>
         TERMINATE_TASK (MY_NAME => MY_NAME,
                         ABORTED => ABORTED,
                         MY_NODE => MY_NODE);
end T;
```

The three calls introduced by this transformation are a call to ELABORATE_TASK (to synchronize the beginning of activation with the master reaching its **begin**), a call to ACTIVATE_TASK (to inform the master of the end of activation), and calls to TERMINATE_TASK (to terminate the thread of control). Notice that calls to TERMINATE_TASK are added to each exception handler, and an **others** handler is added if it does not otherwise exist. Except for the call to ELABORATE_TASK, MY_NAME and MY_NODE are parameters to every supervisor call made by the task. MY_FRAME_POINTER and CURRENT_SCOPE are used in the declaration of dependent tasks. ABORTED is used in most supervisor calls to test for abnormality. ACCEPTED, IN_PARAMS, OUT_PARAMS and ACCEPT_INDEX are used in transformed entry calls, **accept** statements and **select** statements; their use will be described further below.

9.2 Task Types and Task Objects

Declarations of tasks within object declarations are transformed to calls to CHILD_TASK; the transformations are shown assuming the existence of a previously declared task type called TT:

```
-- Original source text—DECLARATION OF TASK OBJECTS:
TT_OBJECT  : TT;
TT_ARRAY   : array (discrete_ranges) of TT;

type TT_REC is
    record
      TT_COMPONENT : TT;
    end record;
TT_RECORD : TT_REC;



-- Transformed source text:
TT_OBJECT : constant TASK_NAME :=
  CHILD_TASK
      (MY_NAME           => MY_NAME,
       MASTER_NAME       => MY_NAME,
       ENTRY_COUNT       => CARDINALITY_OF_ENTRY_LIST,
       MASTER_SCOPE      => CURRENT_SCOPE,
       DEP_ENTRY_ADDR    => TT'ADDRESS,
       GLOBAL_FRAME_PTR  => TT_FRAME_POINTER,
       MY_NODE           => MY_NODE);

TEMP_TT_ARRAY : constant
        TASK_NAME_ARRAY (1 .. CARDINALITY_OF_TT_ARRAY) :=
  CHILD_TASK
      (MY_NAME           => MY_NAME,
       MASTER_NAME       => MY_NAME,
       DEPENDENT_COUNT   => CARDINALITY_OF_TT_ARRAY,
          ENTRY_COUNT       => CARDINALITY_OF_ENTRY_LIST,
          MASTER_SCOPE      => CURRENT_SCOPE,
          DEP_ENTRY_ADDR    => TT'ADDRESS,
          GLOBAL_FRAME_PTR  => TT_FRAME_POINTER,
          MY_NODE           => MY_NODE);
TT_ARRAY : constant array (discrete_ranges) of
             TASK_NAME := REMAP (TEMP_TT_ARRAY);

type TT_REC is
   record
      TT_COMPONENT : TASK_NAME;
   end record;

TT_RECORD : TT_REC := (TT_COMPONENT =>
  CHILD_TASK
         (MY_NAME           => MY_NAME,
          MASTER_NAME       => MY_NAME,
          ENTRY_COUNT       => CARDINALITY_OF_ENTRY_LIST,
          MASTER_SCOPE      => CURRENT_SCOPE,
          DEP_ENTRY_ADDR    => TT'ADDRESS,
          GLOBAL_FRAME_PTR  => TT_FRAME_POINTER,
          MY_NODE           => MY_NODE));
```

In the transformation above, the cardinality of TT_ARRAY is equal to the product of the cardinalities of the individual *discrete_ranges*; these values may be calculated as was done for the cardinality of entry families. The *REMAP* operation can be any one-to-one mapping of values from the one-dimensional array TEMP_TT_ARRAY to the multi-dimensional array TT_ARRAY. The *REMAP* operation is necessitated by the fact that it is much more efficient to initiate all of the components of TT_ARRAY with a single call to CHILD_TASK, which can initiate the tasks in parallel.

The transformations for objects and record subcomponents of named array types are identical to the transformation shown above for the object TT_ARRAY of an anonymous array type. Notice that in the case of objects and subcomponents of a task type, the activating thread of control is also the master thread of control; thus, MY_NAME is used for the first two parameters. The innermost enclosing scope for the new tasks is accessible using TT_FRAME_POINTER, which would have been declared for type TT as in the previous section.

Allocators of an access type whose designated subtype is a task type are also transformed to calls to the CHILD_TASK function, while such access types themselves are modified to have TASK_NAME as their designated subtype. The transformations shown below again assume the existence of a previously declared task type called TT:

```
-- Original source text—ACCESS TYPES DESIGNATING A
--                      TASK TYPE:
declare
  type TT_ACC is access TT;
  TT_ACCESS : TT_ACC;
begin
  TT_ACCESS := new TT;
end;

-- Transformed source text:
declare
  type TT_ACC is access TASK_NAME;
  TT_ACC_MASTER : TASK_NAME renames MY_NAME;
  TT_ACC_SCOPE  : SCOPE_NUMBER_TYPE
                  renames CURRENT_SCOPE;

  TT_ACCESS : TT_ACC;
begin
  TT_ACCESS := new TASK_NAME'
    (CHILD_TASK
        (MY_NAME          => MY_NAME,
         MASTER_NAME      => TT_ACCESS_MASTER,
         ENTRY_COUNT      => CARDINALITY_OF_ENTRY_LIST,
         MASTER_SCOPE     => TT_ACCESS_SCOPE,
         DEP_ENTRY_ADDR   => TT'ADDRESS,
         GLOBAL_FRAME_PTR => TT_FRAME_POINTER,
            MY_NODE          => MY_NODE,
            IMMEDIATE        => TRUE,
            PRE_BODY         => FALSE));
    end;
```

The transformation for the allocator is used for *all* allocators for the access type. The transformations extend in the obvious way for arrays and records containing components of such access types. If such an allocator appears prior to the declaration of the corresponding task body, the PRE_BODY parameter in the call to CHILD_TASK must be TRUE instead of FALSE.

9.3 Task Execution—Task Activation

No transformations are required.

9.4 Task Dependence—Termination of Tasks

The first section of this appendix showed how the outermost scope of a task body is transformed to use the Distributed Ada Supervisor. In general, every declarative region of an Ada program requires transformation so that the activation and termination of dependents of each such scope may be synchronized correctly according to the semantics of Ada. The following transformation applies to all subprogram bodies, packages and block statements; the transformation is shown for a block statement:

```
    -- Original source text—DECLARATIVE REGIONS:
    declare
      DECLARATIONS;
    begin
      STATEMENTS;
    end;

    -- Transformed source text:
    declare
```

```
  MY_FRAME_POINTER : SYSTEM.ADDRESS :=
      NET_SERVICES.CURRENT_FRAME_POINTER;
  CURRENT_SCOPE : constant SCOPE_NUMBER_TYPE :=
      ENTER_NEW_SCOPE (MY_NAME => MY_NAME,
                       MY_NODE => MY_NODE);
  IN_PARAMS, OUT_PARAMS : PARAM_LIST_REF;
  ACCEPTED              : BOOLEAN;
  ABORTED               : BOOLEAN;    -- Not inserted in outermost
                                      -- declarative parts of
                                      -- subprograms.

  DECLARATIONS;
begin
  ACTIVATE_TASK (MY_NAME => MY_NAME,
                 ABORTED => ABORTED,
                 MY_NODE => MY_NODE);
  if ABORTED then
    goto END_OF_SCOPE_n;
  end if;
  STATEMENTS;
 <<END_OF_SCOPE_n>>
  TERMINATE_TASK (MY_NAME => MY_NAME,
                  ABORTED => ABORTED,
                  MY_NODE => MY_NODE);
exception
  when SOME_EXCEPTION =>
        SOME_EXCEPTION_HANDLER;
        TERMINATE_TASK (MY_NAME => MY_NAME,
                        ABORTED => ABORTED,
                        MY_NODE => MY_NODE);
  when others =>
        TERMINATE_TASK (MY_NAME => MY_NAME,
                        ABORTED => ABORTED,
                          MY_NODE => MY_NODE);
end;
```

Note that the ABORTED variable is *not* inserted in the outermost declarative parts of subprograms, since subprograms are transformed to have an ABORTED parameter (see below).

In the case of a package, a dummy body is supplied if the package has no body; for the purposes of the transformation, the declarative part of the package begins at the beginning of the package visible part and ends at the end of the declarative part of the package body. The sequence of statements of the package body is the sequence of statements to which the transformation is applied.

Since the labels within a program unit (subprograms, packages, tasks, generic units) must be unique, END_OF_SCOPE labels are generated during transformation by sequentially numbering the scopes within the source text of the program unit; this is the purpose of the *n* value appearing in the label of the above transformation. The outermost scope is numbered zero.

Subprogram declarations, subprogram bodies (except for the main program) and subprogram calls require one extra transformation in their parameter parts, so that the task calling a subprogram can be identified correctly:

```
-- Original source text—SUBPROGRAM PARAMETERS:
declare
  ...
  procedure P (FORMALS);
  ...
begin
  ...
  P (ACTUALS);
  ...
end;

-- Transformed source text:
declare
  ...
  procedure P (MY_NAME  : in     TASK_NAME;
               MY_NODE  : in out NODE_ID_TYPE;
               ABORTED  : in out BOOLEAN;
               FORMALS);
  ...
begin
  ...
  P (MY_NAME => MY_NAME,
     MY_NODE => MY_NODE,
     ABORTED => ABORTED,
     FORMALS => ACTUALS);
  if ABORTED then
    goto END_OF_SCOPE_n;
  end if;
  ...
end;
```

The transformation of the parameter part of a subprogram body is identical to the transformation shown above for the subprogram declaration. Since functions may only have mode **in** formal parameters, the two **in out**-mode parameters of the procedure transformation appear as **in**-mode parameters of an access type for NODE_ID_TYPE and BOOLEAN. Renaming declarations are provided for the objects designated by these pointers so that no special form of the other transformations need be performed inside functions.

Scopes may also be exited through execution of an **exit** statement, a **return** statement or a **goto** statement. The transformation for such cases requires the insertion of one or more calls to TERMINATE_TASK. The transformation is shown below for a **return** statement, but it is identical to the transformation for **exit**s and **goto**s:

```
-- Original source text—SCOPE EXITS:
return X;

-- Transformed source text:
TERMINATE_TASK (MY_NAME => MY_NAME,
                ABORTED => ABORTED,
                     MY_NODE => MY_NODE);
...
TERMINATE_TASK (MY_NAME => MY_NAME,
                ABORTED => ABORTED,
                MY_NODE => MY_NODE);
return X;
```

The number of calls that are inserted is equal to the number of scopes that are being exited; this number can be determined from static examination of the source text.

9.5 Entries, Entry Calls, and Accept Statements

Transformation of simple entry calls and **accept** statements are special cases of the transformation of timed entry calls and selective waits and are shown in sections below.

9.6 Delay Statements, Duration and Time

A simple **delay** statement (i.e., one not appearing as the first statement of a **delay** alternative of a **select** statement) is transformed to a call to the DELAY_TASK procedure as follows:

```
-- Original source text—SIMPLE DELAY STATEMENT:
delay DELAY_AMOUNT;

-- Transformed source text:
DELAY_TASK (MY_NAME => MY_NAME,
            A_WHILE  => DELAY_AMOUNT,
            ABORTED => ABORTED,
            MY_NODE => MY_NODE);
if ABORTED then
  goto END_OF_SCOPE_n;
end if;
```

9.7 Select Statements

9.7.1 Selective Waits

A selective wait has k (where $k \geq 1$) **accept** alternatives. In addition a selective wait may optionally have one or more **delay** alternatives, or a single **terminate** alternative, or an **else** part; these three optional parts are mutually exclusive. Selective waits are transformed into a call to the Supervisor procedure ACCEPT_BEGIN, followed by a call to the ACCEPT_END procedure if an **accept** alternative is selected. For the transformation shown below, each of the $k+1$ alternatives is numbered consecutively from 1 to $k+1$; these numbers are the index numbers (the value of the formal parameter INDICES) which ACCEPT_BEGIN chooses from to indicate which alternative was selected. The chosen index number is then used as the switch value for a **case** statement which follows the call to ACCEPT_BEGIN. The transformation shown below is for a selective wait with a single **delay** alternative; assume that MY_NAME has declared n entries:

```
-- Original source text—SELECTIVE WAITS:
select
  when GUARD_1 =>
    accept E1 (FORMALS_1) do
      STATEMENTS_1_a;
    end E1;
    STATEMENTS_1_b;
or
...
or
  when GUARD_k =>
    accept Ek (FORMALS_k) do
      STATEMENTS_k_a;
    end Ek;
    STATEMENTS_k_b;
or
  when GUARD_other =>
    delay TIME_OUT;
    STATEMENTS_other;
```

```
   end select;

-- Transformed source text:
ACCEPT_BEGIN (MY_NAME   => MY_NAME,
              GUARDS    => (POSITION_OF_E1     => GUARD_1,
                            ...,
                            POSITION_OF_Ek     => GUARD_k,
                            N + 1              => GUARD_other,
                            OTHER_POSITIONS    => FALSE),
              INDICES   => (POSITION_OF_E1     => 1,
                            ...,
                            POSITION_OF_Ek     => k,
                            N + 1              => k + 1,
                            OTHER_POSITIONS    => 0),
              TIME_OUT  => TIME_OUT,
              TERM_ALT  => FALSE,
              IN_PARAMS => IN_PARAMS,
              INDEX     => ACCEPT_INDEX,
              ABORTED   => ABORTED,
              MY_NODE   => MY_NODE);
if ABORTED then
  goto END_OF_SCOPE_n;
end if;

case ACCEPT_INDEX is
   when 1 =>
      UNPACK (IN_PARAMS);
      STATEMENTS_1_a;
      ACCEPT_END (MY_NAME    => MY_NAME,
                  OUT_PARAMS => out_MODE_FORMALS_1_PACKED,
                  ABORTED    => ABORTED,
                  MY_NODE    => MY_NODE);
      if ABORTED then
        goto END_OF_SCOPE_n;
```

```
            end if;
            STATEMENTS_1_b;
        . . .
        when k =>
            UNPACK (IN_PARAMS);
            STATEMENTS_k_a;
            ACCEPT_END (MY_NAME      => MY_NAME,
                        OUT_PARAMS   => out_MODE_FORMALS_k_PACKED,
                        ABORTED      => ABORTED,
                        MY_NODE      => MY_NODE);
            if ABORTED then
              goto END_OF_SCOPE_n;
            end if;
            STATEMENTS_k_b;
        when k + 1 =>
            STATEMENTS_other;
end case;
```

The value *POSITION_OF_E* indicates the numerical position of entry E in the entry list for task MY_NAME according to the numbering scheme described earlier.

Any **accept** statement without a body is equivalent to one whose body is a single **null** statement. An unguarded alternative is equivalent to a guarded alternative whose guard is TRUE. A simple **accept** statement is equivalent to a selective wait with a single **accept** alternative whose guard is TRUE. An **else** part is equivalent to a **delay** alternative whose guard is TRUE and whose *TIME_OUT* is 0.0. A selective wait containing only **accept** alternatives is equivalent to a selective wait with the same **accept** alternatives plus a single **delay** alternative whose guard is FALSE.

For a selective wait with a **terminate** alternative, the TERM_ALT parameter is set to TRUE (instead of FALSE, as in the above forms of selective wait), and the TIME_OUT parameter is set to DURATION'LAST. The *STATEMENTS_other* for the **terminate** alternative is simple "**goto** END_OF_SCOPE_0".

As indicated in the above transformation, rendezvous parameter values are *packed* into arrays of bytes and *unpacked* from these arrays back into program objects. No general algorithms for this purpose was developed, but the general idea is illustrated below by the packing and unpacking of an INTEGER value:

```
-- Transformation for RENDEZVOUS PARAMETER PACKING:
declare

    I      : INTEGER := FOO;
    TEMP_I : NET_SERVICES.BYTE_ARRAY (1 .. 4);
begin
    -- Packing I into TEMP_I:
    TEMP_I := NET_SERVICES.BYTE (I / 16777216) &
                NET_SERVICES.BYTE ((I / 65536) mod 256) &
                  NET_SERVICES.BYTE ((I / 256) mod 256) &
                    NET_SERVICES.BYTE (I mod 256);

    -- Unpacking TEMP_I into I:
    I := INTEGER (TEMP_I (4)) +
          256 * (INTEGER (TEMP_I (3)) +
            256 * (INTEGER (TEMP_I (2)) +
            256 * (INTEGER (TEMP_I (1)))));
end;
```

The above transformation assumes that INTEGERs are stored as 32-bit values.

9.7.2 Conditional Entry Calls

A conditional entry call is treated as a timed entry call with a timeout value of zero; the transformation of timed entry calls is shown in the next section.

9.7.3 Timed Entry Calls

Simple entry calls, conditional entry calls and timed entry calls are all transformed to calls to the ENTRY_CALL procedure:

```
-- Original source text—ENTRY CALLS:
select
```

```
  T.E (ACTUALS);
  STATEMENTS_1;
or
  delay TIME_OUT;
  STATEMENTS_2;
end select;

-- Transformed source text:
ENTRY_CALL (MY_NAME          => MY_NAME,
            CALLEE_T_NAME    => T,
            CALLEE_E_NAME    => POSITION_OF_E,
            IN_PARAMS        => in_MODE_ACTUALS_PACKED,
            TIME_OUT         => TIME_OUT,
            OUT_PARAMS       => OUT_PARAMS,
            ACCEPTED         => ACCEPTED,
            ABORTED          => ABORTED,
            MY_NODE          => MY_NODE);
if ABORTED then
  goto END_OF_SCOPE_n;
elsif ACCEPTED then
  UNPACK (OUT_PARAMS);
  STATEMENTS_1;
else
  STATEMENTS_2;
end if;
```

A simple entry call is equivalent to a timed entry call with a *TIME_OUT* of DURATION'LAST; a conditional entry call is a timed entry call with a *TIME_OUT* of 0.0. The same comments that were made about packing and unpacking of parameters for a selective wait apply to timed entry calls also.

9.8 Priorities

An explicit priority for a task is declared in the entry list for the task; this value is passed as the last parameter of the call to CHILD_TASK:

```
-- Original source text—PRIORITIES:
task T is
     pragma PRIORITY (T_PRIORITY);
end T;

-- Transformed source text:
procedure T_BODY ( ... );        -- As before.
T : constant TASK_NAME :=
   CHILD_TASK (...,        -- As before.
                     DEP_PRIORITY => T_PRIORITY);
```

9.9 Task and Entry Attributes

References to task and entry attributes appear within expressions; these references are directly transformed to calls to corresponding Supervisor functions as follows:

```
-- Original source text—ATTRIBUTE EVALUATIONS:
... T'CALLABLE ...
... T'TERMINATED ...
... E'COUNT ...

-- Transformed source text:
... CALLABLE_ATTR (MY_NAME => MY_NAME,
                   C_NAME  => T,
                   MY_NODE => MY_NODE) ...
... TERMINATED_ATTR (MY_NAME => MY_NAME,
                     T_NAME  => T,
                     MY_NODE => MY_NODE) ...
... COUNT_ATTR (MY_NAME  => MY_NAME,
                MY_ENTRY => POSITION_OF_E,
                    MY_NODE    => MY_NODE) ...
```

Again, the value *POSITION_OF_E* indicates the numerical position of entry E in the entry list for task MY_NAME according to the numbering scheme described earlier.

9.10 Abort Statements

Abort statements are transformed into calls to the ABORT_TASK procedure; the VICTIMS parameter is supplied as an aggregate list of task ID variables:

```
-- Original source text—ABORT STATEMENT:
abort T1, T2, ..., Tn;

-- Transformed source text:
ABORT_TASKS (MY_NAME => MY_NAME,
             VICTIMS  => (T1, T2, ..., Tn),
             ABORTED => ABORTED,
             MY_NODE => MY_NODE);
if ABORTED then
  goto END_OF_SCOPE_n;
end if;
```

9.11 Shared Variables

Pragma SHARED is not supported.

Appendix F

Ada Tasking Events

F.1 Action Declarations for the Tasking Events

This section declares the TSL **actions** for the tasking events of Ada. Comments indicate where the actions are to be performed. The **perform** statements must be inserted into each test program as noted, possibly using an automatic preprocessor.

−− *Performed after each* **abort** *statement, once per aborted task:*
−−+ **action** ABORTED (A_TASK : **task**);

−− *Performed before each* **abort** *statement, once per aborted task:*
−−+ **action** BEGIN_ABORT (A_TASK : **task**);

−− *Performed before each activation of tasks which are designated by*
−− *access values; access types are identified by unique integers:*
−−+ **action** BEGIN_ACTIVATE_DYNAMIC (D_TASK : **task**; A_TYPE : INTEGER);

−− *Performed at the beginning of the declarative part of a task body:*
−−+ **action** BEGIN_ACTIVATION;

−− *Performed as the first statement of the sequence of statements*
−− *of a new declarative region:*
−−+ **action** BEGIN_BLOCK;

```
--   Performed before each simple delay statement. TIMESTAMP must be
--   set to CALENDAR.CLOCK in all corresponding perform statements:
--+ action BEGIN_DELAY (TIMEOUT : DURATION; TIMESTAMP : CALENDAR.TIME);

--   Performed as the first statement of each accept statement body:
--+ action BEGIN_RENDEZVOUS (CALLER : task; CALLED_E : entry);

--   Performed after each evaluation of attribute CALLABLE:
--+ action CHECKED_CALLABLE (C_TASK : task; VALUE : BOOLEAN);

--   Performed after each evaluation of attribute COUNT:
--+ action CHECKED_COUNT (C_ENTRY : entry; VALUE : NATURAL);

--   Performed after each evaluation of attribute TERMINATED:
--+ action CHECKED_TERMINATED (T_TASK : task; VALUE : BOOLEAN);

--   Performed immediately after evaluation of guards in a selective wait
--   with no else part and all false guards:
--+ action CLOSED_SELECT;

--   Performed as the last statement executed inside each task body:
--+ action COMPLETE;

--   Performed at each declaration of an access type whose designated subtype is
--   a task type; access types are identified by unique integers:
--+ action DECLARE_DYNAMIC_TASK_TYPE (A_TYPE : INTEGER; T_TYPE : task_type);

--   Performed at each declaration of a single task:
--+ action DECLARE_SINGLE_TASK (S_TASK : task);

--   Performed at each declaration of a task declared by an object declaration:
--+ action DECLARE_TASK_OBJECT (T_TASK : task; T_TYPE : task_type);
```

```
--   Performed at each declaration of a task type:
--+ action DECLARE_TASK_TYPE (T_TYPE : task_type);

--   Performed immediately following each delay statement. TIMESTAMP must
--   be set to CALENDAR.CLOCK in all corresponding perform statements:
--+ action DELAYED (TIMESTAMP : CALENDAR.TIME);

--   Performed by the supervisor each time an entry call is removed from a call queue
--   before being accepted:
--+ action DEQUEUE_CALL (C_TASK : task; Q_TASK : task; Q_ENTRY : entry);

--   Performed after each activation of tasks which are designated by
--   access values:
--+ action END_ACTIVATE_DYNAMIC (D_TASK : task; A_TYPE : INTEGER);

--   Performed immediately prior to beginning the sequence of
--   statements of a task body:
--+ action END_ACTIVATION;

--   Performed immediately following each entry call:
--+ action END_CALL (CALLEE : task; CALLED_E : entry);

--   Performed as the last statement of each accept statement body:
--+ action END_RENDEZVOUS (CALLER : task; CALLED_E : entry);

--   Performed by the supervisor each time an entry call arrives at a call queue:
--+ action ENQUEUE_CALL (C_TASK : task; Q_TASK : task; Q_ENTRY : entry);

--   Performed as the first statement upon entering a new declarative
--   region other than the outermost scope of a task:
--+ action ENTER_BLOCK;
```

```
--   Performed as the first statement after exiting a declarative region other
--   than the outermost scope of a task:
--+ action EXITED_BLOCK;


--   Performed within each exception handler:
--+ action HANDLE_EXCEPTION;

--   Performed by the supervisor when the abnormality of a task is registered:
--+ action MAKE_ABNORMAL (A_TASK : task);

--   Performed immediately after evaluation of guards in a selective wait with no else
--   part and whose only open alternatives are accept statements:
--+ action OPEN_SELECT;

--   Performed immediately before each raise statement:
--+ action RAISE_EXCEPTION;

--   Performed immediately after evaluation of guards in a selective wait
--   with an open delay alternative; the delay value passed is the
--   smallest of the open delay statements. TIMESTAMP must be set to
--   CALENDAR.CLOCK in all corresponding perform statements:
--+ action SELECT_DELAY (TIMEOUT : DURATION; TIMESTAMP : CALENDAR.TIME);

--   Performed immediately after evaluation of guards in a selective
--   wait with an else part:
--+ action SELECT_ELSE;

--   Performed immediately after evaluation of guards in a selective
--   wait with an open terminate alternative:
--+ action SELECT_TERMINATE;

--   Performed immediately before each simple accept statement:
```

```
--+ action SIMPLE_ACCEPT (ACCEPTED_E : entry);

--  Performed immediately before each simple entry call:
--+ action SIMPLE_CALL (CALLEE : task; CALLED_E : entry);

--  Performed immediately following each pragma PRIORITY statement:
--+ action SPECIFY_PRIORITY (P_TASK : task; P : PRIORITY);

--  Performed inside the supervisor when a task is completed and all its
--  dependents have terminated:
--+ action TERMINATED;

--  Performed immediately before each conditional or timed entry call
--  statement. TIMESTAMP must be set to CALENDAR.CLOCK in
--  all corresponding perform statements:
--+ action TIMED_CALL (TIMEOUT : DURATION; TIMESTAMP : CALENDAR.TIME);

--  Performed immediately following each shared variable update;
--  shared variables are identified by unique integers:
--+ action UPDATE (OBJ : INTEGER);
```

F.2 TSL Specification of the Tasking Events

This section lists the tasking events of Ada in the order in which they are mentioned in the Language Reference Manual. Both an informal English description and a TSL expression are given for each event.

9.1 Task Specifications and Task Bodies

1 Single Task Declaration

Ada: Task *T* declares dependent *D*.

TSL: T **performs** DECLARE_SINGLE_TASK (S_TASK => D);

2 Task Type Declaration

Ada: Task T declares task type TT.

TSL: T **performs** DECLARE_TASK_TYPE (T_TYPE => TT);

9.2 Task Types and Task Objects

1 Declaration of a Task By an Object Declaration

Ada: Task T declares dependent D of type TT.

TSL: T **performs** DECLARE_TASK_OBJECT (T_TASK => D, T_TYPE => TT);

2 Access Type Declaration

Ada: Task T declares access type PT referencing task type TT.

TSL: T **performs** DECLARE_DYNAMIC_TASK_TYPE (A_TYPE => PT, T_TYPE => TT);

9.3 Task Execution—Task Activation

1 Beginning of Activation

Ada: Task T begins activation.

TSL: T **performs** BEGIN_ACTIVATION;

2 End of Activation

Ada: Task T ends activation.

TSL: T **performs** END_ACTIVATION;

3 Begin Dynamic Activation

Ada: Task T will activate task P.**all**, where P is of type PT.

TSL: T **performs** BEGIN_ACTIVATE_DYNAMIC (D_TASK => P.**all**, A_TYPE => PT);

4 End Dynamic Activation

Ada: Task T activated task P.**all**, where P is of type PT.

TSL: T **performs** END_ACTIVATE_DYNAMIC (D_TASK => P.**all**, A_TYPE => PT);

9.4 Task Dependence—Termination of Tasks

1 Completion

Ada: Task *T* becomes completed.

TSL: T **performs** COMPLETE;

2 Termination

Ada: Task *T* becomes terminated.

TSL: T **performs** TERMINATED; (*Must be performed by tasking supervisor.*)

3 Block Entry

Ada: Task *T* enters a new nested block.

TSL: T **performs** ENTER_BLOCK;

4 Block Execution

Ada: Task *T* begins execution of the sequence of of statements of a new block.

TSL: T **performs** BEGIN_BLOCK;

5 Block Exit

Ada: Task *T* exited a nested block.

TSL: T **performs** EXITED_BLOCK;

9.5 Entries, Entry Calls, and Accept Statements

1 Entry Call Execution/Suspension of Caller

Ada: Task *C* calls task *T* at entry *E*.

TSL: C **performs** SIMPLE_CALL (CALLEE => T, CALLED_E => E);

2 Entry Call Arrival

Ada: A call from task *C* is placed on the queue of entry *E* at task *T*.

TSL: **any performs** ENQUEUE_CALL (C_TASK => C, Q_TASK => T,
Q_ENTRY => E); (*Must be performed by tasking supervisor.*)

3 Accept Execution/Suspension of Accepting Task

Ada: Task T executes an accept for entry E.

TSL: T **performs** SIMPLE_ACCEPT (ACCEPTED_E => E);

4 Begin Rendezvous/Resumption of Accepting Task

Ada: Task C begins rendezvous at entry E of Task T.

TSL: T **performs** BEGIN_RENDEZVOUS (CALLER => C, CALLED_E => E);

5 End Rendezvous/End of Accept Statement

Ada: Task C ends rendezvous at entry E of Task T.

TSL: T **performs** END_RENDEZVOUS (CALLER => C, CALLED_E => E);

6 Resumption of Caller/End of Entry Call Statement

Ada: Task C ends rendezvous at entry E of Task T.

TSL: C **performs** END_CALL (CALLEE => T, CALLED_E => E);

9.6 Delay Statements, Duration and Time

1 Beginning of Delay

Ada: Task T begins execution of a delay statement with duration TO.

TSL: T **performs** BEGIN_DELAY (TIMEOUT => TO, TIMESTAMP => CALENDAR.CLOCK);

2 End of Delay

Ada: Task T finishes execution of a delay statement.

TSL: T **performs** DELAYED (TIMESTAMP => CALENDAR.CLOCK);

9.7 Select Statements

9.7.1 Selective Waits

1 Selective Wait with Else

Ada: Task *T* is executing **select/else**.

TSL: T **performs** SELECT_ELSE;

2 Selective Wait with Terminate

Ada: Task *T* is executing **select/terminate**.

TSL: T **performs** SELECT_TERMINATE;

3 Selective Wait with Delay

Ada: Task *T* is executing **select/delay** with timeout *TO*.

TSL: T **performs** SELECT_DELAY (TIMEOUT => TO, TIMESTAMP => CALENDAR.CLOCK);

4 Selective Wait with No Else Part and Only Open Accept Alternatives

Ada: Task *T* is executing **select** only with open accept alternatives.

TSL: T **performs** OPEN_SELECT;

5 Selective Wait with All Guards False and No Else Part

Ada: Task *T* is executing closed **select**.

TSL: T **performs** CLOSED_SELECT;

9.7.2 Conditional Entry Calls

1 Conditional Entry Call Execution

Ada: Task *C* begins execution of a timed entry call with timeout 0.0.

TSL: C **performs** TIMED_CALL (TIMEOUT => 0, TIMESTAMP => CALENDAR.CLOCK);

9.7.3 Timed Entry Calls

1 Timed Entry Call Execution

Ada: Task *C* begins execution of a timed entry call with timeout *TO*.

TSL: C **performs** TIMED_CALL (TIMEOUT => TO, TIMESTAMP => CALENDAR.CLOCK);

2 Timed Entry Call Cancellation

Ada: A call from task C is removed from the queue of entry E at task T.

TSL: **any performs** DEQUEUE_CALL (C_TASK => C, Q_TASK => T, Q_ENTRY => E); (*Must be performed by tasking supervisor.*)

9.8 Priorities

1 Priority Specification

Ada: Priority P is specified for task T.

TSL: **any performs** SPECIFY_PRIORITY (P_TASK => T, P => P);

9.9 Task and Entry Attributes

1 CALLABLE Attribute Evaluation

Ada: Task T evaluated attribute CALLABLE of task C to be value V.

TSL: T **performs** CHECKED_CALLABLE (C_TASK => C, VALUE => V);

2 TERMINATED Attribute Evaluation

Ada: Task T evaluated attribute TERMINATED of task C to value V.

TSL: T **performs** CHECKED_TERMINATED (T_TASK => C, VALUE => V);

3 COUNT Attribute Evaluation

Ada: Task T evaluated attribute COUNT of entry E to value V.

TSL: T **performs** CHECKED_COUNT (C_ENTRY => E, VALUE => V);

9.10 Abort Statements

1 Beginning of Abort Statement

Ada: Task T executes an **abort** statement to abort of task A.

TSL: T **performs** BEGIN_ABORT (A_TASK => A);

2 End of Abort Statement

Ada: Task T finishes execution of an **abort** statement which aborted task A.

TSL: T **performs** ABORTED (A_TASK => A);

3 Abnormality

Ada: Task A becomes abnormal.

TSL: **any performs** MAKE_ABNORMAL (A_TASK => A); (*Must be performed by tasking supervisor.*)

4 Exception Raising—*TASKING_ERROR, PROGRAM_ERROR*

Ada: Task T raises exception X.

TSL: T **performs** RAISE_EXCEPTION; (*no exception name*)

5 Exception Handling—*TASKING_ERROR, PROGRAM_ERROR*

Ada: Task T handles exception X.

TSL: T **performs** HANDLE_EXCEPTION; (*no exception name*)

9.11 Shared Variables

1 Shared Object Update

Ada: Task T updates object object O with value V.

TSL: T **performs** UPDATE (OBJ => O);

Appendix G

TSL Formalization of a Subset of Ada Tasking

This appendix presents the TSL formalization of the subset of the Ada tasking semantics defined in Chapter 6. The formal definition of the Ada tasking events were presented in Appendix F as TSL **action** declarations. The TSL formalization of the semantics of the **accept** statement was presented in Chapter 6.

G.1 Task Activation

This section presents the TSL formalization of the semantics of the activation of single tasks and tasks declared by object declarations.

G.1.1 Property Declarations

The property declarations for the predicates *ACTIVATED (T)* and *COMPLETED (T)* were given in Figure 51 of Chapter 6.

The function *SCOPE (T)* is used to indicate how many nested scopes a T has entered; it is initially zero, is incremented by each scope entry, and is decremented by each scope exit:

−− *SCOPE function:*

```
--+ property S_SCOPE (task) : NATURAL := 0
--+ is
--+   when ?T performs ENTER_BLOCK then
--+         set S_SCOPE (?T) := S_SCOPE (?T) + 1;
--+   when ?T performs EXITED_BLOCK then
--+         set S_SCOPE (?T) := S_SCOPE (?T) - 1;
--+ end S_SCOPE;
```

The predicate *DEPENDENT (T, D)* is initially false and is defined to become true when D is declared as a dependent of T and false again when D terminates:

```
--    DEPENDENT predicate:
--+ property S_DEPENDENT (task, task) : BOOLEAN := FALSE
--+ is
--+   when (?T performs DECLARE_SINGLE_TASK (S_TASK => ?D) or
--+          ?T performs DECLARE_TASK_OBJECT (T_TASK => ?D))
--+   then
--+         set S_DEPENDENT (?T, ?D) := TRUE;
--+   when (?T performs DECLARE_SINGLE_TASK (S_TASK => ?D) or
--+          ?T performs DECLARE_TASK_OBJECT (T_TASK => ?D))
--+   then ?D performs TERMINATED
--+         set S_DEPENDENT (?T, ?D) := FALSE;
--+ end S_DEPENDENT;
```

The predicate *ACTIVATING (T)* is initially false and is defined to become true when T is activated by its master and false again when it ends its activation:

```
--    ACTIVATING predicate:
--+ property S_ACTIVATING (task) : BOOLEAN := FALSE
--+ is
--+   when ?T performs BEGIN_ACTIVATION then
--+         set S_ACTIVATING (?T) := TRUE;
--+   when ?T performs END_ACTIVATION then
--+           set S_ACTIVATING (?T) := FALSE;
--+ end S_ACTIVATING;
```

Finally, the predicate *DECLARED (TT)* is initially false and is defined to become true when TT is declare and false when the scope of TT is exited:

```
--   DECLARED predicate:
--+ property S_DECLARED (task_type) : BOOLEAN := FALSE
--+ is
--+   when any performs DECLARE_TASK_TYPE (T_TYPE => ?TT) then
--+           set S_DECLARED (?TT) := TRUE;
--+   when ?T performs DECLARE_TASK_TYPE (T_TYPE => ?TT)
--+               where ?S = S_SCOPE (?T)
--+   then  (?T performs EXITED_SCOPE
--+                where S_SCOPE (?T) = ?S or
--+             ?T performs TERMINATED
--+                where ?S = 0)
--+           set S_DECLARED (?TT) := FALSE;
--+ end S_DECLARED;
```

G.1.2 Functional Specification

Two functional specifications are constructed. The first is for the activation of a single task:

```
--+ <<SINGLE_TASK_ACTIVATION>>
--+ when ?T performs DECLARE_SINGLE_TASK (S_TASK => ?D)
--+             where ?S = S_SCOPE (?T) and S_ACTIVATED (?T) and
--+                     not S_COMPLETED (?T)
--+ then (?D performs BEGIN_ACTIVATION
--+             where S_SCOPE (?T) = ?S and S_DEPENDENT (?T, ?D)
--+         =>
--+         ?D performs END_ACTIVATION
--+             where S_SCOPE (?T) = ?S and S_ACTIVATING (?D)
--+         =>
--+         ?T performs BEGIN_BLOCK
--+            where S_SCOPE (?T) = ?S and S_ACTIVATED (?D)
--+        )
--+ until ?T performs BEGIN_BLOCK;
```

The second is for the activation of a task declared by an object declaration:

```
--+ << TASK_OBJECT_ACTIVATION >>
--+ when ?T performs DECLARE_TASK_OBJECT (T_TASK => ?D,
--+                                       T_TYPE => ?TT)
--+            where ?S = S_SCOPE (?T) and S_DECLARED (?TT) and
--+                  S_ACTIVATED (?T) and not S_COMPLETED (?T)
--+ then (?D performs BEGIN_ACTIVATION
--+            where S_SCOPE (?T) = ?S and S_DEPENDENT (?T, ?D)
--+         =>
--+         ?D performs END_ACTIVATION
--+            where S_SCOPE (?T) = ?S and S_ACTIVATING (?D)
--+         =>
--+         ?T performs BEGIN_BLOCK
--+            where S_SCOPE (?T) = ?S and S_ACTIVATED (?D)
--+        )
--+ until ?T performs BEGIN_BLOCK;
```

G.1.3 Safety Specifications

One safety specification is derived from each different guarded event in the above functional specifications. The safety specifications for the *Begin Activation* event and one for the *Begin Block* event require a special construction to put the events in the appropriate context:

```
--   The Declare Single Task event must not occur
--   in an incorrect state:
--+ not ?T performs DECLARE_SINGLE_TASK
--+          where not (S_ACTIVATED (?T) and
--+                     not S_COMPLETED (?T));
```

```
--   The Declare Task Object event must not occur
--   in an incorrect state:
--+ not ?T performs DECLARE_TASK_OBJECT (T_TYPE => ?TT)
--+          where not (S_DECLARED (?TT) and S_ACTIVATED (?T) and
--+                          not S_COMPLETED (?T));

--   The Begin Activation event must not occur in an incorrect state:
--+ when (?T performs DECLARE_SINGLE_TASK (S_TASK => ?D)
--+              where ?S = S_SCOPE (?T) or
--+          ?T performs DECLARE_TASK_OBJECT (T_TASK => ?D)
--+              where ?S = S_SCOPE (?T))
--+ then not ?D performs BEGIN_ACTIVATION
--+                where not (S_SCOPE (?T) = ?S);

--   The Begin Block event must not occur in an incorrect state:
--+ when (?T performs DECLARE_SINGLE_TASK (S_TASK => ?D)
--+              where ?S = S_SCOPE (?T) or
--+          ?T performs DECLARE_TASK_OBJECT (T_TASK => ?D)
--+              where ?S = S_SCOPE (?T))
--+ then not ?T performs BEGIN_BLOCK
--+                where S_SCOPE (?T) = ?S and
--+                          not (S_ACTIVATED (?D));
```

The S_SCOPE property is not needed in the guards of the first two safety specifications since the placeholder ?S is *set* to the value of this property when it is first matched.

G.2 Task Termination

This section presents the TSL formalization of the semantics of the termination of single tasks and tasks declared by object declarations.

G.2.1 Property Declarations

The property declarations for the predicates *ACTIVATED (T)* and *COMPLETED (T)* were given in Figure 51. The property declaration for the function *SCOPE (T)* was given in Section G.1.1.

The predicate *TERMINATED (T)* is initially false and is defined to become true when *T* terminates:

```
--   TERMINATED predicate:
--+ property S_TERMINATED (task) : BOOLEAN := FALSE
--+ is
--+   when ?T performs TERMINATED then
--+           set S_TERMINATED (?T) := TRUE;
--+ end S_TERMINATED;
```

G.2.2 Functional Specification

Two functional specifications are constructed. The first is for the termination of a task declared in the outermost scope of its master:

```
--+ << OUTERMOST_TASK_TERMINATION >>
--+ when (?T performs DECLARE_SINGLE_TASK (S_TASK => ?D)
--+              where S_SCOPE (?T) = 0 and S_ACTIVATED (?T) and
--+                    not S_COMPLETED (?T) or
--+          ?T performs DECLARE_TASK_OBJECT (T_TASK => ?D)
--+              where S_SCOPE (?T) = 0 and S_ACTIVATED (?T) and
--+                    not S_COMPLETED (?T))
--+ then (?D performs TERMINATED
--+              where S_SCOPE (?T) >= 0 and S_DEPENDENT (?T, ?D)
--+          =>
--+          ?T performs TERMINATED
--+              where S_SCOPE = 0 and S_TERMINATED (?D)
--+       )
--+ until ?T performs TERMINATED;
```

The second is for the termination of a task declared in an inner scope of its master:

```
--+ <<INNER_TASK_TERMINATION>>
--+ when (?T performs DECLARE_SINGLE_TASK (S_TASK => ?D)
--+             where ?S = S_SCOPE (?T) and S_SCOPE (?T) > 0 and
--+                     S_ACTIVATED (?T) and not S_COMPLETED (?T)
--+         or
--+         ?T performs DECLARE_TASK_OBJECT (T_TASK => ?D)
--+             where ?S = S_SCOPE (?T) and S_SCOPE (?T) > 0 and
--+                     S_ACTIVATED (?T) and not S_COMPLETED (?T))
--+ then (?D performs TERMINATED
--+             where S_SCOPE (?T) >= ?S and S_DEPENDENT (?T, ?D)
--+         =>
--+         ?T performs EXITED_BLOCK
--+             where S_SCOPE = ?S and S_TERMINATED (?D)
--+       )
--+ until ?T performs EXITED_BLOCK
--+             where S_SCOPE = ?S;
```

G.2.3 Safety Specifications

Safety specifications for the *Declare Task* events were given in Section G.1.3. Two other safety specifications are derived from the above functional specifications, one for the *Exited Scope* event and one for the *Terminates* event. These two require a special construction to put the events in the appropriate context:

```
--  The Terminates event must not occur in an incorrect state:
--+ when (?T performs DECLARE_SINGLE_TASK (S_TASK => ?D)
--+             where S_SCOPE (?T) = 0 or
--+         ?T performs DECLARE_TASK_OBJECT (T_TASK => ?D)
--+             where S_SCOPE (?T) = 0)
--+ then not ?T performs TERMINATED
--+                 where not (S_TERMINATED (?D));
```

```
--    The Exited Block event must not occur in an incorrect state:
--+ when (?T performs DECLARE_SINGLE_TASK (S_TASK => ?D)
--+            where ?S = S_SCOPE (?T) and S_SCOPE (?T) > 0 or
--+          ?T performs DECLARE_TASK_OBJECT (T_TASK => ?D)
--+            where ?S = S_SCOPE (?T) and S_SCOPE (?T) > 0)
--+ then not ?T performs EXITED_BLOCK
--+               where S_SCOPE (?T) = ?S and
--+                     not (S_TERMINATED (?D));
```

G.3 Task Execution

This section presents the TSL formalization of the semantics of task execution, specifying the various stages that all tasks pass through during their lifetime between activation and termination.

G.3.1 Property Declarations

The property declarations for the predicates *ACTIVATED (T)* and *COMPLETED (T)* were given in Figure 51. The property declarations for the function *SCOPE (T)* and the predicate *ACTIVATING (T)* were given in Section G.1.1. The property declaration for the predicate*TERMINATED (T)* was given in Section G.2.1.

G.3.2 Functional Specification

The functional specification is as follows:

```
--+ <<TASK_EXECUTION>>
--+ when ?T performs BEGIN_ACTIVATION
--+            where not S_ACTIVATING (?T) and not S_ACTIVATED (?T)
--+ then (?T performs END_ACTIVATION
--+            where S_ACTIVATING (?T) and S_SCOPE (?T) = 0
--+          =>
--+          ?T performs COMPLETE
--+            where S_ACTIVATED (?T) and S_SCOPE (?T) = 0
--+          =>
--+      ?T performs TERMINATED
--+         where S_COMPLETED (?T) and S_SCOPE (?T) = 0
--+     )
--+ until ?T performs TERMINATED;
```

G.3.3 Safety Specifications

One safety specification is derived from each guarded event of the above functional specification:

```
--  The Begin Activation event does not occur in an incorrect state:
--+ not ?T performs BEGIN_ACTIVATION
--+        where not (not S_ACTIVATING (?T) and
--+                   not S_ACTIVATED (?T));

--  The End Activation event does not occur in an incorrect state:
--+ not ?T performs END_ACTIVATION
--+        where not (S_ACTIVATING (?T) and
--+                   not S_SCOPE (?T) = 0);

--  The Completes event does not occur in an incorrect state:
--+ not ?T performs COMPLETE
--+        where not (S_ACTIVATED (?T) and
--+                   not S_SCOPE (?T) = 0);

--  The Terminates event does not occur in an incorrect state:
--+ not ?T performs TERMINATED
--+        where not (S_COMPLETED (?T) and
--+                   S_SCOPE (?T) = 0);

--  A terminated task does nothing:
--+ not ?T performs any
--+        where S_TERMINATED (?T);
```

Note that a second safety specification has been defined for both the *Begin Activation* and the *Terminates* events; in each case, the pair of safety specifications would be checked separately, and both specifications must be satisfied.

G.4 The Entry Call Statement

This section presents the TSL formalization of the semantics of the entry call statement.

G.4.1 Property Declarations

The property declarations for the predicates *ACTIVATED (T)* and *COMPLETED (T)* were given in Figure 51. The property declarations for the predicates *ACCEPTING (T)* and *IN_RENDEZVOUS (T)* were given in Figure 53.

The predicate *CALLING (C, T, E)* is initially false and is defined to be true between the beginning and end of a call to entry E of task T by task C:

```
--   CALLING predicate:
--+ property S_CALLING (task, task, entry) : BOOLEAN := FALSE
--+ is
--+    when ?C performs SIMPLE_CALL (CALLEE => ?T,
--+                                  CALLED_E => ?E) then
--+          set S_CALLING (?C, ?T, ?E) := TRUE;
--+    when ?C performs END_CALL (CALLEE => ?T,
--+                               CALLED_E => ?E) then
--+          set S_CALLING (?C, ?T, ?E) := FALSE;
--+ end S_CALLING;
```

The predicate *QUEUED (C, T, E)* is initially false and is defined to be true between the arrival of a call from task C to task T at entry E and the beginning of the corresponding rendezvous:

```
--   QUEUED predicate:
--+ property S_QUEUED (task, task, entry) : BOOLEAN := FALSE
--+ is
--+   when any performs ENQUEUE_CALL (C_TASK => ?C,
--+                                     Q_TASK => ?T,
--+                                     Q_ENTRY => ?E) then
--+         set S_QUEUED (?C, ?T, ?E) := TRUE;
--+   when ?T performs BEGIN_RENDEZVOUS (CALLER => ?C,
--+                                        CALLED_E => ?E) then
--+         set S_QUEUED (?C, ?T, ?E) := FALSE;
--+ end S_QUEUED;
```

Finally, the predicate *RENDEZVOUSED (C, T, E)* is initially false and is defined to be true between the end of a rendezvous between task C and task T at T's entry E and the end of the corresponding entry call executed by C:

```
--   RENDEZVOUSED predicate:
--+ property S_RENDEZVOUSED (task, task, entry) : BOOLEAN
--+                                              := FALSE
--+ is
--+   when ?T performs END_RENDEZVOUS (CALLER => ?C,
--+                                     CALLED_E => ?E) then
--+         set S_RENDEZVOUSED (?C, ?T, ?E) := TRUE;
--+   when ?C performs END_CALL (CALLEE => ?T,
--+                               CALLED_E => ?E) then
--+         set S_RENDEZVOUSED (?C, ?T, ?E) := FALSE;
--+ end S_RENDEZVOUSED;
```

G.4.2 Functional Specification

The functional specification uses the function *QUEUE_SIZE (T, E)* to count how many rendezvous take place before C's call is accepted. These other rendezvous are matched by a TSL **macro** (a named compound event) called OTHER_RENDEZVOUS, which is used to enable rebinding of the placeholder *?C2* for each such rendezvous. The terminator for this specification covers the requirement that T must be suspended (i.e., T must perform no action) until the entry call is finished:

```
--+ macro OTHER_RENDEZVOUS (?T : task; ?E : entry;
--+                                  ?C : task) is
--+        ?T performs END_RENDEZVOUS (CALLER => ?C2,
--+                                        CALLED_E => ?E)
--+            where S_QUEUED (?C, ?T, ?E) and
--+                    S_ACCEPTING (?T, ?E) and ?C2 /= ?C
--+        =>
--+        ?T performs SIMPLE_ACCEPT (ACCEPTED_E => ?E)
--+            where S_QUEUED (?C, ?T, ?E) and
--+                    not S_ACCEPTING (?T, ?E);
--+ end OTHER_RENDEZVOUS;

--+ <<SIMPLE_ENTRY_CALL>>
--+ when ?C performs SIMPLE_CALL (CALLEE => ?T,
--+                                   CALLED_E => ?E)
--+           where S_ACTIVATED (?C) and
--+                  not S_COMPLETED (?C)
--+ then ((any performs ENQUEUE_CALL (C_TASK => ?C,
--+                                        Q_TASK => ?T,
--+                                        Q_ENTRY => ?E)
--+             where S_CALLING (?C, ?T, ?E) and
--+                     ?Q = S_QUEUE_SIZE (?T, ?E) and
--+                     S_ACCEPTING (?T, ?E)
--+        =>
--+        (OTHER_RENDEZVOUS (?T, ?E, ?C)) ↑ ?Q
--+       )
--+       or
--+       (any performs ENQUEUE_CALL (C_TASK => ?C,
--+                                        Q_TASK => ?T,
--+                                        Q_ENTRY => ?E)
--+             where S_CALLING (?C, ?T, ?E) and
--+                     ?Q = S_QUEUE_SIZE (?T, ?E) and
--+                     not S_ACCEPTING (?T, ?E)
--+        =>
--+         ?T performs SIMPLE_ACCEPT (ACCEPTED_E => ?E)
--+            where S_QUEUED (?C, ?T, ?E) and
--+                  not S_ACCEPTING (?T, ?E)
--+         =>
--+         (OTHER_RENDEZVOUS (?T, ?E, ?C)) ↑ ?Q
--+        )
--+       )
--+       =>
--+       ?T performs BEGIN_RENDEZVOUS (CALLER => ?C,
--+                                     CALLED_E => ?E)
--+          where S_QUEUED (?C, ?T, ?E) and
--+                S_ACCEPTING (?T, ?E)
--+       =>
--+       ?T performs END_RENDEZVOUS (CALLER => ?C,
--+                                   CALLED_E => ?E)
--+          where S_IN_RENDEZVOUS (?C, ?T, ?E)
--+       =>
--+       ?C performs END_CALL (CALLEE => ?T,
--+                             CALLED_E => ?E)
--+          where S_RENDEZVOUSED (?C, ?T, ?E)
--+ until ?C performs any;
```

G.4.3 Safety Specifications

The safety specification for the *End Rendezvous* event was given in Figure 56. One safety specification is derived from each of the other guarded events of the functional specification:

```
--  The Begin Call event does not occur in an incorrect state:
--+ not ?C performs SIMPLE_CALL
--+        where not (S_ACTIVATED (?C) and not S_COMPLETED (?C));

--  The Call Arrival event does not occur in an incorrect state:
--+ not any performs ENQUEUE_CALL (C_TASK => ?C,
--+                                     Q_TASK => ?T,
--+                                     Q_ENTRY => ?E)
--+           where not (S_CALLING (?C, ?T, ?E));

--   The Begin Rendezvous event does not occur in an incorrect state:
--+ not ?T performs BEGIN_RENDEZVOUS (CALLER => ?C,
--+                                       CALLED_E => ?E)
--+          where not (S_QUEUED (?C, ?T, ?E) and
--+                     S_ACCEPTING (?T, ?E));

--   The End Call event does not occur in an incorrect state:
--+ not ?C performs END_CALL (CALLEE => ?T,
--+                           CALLED_E => ?E)
--+          where not (S_RENDEZVOUSED (?C, ?T, ?E));
```

Note that a second safety specification has been defined for the *Begin Rendezvous* event (the first was defined in Chapter 6); the two safety specifications would be checked separately, and both must be satisfied.

G.5 The Delay Statement

This section presents the TSL formalization of the semantics of the **delay** statement.

G.5.1 Property Declarations

The property declarations for the predicates *ACTIVATED (T)* and *COMPLETED (T)* were given in Figure 51. The function *CURRENT_TIME* is equivalent to the function CLOCK of the predefined Ada package CALENDAR. As mentioned in Appendix F, since all notion of time is lost in the merging of events into the TSL global event stream, the value of *CURRENT_TIME* must be passed as a constituent of both the *Begin Delay* and *End Delay* events when they are performed.

G.5.2 Functional Specification

The terminator for the functional specification specifies the requirement that T must be suspended (i.e., T must perform no action) until the delay is finished.

```
--+ <<SIMPLE_DELAY>>
--+ when ?T performs BEGIN_DELAY (TIMEOUT => ?D,
--+                                TIMESTAMP => ?TS1)
--+        where S_ACTIVATED (?T) and not S_COMPLETED (?T)
--+ then ?T performs DELAYED (TIMESTAMP => ?TS2)
--+        where ?TS2 - ?TS1 >= ?D
--+ until ?T performs any;
```

G.5.3 Safety Specifications

A safety specification is derived for the *Execute Delay* event of the functional specification; the safety specification is equivalent to the functional specification.

```
--  The Begin Delay event does not occur in an incorrect state:
--+ not ?T performs BEGIN_DELAY
--+       where not (S_ACTIVATED (?T) and
--+                  not S_COMPLETED (?T));
```

G.6 Other Specifications

Three other specifications must be written in order to fully specify the semantics of the subset of Ada tasking under consideration.

The first specification says that a task executing an **accept** statement for an entry whose queue is empty is suspended until the arrival of a call to the entry; the arrival of the call allows the task to begin a rendezvous. In fact, once a task begins an **accept** statement, it can do nothing until it begins a rendezvous:

```
--+ <<ACCEPTOR_SUSPENSION>>
--+ when ?T performs SIMPLE_ACCEPT (ACCEPTED_E => ?E)
--+ then ?T performs BEGIN_RENDEZVOUS (CALLED_E => ?E)
--+ until ?T performs any;
```

The second specification says that a task cannot execute nested **accept** statements for the same entry:

```
--+ << NESTED_ACCEPTS >>
--+ not ?T performs SIMPLE_ACCEPT (ACCEPTED_E => ?E)
--+         where S_ACCEPTING (?T, ?E);
```

The final specification says that entry queues respect a first-come-first-served (FCFS) discipline; that is, entry calls are processed in the order of their arrival:

```
--+ << FCFS_ENTRY_QUEUES >>
--+ when (?S performs ENQUEUE_CALL (C_TASK => ?C1,
--+                                 Q_TASK => ?T,
--+                                 Q_ENTRY => ?E)
--+        =>
--+        ?S performs ENQUEUE_CALL (C_TASK => ?C2,
--+                                 Q_TASK => ?T,
--+                                 Q_ENTRY => ?E)
--+           where ?C2 /= ?C1 and S_QUEUED (?C1, ?T, ?E)
--+       )
--+ then ?T performs BEGIN_RENDEZVOUS (CALLER => ?C1,
--+                                    CALLED_E => ?E)
--+ before ?T performs BEGIN_RENDEZVOUS (CALLER => ?C2,
--+                                      CALLED_E => ?E);
```

Bibliography

[ABB*86] M. Acetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: a new kernel foundation for UNIX development. In *Proceedings of the USENIX 1986 Summer Technical Conference*, pages 93–112, June 1986.

[ACF87] Yeshayahu Artsy, Hung-Yang Chang, and Raphael A. Finkel. Interprocess communication in Charlotte. *IEEE Software*, 4(1):22–28, January 1987.

[ACG86] Sudhir Ahuja, Nicholas Carriero, and David Gelernter. Linda and friends. *IEEE Computer*, 19(8):26–34, August 1986.

[Ack79] William B. Ackerman. Data flow languages. In *Proceedings of the National Computer Conference, Volume 48*, pages 1087–1095, AFIPS, June 1979.

[Ada83] *The Ada Programming Language Reference Manual.* US Department of Defense, US Government Printing Office, February 1983. ANSI/MIL-STD-1815A-1983.

[Ada87] Ada Joint Program Office. Ada validation procedures and guidelines. *Ada Letters*, 7(2):29–57, March–April 1987.

[AFdR80] Krzysztof R. Apt, Nissim Francez, and Willem P. de Roever. A proof system for communicating sequential processes. *ACM Transactions on Programming Languages and Systems*, 2(3):359–385, July 1980.

[AGB*77] Allen L. Ambler, Donald I. Good, James C. Browne, Wilhelm F. Burger, Richard M. Cohen, Charles G. Hoch, and Robert E. Wells. GYPSY: a language for specification and implementation of verifiable programs. In *Proceedings of a Conference on Language Design for Reliable Software*, pages 1–10, ACM SIGPLAN, March 1977. Appears in *SIGPLAN Notices* 12(3), March 1977.

[AGH*87] Larry M. Augustin, Benoit A. Gennart, Youm Huh, David C. Luckham, and Alec Stanculescu. VAL: an annotation language for VHDL. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD'87)*, pages 418–421, IEEE Computer Society, November 1987.

[Apt81] Krzysztof R. Apt. Ten years of Hoare's logic: a survey—part I. *ACM Transactions on Programming Languages and Systems*, 3(4):431–483, October 1981.

[AS83] Gregory R. Andrews and Fred B. Schneider. Concepts and notations for concurrent programming. *ACM Computing Surveys*, 15(1):3–44, March 1983.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1986.

[Bak86] Theodore P. Baker. An Ada runtime system interface. In *Proceedings of the Second International Conference on Ada Applications and Environments*, pages 99–110, IEEE Computer Society, April 1986.

[Bar85] Howard Barringer. *A Survey of Verification Techniques for Parallel Programs. Lecture Notes in Computer Science No. 191*, Springer-Verlag, 1985.

[BBC*83] Eric J. Berglund, Kenneth P. Brooks, David R. Cheriton, David R. Kaelbling, Keith A. Lantz, Timothy P. Mann, Robert J. Nagler, William I. Nowicki, Marvin M. Theimer, and Willy E. Zwaenepoel. *V-System 4.0 Reference Manual.* Stanford University, Computer Systems Laboratory, November 1983.

[BdFV86] Fabrizio Baiardi, Nicoletta de Francesco, and Gigliola Vaglini. Development of a debugger for a concurrent language. *IEEE Transactions on Software Engineering*, SE-12(4):547–553, April 1986.

[BF81] Raymond M. Bryant and Raphael A. Finkel. A stable distributed scheduling algorithm. In *Proceedings of the 2nd International Conference on Distributed Computing Systems*, pages 314–323, IEEE Computer Society, April 1981.

[BGHL87] Andrew D. Birrell, John V. Guttag, James J. Horning, and Roy Levin. Synchronization primitives for a multiprocessor: A formal specification. In *Proceedings of the 11th Symposium on Operating Systems Principles*, pages 94–102, ACM SIGOPS, November 1987.

[BH75] Per Brinch Hansen. The programming language Concurrent Pascal. *IEEE Transactions on Software Engineering*, SE-1(2):199–207, June 1975.

[BH77] Per Brinch Hansen. *The Architecture of Concurrent Programs*. Prentice-Hall, 1977.

[BH78] Per Brinch Hansen. Distributed processes: a concurrent programming concept. *Communications of the ACM*, 21(11):934–941, November 1978.

[BM82] Howard Barringer and Ian Mearns. Axioms and proof rules for Ada tasks. *IEE Proceedings Part E*, 129(2):38–48, March 1982.

[BM83] David L. Bird and Carlos U. Muñoz. Automatic generation of random self-checking test cases. *IBM Systems Journal*, 22(e):229–245, 1983.

[BN84] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[BR85] Theodore P. Baker and Gregory A. Riccardi. Ada tasking: from semantics to efficient implementation. *IEEE Software*, 2(2):34–46, March 1985.

[BR86] Theodore P. Baker and Gregory A. Riccardi. Implementing Ada exceptions. *IEEE Software*, 3(5):42–51, September 1986.

[BW82] Peter C. Bates and Jack C. Wileden. EDL: a basis for distributed systems debugging tools. In *Proceedings of the 15th Hawaii International Conference on System Sciences*, pages 86–93, January 1982.

[CH74] R. H. Campbell and A. N. Habermann. The specification of process synchronization by path expressions. In *Proceedings of an International Symposium on Operating Systems*, pages 89–102, Springer-Verlag (Lecture Notes in Computer Science No. 16), April 1974.

[Che82] David R. Cheriton. *The Thoth System: Multi-Process Structuring and Portability*. Elsevier Science Publishing Co., 1982.

[Che83] David R. Cheriton. Local networking and internetworking in the V-system. In *Proceedings of the 8th Data Communications Symposium*, pages 9–16, ACM, October 1983.

[Che84] David R. Cheriton. The V kernel: a software base for distributed systems. *IEEE Software*, 1(2):19–42, April 1984.

[Cle82] Geert B. Clemmensen. A formal model of distributed Ada tasking. In *Proceedings of the AdaTEC Conference*, pages 224–237, ACM SIGPLAN—AdaTEC, October 1982.

[CM84] W. F. Clocksin and C. S. Mellish. *Programming in Prolog*. Springer-Verlag, 1984. Second Edition.

[CM86] David R. Cheriton and Timothy P. Mann. *A Decentralized Naming Facility*. Technical Report 86-1098, Department of Computer Science, Stanford University, February 1986.

[CMMS79] David R. Cheriton, Michael A. Malcolm, Lawrence S. Melen, and Gary R. Sager. Thoth, a portable real time operating system. *Communications of the ACM*, 22(2):105–115, February 1979.

[Cor84] Dennis Cornhill. Four approaches to partitioning Ada programs for execution on distributed targets. In *Proceedings of the Conference on Ada Applications and Environments*, pages 153–162, IEEE Computer Society, October 1984.

[CW82] R. Curtis and L. Wittie. BugNet: a debugging system for parallel programming environments. In *Proceedings of the 3rd International Conference on Distributed Computing Systems*, pages 394–399, IEEE Computer Society, October 1982.

[CZ83] David R. Cheriton and Willy E. Zwaenepoel. *The Distributed V Kernel and Its Performance for Diskless Workstations*. Technical Report 83-973, Department of Computer Science, Stanford University, July 1983. Computer Systems Laboratory Report 83-246.

[CZ84] David R. Cheriton and Willy E. Zwaenepoel. *One-to-Many Interprocess Communication in the V-System.* Technical Report 84-264, Computer Systems Laboratory, Stanford University, August 1984. Department of Computer Science Technical Report 84-1011.

[CZ85] David R. Cheriton and Willy E. Zwaenepoel. Distributed process groups in the V kernel. *ACM Transactions on Computer Systems*, 3(3):77–107, May 1985.

[DH72] Ole-Johan Dahl and C. A. R. Hoare. Hierarchical program structures. In *Structured Programming*, pages 175–220, Academic Press, 1972.

[Dij68] Edsger W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages*, Academic Press, 1968.

[Dij75] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, August 1975.

[EBGW83] A. Evans, K.J. Butler, G. Goos, and W. A. Wulf. *DIANA Reference Manual, Revision 3.* Tartan Laboratories, Inc., Pittsburgh, PA, 1983.

[Fal82] Edward Falis. Design and implementation in Ada of a runtime task supervisor. In *Proceedings of the AdaTEC Conference*, pages 1–9, ACM SIGPLAN—AdaTEC, October 1982.

[FFS77] R. J. Fuller, S. H. Fuller, and D. P. Siewiorek. Cm*—a modular, multi-microprocessor. In *Proceedings of the National Computer Conference, Volume 46*, pages 637–644, AFIPS, June 1977.

[Fis78] David A. Fisher. DoD's common programming language effort. *IEEE Computer*, 11(3):24–33, March 1978.

[Flo67] Robert W. Floyd. Assigning meanings to programs. In *Proceedings of the Symposium in Applied Mathematics, Volume XIX*, pages 19–32, American Mathematical Society, April 1967.

[FS81] Lawrence Flon and Norihisa Suzuki. The total correctness of parallel programs. *SIAM Journal on Computing*, 10(2):227–246, May 1981.

[FW86] David A. Fisher and Richard M. Weatherly. Issues in the design of a distributed operating system for Ada. *IEEE Computer*, 19(5):38–47, May 1986.

[GdR84] Rob Gerth and Willem P. de Roever. A proof system for concurrent Ada programs. *Science of Computer Programming*, 4:159–204, 1984.

[Ger82] Rob Gerth. *A Sound and Complete Hoare Axiomatization of the Ada Rendezvous.* Technical Report RUU-CS-82-5, Department of Computer Science, University of Utrecht, April 1982. Extended Abstract.

[GHL82] Steven M. German, David P. Helmbold, and David C. Luckham. Monitoring for deadlocks in Ada tasking. In *Proceedings of the AdaTEC Conference*, pages 10–25, ACM SIGPLAN—AdaTEC, October 1982.

[GHW85] John V. Guttag, James J. Horning, and Jeannette M. Wing. The Larch family of specification languages. *IEEE Software*, 2(5):24–36, September 1985.

[GM84] Richard P. Gabriel and John McCarthy. *Queue-based Multi-processing Lisp.* Technical Report 84-1007, Computer Science Department, Stanford University, 1984.

[Gro78] High Order Language Working Group. *Requirements for High Order Computer Programming Languages—STEELMAN.* Technical Report, US Department of Defense, 1978.

[Hai80] Brent T. Hailpern. *Verifying Concurrent Processes Using Temporal Logic.* PhD thesis, Computer Systems Laboratory, Stanford University, August 1980. Technical Report 80-195.

[HB84] Kai Hwang and Fayé A. Briggs. *Computer Architecture and Parallel Processing.* McGraw-Hill, 1984.

[HL83] David P. Helmbold and David C. Luckham. *Runtime Detection and Description of Deadness Errors in Ada Tasking.* Technical Report 83-249, Computer Systems Laboratory, Stanford University, November 1983. Program Analysis and Verification Group Report 22.

[HL85a] David P. Helmbold and David C. Luckham. Debugging Ada tasking programs. *IEEE Software*, 2(2):47–57, March 1985.

[HL85b] David P. Helmbold and David C. Luckham. TSL: Task Sequencing Language. In *Ada in Use: Proceedings of the Ada International Conference*, pages 255–274, Cambridge University Press, May 1985.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–581, October 1969.

[Hoa74] C. A. R. Hoare. Monitors: an operating system structuring concept. *Communications of the ACM*, 17(10):549–557, October 1974.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[HW73] C. A. R. Hoare and Niklaus Wirth. An axiomatic definition of the programming language Pascal. *Acta Informatica*, 2(4):335–355, 1973.

[ILL75] Shigeru Igarashi, Ralph L. London, and David C. Luckham. Automatic program verification I: a logical basis and its implementation. *Acta Informatica*, 4:145–182, 1975.

[JK85] Rakesh Jha and Dennis Kafura. *Implementation of Ada Synchronization in Embedded Distributed Systems*. Technical Report, Virginia Polytechnic Institute, 1985.

[Jon85] Geraint Jones. *Programming in 'occam'—A Tourist Guide to Parallel Programming*. Technical Report PRG-43, Oxford University Computing Laboratory, March 1985.

[Kaf85] Dennis Kafura. *Termination of Ada Tasks in a Distributed Environment*. Technical Report, Virginia Polytechnic Institute, 1985.

[Kla85] Nils Klarund. *Formal Concepts for Specification and Automatic Testing of Ada Tasks*. Technical Report, DDC International A/S, 1985.

[Kri83] Bernd Krieg-Brückner. Consistency checking in Ada and Anna: a transformational approach. *Ada Letters*, 3(2):46–54, September-October 1983.

[KU87] John C. Knight and John I. A. Urquhart. On the implementation and use of Ada on fault-tolerant distributed systems. *IEEE Transactions on Software Engineering*, SE-13(5):553–563, May 1987.

[Lea84] J. F. Leathrum. Design of an Ada run-time system. In *Proceedings of the Conference on Ada Applications and Environments*, pages 4–13, IEEE Computer Society, October 1984.

[Lev80] Gary Marc Levin. *Proof Rules for Communicating Sequential Processes.* PhD thesis, Department of Computer Science, Cornell University, August 1980. Technical Report 80-435.

[LGvH*79] David C. Luckham, S. M. German, Friedrich W. von Henke, R. A. Karp, P. W. Milne, D. C. Oppen, W. Polak, and W. L. Scherlis. *Stanford Pascal Verifier User Manual.* Technical Report 79-731, Department of Computer Science, Stanford University, March 1979. Program Analysis and Verification Group Report 11.

[LHM*87] David C. Luckham, David P. Helmbold, Sigurd Meldal, Douglas L. Bryan, and Michael A. Haberler. Task Sequencing Language for specifying distributed Ada systems. In A. Nico Habermann and Ugo Montanari, editors, *System Development and Ada: Proceedings of the CRAI (Consorzio per le Ricerche e le Applicazioni di Informatica) Workshop on Software Factories and Ada*, pages 249–305, Springer-Verlag (Lecture Notes in Computer Science No. 275), 1987. Presented at the Workshop in Capri, Italy, May 26–30, 1986.

[LLSvH83] David C. Luckham, H. J. Larsen, D. R. Stevenson, and Friedrich W. von Henke. *ADAM—An Ada-Based Language for Multi-Processing.* Technical Report 83-240, Computer Systems Laboratory, Stanford University, May 1983. Reprint of Department of Computer Science Technical Report 81-867, July 1981.

[LM87] Thomas J. LeBlanc and John M. Mellor-Crummey. Debugging parallel programs with Instant Replay. *IEEE Transactions on Computers*, C-36(4):471–482, April 1987.

[LNR87] David C. Luckham, Randall B. Neff, and David S. Rosenblum. An environment for Ada software development based on formal specification. *Ada Letters*, 7(3):94–106, March–April 1987. Also Stanford University Computer Systems Laboratory Technical Report 86-305 (Program Analysis and Verification Group Report 31).

[LO83] Amy L. Lansky and Susan S. Owicki. *Gem: A Tool for Concurrency Specification and Verification.* Technical Report 83-251, Computer Systems Laboratory, Stanford University, November 1983.

[LP80a] David C. Luckham and Wolfgang Polak. *Ada Exceptions: Specification and Proof Techniques.* Technical Report 80-789, Computer Science Department, Stanford University, February 1980. Stanford Verification Group Report 16.

[LP80b] David C. Luckham and Wolfgang Polak. A practical method of documenting and verifying Ada programs with packages. In *Proceedings of the Symposium on the Ada Programming Language*, pages 113–122, ACM SIGPLAN, December 1980. Appears in *SIGPLAN Notices* 15(11), November 1980.

[LR85] Richard J. LeBlanc and Arnold D. Robbins. Event-driven monitoring of distributed programs. In *Proceedings of the 5th International Conference on Distributed Computing Systems*, pages 515–522, IEEE Computer Society, May 1985.

[LvH85] David C. Luckham and Friedrich W. von Henke. An overview of Anna, a specification language for Ada. *IEEE Software*, 2(2):9–23, March 1985.

[LvHKO87] David C. Luckham, Friedrich W. von Henke, Bernd Krieg-Brückner, and Olaf Owe. *Anna—A Language for Annotating Ada Programs. Lecture Notes in Computer Science No. 260*, Springer-Verlag, 1987.

[Mat85] Nicholas Matelan. The FLEX/32 MultiComputer. In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, pages 209–213, IEEE Computer Society and ACM SIGARCH, June 1985.

[Mel87] Sigurd Meldal. *An Axiomatic Semantics of Spawning.* Technical Report, Institute of Informatics, University of Oslo, 1987. In preparation.

[MP81a] Zohar Manna and Amir Pnueli. *Verification of Concurrent Programs, Part I: The Temporal Framework.* Technical Report 81-836, Department of Computer

Science, Stanford University, June 1981.

[MP81b] Zohar Manna and Amir Pnueli. *Verification of Concurrent Programs, Part II: Temporal Proof Principles*. Technical Report 81-843, Department of Computer Science, Stanford University, September 1981.

[MW85] Zohar Manna and Richard Waldinger. *The Logical Basis for Computer Programming, Volume 1: Deductive Reasoning*. Addison-Wesley, 1985.

[Nat86] N. Natarajan. A distributed scheme for detecting communication deadlocks. *IEEE Transactions on Software Engineering*, SE-12(4):531–537, April 1986.

[Nau63] Peter Naur (Ed.). Revised report on the algorithmic language ALGOL60. *Communications of the ACM*, 6(1):1–17, January 1963.

[NGO85] Van Nguyen, David Gries, and Susan S. Owicki. *A Model and Temporal Proof System for Networks of Processes*. Technical Report 85-270, Computer Systems Laboratory, Stanford University, February 1985.

[OG76] Susan S. Owicki and David Gries. Verifying properties of parallel programs: an axiomatic approach. *Communications of the ACM*, 19(5):279–285, May 1976.

[Oli87] Dave Olien. Parallel Ada tasking on the Balance 8000. In *Uniforum Proceedings*, Winter 1987.

[OSvdG86] D. P. O'Leary, G. W. Stewart, and Robert van de Geijn. *DOMINO—A Message Passing Environment for Parallel Computation*. Technical Report TR-1648, Department of Computer Science, University of Maryland, April 1986.

[Ous82] John K. Ousterhout. Scheduling techniques for concurrent systems. In *Proceedings of the 3rd International Conference on Distributed Computing Systems*, pages 22–30, IEEE Computer Society, October 1982.

[Pdb86] *DYNIX Pdbx Debugger User's Manual*. Sequent Computer Systems, Inc., September 1986.

[PdR82] Amir Pnueli and Willem P. de Roever. Rendezvous with Ada—a proof theoretical view. In *Proceedings of the AdaTEC Conference*, pages 129–137, ACM SIGPLAN—AdaTEC, October 1982.

[PKL80] David A. Padua, David J. Kuck, and Duncan H. Lawrie. High-speed multiprocessors and compilation techniques. *IEEE Transactions on Computers*, C-29(9):763–776, September 1980.

[PPL86] *Balance Guide to Parallel Programming*. Sequent Computer Systems, Inc., September 1986.

[PS83] F. N. Parr and Robert E. Strom. NIL: a high-level language for distributed systems programming. *IBM Systems Journal*, 22(1/2):111–127, 1983.

[PWC*81] G. Popek, B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin, and G. Thiel. LOCUS: a network transparent, high reliability distributed system. In *Proceedings of the 8th Symposium on Operating Systems Principles*, pages 169–177, ACM SIGOPS, December 1981.

[RB84] Gregory A. Riccardi and Theodore P. Baker. A runtime supervisor to support Ada task activation, execution and termination (preliminary report). In *Proceedings of the Conference on Ada Applications and Environments*, pages 14–22, IEEE Computer Society, October 1984.

[RB85] Gregory A. Riccardi and Theodore P. Baker. A runtime supervisor to support Ada tasking: rendezvous and delays. In *Ada in Use: Proceedings of the Ada International Conference*, pages 329–342, Cambridge University Press, May 1985.

[Ros85] David S. Rosenblum. A methodology for the design of Ada transformation tools in a DIANA environment. *IEEE Software*, 2(2):24–33, March 1985. Also Stanford University Computer Systems Laboratory Technical Report 85-269 (Program Analysis and Verification Group Report 27).

[Ros87] David S. Rosenblum. An efficient communication kernel for distributed Ada runtime tasking supervisors. *Ada Letters*, 7(2):102–117, March–April 1987.

[RSL86] David S. Rosenblum, Sriram Sankar, and David C. Luckham. Concurrent runtime checking of Annotated Ada programs. In *Proceedings of the 6th Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 10–35, Springer-Verlag (Lecture Notes in Computer Science No.

241), December 1986. Also Stanford University Computer Systems Laboratory Technical Report 86-312 (Program Analysis and Verification Group Report 33).

[SH86] Vivek Sarkar and John Hennessy. Compile-time partitioning and scheduling of parallel programs. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, pages 17–26, ACM SIGPLAN, June 1986. Appears in *SIGPLAN Notices* 21(7), July 1986.

[SMV83] Richard L. Schwartz, P. M. Melliar-Smith, and Friedrich H. Vogt. *An Interval Logic for Higher-Level Temporal Reasoning.* Technical Report CSL-138, Computer Science Laboratory, SRI International, Menlo Park, CA, February 1983.

[SR86] Sriram Sankar and David S. Rosenblum. *The Complete Transformation Methodology for Sequential Runtime Checking of an Anna Subset.* Technical Report 86-301, Computer Systems Laboratory, Stanford University, June 1986. Program Analysis and Verification Group Report 30.

[SRN85] Sriram Sankar, David S. Rosenblum, and Randall B. Neff. An implementation of Anna. In *Ada in Use: Proceedings of the Ada International Conference*, pages 285–296, Cambridge University Press, May 1985.

[SS71] Dana Scott and Christopher Strachey. Toward a mathematical semantics for computer languages. In Jerome Fox, editor, *Proceedings of the Symposium on Computers and Automata*, pages 19–46, The Polytechnic Institute of Brooklyn and The Microwave Research Institute, Wiley-Interscience, April 1971.

[Sta84] William Stallings. Local networks. *ACM Computing Surveys*, 16(1):3–41, March 1984.

[Ste80] D. R. Stevenson. Algorithms for translating Ada multitasking. In *Proceedings of the Symposium on the Ada Programming Language*, pages 166–175, ACM SIGPLAN, December 1980. Appears in *SIGPLAN Notices* 15(11), November 1980.

[SY83] Robert E. Strom and Shaula Yemini. NIL: an integrated language and system for distributed programming. In *Proceedings of the SIGPLAN '83 Symposium* *on Programming Language Issues in Software Systems*, pages 73–81, ACM SIGPLAN, June 1983. Appears in *SIGPLAN Notices* 18(6), June 1983.

[Tan81] Andrew S. Tanenbaum. *Computer Networks*. Prentice-Hall, 1981.

[Tan87] Andrew S. Tanenbaum. *Operating Systems: Design and Implementation*. Prentice-Hall, 1987.

[Ten76] R. D. Tennent. The denotational semantics of programming languages. *Communications of the ACM*, 19(8):437–453, August 1976.

[TLC85] Marvin M. Theimer, Keith A. Lantz, and David R. Cheriton. *Preemptable Remote Execution Facilities for the V-System*. Technical Report 85-288, Computer Systems Laboratory, Stanford University, September 1985. Department of Computer Science Technical Report 85-1087.

[TO86] Kuo-Chung Tai and Evelyn E. Obid. Reproducible testing of Ada tasking programs. In *Proceedings of the Second International Conference on Ada Applications and Environments*, pages 69–79, IEEE Computer Society, April 1986.

[TvR85] Andrew S. Tanenbaum and Robbert van Renesse. Distributed operating systems. *ACM Computing Surveys*, 17(4):419–470, December 1985.

[vHLKO85] Friedrich W. von Henke, David C. Luckham, Bernd Krieg-Brückner, and Olaf Owe. Semantic specification of Ada packages. In *Ada in Use: Proceedings of the Ada International Conference*, pages 185–196, Cambridge University Press, May 1985.

[VM87] Richard A. Volz and Trevor N. Mudge. Timing issues in the distributed execution of Ada programs. *IEEE Transactions on Computers*, C-36(4):449–459, April 1987.

[VMNM85] Richard A. Volz, Trevor N. Mudge, Arch W. Naylor, and John H. Mayer. Some problems in distributing real-time Ada programs across machines. In *Ada in Use: Proceedings of the Ada International Conference*, pages 72–84, Cambridge University Press, May 1985.

[Wea84a] Richard M. Weatherly. *Design of a Distributed Operating System for Ada*. PhD thesis, Clemson University, Clemson, SC, August 1984.

[Wea84b] Richard M. Weatherly. A message-based kernel to support Ada tasking. In *Proceedings of the Conference on Ada Applications and Environments*, pages 136–144, IEEE Computer Society, October 1984.

[Weg72a] Peter Wegner. Operational semantics of programming languages. In *Proceedings of a Conference on Proving Assertions About Programs*, pages 128–141, ACM SIGPLAN and SIGACT, January 1972. Appears in *SIGPLAN Notices* 7(1), January 1972.

[Weg72b] Peter Wegner. The Vienna definition language. *ACM Computing Surveys*, 4(1):5–63, March 1972.

[Win87] Jeannette M. Wing. Writing Larch interface language specifications. *ACM Transactions on Programming Languages and Systems*, 9(1):1–24, January 1987.

[Wir71] Niklaus Wirth. The programming language Pascal. *Acta Informatica*, 1(1):35–63, 1971.

[Wir82] Niklaus Wirth. *Programming in Modula-2. Texts and Monographs in Computer Science*, Springer-Verlag, 1982.

[Woo87] M. Woodger. Origins of Ada features. *Ada Letters*, 7(1):59–70, January–February 1987.

[WS83] Peter Wegner and Scott A. Smolka. Processes, tasks, and monitors: a comparative study of concurrent programming primitives. *IEEE Transactions on Software Engineering*, SE-9(4):446–462, July 1983.

[ZGMB82] H. Zimmerman, M. Guillemont, G. Morisset, and J. S. Banino. *CHORUS: A Communication and Processing Architecture for Distributed Systems.* Technical Report EXP 4 525, INRIA, Rocquencourt, 78150 Le Chesnay, France, 1982.

[Zim80] Hubert Zimmerman. OSI reference model—the ISO model of architecture for open systems interconnection. *IEEE Transactions on Communications*, COM-28(4):425–432, April 1980.