A KNOWLEDGE-BASED SYSTEM FOR DEBUGGING CONCURRENT SOFTWARE

Carol Helfgott LeDoux

March 1986
CSD-860060

Report No.

A Knowledge-Based System For Debugging Concurrent Software

Carol Helfgott LeDoux

Computer Science Department
School of Engineering and Applied Science
University of California
Los Angeles, California 90024

ABSTRACT

A Knowledge-Based System For Debugging Concurrent Software

by

Carol Helfgott LeDoux
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 1985
Professor D. Stott Parker, Jr., Chair

The recent development of high-level concurrent programming languages has emphasized the problem of limited debugging tools to support the development of applications using these languages. A new approach is necessary to improve the efficacy of debugging tools and to adapt them to the framework of a concurrent software environment.

A knowledge-based debugging approach is presented that aids diagnosis of a variety of run-time errors that can occur in concurrent programs written in the Ada* programming language. In this approach, an event stream of program activity is captured in an historical database and accessed using Prolog queries extended with temporal-logic predicates. Diagnosis is aided by applying rule-based descriptions of some common classes of software errors and by matching program specifications against the trace database.

This approach was used in building a prototype debugger, called Your Own Debugger for Ada (YODA). The design of YODA is described and analyses of several sample Ada programs are presented to illustrate diagnosis of errors associated with concurrency, including deadness errors and misuse of shared data.

---

*Ada is a registered trademark of the U.S. Government – Ada Joint Program Office

FOREWORD

This report reproduces a dissertation submitted in partial satisfaction of the requirements for the degree of Doctor of Philosophy in Computer Science at the University of California, Los Angeles, on 2 December 1985. The author thanks her doctoral committee, which consisted of Professors D. Stott Parker, Jr. (Chair), Walter J. Karplus, David F. Martin, Bruce Rothschild, and R. Clay Sprowls. This research was supported in part by an Aerospace Doctoral Fellowship. Preparation of this document was supported in part by the Aerospace Sponsored Research Program.

CONTENTS

FIGURES

TABLES

1. INTRODUCTION

Debugging, the process of finding and removing errors in programs, is a major factor in the cost of software development and maintenance. It represents 25 to 50 per cent of the total effort of developing large-scale software (Ref. 1). Programmers spend three times longer debugging their programs than initially writing them. Errors that remain can be costly. Neuman publishes a monthly list of software-related catastrophies and mishaps (e.g., (Ref. 2, 3)).

All software is prone to errors, but debugging techniques have advanced little in the past two decades. Some automated techniques are available, but informal techniques prevail. As noted previously (Ref. 4), debugging has rarely been the subject of theoretical investigation. Until recently, debugging has received little attention in computer science (Ref. 5, 6).

Traditional debugging practices are often ill-suited to modern programming techniques, such as *concurrency*. Unlike purely sequential programs, a concurrent program has several distinct threads of control, called *processes*, each of which executes independently. A concurrent program is *distributed* if the individual processes execute in parallel on multiple processors or on multiple computers with communication between processes. Concurrency supports the development of processing systems for real-time applications (e.g., database management systems and air-traffic control systems), transaction-processing applications (e.g., airline reservation systems), and large-scale, parallel, scientific computations.

Andrews and Schneider have reviewed concurrency techniques and cite the increasing economic feasibility of distributed systems and multi-processors as a motivation for the recent trend toward concurrent programming (Ref. 7). Distributed software offers advantages in survivability, reliability, and functional modularity.

New programming notations have been defined to simplify the expression of concurrency in high-level programming languages. Concurrent Pascal, Modula-2, Concurrent Prolog, Communicating Sequential Processes (CSP), and Ada* are examples of modern, high-level languages that provide explicit notations for concurrency.

This dissertation introduces new techniques for isolating the cause of software errors that are associated with *multitasking*, the concurrency feature of the Ada programming language. Ada is a general-purpose language designed for the U.S. Department of Defense (DoD) as the standard language for *mission-critical* software, that is, all software required for the conduct of the military mission of the DoD, including real-time systems.

An individual process in Ada is called a *task*. The conceptual framework of multitasking permits a

---

*Ada is a registered trademark of the U.S. Department of Defense – Ada Joint Program Office.

designer to isolate possibly simultaneous or inherently asynchronous events (Ref. 8). Multitasking permits mutual exclusion of processes, temporary unavailability of processes, and ordering of execution.

Our approach to debugging is to collect a *trace history* that captures the time-dependent relationships of events that can occur in the execution of a concurrent program. We develop techniques for automating *trace analysis*, i.e., for accessing a trace database to test program properties. Although our focus is on multitasking, our approach is extensible to other notations for concurrency.

We identify common causes of errors in Ada tasking programs and investigate strategies for diagnosing them. To show the feasibility of our approach, we have built a prototype debugger, called Your Own Debugger for Ada (YODA).

1.1. Debugging Defined

The origin of the word "bug" for a program error has been attributed to an incident in which a moth flew into an early computer and wrecked havoc. Instead, a *bug* has meant "an unexpected defect, fault, flaw, or imperfection" as early as 1889 (Ref. 3).

Model (Ref. 4) divides the process of debugging into five phases:

1. observing program behavior (either by monitoring the execution or by hand simulation),
2. comparing observed behavior with expected behavior,
3. analyzing the differences that have been detected (discovering the cause of a detected error),
4. devising changes to make the program conform to intended behavior, and
5. changing the program to correct the error (e.g., with a text editor).

Of the five phases of debugging, monitoring is the main function of traditional automated debugging tools. Model argued that monitoring has the most potential for improvement -- error localization and error correction are *creative* activities that cannot be automated easily. Isolating the cause of an error remains a trial-and-error process. On the contrary, we argue that error localization can benefit from automated aids.

1.2. Why Is Debugging Difficult?

Debugging is a difficult process because programmers need to extract relevant data from the information available about the program's execution history. Another difficulty is that programmers must observe execution at a high level, such as the calling sequence of program units, as well as at a detailed level, such as the history of values assigned to program variables.

Debugging is labor intensive. The length of time required to find the cause of an error is related to the coherency of the error message and to the expertise of the programmer in recognizing error symptoms.

Error messages, unexpected program behavior, and incorrect results often give few clues to the cause of the error. Debugging requires skills of ingenuity, intuition, patience, and feature recognition.

1.3. Problems of Debugging Concurrent Software

Debugging a concurrent program is more difficult than debugging a sequential program for several reasons. Garcia-Molina (Ref. 9) partitions these reasons into four categories:

1. Having various loci of control makes a system more complex and, thus, harder to understand and more prone to errors, especially intermittent errors.
2. If processes execute in parallel on multiple processors, then debugging requires discovering on which processor a process has failed.
3. If the software is distributed onto processors that are geographically dispersed, then communication delays between processors hinder access to information necessary for debugging.
4. Applications that are distributed tend to be large, and exhaustive testing of them is costly and time-consuming.

We have chosen to focus on the first of these issues -- debugging concurrent software, as separate from the problems of debugging on multiple processors or on a distributed system. We divide the complications that arise into three categories: error detection, testing, and fault localization.

1.3.1. Complexity of Error Detection

The behavior of a concurrent program is less predictable than that of a sequential program because of the introduction of non-determinism, e.g., in scheduling of processes. Program errors can be intermittent and difficult to reproduce.

1.3.2. Complexity of Testing

The execution of a concurrent program is more complex than that of a sequential program. For example, the number of possible execution paths is increased significantly. Errors can be more subtle because of the increased complexity of the execution. For example, if a process terminates because of an error, then other processes will fail if they try to communicate with the errant process.

1.3.3. Complexity of Error Localization

Concurrency increases the amount of data programmers need to examine in locating the cause of an error and, thus, can increase the amount of time and expertise needed for debugging. For example, programmers need to examine data at the level of interprocess communication, as well as at the intraprocess level. Concurrency introduces new classes of errors, e.g., errors caused by communication failures between processes and by race conditions.

1.4. Notations for Concurrency

To cooperate, concurrent processes must *communicate* and *synchronize*. Communication requires several actions to occur in a specified order; however, processes can differ in their rate of execution. *Synchronization mechanisms* delay the execution of processes to constrain the ordering of events in interprocess communication. Processes communicate by either *shared variables* or *message passing*. Variables are *shared* if they can be referenced by more than one process. A variable is *referenced* when its value is used in the evaluation of an expression. A variable is *defined* when it obtains a new value as the result of the execution of a statement. In *message passing*, processes communicate by sending and receiving messages.

Shared variables provide concurrency in procedure-oriented languages, e.g., Concurrent Pascal and Modula, in which *monitors* control synchronization. A *monitor* is a collection of shared resources with procedures for implementing controlled operations on a shared resource.

Message passing can be *synchronous*, *asynchronous*, or *buffered*. If the process sending a message is never delayed, then message passing is *asynchronous*. If the process sending a message is always delayed until receiving a corresponding response, then message passing is *synchronous*. If messages are held in a bounded buffer, then message passing is *buffered*. CSP (Ref. 10) is a programming notation based on synchronous message passing.

1.5. Ada Concurrency: The Rendezvous Mechanism

This dissertation assumes some familiarity with the Ada language. In this section we give a brief description of Ada concurrency. In later chapters we describe other features of Ada, e.g., exception handling, modularity, nesting, and strong typing. For a detailed description of the language, see the Ada Language Reference Manual (ALRM) (Ref. 11). For a tutorial treatment of the language, many good references are available (e.g., (Ref. 12, 13)).

Ada multitasking is implemented primarily by *remote procedure call* between tasks, although Ada permits access to shared data. (Ada is a procedure-oriented language.) In *remote procedure call*, message passing is implemented by a procedure call from one process to another. Messages are exchanged between processes by the arguments in the call statement. No explicit synchronization primitives are provided by Ada for imposing concurrency restrictions on shared variables.

Ada's main innovation in concurrency is the *rendezvous* mechanism for handling synchronization of tasks. Synchronization is achieved by an *entry call* from a task to an *accept* statement in another task. A FIFO queue is associated with each entry. Each entry can have one or more corresponding accept statements.

Accept statements can be prefaced with *guards* to specify conditional execution. Rendezvous requires that the entry be ready for execution and that a call be made from another task to this entry. The calling

task is suspended until rendezvous completes. Scheduling and synchronization of tasks are non-deterministic.

Two variations of the simple entry call are the *timed* entry call and the *conditional* entry call. The timed entry call specifies a minimum duration to wait for rendezvous before canceling the call. A conditional entry call specifies that a call is to be canceled if there is no open alternative for the corresponding accept statement. A timed entry call with a zero delay is equivalent to a conditional entry call.

Multitasking can be implemented on a distributed system or on a multiprocessor, as well as with interleaved execution on a single hardware processing element. The implementation is intended to be transparent to the Ada programmer.

1.6. Common Bugs in Ada Tasking Programs

The semantics of the Ada language was specified independently of the implementation of any translator, thus reducing the risk of inconsistency between the semantics of the code in production and the semantics of the program executed in a debugging environment. An evolving set of test suites, the Ada Compiler Validation Capability (ACVC) (Ref. 14), is maintained by the DoD for checking consistency.

Expectations are high for improving software productivity and reliability by writing programs in Ada, as opposed to writing them in older languages, such as Fortran, Jovial, or assembly language. Claims of the advantages of using Ada are exemplified by the following advertisement for an Ada compiler:

> This reusable, high-order language can put an end to the Software Crisis. Ada decreases skyrocketing software costs, improves management and control, reduces life cycle costs, boosts productivity, dramatically reduces errors and cuts training costs (Ref. 15).

The refutation or validation of these claims is outside the scope of our research. Ada eliminates some classes of errors, e.g., strong typing detects exceeded array bounds and other type-conversion errors. We address the difficulties of debugging Ada programs that use language features that are prone to errors. Two kinds of errors associated with multitasking are *deadness errors* and *misuse of shared data*.

A *deadness error* occurs when tasks of a concurrent program reach a state from which execution cannot continue (owing to a task-communication failure), although the tasks have not yet terminated. *Cyclic*, or *circular*, *deadlock* is one class of deadness errors. These errors occur when a cyclic path of entry calls is executed, e.g., when a single task calls itself. In Chapter Six, the Ada program in Figure 6-1 exhibits cyclic deadlock.

Other classes of deadness errors are possible, e.g., *system lockup*. These errors occur when a task makes an untimed, unconditional call to an entry for which no corresponding accept alternative becomes open. The calling task remains suspended indefinitely. Unlike cyclic deadlock, system lockup cannot be diagnosed by testing for a cyclic path of entry calls and, thus, is substantially different from a debugging viewpoint. Also, cyclic deadlock is likely to originate from a design error; whereas system lockup is more

likely the result of a coding error.

The ordering of read/write access to shared data can affect its integrity. Examples of the *misuse of shared data* have been described previously (Ref. 16): referencing an uninitialized shared variable, assigning a new value to a shared variable before the previous value is referenced, and assigning values to a shared variable in two different tasks acting in parallel. In Chapter Six, Figure 6-6 shows an Ada program that exhibits misuse of shared data.

1.7. Automated Techniques for Debugging Concurrent Programs

Automated debugging techniques can be divided by the strategies used in observing program behavior:

1. *Dynamic* debugging tools allow programmers to observe and control program behavior interactively.
2. A *monitor* is a debugging tool that extracts information about the computation of a program as it executes, without providing control over the execution.

Applying conventional dynamic debugging techniques in a concurrent programming environment raises several research questions (Ref. 17). For example, automated debuggers traditionally support monitoring of an individual process, but provide little or no support for monitoring process interactions. Another difficulty is that dynamic debugging assumes that the entire state of the program can be examined and controlled (e.g., all Ada tasks can be halted at once). This assumption is often difficult to implement in a concurrent environment and is inappropriate for distributed systems.

For real-time applications, one environment may be used for development (the *host*) and another for production (the *target*). Debugging tools are often tied to the host environment and may be unavailable in the target environment. An error that has occurred on the target machine may be difficult to reproduce on the host machine. Errors can depend on hardware characteristics, the compiler, data input, as well as non-deterministic execution.

For debugging concurrent software, the following facilities are desirable:

- monitoring individual processes in a concurrent environment,
- monitoring and displaying interprocess communication,
- maintaining audit trails of execution, and
- abstracting from observations of the program's behavior.

1.8. Approach and Scope of this Dissertation

This dissertation presents a *retrospective* approach to debugging. Unlike dynamic debugging, retrospective debugging provides tools for *post-mortem* analysis of program execution (i.e., after the program has terminated). We develop automated techniques for observing and analyzing a program's

execution history.

We view debugging as the problem of extracting relevant information about a program's structure (e.g., the symbol table) and a history of the program's past behavior. In our approach, this information is maintained in an historical database that the programmer can access to test assertions about the program's behavior.

Since this work is the first such effort, it was important to limit its scope. The Ada language is large and complex, and there are many classes of program errors. Future developers will want to extend this work to deal with other varieties of errors and to aid in detecting or avoiding errors.

1.8.1. The Trace Database Approach

In our approach to debugging we capture *trace data* (information about the dynamic activity of a program) as an event stream and store it into an historical database. For Ada, we capture events associated with task synchronization, task status, variable reference, and variable definition.

Collecting trace data from Ada software raises many interesting problems. The overhead of monitoring real-time software will perturb its behavior, as noted previously (Ref. 17, 18). In producing traces for debugging, there is a "Heisenberg uncertainty principle": Whatever mechanisms are introduced to elicit trace information will interfere with program performance and may eventually modify the behavior of the program. In a non-deterministic programming environment, tracing may be acceptable only when debugging occurs in simulated time.

1.8.2. Trace Analysis

To access information in a dynamic environment, a temporal-logic approach is taken. Temporal logic provides the mechanics for precise reasoning about references to time. Temporal logic has been applied in specifying and verifying the behavior of concurrent programs, i.e., in proving that a program satisfies specified properties.

We use temporal specifications in testing assertions against a trace database. To answer queries about information not explicitly stored in the symbol table or trace database, we propose a *knowledge base* containing strategies for validating historically common program errors, e.g., cyclic deadlock and misuse of shared data.

In contrast to databases, *knowledge bases* contain not only facts, but also *rules* from which new facts can be generated. A rule can specify the *essential* conditions for establishing new facts, or it can specify *heuristics* for deducing new facts. *Knowledge-based* systems access a knowledge base containing facts and rules in a specialized problem domain.

1.8.3. Error Hierarchies and Diagnostic Reasoning

To aid in generating candidate diagnoses, we develop a classification scheme for organizing errors into hierarchies. Codifying historically common errors and debugging strategies is a prerequisite to automating diagnosis. Understanding more about the process of diagnosis can lead to improvements in program design. We propose organizing errors by assertions that can confirm their presence.

1.9. A Prototype Debugger for Ada: YODA

YODA is a stand-alone system, although it can serve as the basis for a debugger that is integrated with a specific translator and run-time system. YODA parses an Ada program, generates a symbol table, and embeds diagnostic output statements into a copy of the source program. When the annotated program is compiled and executed, the diagnostic statements invoke a program monitor to capture trace data. Figures 1-1, 1-2, and 1-3 show the system-level design of YODA.

YODA consists of the following components:

- A lexical scanner.
- A top-down parser to produce an abstract syntax tree.
- A semantic analyzer to build a symbol table.
- An annotator to augment the source code with diagnostic output statements.
- A pretty-printer that outputs the annotated source.
- A program monitor to build the trace database.
- A trace query processor that supports references to time.

All components were written in "standard" Prolog (Ref. 19), except the program monitor, which was written in ANSI Ada (Ref. 11). YODA was implemented on a Digital Equipment Corporation (DEC) VAX 11/780 computer under the Berkeley UNIX* 4.1 operating system. All Ada programs presented in this dissertation were translated and executed using the validated New York University Ada translator and interpreter (Ada/ED ANSI Version 1.1). All Prolog programs were executed using C-Prolog (Ref. 20).

Prolog is a logic programming language. It was developed around 1972 by Colmerauer and his colleagues at the University of Marseille (Ref. 21, 22) and is based on work by Kowalski (Ref. 23). Prolog's suitability for database systems has been described previously (Ref. 24, 25).

We extend Prolog with temporal operators for testing time-dependent properties of programs. Prolog is well-suited as a general query language (Ref. 26, 27). Unlike a relational-calculus language, such as Quel, Prolog supports queries on assertions as well as instantiations. The user can add new queries and definitions.

---

* UNIX is a registered trademark of AT&T Bell Laboratories.

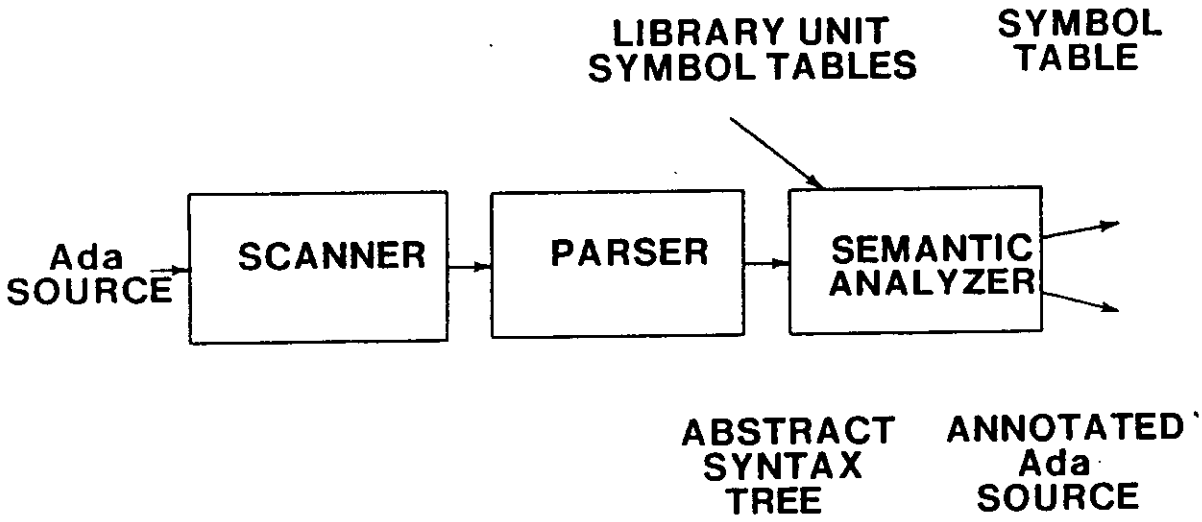

Figure 1-1: Design of YODA's Preprocessor

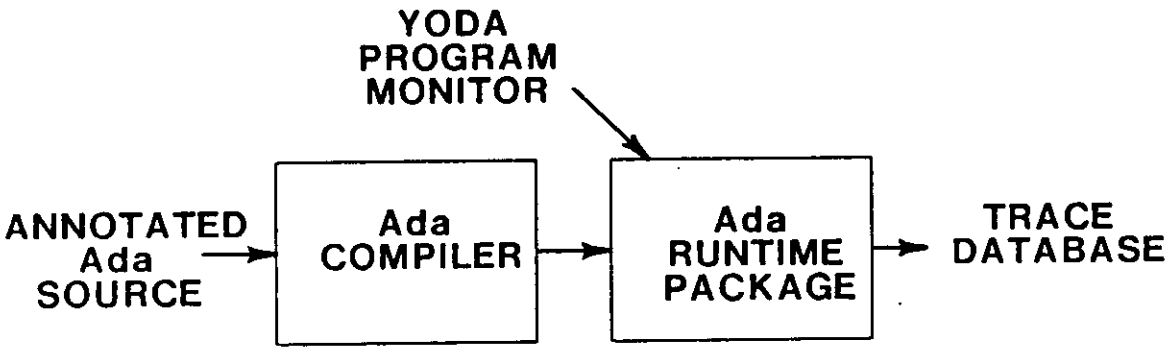

Figure 1-2: Design of YODA's Monitor

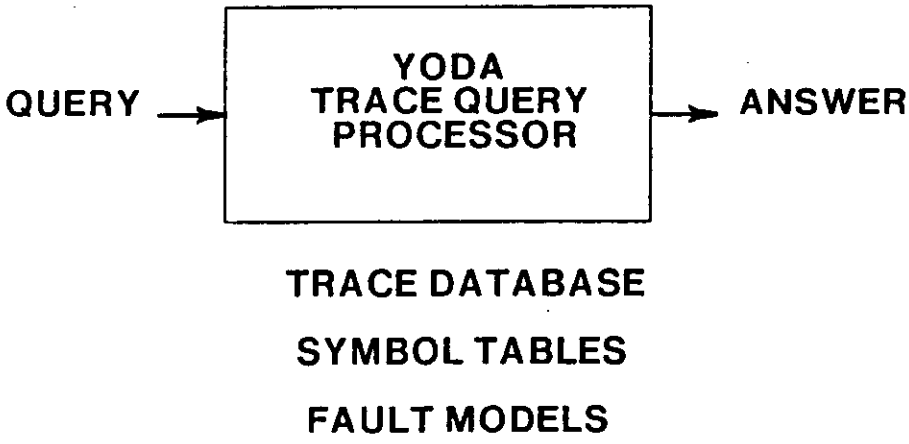

Figure 1-3: Design of YODA's Query Processor

1.10. Contributions

A fundamental goal of this research was to adapt debugging to the framework of concurrent software. An understanding of time is required for providing a formal semantics for debugging concurrent programs. We have developed techniques for analyzing program behavior and have designed and built mechanisms for supporting time-related queries on a program's behavior. This approach supports a high-level abstraction of program behavior based on an event stream.

The trace database approach improves on previous work in debugging because knowledge about time is built into the system. Our approach organizes trace data into a logical structure and supports flexible, controlled access to both static and dynamic program data. A database approach to debugging mechanizes the task of analyzing the trace of a program's behavior.

This dissertation provides not only a formal and conceptual framework on which to model program behavior, but also a practical approach to building a debugger that programmers will find useful. YODA represents a comprehensive approach to debugging. It supports effective diagnostic techniques for several classes of errors associated with concurrency.

A retrospective approach to debugging reduces the host-target problem to executing *annotated* Ada programs on the target system. Once captured, the trace database can be ported to the host for analysis.

Major accomplishments of this dissertation are as follows:

- Formalization of a knowledge-representation scheme for describing the behavior of concurrent programs.
- Simplification of the task of analyzing a program's past behavior by automating the selection of "interesting" events, e.g., interprocess communication. (Note that changing the events selected means changing the class of errors that can be recognized.)
- Development of automated techniques for examining and analyzing the execution history of concurrent programs.
- Development of automated techniques for localizing certain classes of errors, e.g., by identifying useful abstractions of a program's behavior.
- Implementation of a knowledge-based prototype tool for debugging Ada tasking programs.

1.11. Organization of this Dissertation

This dissertation integrates research in several areas, including debugging, temporal logic, historical databases, programming languages, and knowledge-based systems. The differences in terminology for each of these areas and their separate roles in support of this research motivated the organization of our presentation.

Chapter Two presents an historical perspective of debugging techniques and tools. We review both informal and automated debugging techniques. We elaborate on the deficiencies of traditional debugging tools for supporting effective debugging. We describe recent trends and innovations in debugging.

These include debugging tools intended for concurrent software, as well as tools that support either a database or a knowledge-based approach to examining program behavior.

Chapter Three provides the theoretical foundation for the research presented in this dissertation. We review the semantics of temporal logic and its application in specifying and verifying concurrent program behavior. We evaluate various temporal models of program behavior by their expressive power and by minimality of primitives, e.g., path expressions (regular expressions for specifying allowable sequences of events). We develop an *interval temporal logic* (a temporal logic based on bounded intervals of time) for analyzing program behavior.

Chapter Four introduces the major results of this dissertation: the development of a system for maintaining and accessing a trace database. We discuss the practical advantages that trace analysis offers over program verification for improving program reliability. We show how formal reasoning about time can be applied in recognizing faulty behavior patterns. The debugging approach that we present incorporates both interval logic and path expressions in testing assertions against a trace database.

Chapter Five presents the details of the implementation of YODA. We give an overview of Ada programming features and describe the major components of YODA. We identify relevant program events and define abstractions on events for describing the behavior of Ada tasking programs. We discuss the technical issues that we encountered in the process of designing and implementing a tool to capture the event history of Ada programs. A summary of Chapter Five has been published previously (Ref. 28).

In Chapter Six we introduce a functional model for classifying program errors: a hierarchy based on the complexity of queries needed to validate the class of errors to which a particular error belongs. We compare our model with previous error models, such as those based on language features and error symptoms. We present several examples of Ada tasking programs that contain errors and apply the diagnostic features of YODA in locating these errors. We analyze programs with the following errors: cyclic deadlock and non-serializability.

General conclusions and open problems are presented in Chapter Seven.

Appendix A gives an introduction to Prolog syntax and semantics. Appendix B presents a Prolog program for converting path expressions into a grammar-rule notation that can be applied in matching program specifications against a trace database. Appendix C presents our Prolog-based, top-down parser for Ada. Appendix D provides a YODA User's Guide. Appendix E presents an example of a complex Ada tasking program. Appendix F presents the symbol tables generated by YODA for the sample program in Appendix E.

2. DEBUGGING TOOLS

Techniques and tools for debugging are diverse. In this chapter we describe debugging techniques and strategies (Section 2.1). We review some tools for debugging concurrent programs, including programs written in Ada (Section 2.2). We describe recent work in applying database and knowledge-based techniques in debugging (Section 2.3). We conclude with a discussion on the infrequent use of available debugging tools and the need for exploring new debugging techniques, such as trace analysis (Section 2.4).

2.1. Debugging Techniques

Most strategies that programmers apply when debugging are informal, or *ad hoc*. These informal techniques prevail, although they are inefficient, unreliable, and often ineffective.

2.1.1. Informal Debugging Techniques

As noted previously (Ref. 29), the most traditional debugging technique is *desk checking*, that is, manually simulating program execution (while sitting at one's desk). This technique involves inspecting the source listing and performing hand calculations to compare with output listings.

Another informal debugging technique is to submit *test runs*, that is, to isolate the location of an error by iterative testing. On each successive run, either the input data are varied (to identify the conditions that may cause the error), or a modified version of the program is executed. Test runs are time-consuming because they involve a trial-and-error process. Also, the disappearance of an error may be fortuitous, without the programmer having located its cause.

A programmer can reduce the complexity of isolating an error by reducing the size of the program under analysis. Often a programmer can isolate a fragment of code or a particular statement type that is suspect, without identifying the faulty statement. The programmer can then write and test a separate, smaller program containing the suspect code. Gauss provides a tongue-in-cheek description of the process of partitioning a program into increasingly smaller pieces until the error is found (called the "wolf fence" technique) (Ref. 30).

Another informal debugging strategy is to sprinkle the program with diagnostic output statements to capture information about program execution. The usefulness of this technique depends on the programmer's expertise at identifying probable sources of errors.

2.1.2. Automated Support for Debugging

Automated tools that support debugging include *compilers*, *static analyzers*, *exception handlers*, and *debuggers*. These tools provide the following kinds of support for debugging:

- Compilers automate the detection of a restricted class of program errors, called *compile-time*

errors.

- Static analyzers automate techniques for detecting software errors that cannot be detected at compile time, that is, *run-time* errors.
- Exception handlers allow a program to continue executing after a run-time error has been detected.
- Debuggers help a programmer to observe the execution of a program (i.e., to inspect the program state), to detect erroneous program behavior and to locate its cause.

The purpose of an automated debugger is to help programmers to locate run-time errors, *logical* errors, and *portability* errors. *Logical* errors are those that need not be detected at compilation or execution, but can prevent the program from conforming with its specifications. A *portability* error results from relying on programming features that are outside the language standard, including features that are left as implementation dependent by the standard. Automated debugging tools include *dynamic* (interactive) debuggers and *monitors.*

2.1.2.1. Static Analyzers

The purpose of a static analyzer is to find potential sources of errors that cannot be detected by the compiler, but can be detected without executing the program. For example, static analysis can enforce coding standards, monitor the quality of the code, and test for adherence to programming standards.

The two basic techniques for static analysis are *control flow analysis* and *data flow analysis.* In control flow analysis, a graph is built to show the allowed flow of control between statements or sections of code. In data flow analysis, a graph is also built, but each node denotes a single statement, the execution of which can cause a variable to be updated. Data flow analysis can detect *program anomalies*, such as undefined or unreferenced variables.

For example, Taylor used static program analysis for reporting deadness errors, such as cyclic deadlock, in Ada tasking programs (Ref. 31). He was constrained to debugging simple programs because every possible state must be examined.

2.1.2.2. Exception Handlers

Exception handling permits a program to continue from an abnormal state. An exception is *raised* when a program reaches a specified state. Exception handlers specify a block of code to be executed when an exception is raised. Following execution of the handler code, a variety of responses are possible. Yemini and Berry provide a review of the various handler responses: termination, resuming execution, retry, propagating the exception, and transfer of control (Ref. 32). Exception handlers can provide additional features, e.g., parameterization. We consider two languages that support exception-handling: PL/1 and Ada. Neither of these languages supports parameterized exceptions.

PL/I supports *resumption of execution* after exception handling (Ref. 33). PL/I provides system-defined exception handlers only, e.g.,

End_Error Attempt to read past end-of-file.

Data_Error Input value does not match required type (READ) or required syntax (GET).

Ada supports *termination of execution* following exception handling. If the exception is handled in a task, for example, the task terminates, and execution resumes at the calling task. If an exception has no handler in the program unit in which it is raised, the exception is propagated dynamically. The system and the user can define Ada exceptions. Examples of exceptions that are defined by the system include run-time checks for subscript out of range and for arithmetic underflow/overflow.

2.1.2.3. Dynamic Debugging

Dynamic-debugging techniques include setting *breakpoints*, viewing intermediate data values or calling sequences by *tracing*, and examining control flow by *single-stepping*. Johnson provides an in-depth glossary (Ref. 34) of debugging terminology:

breakpoint "A location in a program's execution at which either some debugging command is to be performed or the user wishes to gain control."

data breakpoint "A breakpoint that can be associated with the access of data values." (Also called a *demon*.)

code breakpoint "A breakpoint that can be associated with the execution of program code segments."

conditional breakpoint "A breakpoint that is initiated only if some location-dependent predicate evaluates to true."

trace "A display of the dynamic activity of some aspect of a program."

retrospective trace "An historic display of the execution path of a program."

single-stepping "The ability to dynamically step through program execution, stopping periodically, so the user can interrogate the program's state."

Medina-Mora and Feiler have identified a hierarchy of desirable features for dynamic debugging in an integrated environment (Ref. 35). These features were incorporated into their design of an interactive debugger for the Incremental Programming Environment. These facilities include the following kinds of support:

- continuing the execution of a program at the point where execution was suspended by a breakpoint,
- unwinding a stack of procedure activations to modify a procedure in the stack and then resuming, and
- restoring the execution state to a previous point.

Assertion monitoring compares expected and observed behavior to trigger conditional breakpoints. For example, in Preliminary Ada (Ref. 36), conditional breakpoints could be set by the **assert** statement. This feature has since been removed from the Ada language. A major criticism of the Ada **assert** statement was that, to some extent, it duplicated exception handling, but provided less information. Unlike exception handlers, the **assert** statement provided no mechanism for showing the assertions that had failed, e.g., parameterization was not allowed.

2.1.2.4. Monitors

Recall that a *monitor* is a debugging tool that extracts information about the computation of a program as it executes, without providing control over the execution. *Message monitoring* allows programmers to examine the "external" behavior of programs, i.e., the execution of process interactions. Message passing can be monitored by tracing states of individual processes (because a message resides in a process just before it is sent or just after it is received). Yet, the state of an individual process contains information about the values of all variables to which the process has access. Harter (Ref. 37) argues that internal states provide more information than necessary for monitoring message passing. (Often the programmer is interested in only those variables containing the message being passed.)

Trace analysis is the process of analyzing an execution history that is collected by monitoring a program. Trace analysis can be applied either at a breakpoint as a dynamic debugging aid or, retrospectively, after the program has terminated.

Retrospective debugging techniques obviate many of the difficulties associated with dynamic debugging of concurrent programs. For example, single-stepping can be simulated by "replaying" the program's execution history, without altering the ordering of the computation.

Audit trails are important for debugging, in general, and are crucial for debugging concurrent programs. Having a record of the past helps in understanding the conditions leading to an error. In a non-deterministic programming environment, the timing and sequencing of events that caused an error may be difficult to reproduce. Errors depend on data input, scheduling algorithms, and timing dependencies, e.g., the implication of race conditions.

In comparison with other debugging aids, trace analysis provides many advantages:

- Unlike the results of static analysis, traces represent event sequences that can occur.
- Unlike assertion monitoring, in which expected and observed behavior are compared to trigger conditional breakpoints, trace analysis supports *post-mortem* debugging. A trace database provides a record of what happened before things started going wrong. The programmer can "replay" the results without reproducing them.
- When interactive debugging is impractical on a target machine, a trace can be collected on the target machine and ported to the host machine for post-mortem analysis.

2.2. Concurrent Debuggers

The following is a discussion of some systems for debugging high-level, concurrent software. These tools extend interactive debugging techniques for concurrency. This is not a complete survey of existing systems, but a description of current work in the development of experimental, prototype debuggers for concurrent languages. Satterthwaite (Ref. 38) describes earlier debugging systems, and Smith (Ref. 39) provides an abbreviated history of multiprocess, message-based systems.

BAIL and Defence support debugging for individual processes in a multiprocess environment, but no

interprocess features, such as monitoring message traffic, starting and stopping processes, or sending and receiving messages. The "checkpoint debugging" approach and the SPIDER debugger support interprocess debugging in a multiprocess environment. The Rolm Ada debugger and the Arcturus debugger support monitoring and controlling individual Ada tasks and monitoring task communication in a single-processor environment. Stanford's Ada debugger monitors task-state information.

2.2.1. BAIL

BAIL is an interactive debugger for accessing the multi-process environment of a SAIL program (Ref. 40). SAIL is an extended dialect of ALGOL60 that runs on the DEC PDP-10 computer. BAIL allows the user to insert breakpoints and access variables within a single process. BAIL was developed in the early 1970s at Stanford University. It runs under the TENEX and TOPS-10 operating systems.

2.2.2. Defence

Defence is a prototype debugger for Concurrent Euclid (Ref. 41). Users can monitor the execution of concurrent processes to determine if a process is running, in what queues the remaining processes are waiting, and the next statement to be executed in each process. Within a process, users can examine and modify variables, set trace areas, set conditional breakpoints, and invoke single-stepping. Defence allows monitoring of concurrent programs in a single-processor environment, but provides no control over message traffic or interprocess events.

2.2.3. Checkpoint Debugging

Checkpoint debugging has been investigated for debugging distributed software with real-time constraints (Ref. 42). This method requires taking regular checkpoints of a program. A *checkpoint* consists of a snapshot of a relevant program state and a sequential recording of all program input since the time of the previous program snapshot. A program failure can be repeated (deterministically) by returning to a previous checkpoint.

A disadvantage of this approach is the cost of execution and storage overhead required for taking checkpoints. Also, because checkpoints are synchronized with a clock instead of with events, there is no guarantee that important events would be detected.

2.2.4. SPIDER

SPIDER is an interactive debugger that aids in locating errors in communicating, loosely-coupled processes (a multiprocess, non-distributed system) (Ref. 39). SPIDER deals with interprocess events and treats processes as separate, communicating *black boxes*. Debugging tools include a debugger and demons (which automatically filter events).

Debugging techniques include monitoring, controlling, and testing processes. The programmer can

alter the contents of messages; create, access, and modify interprocess objects; preview, single-step, and replace individual interprocess events; and enable and disable debugging demons. SPIDER helps detect faulty interprocess communication, but not timing-related interprocess bugs.

2.2.5. Rolm Ada Debugger

The Rolm Ada debugger was developed as part of the Rolm Ada Work Center, which includes a DoD-validated ANSI Ada compiler running on the Rolm MSE/800 and the Data General Eclipse-800 computers (Ref. 43). The Rolm Ada compiler runs on a single processor with interleaved execution. The Rolm debugger allows programmers to determine the status of currently active tasks (e.g., waiting for rendezvous to begin) and to single-step through task interactions. Within a task, programmers can examine and modify variables, set traces, insert breakpoints, invoke single-stepping, and display program history. Only scalar variables and parameters can be modified. Traces can be set on statements, subprogram calls, or exceptions raised. Access is provided for active tasks only.

The Rolm Ada debugger allows breakpoints to be embedded in the program or entered through a keyboard interrupt. Breakpoints and single-stepping can be applied to the *selected* task only; other tasks continue executing. If the selected task is set to default to the currently executing task, then the interleaved history of all tasks can be displayed (if the history buffer is large enough), and task interactions can be monitored by single-stepping through rendezvous. Tracing of shared variables could, presumably, be done by setting traces in each task.

Raised exceptions and program deadlock can automatically invoke the debugger. When execution is continued from a breakpoint, exceptions are propagated normally. The history can be displayed for statements executed, variable declarations, and the stack of subprogram activations for an individual task. The history display can include the following tasking events:

- elaboration of task declaration,
- elaboration of task body declaration,
- execution of delay statement,
- execution of entry call,
- rendezvous started,
- rendezvous finished,
- execution of select statement,
- execution of timed entry call,
- execution of conditional entry call,
- task initiated,
- task aborted,
- task completed (ready to terminate), and
- task terminated.

2.2.6. Arcturus Debug Facility

Arcturus is a programming environment for developing Ada software. It was built as a research tool and does not yet support full Ada (Ref. 44). Arcturus includes a breakpoint and trace facility, which was intended for sequential programs, originally. The breakpoint package has since been extended for interactively debugging Ada tasking programs (Ref. 45).

The breakpoint package of Arcturus provides for traces and for entering Ada statements or expressions to be evaluated. Statements already existing in the program cannot be modified. Break commands can be entered during program execution (by a keyboard interrupt), or can be embedded in an Ada program, anywhere that a statement is allowed. In addition, a breakpoint is automatically invoked if an exception is raised for which there is no exception handler in the module that raised the exception or in any of the modules through which the exception is dynamically propagated. When this occurs, the run-time stacks are left untouched, so that no information is lost by propagating the exception.

2.2.7. An Ada Run-time Monitor

A tool for monitoring Ada programs has been developed at Stanford University (Ref. 46, 47, 48, 49). This debugger monitors current task-state information and diagnoses some deadness errors (such as cyclic deadlock) from simple diagnostic descriptions. Results of this effort showed that current task-state information is inadequate for diagnosing many run-time tasking errors.

2.3. Database/Knowledge Base Approaches to Debugging

Experience has shown that often a programmer can locate the cause of an error simply by obtaining a viewpoint different from the one used in implementing the program. For example, Brown and Sampson observed that the cause of a difficult error often becomes clear when a programmer discusses the error with another programmer (Ref. 1). Adrion (Ref. 29) observed that during group reviews of a program (e.g., a design review) a programmer "...finds many errors just by the simple review act of reading aloud." Adrion drew the following conclusion:

> Since seeing one's own errors is difficult, it is more effective if a second party does the desk checking.

In a survey of research on human factors in programming, Sheil cites empirical evidence to support the notion that the programmer's expertise is composed of a large *knowledge base* of interrelated, structured items (Ref. 50). For example, Shneiderman showed that expert programmers can memorize a structured program more easily than a novice can, but if the statements are shuffled, the advantage is reduced (Ref. 51). Other empirical studies (Ref. 52, 53, 54) also support programming folklore in suggesting that the presentation of information about a program's structure and behavior significantly influences a programmer's success in debugging it, particularly when the programmer is experienced.

In this section we discuss some prototype debugging tools that provide either database or knowledge-based support for debugging *sequential* programs. The OMEGA debugger supports queries on a

database of static program information. Sniffer supports queries on a database of a history of program states. FALOSY and the Prolog debugger use a database of expected output behavior for isolating discrepancies between intended and observed program behavior. In Chapter Four we describe the TQuel debugger (Ref. 55), which provides database support for debugging *concurrent* programs.

2.3.1. Debugger for Omega

The OMEGA database-system interface is a prototype debugging tool for Pascal-like programs (Ref. 56, 57). This system uses the conventional relational database management system Ingres, with its query language, Quel, to store and query static program information (the symbol table and parse tree), e.g., "Find all source statements that call procedure P." (All OMEGA queries are expressed in Quel.) This debugger was designed as a component of the OMEGA programming environment, in development at the University of California, Berkeley. Initial experience with this prototype showed it to be too slow to be useful.

The original design for this system allowed for dynamic access to current program-state information during program execution. For example, to access the value of a variable, the user would interrogate the database, thus invoking a request to the debugger to return the current value in real time. Quel would need to be altered both semantically and syntactically for expressing queries that trigger conditional breakpoints, e.g., "Suspend execution when P1 calls P2 and display the parameters passed."

2.3.2. Prolog Debugger

At Yale University, Shapiro developed a set of interactive diagnosis algorithms and bug-correction algorithms for identifying and fixing bugs in Prolog programs (Ref. 58). The diagnosis algorithms, written in Prolog, take as input the program to be debugged and a list of input data samples with expected output behavior. While single-stepping through procedure calls, these algorithms query the user for the correctness of intermediate results, to narrow the search for a bug. The system maintains a database of the result of queries for each debugged program, to minimize the number of queries needed for debugging a modification to an existing program.

A disadvantage of this approach is that it is not extensible to procedure-oriented languages, such as Ada, in which computation can take place in assignment statements as well as in procedure calls.

2.3.3. Sniffer

Sniffer is a knowledge-based interactive debugging aid that applies program analysis "by inspection" to diagnose a narrow class of program errors (Ref. 59). This tool was developed at the M.I.T. Artificial Intelligence Laboratory as part of the Abstraction, Inspection and Debugging programming environment (Ref. 52). Sniffer was implemented in Lisp on the MIT Lisp Machine for debugging Lisp Programs.

The debugging knowledge in Sniffer is organized into individual units containing expert information

about specific errors. Each expert, or "sniffer," independently examines the user-supplied description of the bug and applies a feature-recognition process to the program under analysis and to the events that took place during program execution. The recognition process is supported by two systems: the *cliche finder*, which identifies fragments of algorithms in the code by matching them against typical programming plans (cliches), and the *time rover*, which supports queries about the history of program states that occurred during the program's execution.

A disadvantage of this approach is that it requires at least as much effort to write the bug description as it does to write the program. Also the feature-recognition process is time-consuming, even for small programs.

2.3.4. FALOSY

The FAult LOcalization SYstem (FALOSY) (Ref. 60) is a knowledge-based prototype debugging model that was developed at the University of Minnesota. FALOSY was implemented for localizing errors in master-file update programs written in Pascal. The FALOSY tools include discrepancy analysis, strategy selection, recognition heuristics, and hypothesis generation. FALOSY compares expected output and observed output of programs (supplied by the user) and displays the discrepancies.

The *discrepancy analyzer* selects one of several strategies for looking at the code. The *localization strategy*, for example, looks for expected structural defects, such as a missing initialization statement. *Program slicing* locates statements referencing a variable and checks the variable's computation. Fault hypotheses are generated using a collection of fault models that embody knowledge about a fault or a class of faults that are domain-specific. FALOSY assumes that the fault is in only one program statement.

2.4. Summary of Previous Work and Conclusions

Recent research in debugging has focused on two areas:

1. Extending conventional dynamic debugging techniques for modern programming techniques, e.g., debugging distributed software.
2. Developing innovative approaches for debugging, e.g., database and knowledge-based support.

A study by Hanson and Rosinski has shown that, of the kinds of programming tools available (e.g., screen editors, data dictionaries, pretty-printers, and configuration managers), debugging tools are preferred over all others (Ref. 61). That is, programmers perceive debugging tools as the most important tool for improving their productivity.

Yet, during debugging sessions, interactive debugging facilities, when available, are used *infrequently*. In an empirical study on debugging, Gould observed that programmers preferred to try to locate an error without an automated aid (Ref. 62). Gould allowed subjects in this study at most 45 minutes to locate an

error, because he observed that few errors were found after the first 30 minutes of searching for them. He concluded that using a debugger often involves devoting more time to debugging and seems unwarranted if the programmer believes the bug can be found in 10 minutes. In a more recent study on debugging, programmers relied on test runs and desk checking 90 per cent of the time, although debugging facilities were available (Ref. 63).

To effectively use conventional dynamic debugging techniques, the programmer must gain some intuition about which lines of source code are suspect and, thus, which data values need to be examined. That is, there is an implicit assumption that the programmer knows where to place the breakpoints and which variables to trace. Garcia and Berman call this assumption the *sorcery property* of debugging because it makes debugging more of an art than a science (Ref. 64).

3. TEMPORAL MODELS OF PROGRAM BEHAVIOR

"Contrariwise," continued Tweedledee, "if it was so, it might be; and if it were so, it would be; but as it isn't, it ain't. That's logic."

-- Lewis Carroll (*Through the Looking-Glass*)

Specifying the behavior of a software system is complicated because it entails an abstract and precise description of the requirements, separate from the implementation. Each specification model must make some assumptions about the class of programs to be investigated, e.g., sequential or concurrent, terminating or cyclic, and whether the program is a finite-state system. Another difficulty is that each model must make assumptions about the environment in which the program will execute, e.g., on a distributed system, on a multi-processor, or on a single processor.

A general model for specifying requirements for concurrent programs is difficult to develop because each programming language uses a different semantics for concurrency, e.g., communication by shared variables versus message passing. Specification languages can be evaluated by the number and scope of *program properties* that can be defined and proved. For example, Sajkowski has applied these criteria in evaluating formal techniques for specifying and verifying communication protocols (Ref. 65). For concurrent programs, properties that have been investigated in the literature include mutual exclusion of processes, freedom from deadlock, absence of individual starvation, and synchronization of accesses to a shared resource.

Program *verification* consists of proving the correctness of a program's specification, that is, proving that all possible executions of the program eventually achieve a specific goal or satisfy a specific property. Properties are expressed in *assertions*, which are statements about the program's variables and sequence of execution.

Concurrency complicates verification. For example, concurrent programs can be non-deterministic and, thus, can exhibit more than one correct behavior for the same input data. Also, proving *total correctness* of a sequential program requires proof of program termination; however, some concurrent programs are intended to be cyclic and non-terminating, e.g., operating systems.

Temporal reasoning has been applied in proving properties of concurrent systems because other proof techniques (e.g., axiomatic, denotational, and operational) are inadequate for reasoning about concurrent behavior and delayed processing (Ref. 66). A *temporal model* is a formal system for representing knowledge about the relationships of events or activities that can take place within time. For program verification, a temporal model is a system that uses *temporal operators* in proving properties of programs.

Temporal operators specify the ordering of events and relationships among them. An event may follow, precede, or be contemporaneous with another event. A given event may cause another event to occur. A sequence of events or activities may recur in a specific pattern.

As temporal operators are added to a specification language, increasingly more complicated properties can be expressed in a "natural" and convenient form. We examine some completeness results, which suggest limitations on the expressiveness that can be achieved.

It has been argued that as the number of temporal operators increases in a specification, the meaning of the specification becomes more difficult to understand (Ref. 67, 68). In defining a specification language, a central goal is to maximize expressive power while minimizing the number of primitive operators.

3.1. Introduction

In this chapter we review four temporal models that have been used widely for specifying and verifying the behavior of concurrent programs:

1. A *temporal formula* for a program is an assertion that specifies all possible sequences of events or states that can take place in the program's execution (Ref. 69).
2. An *interval formula* for a program is an assertion that specifies all possible sequences of events or states that can take place within a specified interval of time during a program's execution (Ref. 70, 71, 72, 73).
3. A *path expression* is a regular expression specifying allowable sequences of events (Ref. 74, 75).
4. A *Petri net* is a directed graph specifying the set of all possible sequences of concurrent state transitions of a program (Ref. 76).

In the remainder of this section we discuss advantages and disadvantages of each of these models and highlight their differences. At the end of this section, we present the organization for the remainder of this chapter.

3.1.1. Temporal Formulas

An advantage of temporal formulas is that their semantics is precisely defined by *temporal logic*, which is a formal model for reasoning about time. Various systems of temporal logic have been developed, including *interval logic*, which extends temporal logic for reasoning about intervals of time. In recent years, temporal logic has been adapted to reasoning about timing dependencies of concurrent programs, e.g., "If X occurs, then eventually the program will satisfy property P."

Burstall was the first to introduce temporal logic in defining the semantics of computer programs (Ref. 77). Pnueli was the first to formalize a methodology for temporal reasoning about programs (Ref. 69). Temporal logic provides the mechanics for reasoning about both the past and the future; however, in specifying program behavior Pnueli used only the future fragment of temporal logic. Throughout this chapter we restrict our discussion of temporal formulas to assertions on the current state and future states of a program, although assertions on the past are also possible.

Temporal logic has been applied in proving program properties of various concurrent systems, such as

- communication protocols (Ref. 78);
- distributed systems, including communication systems, process control systems, and a prime-number generator (Ref. 68);
- multi-process programs written in Communicating Sequential Processes (CSP) (Ref. 79, 80);
- operating systems, e.g., UNIX, Modula, and MESA (Ref. 81);
- access protocols of a local area network (Ref. 82).

Temporal formulas are useful for reasoning about eventualities (requirements that specific properties eventually become true) and invariances (requirements that specific properties are always true). A disadvantage of temporal formulas is that they are complex and difficult to understand. Furthermore, the literature is complicated: notation differs; terminology varies; and controversy continues over the choice of a temporal-logic system.

The lack of conformity in notation can be attributed partly to variations in typesetting facilities, but also involves redefinitions of basic temporal operators. In addition, the terminology of program properties differs, e.g., "absence of individual starvation" is called "response to insistence" and "weak eventual fairness." It is often unclear whether terms such as these have identical semantics, or if, instead, they differ in their underlying assumptions.

Although it is widely agreed that temporal logic is useful in concurrent program verification, researchers disagree on the choice of a temporal-logic system. The key controversies are

1. state-based specifications with history variables *versus* event-based specifications,
2. linear-time logic *versus* branching-time logic,
3. temporal logic *versus* interval logic.

Briefly, these systems are defined as follows.

State-based temporal specifications model program behavior as successions of state of some abstract machine. A *history variable* is a state encoding of the sequence of values that will be obtained by a program variable before reaching the current state. *Event-based* temporal specifications model program behavior as sequences of events.

In *linear-time* logic we consider all possible future paths of an execution, but consider each possible path independently, so that each state has exactly one successor. (Linear-time operators describe events along a *single* future.) In *branching-time* logic we consider all possible paths at once, so that each state may have many successors. (Branching-time operators allow quantification of events over possible futures.)

Temporal-logic operators express properties that extend into an infinite future. *Interval-logic* operators express properties that hold over a bounded future.

These controversies over temporal logics have been motivated by the following arguments:

- A temporal formula is an assertion about the future progress of the program, although not necessarily from the first state of a computation. Some mechanism is required for establishing prior history, e.g., history variables, event specifications, or interval logic.
- Lamport, Emerson, and others argued that a branching-time logic is needed to capture the semantics of distributed programs. Several branching-time logics have been proposed for program verification (Ref. 67, 83, 84).
- Wolper (Ref. 79) argued that event operators are needed to express arbitrary regular properties, such as: "*p* must be true in every even state of a sequence." Wolper's claim has been refuted (Ref. 85); however, other arguments have been put forth in favor of event operators, e.g., their suitability for reasoning about concurrent programs that communicate via message passing.
- Lamport argued that *short-term fairness* is expressible neither in linear-time nor in branching-time logic (Ref. 67). Short-term fairness expresses the requirement: "For some fixed integer N, a process cannot hold a request for more than N consecutive states without receiving a response." The property of short-term fairness can be expressed in interval logic.

Controversy over the best temporal-logic system reflects the difficulty of representing knowledge about time-related events. By restricting our application of temporal logic to the behavior of programs, we can focus on practical issues and avoid many of the philosophical difficulties that complicate a more general theory of time. These difficulties include reasoning about continuous change (as opposed to discrete changes), outdated beliefs, events that cause no change in state ("no-ops"), and tensed verbs in natural-language statements.

A detailed discussion of issues in a general model of time can be found in work on cognitive theories of time (Ref. 86, 87, 88), on the design of historical databases (Ref. 89), and on natural-language processing (Ref. 90). Turner reviews various temporal logics and discusses their importance in artificial intelligence (Ref. 91).

3.1.2. Interval Formulas

Like temporal formulas, interval formulas have a formal semantics. An important advantage of interval formulas is that they allow specifying behavior that must occur in the "near" future and not before the end of time. Several systems of interval logic have been developed.

In this chapter we develop an interval-logic system, called ℭ, that is based on linear time and supports both state-based and event-based specification techniques. The differences between our approach and those reviewed in this chapter stem from our interest in Ada's concurrency paradigm and from our emphasis on debugging.

3.1.3. Path Expressions

While temporal-logic specifications are based on an axiomatization of time, path expressions rely on an operational definition of time. An advantage of path expressions is that they are easy to understand. Although regular expressions are well understood, the semantics of path expressions is less well-defined.

Furthermore, unlike temporal formulas and interval formulas, path expressions provide no mechanism for requiring that a sequence of events ever occur; instead, they specify sequences of events that are *allowed* to occur.

Path expressions are useful for specifying a sequence of alternating events (e.g., message receipt follows message transmission). We show that a restricted class of path expressions can be transformed into temporal formulas.

3.1.4. Petri Nets

Advantages of Petri nets are their graphical form and their amenability to mechanization. Also, unlike path expressions, Petri nets can express eventualities, i.e., requirements that specific events occur. The disadvantage of Petri nets is state explosion, a problem of any state-transition model.

Lauer and Campbell have shown that any arbitrary path expression can be translated into a corresponding Petri net (Ref. 92). The transitions of a Petri net correspond to the actions of a path expression. A Petri net is said to *simulate* a path expression if and only if the set of strings it generates is exactly the set of strings accepted by a path expression.

3.1.5. Organization of this Chapter

The remainder of this chapter is organized as follows. Section 3.2 presents an overview of temporal logic. In Section 3.3 we discuss the temporal semantics of states and events and distinguish between state-based and event-based specification techniques. Section 3.4 reviews state-based temporal-logic specification techniques for modeling the semantics of concurrent programs. Section 3.5 describes event-based specification techniques (including path expressions) and state-transition specification techniques (including Petri nets). In Section 3.6 we investigate the relationships between path expressions, event expressions, and temporal formulas. We present a formal method for transforming a restricted class of path expressions into temporal formulas. Section 3.7 reviews various interval-logic systems for reasoning about temporal relationships among program states. Section 3.8 introduces our interval-logic system, $\mathbb{C}$. We summarize our conclusions in Section 3.9.

In Section 3.4 we consider the following state-based specification systems:

1. Systems D, DX, and *Propositional Temporal Logic* (PTL) use linear-time logic for reasoning about concurrent programs that communicate via shared memory (Ref. 69, 66, 93).
2. *Computation Tree Logic* (CTL and CTL*) uses a branching-time logic for reasoning about distributed programs that communicate via shared memory (Ref. 83, 84).
3. *Extended PTL* (EPTL) uses a linear-time logic for reasoning about concurrent programs that communicate via message passing (Ref. 79).

In Section 3.5 we consider the following event-based specification techniques:

1. *The Event-Based Specification* Language (EBS) uses a linear-time logic for reasoning

about distributed programs that communicate via message passing (Ref. 68).

2. Vogt's *event specifications* use a linear-time logic for reasoning about concurrent programs that communicate via message passing (Ref. 94).
3. *Path expressions* use regular expressions for reasoning about concurrent programs that communicate via shared variables.
4. *Petri net graphs* use state transitions for reasoning about distributed programs that communicate via shared memory.

EPTL and Vogt's event specifications are closely related, forming a bridge from state models to event models.

Table 3-1 summarizes our presentation. This table shows the program properties that can be specified in the various temporal-specification languages that we review. For each property that can be specified in a language, we show the temporal operator that the language provides to express that property. These properties, languages, and operators are defined in the body of this chapter. Tables 3-2 and 3-3 summarize the definitions of temporal-logic operators. Tables 3-4, 3-5, and 3-6 summarize important properties for concurrent programs.

3.2. Temporal Logic

Temporal logic is a branch of philosophical logic that deals with sentences that can *become* true or false with the passage of time. The goal of temporal logic is to formalize reasoning about sentences that have a temporal context.

The history of temporal logic dates back to the ancient Greeks, but interest in it was revived in the late 1940's, when the R-calculus was introduced as a basic formal model for temporal reasoning. Rescher and Urquhart (Ref. 95) provide an historical perspective of temporal logic from the time of antiquity and present a formal development of various temporal-logic systems. The formalism presented here is drawn largely from this source. (We use their notation as well, with minor variations.)

3.2.1. Defining Temporal Logic

The essence of temporal logic is

- statements will vary in truth-status over time, and
- operations can be defined on time-dependent statements.

A statement is *temporally definite* if its truth or falsity is independent of when it is asserted, e.g., "It sometimes rains in Los Angeles." That is, if this statement is true today, it must have been true always in the past and it always will continue to be true in the future. Otherwise a statement is *temporally indefinite*, e.g., "It is raining in Los Angeles today." That is, the truth-status of a temporally indefinite statement depends on when it is asserted.

Table 3-1: A Comparison of Expressiveness in Temporal Models

Program Property	Specification Languages				
	System D	System DX	State Formulas (PTL)	Path Formulas (CTL)	Extended Formulas (EPTL)
Safety Properties	□	□	□	E□	□
Liveness Properties	◊	◊	◊	E◊	◊
Immediate Responsiveness (no intervening events)	none	O	O	EO (partial ordering)	O
Precedence Properties (FIFO ordering)	none	none	U ("until")	EU	U ("until")
Sequencing	none	O	O	EO	";"
Exclusive Or	~(α∧β)	~(α∧β)	~(α∧β)	~(α∧β)	"+"
Iteration	none	none	none	none	Kleene star
Short-Term Fairness	none	none	none	none	none
Distributed Properties (simultaneous events)	none	none	none	A□, A◊ AU, AO	none

Table 3-1, concluded

Program Property	Specification Languages: Vogt's Event Spec's	Path Expressions	Petri Nets	EBS	Schwartz's Interval Formulas
Safety Properties	□	**path** α **end**	"holds"	->	[I]□
Liveness Properties	◊	none	"enables"	=>	[I]◊
Immediate Responsiveness (no intervening events)	O	none	none	none	yes
Precedence Properties (FIFO ordering)	U	";"	"enables"	=>	<=, =>
Sequencing	none	";"	"enables"	->	<=, =>
Exclusive Or	~(α∧β)	"+"	yes	~(α∧β)	~(α∧β)
Iteration	none	Kleene star	yes	none	none
Short-Term Fairness	none	none	none	none	yes
Distributed Properties (simultaneous events)	none	yes	yes	->, =>	yes

A *definite date* is a time specification that is *chronologically stable.* That is, the time at which a definite date is referenced does not change its specification, e.g., "30 November 1985." A time specification that changes with respect to the occasion of reference is a *pseudo-date*, e.g., "yesterday" or "now." A pseudo-date is *chronologically unstable.*

The introduction of "now" separates temporal from *chronological logic.* Temporal logic assumes a dating scheme based on a reference point that is a pseudo-date; thus, all dates in the scheme are pseudo-dates, e.g., "now," "yesterday," and "tomorrow." On the other hand, chronological logic assumes a dating scheme based on a reference point that is a definite date; thus, all dates in the scheme are chronologically stable. A chronological logic provides the added expressibility of metrics; that is, we can differentiate between the "near future" and the "distant future."

Time can be regarded as either *discrete* or *continuous.* In discrete time, the ordering of temporal instants is isomorphic to that of the integers. Regarding time as discrete is to view it as consisting of a series of distinct instants, such that instants are comparable on their relative position on a time-line. Regarding time as continuous is to assume that between any two instants there will always be a third one. With continuous time, it makes no sense to speak of the *next* instant after the given one. We assume discrete time throughout this work, a natural assumption in computer systems.

3.2.2. The Basic Temporal Model: System $\mathbb{R}$

Rescher and Urquhart have shown that the basic temporal-logic system, $\mathbb{R}$, is *complete* and *decidable.* A system is *complete* if every valid assertion in that system is a theorem. In system $\mathbb{R}$ the temporal context of a statement is provided by the operation of *temporal realization.* The operator R asserts that statement A holds (is realized) at the particular instant *t*, written $R_t(A)$. By convention, if A is a temporally definite statement, then $A \equiv (\forall\, t)\ R_t\ (A)$. If *t* is a pseudo-date, and A is some temporally indefinite statement, then $R_t(A)$ is temporally indefinite.

A set of rules for the R-operator constitutes a temporal calculus over R, called system $\mathbb{R}$. Sentences in $\mathbb{R}$ are composed from the following set of primitives:

1. Propositional variables, such as A and B, ranging over both temporally definite and indefinite statements.
2. Variables, such as *t*, *s*, and *u*, for time instants, i.e., as either definite or pseudo-dates.
3. The variable *n*, for the pseudo-date, "now."
4. The connectives of propositional logic, including ~ for negation, $\wedge$ for conjunction, $\Rightarrow$ for implication, and $\equiv$ for equivalence.
5. The quantifiers, $\forall$ (the universal quantifier) and $\exists$ (the existential quantifier), ranging over time. (In Section 3.2.9 we will extend $\forall$ and $\exists$ to range over other entities as well.)
6. The identity predicate, =, for comparing variables or *n*.
7. The temporal operator R.

The usual principles of logic are assumed for sentences in $\mathbb{R}$ that do not involve n or R. Thus, the following rules specify sentences that are *well-formed formulas* (wffs) in $\mathbb{R}$:

1. A propositional variable is a wff.
2. If α is a wff, then $\sim\alpha$ is a wff.
3. If α and β are wffs, then $\alpha \wedge \beta$ is a wff.
4. If α and β are wffs, then $\alpha \Rightarrow \beta$ is a wff.
5. If α and β are wffs, then $\alpha \equiv \beta$ is a wff.

System $\mathbb{R}$ is based on the following axioms:

$$R_t(\sim A) \equiv \sim R_t(A) \tag{1}$$

$$R_t(A \wedge B) \equiv [R_t(A) \wedge R_t(B)] \tag{2}$$

$$R_n(A) \equiv A \tag{3}$$

$$R_s[(\forall t)A] \equiv (\forall t)\,[R_s(A)] \tag{4}$$

$$R_s[R_t(A)] \equiv R_t(A) \tag{5}$$

$$R_t(n=s) \equiv (t=s) \tag{6}$$

$$R_t(s=u) \equiv (s=u) \tag{7}$$

$$(\forall t)A \Rightarrow A^{t/n} \tag{8}$$

and the following rules of inference:

$$\text{If A is a tautology (i.e., a valid wff) then } \vdash A \tag{9}$$

$$\text{If } \vdash A \text{ then } \vdash (\forall t)\, R_t(A) \tag{10}$$

$$\text{If } \vdash (A \equiv B) \text{ then } (\vdash A) \equiv B \tag{11}$$

In Axiom 4, t and s must be distinct. In Axiom 8, $A^{t/n}$ means that n can be substituted for every free occurrence of t in A. It assumes that t does not occur within the scope of an R-operator in A.

3.2.3. Temporal Precedence

A genuine temporal logic requires not only the introduction of presentness ("now"), but also the relationship of *temporal precedence*: $t < s$ for "time t is before time s." Temporal precedence is introduced into system $\mathbb{R}$ by including the precedence relation as a primitive and adding the following axioms:

$$R_t(n < s) \equiv t < s$$

$$R_t(s < n) \equiv s < t$$

$$R_t(s < u) \equiv s < u \text{ whenever } s \neq n,\ u \neq n$$

3.2.4. Linear Time versus Branching Time

Temporal logic is commonly based on a *linear series*, that is, a single course of time, but can be based also on *branching time*, that is, possible future courses of events. If we assume branching time, then temporal precedence determines only a partial ordering on events, i.e., $t < s$ means that a set of events may occur at time s, which is later than time t. Branching time is represented by a tree structure.

A linear series is the standard picture of time. If we think of time as linear, then instants are comparable on their relative ordering of earlier, contemporary, or later. A temporal logic is *complete* with respect to linear time, if (in addition to the axioms of system $\mathbb{R}$) it requires *transitivity* and *connectedness* of the precedence relation:

$$(t < s) \wedge (s < u) \Rightarrow t < u$$

$$(t < s) \vee (t = s) \vee (s < t)$$

3.2.5. Tense Logic

Temporal precedence establishes an ordering of instants in terms of earlier or later. In combination with presentness ("now"), temporal precedence establishes a temporal ordering with respect to "now," i.e., *past*, *present*, and *future* times. The basic set of *tense* operators includes F ("future"), P ("past"), G ("henceforth"), and H ("heretofore"). Table 3-2 defines these basic tenses as functions on propositional variables p and q. A *tense* is a function on a set of propositions, such that the truth-status of the function is given by an equivalence to a wff in $\mathbb{R}$.

Not all relationships within $\mathbb{R}$ can be expressed in tense logics, e.g., some properties of temporal ordering cannot be expressed, although *continuity* can be. Continuity means that, for continuous time, no gaps exist between instants, i.e., there is a continuum of instants between any two instants.

Not every tense operator can be defined by the basic set of tense operators. For example, Kamp (Ref. 96) proved that S ("since") and U ("until"), defined in Table 3-2, are two binary tenses that cannot be expressed with only F, P, G, H, and propositional connectives. Both F and P, however, can be defined by S and U:

$$Fp \equiv (p \vee \sim p)\ U\ p$$

$$Pp \equiv (p \vee \sim p)\ S\ p$$

Kamp also showed that if time is linear, dense, and infinite in both directions (past and future), then every tense can be expressed in terms of S and U.

3.2.6. Temporal Modality

Modal logic provides another basis for tense logic. Hughes and Cresswell present an introduction to the theory of modal logic, of which *temporal modality* can be considered a part (Ref. 97). Modality is the logic of *necessity* ("It must be the case.") and *possibility* ("It may be the case.").

Table 3-2: Basic Tense Operators

Operator	Tense	Definition
The basic tense operators are		
Fp	"It will be that p" (future)	$(\exists t)\ [n<t \wedge R_t(p)]$
Pp	"It has been that p" (past)	$(\exists t)\ [t<n \wedge R_t(p)]$
Gp	"Henceforth always p" (p will always be true in the future)	$(\forall t)\ [n<t \Rightarrow R_t(p)]$
Hp	"Heretofore always p" (p has always been true in the past)	$(\forall t)\ [t<n \Rightarrow R_t(p)]$
Two binary tense operators are		
pSq	"p since q" (p has been true since q)	$(\exists t)\ \{t<n \wedge R_t(q) \wedge (\forall s)\ [(t<s<n) \Rightarrow R_s(p)]\}$
pUq	"p until q" (p will be true at least until q. (Eventually q holds.)	$(\exists t)\ \{n<t \wedge R_t(q) \wedge (\forall s)\ [(n<s<t) \Rightarrow R_s(p)]\}$
The tense operators of modal logic are		
$\Diamond p$	"Eventually p"	$p \vee Fp$ $(\exists t)\ [n \leq t \wedge R_t(p)]$
$\Box p$	"Always p"	$p \wedge Gp$ $(\forall t)\ [n \leq t \Rightarrow R_t(p)]$

The notion of a temporal modality dates back to classical antiquity and is attributed to Diodorus Cronus, a Stoic logician. Cronus provided a tensed semantics for modality:

Necessity	It is true *now and always* will be true.
Possibility	It is true *now or* will be realized *some time* in the *future*. (It may become false again after becoming true.)

The temporal modal operators are □ for necessity ("always") and ◊ for possibility ("eventually"), as defined in Table 3-2. Modal operators can be combined, e.g., ◊ □ ("eventually henceforth") and □ ◊ ("infinitely often"). In linear time, the operators ◊ and □ are duals:*

$$\Diamond p \equiv \sim\Box\sim p$$

Temporal modality is also called "classical temporal logic" in the literature, because it extends classical (propositional) logic.

Other tense operators can be defined with □, ◊, and U. (See Table 3-3.) For example, Lamport introduced a binary version of the □ operator, meaning "as long as." (Note that the unary operation $\Box p$ is expressible as "true □ *p*.") The binary operator N ("unless") was introduced by Nguyen (Ref. 98). Schwartz (Ref. 70) introduced the operator **latches_until**, a variation on the U-operator. Ramamritham and Keller (Ref. 99) derived the binary operators **onlyafter** and **after**. The binary operator -> ("precedes") is useful for reasoning about a FIFO ordering.

3.2.7. Chronological Logic: System $\mathbb{R}^+$

If *n* is fixed, then we obtain a system for chronological logic: system $\mathbb{R}^+$. Let T be a set of values of temporal variables, e.g., *t*, *s*, and *u*. The set T is a *metric space* if there is a distance function *d*, defined over all pairs of T-elements, such that

$$d(t,s) = 0 \text{ iff } t = s$$

$$d(t,s) + d(s,u) \geq d(u,t)$$

where we require that $d(u,t) = d(t,u)$. If T is a metric space, then a system based on the R-calculus defined over the set T is said to embody *metric time*.

To move to a chronological system, the standard R-calculus is modified by replacing *n* by the identity element *i*, as a point of reference. System $\mathbb{R}^+$ is derived from the R-calculus as follows:

1. Let ⊕ be a binary operator over T and let *i* be an identity element in T, such that {T, ⊕, *i*} constitutes a group that is commutative and additive. The date $i \oplus t = t$ means "*t* units after *i*," and the date $i \oplus -t = -t$ means "*t* units before *i*."
2. Replace all occurrences of *n* by *i* in Axioms 3, 6, 7, and 8 of system $\mathbb{R}$.
3. Replace Axiom 5 of system $\mathbb{R}$ by the following axiom:

$$R_s[R_t(A)] \equiv R_{s \oplus t}(A)$$

---

*Lamport has shown that this duality does not hold for branching time, but more about this later (Ref. 67). (See Section 3.4.2.)

Table 3-3: Derived Tense Operators

Operator	Tense	Definition
The operators of modal logic can be combined:		
$\Diamond \Box p$	"eventually henceforth p"	
$\Box \Diamond p$	"infinitely often p"	
New tense operators can be derived from existing tense operators:		
$p \Box q$	"q is true as long as p remains true"	$q \mathrm{U} \sim p$
$p \mathrm{N} q$	"p unless q"	$\Box p \vee (p \mathrm{U} q)$
$p \text{ -> } q$	"p precedes q"	$\Diamond p \Rightarrow (\sim q \mathrm{U} p)$
p **latches_until** q		$(p \Rightarrow (p \mathrm{U} q)) \mathrm{U} q$
p **onlyafter** q	"p can become true only after q does" (does not assume eventuality of q)	$\sim p \mathrm{U} q$
p **after** q	"p will become true after q does" (at same time or later, does not assume eventuality of q)	$(\sim p \mathrm{U} q) \wedge \Diamond p$

Various metric times are possible. For example, a metric time that is linear and non-circular is obtained if T is the set of integers, $d(t,s) = |t-s|$, and i is 0. Section 3.7.3 reviews a specification language based on metric time (Ref. 73).

3.2.8. Interval Logic and Temporal Patterns

The truth status of a temporally indefinite statement can hold over a range of times, called an *interval*, e.g., "The temperature in Los Angeles has been above 100 degrees for two weeks." A temporal interval is a range of instants such that the first endpoint is not later than the second endpoint.

Although Rescher and Urquhart provide only a brief discussion of interval logic, several systems have been proposed in recent years. For example, Allen proposed the use of intervals in building hierarchies of temporal knowledge for natural-language understanding (Ref. 100, 87). Allen takes intervals as primitives. To allow for reasoning about time points, he introduced a *point interval*, i.e., an interval whose endpoints coincide. This model assumes that all intervals are bounded (i.e., closed at both ends). Allen defined a set of 13 primitive *temporal relations* between intervals, such as "during," "before," "overlap," "meets," and "equal." Allen and Hayes showed that the only interval relation required is "meets," because the other relations can be defined with the "meets" relation, the existential quantifier, and propositional connectives (Ref. 88). For example, the relationship "before" holds between two intervals if there exists an interval between them.

In other interval logics, time points are primitives. For example, McDermott developed an interval logic based on points in continuous time (Ref. 86). In this system, intervals can be unbounded. Also, Clifford and Warren developed an interval logic in which time points are primitives, and intervals are required to be closed and dense (Ref. 89).

A "process," or activity, occurs over a certain interval of time. Rescher and Urquhart characterize an activity by its *temporal pattern*:

homogeneous	Can go on for all times throughout the interval, e.g., flying an airplane, running around a track.
majoritative	Can go on at most times throughout the interval, e.g., writing a letter, whistling.
occasional	Can go on at some times throughout the interval, e.g., sneezing, a rooster crowing.
wholistic	Can go on for the whole interval, but not for only a part of the interval, e.g., flying an airplane from Los Angeles to New York, running around the track three times, a ball changing its position from location X to location Y. (That is, a description of the activity involves some *initial* and *final* conditions.)

We can observe examples of these temporal patterns in the behavior of an Ada program. For example, a task waiting in an entry queue is a homogeneous activity. Redefinition of variables is majoritative. An example of an occasional activity is reading input data. Executing a rendezvous between any given pair of tasks is an wholistic activity.

In reasoning about program behavior, however, we need consider only homogeneous activities, i.e.,

program states (in the sense of a "snapshot" of a program's behavior), and wholistic activities, i.e., sequences of events (or of state transitions) that trigger the beginning and ending of an activity. Hence, we can provide a simpler semantics for interval logic. Section 3.7 reviews interval logics for specifying program behavior.

3.2.9. Temporal Logic and Predicate Logic

The temporal logic that we have presented so far is an extension of *propositional logic*: It deals with sentences to which a truth value (true or false) can be assigned. Sentences in propositional logic are composed of propositions and connectives. Every proposition in a sentence is either true or false, but not both. *Predicate* temporal logic deals with temporally indefinite sentences that contain occurrences of free variables that may range over domains other than time, e.g., "For some X, X is a task that eventually calls itself."

Predicate temporal logic is an extension of *predicate logic* (also called *quantified* logic), which is concerned with propositions about objects. Propositions in temporal logic are either *atomic* or *compound*. An *atomic proposition* is an expression of the form:

$p(X_1,...,X_k)$

where p is a *predicate symbol*, which expresses the relationship between objects, and the arguments $X_1,...,X_k$ are a set of *terms*, which represent objects. A term can be a constant, a variable, or a *compound term*. A compound term consists of a *function symbol* and an ordered set of terms as arguments. The operators $\exists$ and $\forall$ specify quantification over variables. *Compound propositions* are formed by combining propositions with logical connectives: ~ (negation), $\wedge$ (conjunction), $\vee$ (disjunction), $\Rightarrow$ (implication), and $\equiv$ (equivalence).

Predicate temporal logic requires the following extensions to the primitives of system $\mathbb{R}$:

1. Variables, e.g., X, Y, Z, ..., are introduced for objects that range over domains other than time.
2. A and B are *predicate variables* ranging not only over time, but over the domains of variables X, Y, Z, ..., as well.
3. The quantifiers, $\forall$ and $\exists$, are allowed to range not only over time, but also over other domains.

The usual rules of predicate logic are assumed:

1. An expression consisting of a predicate symbol and a finite number of arguments is a wff. We will write α_X for a predicate consisting of a predicate symbol α with argument X.
2. If α is a wff, so is $\sim\alpha$.
3. If α and β are both wffs, so is $\alpha \vee \beta$.
4. If α is a wff and X is a variable then $(\forall X)\alpha_X$ is a wff.
5. $(\exists X)\alpha_X \equiv \sim(\forall X)\sim\alpha_X$

The following rules can be derived:

$(\forall X)\alpha_X \Rightarrow (\exists X)\alpha_X$

$$\sim(\forall X)\alpha_X \equiv (\exists X)\sim\alpha_X$$

Rescher and Urquhart point out one important difficulty in extending predicate logic with temporal operators: If quantification is allowed to range over domains other than time, then it is necessary to specify the "temporal domain" over which variables will range. If we assume the temporal domain to be either of the following cases:

1. All individuals that exist now.
2. All individuals existing at time *t*.

then the rule:

$$(\forall X)R_t[\alpha_X] \Rightarrow R_t[(\forall X)\alpha_X]$$

is invalid.

For example, let X range over Ada tasks and let α_X be the predicate: "X has not yet been created." From the first case above, we can derive the following invalid proposition: "If all tasks that exist now were not yet created at time *t*, then at time *t* all existing tasks were not yet created." On the other hand, if we take the temporal domain to be any of the following cases:

3. All individuals that have existed up to and including now.
4. All individuals that have existed up to and including *t*.
5. All individuals, including those that have existed some time in the past, that exist now, or that will exist at some time in the future.

then extending system $\mathbb{R}$ to predicate temporal logic requires adding the following axioms:

$$R_t[(\forall X)A_X] \equiv (\forall X)R_t(A_X) \tag{12}$$

$$R_t[(\exists X)A_X] \equiv (\exists X)R_t(A_X) \tag{13}$$

Similarly, in modal logic the "Barcan formula":

$$(\forall X)\ \Box\alpha_X \Rightarrow \Box(\forall X)\alpha_X$$

is controversial, although its converse is easily proved (Ref. 97). If we assume the validity of the Barcan formula, then the following rules can be derived for tense logic:

$$\Box(\forall X)\alpha_X \equiv (\forall X)\Box\alpha_X$$

$$\Diamond(\exists X)\alpha_X \equiv (\exists X)\ \Diamond\alpha_X$$

Program verification takes the temporal domain to be the third case. In verification we always reason from the current state into the future. Hence, the Barcan formula is implicitly assumed.

In debugging we reason about the past and must often choose between cases one and three, or between cases two and four. Hence, the Barcan-formula controversy cannot be avoided in debugging (or in any historical database). In Chapter Four we address the implications of this controversy for debugging.

A second difficulty of predicate temporal logic is that it is *undecidable*. That is, it is not always possible to decide the validity of an arbitrary sentence in predicate temporal logic. Hughes and Cresswell have shown that the equivalent modal-logic system, known as LPC (for lower predicate calculus), is undecidable.

3.3. The Temporal Context of Events and States

The words "event" and "state" have a wide range of interpretations; however, any description of an event or state carries with it a temporal specification. We can say that an event occurs at time *t*, or that a state holds at time *t*. This view of time is called *absolutist*: Times are not differentiated by events; events may be differentiated by times. In absolute time (as opposed to *relative* time) a global clock is assumed, and the same events or states can recur.

In *relative* time, a cluster of events constitutes an instant of time, i.e., events differentiate times. If no events occur, then there is no time. Relative time is a common assumption in cognitive theories of time (Ref. 86, 88). Relative time is assumed also in the debugging techniques of "checkpointing" (Ref. 42) and "sampling" (Ref. 55). In this dissertation we restrict our attention to absolute time, which is appropriate in tracing a program's execution.

The remainder of this section is organized as follows. First, we discuss the definitions of events and states in a general theory of temporal logic. Second, we distinguish between *event-based* and *state-based* approaches to modeling program behavior.

3.3.1. Defining Events and States

The definition of events and states varies according to which is taken as primitive and how each is described temporally. In general, events and states are duals. An event is an operation whose occurrence causes a state transition. A state is a memory of the events that have previously occurred, i.e., an encoding of previous events.

For Rescher and Urquhart, an event is a primitive element that occurs at a single time instant. An event can be described either by its relationship to other events (e.g., retrospective, prospective, contemporary, simultaneous) or without reference to other events (i.e., a "pure" description).

Events are primitives in Petri-net theory, also (Ref. 101). A state is a predicate on a sequence of events. An event is an action that takes place in the system. A state is a set of conditions that control the occurrence of an event.

On the other hand, Allen (Ref. 100, 87, 88) defines events as primitives that take time. An event occurs over an interval. Events that cause no change in state occur over point intervals. An event is a predicate on an ordered pair of states. In this model, time is assumed to be relative and can be either continuous or discrete.

In developing a model of time for planning actions (e.g., for robots), McDermott also defined events as primitives (Ref. 86). In this model, time is continuous, infinite, and branching into the future (many futures; one past). A state is "an instantaneous snapshot of the universe." The universe consists of facts, or propositions, whose validity is determined in each state. Events are actions that cause changes in facts. State changes occur in a single instant; events occur over an interval (but may occur over an instant).

In McDermott's model of time, each occurrence of an event is called a *token*. Many different event tokens can take place over a given interval; thus, a cluster of event tokens constitutes an interval. Hence, as in Allen's model, relative time is assumed. McDermott argued that Allen's approach, in which event tokens are defined on ordered pairs of states, is inadequate for a model in which relative time is assumed. Instead, event tokens are defined as predicates on ordered pairs of events and intervals.

Snodgrass takes both events and states as primitive. An event is a change that occurs in the system state at an instant. A state is a relationship that is valid for the duration of an interval. Events delimit states; states generate events by change. The duality of events and states accommodates two approaches to modeling program behavior: an event stream and successions of state.

3.3.2. State-Based versus Event-Based Temporal Models

In this dissertation we restrict our attention to events and states of a program's execution. A state of a program consists of a *data* component and one *control* component for each process. The data component consists of the current values assigned to program variables. Each control component gives the current location of the next instruction to be executed within that process. An event of a program is an action that causes a state change.

We characterize program-specification methods by the way in which execution is modeled and by the expressive power the language provides for specifying constraints on behavior. Execution models are either *state-based* or *event-based*. *State-based specifications* model program behavior as successive states of some abstract machine. *Event-based specifications* model program behavior as an event stream. State-transition systems, such as Petri nets, model program behavior as changes in state, which are atomic events.

Event models are useful for verifying assertions about the behavior of concurrent programs that communicate via message passing. On the other hand, state-based specifications are more appropriate for describing concurrency via shared memory. We are forced to explore both models because Ada provides communication via both shared memory and message passing.

3.4. State-Based Temporal Models

A *state model* is a triple, (I,α,σ), where

- I is a mapping that assigns elements, functions, and predicates to individual constants, functions, and predicate symbols,
- α is a mapping that assigns constant values to program variables, and
- $\sigma = <s_0, s_1,...>$ is an infinite sequence of states.

Recall that a state consists of a data component and one control component for each process. Let S denote the instruction at program location m_i. To specify the control component of a state, predicates on program labels are introduced:

at m_i	Control is at the beginning of the execution of instruction S.
after m_i	Control is just after instruction S is executed.
in m_i	Control is within instruction S.

These predicates specify control points, i.e., conditions leading into or out of a specified state. For example, "**at** m_i" is true in state *s* if, when in state *s*, the program is about to execute the instruction at location m_i.

Control predicates provide a temporal context for state, such that assertions can be made about the current state, instead of from the "first" state. Thus, a state-based temporal formula assumes that future behavior depends only on the current state, and not on prior states (i.e., not on how the current state was reached).

A proposition is a statement about the current state of the program. Propositions are composed from the following primitives:

1. Program labels: $m_1,...,m_n$.
2. Program variables: $y_1,...,y_k$.
3. Constants: $c_1,...,c_j$.
4. Predicates and function symbols: ϕ, ψ, ξ, ... , ranging over program variables.
5. The predicates **at**, **in**, and **after**, ranging over program labels.

A proposition can be

1. A predicate on a program label.
2. A truth-valued function on a set of atomic predicates over program variables.

3.4.1. Linear Time and State Formulas

A linear-time, state-based temporal formula (state formula) is an assertion about a sequence of states, which can include the current state and future states. Temporal models using state formulas have evolved from the minimal system D to the progressively more expressive systems: DX, DUX (PTL), and

EPTL. These models assume a *global state*, so that no two processes execute at the same time. That is, state formulas require a "single event condition": Each state is obtained from the previous state by the execution of exactly one instruction; each state transition corresponds to one atomic instruction.

3.4.1.1. A Discrete Temporal Model: System D

Using only the operators ◊ ("eventually") and □ ("always"), Pnueli proved properties about a concurrency model that achieves communication via shared memory. He interpreted temporal operations as constraints on the computation of the program:

$\Box p$ is true in state s_i, if p is true in all states s_j for $j \geq i$

$\Diamond p$ is true in state s_i, if for $j \geq i$

there exists a state s_j in which p is true

Here, p is an arbitrary proposition.

A *well-formed state formula* (wfsf) consists of

1. Propositions: $p_1,...,p_k$.
2. The connectives of propositional logic, including ~ for negation, $\wedge$ for conjunction, and $\Rightarrow$ for implication.
3. Temporal operators: □ and ◊.

Each state contains truth assignments to all propositions $p_1,...,p_k$.

For example, the following are wfsfs:

$\Box(\text{at } m_i \Rightarrow \psi)$

$(\text{at } m_1 \wedge \phi) \Rightarrow \Box(\text{at } m_2 \Rightarrow \phi)$

$\Box(\text{at } m_i \Rightarrow \Diamond \text{ at } m_j)$

$\Box(\sim(\text{at } m_1 \wedge \text{at } m_2))$

A temporal-logic system based on the operators □ and ◊ has been studied under the name D (for time taken as discrete) (Ref. 97). System D is equivalent to the general modal-logic system S4.3.1. Continuous time requires a weaker system, known as S4.3, which is contained in D.

System D consists of the following axioms:

$\Box A \Rightarrow A$

$\Box(A \Rightarrow B) \Rightarrow (\Box A \Rightarrow \Box B)$

and inference rules:

if A is a tautology then $\vdash A$

if $\vdash A$ and $\vdash A \Rightarrow B$ then $\vdash B$

if $\vdash A$ then $\vdash \Box A$

where A and B are arbitrary wfsfs. (Rules for $\Diamond$ can be derived from its duality with $\Box$.)

3.4.1.2. Safety and Liveness Properties

The temporal operators $\Box$ and $\Diamond$ partition program properties into two classes: *safety* and *liveness*. Most properties fall into either of these classes. *Safety properties* describe conditions that are always true. *Liveness properties* describe conditions that must eventually become true (but may become false again after becoming true).

Safety properties specify that the program cannot reach an undesirable state, i.e., that "nothing bad will happen." These properties can be expressed by *invariant assertions*, which are assertions that are always true (temporally definite statements). An example of a safety property for sequential programs is *partial correctness*, which asserts that the program is correct if it reaches its desired final state. For concurrent programs, the class of safety properties includes deadlock freedom, clean (error-free) behavior, and mutual exclusion of processes. To exhibit clean behavior, a program must be free of the errors commonly found in sequential programs: type incompatibilities, exceeded array bounds, and ill-formed arithmetic expressions (e.g., zero division and numerical overflow). Table 3-4 summarizes the definitions of these important safety properties.

Table 3-4: Important Safety Properties

partial correctness	The program is correct if it reaches its desired final state.
clean behavior	The program is free of errors commonly found in sequential programs.
deadlock freedom	Future progress is always possible, unless the program terminates. (No process is ever permanently blocked.)
mutual exclusion of processes	Specified actions cannot be executed simultaneously, e.g., a process can respond to only one request at a time.

Liveness properties assert that the program will progress, i.e., that "something good will happen." These properties are requirements that certain events occur. They can be expressed in *commitments*, which are temporally indefinite statements about the future. *Total correctness* is an example of a liveness property for sequential programs. A program is *totally correct* if it is partially correct and eventually terminates. For concurrent programs, the class of liveness properties includes termination, accessibility, responsiveness, and response to insistence (absence of individual starvation). Table 3-5 summarizes the definitions of these important liveness properties.

Fairness is a condition to proving liveness properties. A program is *fair* if every process has many chances to proceed. Varying degrees of fairness (e.g., "justice") have been studied in the literature.

Table 3-5: Important Liveness Properties

termination	All processes will eventually terminate.
total correctness	The program is partially correct and eventually terminates.
accessibility	Every path will eventually lead to some specified goal.
responsiveness	Every request from a process will eventually receive a response.
response to insistence	A permanent holding of a request will eventually receive a response.
response to persistence	If a process has many chances to proceed, it eventually will.
response to an impulse	A single request guarantees a response.

Combinations of tense operators, $\Box \Diamond$ ("infinitely often") and $\Diamond \Box$ ("eventually henceforth"), are useful for describing *recurring* properties, such as properties of non-terminating programs, response to persistence, and trends among program events (e.g., "the buffer is infinitely often not full"). Also, the presence of deadlock is expressed as a combination of tense operators: "eventually henceforth all processes are waiting."

Mutual exclusion, responsiveness, and fairness are desirable properties for programs written in languages that use a concurrency paradigm that achieves communication via shared variables (e.g., monitors and semaphores). These properties express requirements on access to *shared resources*. Ramamritham and Keller present a resource-control model, in which the execution of an operation on a shared resource consists of four distinct phases (Ref. 99):

1. Request	The user program requests a resource and the availability of the resource is determined.
2. Service	If the resource becomes available (is "enabled"), the requester receives a response, granting the resource.
3. Active	Access occurs.
4. Termination	Access is completed; the resource is released.

We digress for a moment to consider the usefulness of these safety and liveness properties in an Ada environment. Recall that an individual Ada process is called a *task*, and that synchronization between tasks is handled by the *rendezvous* mechanism, which implements message passing by remote procedure call to a *task entry*. (See Chapter One.) To apply a resource-control model in Ada, it is helpful to think of a "shared resource" as an Ada task that is treated as an "agent," such that each of the task's entries is a resource. For example, the shared resource can be a buffer implemented as a task with entries for "read" and "write."

Access to a resource is then achieved by rendezvous and can be partitioned into the following phases:

1. Request	An entry call is executed, the calling task is suspended, and its name is placed in the entry queue.
2. Service	At least one select alternative for an accept statement of the corresponding entry becomes open, an alternative is selected, and the caller is removed from the entry queue (rendezvous begins).
3. Active	The accept statement is executed (rendezvous occurs).
4. Termination	The calling task resumes execution (rendezvous completes).

"Responsiveness" defines varying degrees of liveness. From weakest to strongest, the levels of responsiveness are as follows: response to insistence, response to persistence, and response to an impulse. All these levels of responsiveness can be expressed using the temporal operators: ◊, □, and their combinations (Ref. 93). In the following discussion let propositions p and q be defined as follows:

- p is true iff the process is active and waiting to be chosen (in request phase).
- q is true iff the process has been selected (in service phase).

Response to insistence (also called "weak responsiveness," "weak eventual fairness," or "absence of individual starvation") is a requirement that a permanent holding of a request will eventually receive a response. This property can be expressed as $\sim\Box(p\wedge\sim q)$ ("p cannot remain true forever without q ever becoming true."). It can be expressed also as $\Box p \Rightarrow \Diamond q$ ("If p remains true then it will eventually cause q."). These are requirements on the current state only. Response to insistence can be specified over the current state and over all future states: $\Box(\Box p \Rightarrow \Diamond q)$.

Response to persistence (also called, "eventual fairness," "strong eventual fairness," or "failure freedom") requires that if a process has many chances to proceed, it eventually will. This property can be expressed as $\Box\,\Diamond p \Rightarrow \Diamond q$ ("If p is true infinitely often, q will eventually become true."). This is logically equivalent to $\sim\Box(\Diamond p\wedge\sim q)$ ("p cannot become true forever without q becoming true."). Response to persistence can be required over the current state and over all future states, e.g., $\Box(\Box\,\Diamond p \Rightarrow \Diamond q)$.

Response to an impulse is a requirement that a single occurrence of p guarantee q: $p \Rightarrow \Diamond q$. This property can be expressed over all future states as $\Box(p \Rightarrow \Diamond q)$.

3.4.1.3. System D Extended: System DX

Pnueli (Ref. 66) augmented the definition of a state formula to include the unary temporal operator X ("next time"). The operation Xp ("next p") is true at time t, iff p is true at t+1: $R_t(Xp) \Rightarrow R_{t+1}(p)$ (assuming linear metric time). The point-based operator X has since been replaced by the state-based operator O, to specify sequences of states:

Op is true in state s_i iff p is true in state s_{i+1}

System DX is an axiomatization over O, □, and ◊. It contains the axioms and inference rules of system D, as well as the following axioms:

$O(\sim A) \equiv \sim OA$

$O(A \Rightarrow B) \Rightarrow (OA \Rightarrow OB)$

$\Box A \Rightarrow OA$

$\Box A \Rightarrow O\Box A$

$\Box(A \Rightarrow OA) \Rightarrow (A \Rightarrow \Box A)$

Given the single event condition, the state formula $\Box(p \Rightarrow Oq)$ specifies an *immediate response*: "If a process has a chance to proceed, it will do so immediately; it will never be kept waiting."

3.4.1.4. Propositional Temporal Logic: PTL

Gabbay (Ref. 93) showed that some temporal properties of programs cannot be specified using only the operators $\Diamond$, $\Box$, and O. The temporal operator U ("until") is needed for expressing temporal ordering, e.g., p U (q U r) specifies that p, q, and r are in sequence. By considering only the future fragment of tense logic, Gabbay provided a simpler proof than Kamp (Ref. 96) of the need for the U-operator. Lamport (Ref. 67) gave a similar argument for including the binary operation $p \Box q$ ("as long as").

The only temporal operator required (in linear time) is U, because $\Box p \equiv p$ U false, $\Diamond p \equiv$ true U p, $Oq \equiv$ false U q, and $p \Box q \equiv q$ U $\sim p$. For example, the U-operator can express the property of immediate responsiveness: $\Box(\sim q \vee (\sim q \text{ U } p))$ ("Either q is never true or q is not true until p."). Immediate responsiveness can be expressed also with the N-operator: $\Box(q \Rightarrow \sim q \text{ N } p)$.

Propositional Temporal Logic (PTL) is a state-based specification language that extends propositional logic with four temporal operators:

Op	p is true in the next state in the sequence.
$\Box p$	p is true in the current state and in all future states in the sequence.
$\Diamond p$	Either p is true in the current state or there exists some future state in which p is true.
p U q	p is true in every state at least until the first state in which q is true. (Eventuality of q is required.*)

System DUX is an axiomatization of PTL. It consists of all the rules and axioms of DX, plus the following axioms:

$\Box A \Rightarrow A$ U B

A U $B \equiv B \vee (A \wedge O(A \text{ U } B))$

*In some representations of state formulas, eventuality is not required.

3.4.1.5. Precedence Properties

"Until" defines a class of *precedence* properties, which extends the class of liveness properties. Precedence properties include strict (FIFO) fairness and bounded overtaking (also called boundedness *). These properties are defined in Table 3-6.

Table 3-6: Important Precedence Properties

absence of unsolicited response	No response will occur unless preceded by a request. (Any process that makes a request eventually gets a chance to proceed.)
strict fairness	Each process has an equal chance to proceed, e.g., a FIFO discipline or interleaving.
boundedness	The number of messages in each buffer can never exceed the capacity of the buffer.

Gabbay showed that the U-operator is required to express properties of responsiveness augmented with *fairness*, the requirement that each process have many chances to proceed ($\Box \Diamond p$). *Absence of unsolicited response* requires that a resource never be granted unless it is preceded by a request: $\Diamond q \Rightarrow \sim q \ U \ p$ ("Any process that makes a request will eventually get a chance to proceed."). Given the duality of $\Box$ and $\Diamond$ (in linear time), this property is logically equivalent to $\sim q \ N \ p$ ("A resource will not be granted *unless* a request is made."). Recall that $p \ N \ q \equiv \Box p \vee (p \ U \ q)$.

Strict fairness (also called, "strict responsiveness") requires that each process have an *equal chance* to proceed, e.g., a strong FIFO discipline. Consider the following set of propositions for each process P_i, where i=1,2:

- p_i is true iff process P_i is waiting for a resource.
- q_i is true iff process P_i is served.

A strong FIFO discipline requires

$$(\Diamond p_2 \Rightarrow (\sim p_2 \ U \ p_1)) \Rightarrow (\Diamond q_2 \Rightarrow (\sim q_2 \ U \ q_1))$$

which is logically equivalent to each of the following formulas:

$$(\sim p_2 \ N \ p_1) \Rightarrow (\sim q_2 \ N \ q_1)$$

$$(p_1 \dashrightarrow p_2) \Rightarrow (q_1 \dashrightarrow q_2)$$

$$(p_1 \wedge \sim(p_2 \vee q_2)) \Rightarrow (p_1 \ \Box \sim q_2)$$

Recall the following definitions (See Table 3-3 in Section 3-3.):

$$(p \dashrightarrow q) \equiv (\Diamond p \Rightarrow \sim q \ U \ p)$$

$$(p \ \Box \ q) \equiv (q \ U \sim p)$$

* Boundedness is a property of concurrent programs that communicate by *buffered* message passing. Recall that Ada's rendezvous mechanism is based on synchronous message passing; that is, the Ada language defines no bounds on the length of an entry queue. Thus, boundedness has no obvious counterpart for Ada. (See Chapter One.)

Interleaving requires that strict fairness hold throughout a computation, not only for the first access. It requires *at least one p* between consecutive *q*'s:

$(p \text{ -> } q) \wedge \Box(q \Rightarrow (p \text{ -> } q))$

3.4.1.6. A Temporal-Logic Specification Language: SYSL

The SYnchronizer Specification Language (SYSL) is a program-specification language based on predicate temporal logic (Ref. 99). This language uses the temporal-logic primitives of PTL. SYSL has been applied in specifying the semantics of distributed programs that communicate via a shared resource.

In addition, SYSL has been applied in synthesizing the code of a *synchronizer*, a sequential process that guarantees disciplined access to a shared resource. The synthesis algorithms transform specifications into changes to auxiliary variables that are local to processes. The synthesized code specifies necessary conditions for servicing requests and appropriate actions for satisfying fairness. A preprocessor attempts to detect inconsistencies, incompleteness, and deadlock-prone conditions in a given set of specifications.

Statements in SYSL are composed of the following primitives:

1. English equivalents of PTL primitives ($\Diamond$, $\Box$, U, and $\Rightarrow$).
2. PTL propositions.
3. The universal quantifier of predicate logic.
4. Derived temporal operators, including **onlyafter** and **after**.
5. Macros for expressing liveness and safety properties.

Recall the following definitions (where eventuality of *q* is not assumed):

$(p \textbf{ onlyafter } q) \equiv \sim p \text{ U } q$

$(p \textbf{ after } q) \equiv (\sim p \text{ U } q) \wedge (\Diamond p)$

For example, the SYSL specification:

(at m_1**) onlyafter (after** m_1**)**

specifies a temporal ordering on states.

The SYSL macro "EXCLUDE" specifies mutual exclusion:

$\forall t_1, t_2 \in \text{tick}, t_1 \neq t_2, \Box\sim\{\text{active}(t_1) \wedge \text{active}(t_2)\}$

For example, a program that simulates advancing a clock is specified by the following SYSL statement:

Tick Operations EXCLUDE EACH OTHER

where "tick" is an operation that increments time. This statement expresses the requirement that tick operations occur one at a time.

3.4.1.7. Decidability and Expressiveness

A state is a truth-valued function on a set of propositions. The operator $\models$ defines a mapping from each state *s* to the set of propositions that are true in *s*. A state formula is a truth-valued function on states. We write $s \models \psi$ to denote that state formula ψ is true at state *s* (read, "*s* satisfies ψ").

Recall that $\sigma=<s_0,s_1,...>$ denotes an infinite sequence of states. A subsequence of states is denoted $\sigma_i=<s_i,s_{i+1},...>$, for any $i \geq 0$. We write $\sigma \models \psi$ to denote that state formula ψ is true on the sequence σ. This is defined inductively as follows:

For a proposition *p*, $\sigma \models p$ if $s_0 \models p$.

That is, *p* is true on the sequence σ, if *p* is true in state s_0,
written $s_0[p]$=true or $p \in I(s_0)$.

$\sigma \models \psi_1 \vee \psi_2$ iff $\sigma \models \psi_1$ or $\sigma \models \psi_2$

$\sigma \models \sim\psi$ iff $\sim(\sigma \models \psi)$

$\sigma \models \Box\psi$ iff $(\forall k \geq 0)\ \sigma_k \models \psi$

$\sigma \models \Diamond\psi$ iff $(\exists\ k \geq 0)$ such that $\sigma_k \models \psi$

$\sigma \models O\psi$ iff $\sigma_1 \models \psi$

$\sigma \models \psi_1\ U\ \psi_2$ iff

$(\exists\ k \geq 0)$ such that $\sigma_k \models \psi_2$ and $(\forall i,\ 0 \leq i < k)\ \sigma_i \models \psi_1$

Derived operators can be defined also, e.g.,

$\sigma \models \psi_1\ N\ \psi_2$ iff $\sigma \models \Box\psi_1$ or $\sigma \models \psi_1\ U\ \psi_2$

If every computation σ of a program P satisfies a state formula ψ (i.e., $\sigma \models \psi$ for all σ), then ψ is said to be *valid* over P: $P \models \psi$.

Under the assumption of eventual fairness ("If a process is given many chances to proceed, it eventually will."), Pnueli proved that the validity of an arbitrary eventuality, e.g., $\Box(p \Rightarrow \Diamond q)$, is decidable for finite-state systems. Gabbay proved that both DX and DUX are decidable.

When considering predicate logic, additional definitions are required:

$\sigma \models (\forall X)\psi_X$ iff $(\forall X)[\sigma \models \psi_X]$

$\sigma \models (\exists X)\psi_X$ iff $(\exists X)[\sigma \models \psi_X]$

Gabbay showed that PTL is *expressively complete*, i.e., no additional operators are needed for reasoning about the future, provided linear time is assumed. His proof follows that of Kamp, who showed that if time is linear, dense, and infinite (into both the past and the future), then *every* tense operator can be expressed with only the operators "since" and "until."

Gabbay concluded that all important properties of programs can be expressed in PTL; i.e., no additional operators are needed for making assertions about program behavior. This result has been disputed over several key issues, which motivated the development of branching-time specification languages, the introduction of history variables, the development of event-based specification languages, and, most recently, the introduction of interval-logic specification languages.

In the following sections we examine the validity of the arguments against PTL's "completeness" and review the solutions that have been proposed for extending the expressiveness of PTL.

3.4.2. Branching Time and Path Formulas

Lamport was the first to investigate a branching-time logic for program verification (Ref. 67). In Lamport's model, a branching-time formula is an assertion over all possible computations of a program, starting from the current state. Recall that in linear time we consider each possible computation independently, so that each state has exactly one successor.

Lamport argued that linear time and branching time have different expressive powers, but neither is more expressive than the other. They differ in their interpretation of "◊." In branching time ◊p means that p will become true eventually in *every* computation. In linear time ◊p means that there exists a computation in which p will become true.

Lamport argued that the duality of □ and ◊ does not hold in branching time. (Recall that ◊$p \equiv \sim\Box\sim p$ in linear time.) He omitted the details of the proof, but we include them here.

Let $\Box_B$ and $\Diamond_B$ denote the branching-time interpretations of the linear-time operators □ and ◊, i.e.,

$\Box_B\, p \equiv (\forall \text{paths})[\Box p]$

$\Diamond_B\, p \equiv (\forall \text{paths})[\Diamond p]$

Theorem 1: The equivalence $\Diamond_B\, p \equiv \sim\Box_B\, \sim p$ is invalid.

Proof: The proof is by contradiction. Assume that the equivalence holds. Then by our definitions of branching-time operators

$(\forall \text{paths})[\Diamond p] \equiv \sim\{(\forall \text{paths})[\Box\sim p]\}$

$\equiv (\exists \text{path})[\sim(\Box\sim p)]$ (by logical equivalence)

$\equiv (\exists \text{path})[\sim\Box\sim p]$

$\equiv (\exists \text{path})\Diamond p$ (given the duality of □ and ◊)

This implies

$(\forall \text{paths})[\Diamond p] \equiv (\exists \text{path})[\Diamond p]$

which is clearly false.

"Not never" means eventually happening in some possible future; whereas, "eventually" means

eventually happening in every possible future.

Lamport argued for linear time in reasoning about concurrent programs, but for branching time in reasoning about "nondeterministic" programs (i.e., *distributed* programs). He showed that response to persistence cannot be expressed in branching-time logic. On the other hand, properties of programs that execute in parallel (multiple events occurring at any instant) can be expressed only in branching time.

Both interpretations of eventuality have been incorporated into Computation Tree Logic (CTL) (Ref. 83). In CTL, the basic tense operators are either A ("for all futures") or E ("for some future"), followed by the usual linear-time operators: $\Box$, $\Diamond$, O, and U.

CTL was further extended to the language CTL*, in which assertions are expressed in *path formulas* (Ref. 84). A *path formula* is a state formula that can be preceded by a *path quantifier.* Emerson and Lei have shown that CTL* is decidable in triple exponential time.

In CTL*, validity is defined over a branching structure, M. The expression "M, $s \models \psi$ (M, $\sigma \models \psi$)" means that path formula ψ is true in the structure M at state s of path σ. If every state s of every structure M satisfies a path formula ψ (i.e., M, $s \models \psi$), then ψ is said to be valid over M.

Recall that a state formula is an assertion about the current state and future states. A branching-time formula is an assertion about the current state only (not about future states). Hence, in determining whether $\sigma \models \psi$ (for a path formula, ψ), we consider only the truth values of the atomic propositions in the current state.

3.4.3. History Variables

Recall that temporal formulas specify properties that hold from the *current* state through the remainder of the computation. No specific initial state is assumed. Temporal-specification languages use only the future fragment of temporal logic. Hence, either initial conditions must be specified explicitly, or some mechanism is need for reasoning from the beginning of the program, e.g., using embedded "untils."

Hailpern and Owicki introduced *history variables* to establish a sequence of prior states (Ref. 78). A *history variable* (of unbounded length) is a state encoding of the sequence of values that will be obtained by a program variable before reaching the current state. Without introducing the specific operations (events) that cause the changes, history variables describe what changes can occur during execution.

History variables are useful for expressing regular properties of a message-passing system, e.g., "successive messages must have alternating sequence numbers." History variables provide the memory of all previous process interactions, i.e., the sequences of messages that are transmitted and received. The operator "⟨" ("initial subsequence") specifies a relationship between sequences of history variables:

A ⟨ B A and B are history variables, and A is an initial subsequence of B.

Nguyen introduced the concept of "traces" for describing the sequence of I/O operations for ports in a network (Ref. 98). In addition to the initial subsequence operator, Nguyen included regular expressions on traces, e.g.,

$A \in 0^*1$

("The trace A can be generated by the regular expression 0^*1.")

3.4.4. Extended Propositional Temporal Logic: EPTL

As discussed previously (Section 3.1), Wolper (Ref. 79) argued that event operators are needed to express arbitrary regular properties, such as "p must be true in every even state of a sequence." Wolper's claim has been refuted by McLean (Ref. 85): The property of "even states" can be expressed by the formula $p \wedge \Box(p \Rightarrow \bigcirc\bigcirc p)$.

A stronger argument for event operators is that they are more suitable than state formulas for reasoning about concurrent behavior in programs that communicate by message passing, e.g., Ada and CSP programs. Wolper introduced an extension of PTL, called EPTL, for specifying and synthesizing the synchronization part of CSP programs.

EPTL extends PTL with a set of operators corresponding to regular expressions for specifying state sequences. A regular expression in EPTL corresponds to a right-linear grammar that generates allowable sequences of states. The semantics of temporal operators in EPTL is the same as for PTL, except that EPTL does not require the eventuality of q in the U-operation (p U q).

Each atomic proposition in an EPTL formula is associated with the execution of exactly one interprocess I/O operation in a CSP program, the only form of communication in CSP (Ref. 10). A "single event condition" is assumed, i.e., each state transition corresponds to one interprocess I/O operation. Hence, in each state exactly one atomic proposition is true: the proposition that corresponds to the I/O operation currently in execution.

In EPTL, immediate responsiveness (no delay) requires that every response be preceded by a request, which happened after the last response, if any. Recall that immediate responsiveness is defined as $\Box(p \Rightarrow \bigcirc q)$ ("Either q is never true or q is not true until p.").

Wolper showed that the validity of an arbitrary EPTL formula is decidable (i.e., EPTL preserves the decidability properties of PTL).

3.5. Event-Based Temporal Models

Event-based models formalize constraints on events and their relationships. The temporal context of an assertion is established by prior events, as opposed to a state component, such as history variables. Past and future behavior are both specified by event relationships.

3.5.1. The Event-Based Specification Language: EBS

Chen and Yeh introduced an event model, called the Event-Based Specification Language (EBS), for specifying the behavior of distributed systems (Ref. 68). In this model, program behavior is specified by interprocess events (e.g., sending, receiving, and processing messages) and their relationships. An EBS event is an instantaneous, atomic state transition in the execution of a program.

EBS defines two primitive relations over events:

e_1 -> e_2 "e_1 precedes e_2"
e_1 => e_2 "e_1 enables e_2"

where e_1 and e_2 are arbitrary events.

The "precedes" relation defines temporal precedence of events: "e_1 occurs before e_2, and there may be intervening events." Safety properties are specified by the precedes relation.

The "enables" relation defines causality of events: "If e_1 occurs, then eventually e_2 will occur." Liveness properties are specified by the enables relation.

EBS is an extension of predicate logic. It consists of the following primitives:

1. A finite set of events.
2. The names of interface ports (a finite set).
3. The quantifiers and connectives of predicate logic ($\forall$, $\exists$, ~, $\vee$, $\wedge$, $\Rightarrow$, and $\equiv$).
4. Relational operators: $\in$, $=$, $\neq$.
5. Event relations: -> and =>.

The "precedes" and "enables" relations allow partial orderings on events, in the absence of a global clock. Simultaneous events are allowed:

$\sim(e_1 \rightarrow e_2) \wedge \sim(e_2 \rightarrow e_1)$

Unlike in Nguyen's model, event sequences ("traces") are not specified in EBS (owing to the absence of total ordering on events).

3.5.2. Event Specifications

Vogt (Ref. 94) introduced *event specifications* for expressing predicates on event history. He defined only one event class: an *interaction event*, which is an event that causes synchronization between processes.

Vogt's *event model* is a triple, (E,σ,Σ), where

- E is a finite set of interaction events: $\{e_1,...,e_n\}$ (Events can be parameterized.),
- $\sigma=<e_0,e_1,...>$ is an infinite sequence of interaction events (past), and

- Σ is an infinite set of sequences of events (future).

An event specification is composed from the following primitives:

1. Events from the set E.
2. Propositional connectives: ~, ∧, ⇒, ∨.
3. The linear-time operators: □, ◊, O, U.

As in EPTL, a single event condition is assumed: Each state is obtained from its predecessor by the occurrence of exactly one interaction event (i.e., an interprocess event).

3.5.3. Path Expressions

Path expressions model program behavior as possibly non-terminating sequences of events (Ref. 74, 75). Unlike EPTL and event specifications, which are both intended for verifying CSP programs, path notation is an event model for reasoning about concurrent programs that communicate via shared variables. Path expressions were developed originally for the control of the synchronization of concurrent processes, e.g., to restrict the execution of operations on shared objects. Andler extended the use of path expressions to specify and verify concurrent systems (Ref. 102).

Each path expression is delimited by the keywords **path** and **end**, which represent an implicit Kleene star. The primitive terms of a path expression are *process events*, which are actions corresponding to the names of procedures that can be executed by processes. For example, the path expression:

path e_1 ; e_2 **end**

specifies an alternating sequence of process events e_1 and e_2.

Path expressions specify the temporal ordering of events that are *allowed* to occur in the execution of a program. The events named in a path expression must occur in the specified order; however, other events may intervene. For example, the event stream that is represented by the string "acbc" is allowed by the following path expressions:

path (a+b) ; c **end**

path a ; b **end**

path a ; c ; b ; c **end**

Path expressions use the following operators (in precedence order, from highest to lowest):*

( )	change precedence	parentheses
*	"zero or more times"	Kleene star
;	"next"	sequencing
+	"or"	exclusive selection

*Sequencing can be denoted by a blank. Exclusive selection can be denoted by a comma.

The syntax of path expressions is as follows:

path ::= **path** seq **end**

seq ::= seq ; alternate | alternate

alternate ::= alternate + cycle | cycle

cycle ::= element * | element

element ::= process_event | (seq)

Path expressions have been extended to include the Kleene plus operator and an operator for specifying that two events occur in parallel.

A path expression can be identified with the set of strings of events it accepts. A path expression specifies all allowable event sequences of a program. A program is considered correct if all possible execution paths can be accepted by the path expression.

Because regular expressions correspond to finite-state machines, a path expression can be transformed into a state-transition graph, by labeling arcs with the names of process events. A *simple* (or elementary) *path expression* has a graph in which no two arcs carry the same name; that is, each name appears only once in a simple path expression.

Although path expressions were intended for reasoning about access to shared variables, they are suitable for reasoning about process interactions. For example, in Ada, process events can correspond to the names of task entries:

path `BUFFER.READ ; BUFFER.WRITE` **end**

While temporal logic is useful for specifying *eventualities*, path expressions excel at specifying *recurring* behavior patterns, such as a FIFO ordering of events. Path expressions model the behavior of a program as a pattern of events that can recur as the program progresses.

3.5.4. Petri nets

A Petri net is a finite-state machine (Ref. 76). A Petri net *structure* is a four-tuple, (P,T,I,O), where

- P is a finite set of *places*,
- T is a finite set of *transitions*,
- I maps transitions to input places, and
- O maps transitions to output places.

(Each transition may be mapped to more than one input/output place.) No notion of system state is assumed in Petri nets.

A Petri net *graph* is a directed graph such that each node is either a circle (place) or a bar (transition), denoting, respectively, *conditions* and *events*. An event is *enabled* if conditions necessary for its

occurrence hold (pre-conditions). The occurrence of an event may cause other conditions to become true (post-conditions). An arc is drawn from each place denoting a pre-condition of an event (an input place) to the transition denoting the event. Another arc is drawn from each transition denoting an event to each place denoting a post-condition of the event (an output place). If a condition holds in the current state of the system, then a *token* (represented by a dot) is placed in the corresponding place in the net. More than one token can be assigned to a place.

The number and distribution of tokens control execution of the net. A transition is enabled to "fire" if all its input places are marked with tokens. The firing of a transition corresponds to the occurrence of an event. Firing removes tokens from a transition's input places and creates new tokens in its output places. A Petri net can be identified with the set of all possible firing sequences. The placement of tokens determines safety and liveness properties of the program. (See Figure 3-1 (Ref. 92).)

3.6. Relationship Between Path Expressions and Temporal Logic

Not all state formulas can be transformed into path expressions, e.g., eventualities cannot be expressed in path expressions. A natural question to ask is whether an arbitrary path expression can be transformed into a state formula. If so, then EPTL is no more expressive than PTL. That is, we consider whether all regular properties can be expressed in state formulas, without introducing regular expressions.

A path expression can be translated into a finite-state machine. The validity of a state formula is decidable for a finite-state system (assuming propositional logic). Thus, it is decidable whether a given state formula is valid for a given path expression.

Previous work suggests a relationship between path expressions, Petri nets, and temporal formulas. Recall that Lauer and Campbell showed that an arbitrary path expression can be translated into a corresponding Petri net (Ref. 92). Schwartz and Melliar-Smith illustrated a transformation from Petri nets to state formulas, but provided neither a generalized transformation algorithm, nor a formal proof of this result (Ref. 70). Plaisted introduced a low-level language that is a generalization of regular expressions, and into which temporal formulas can be translated; however, he left open the relationship between this language and path expressions (Ref. 103).

In this section we present a formal translation from a restricted class of simple path expressions into PTL formulas. Our results show that most regular properties can be expressed in PTL formulas. The single exception is "unrestricted" iteration, e.g.,

path $e_1 \; ; \; e_2{}^*$ **end**

No mechanism exists in temporal logic for specifying that a proposition become true zero or more times from now until the end of time. On the other hand, we can specify that a proposition become true zero or more times before some other proposition becomes true. Recall that "false" U $q \equiv \Diamond q$. For some

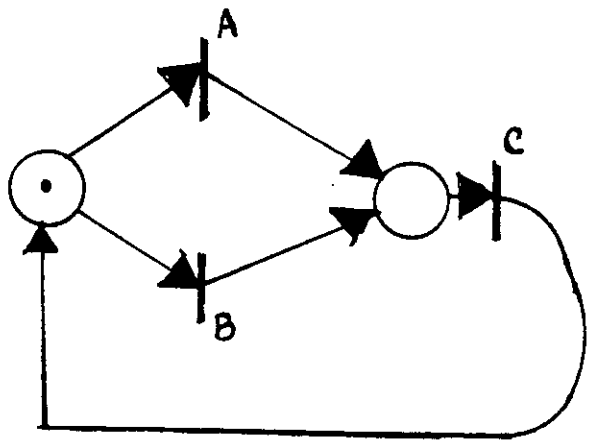

Figure 3-1: Petri net
For path expression: **path** (a+b) ; c **end**

proposition p, p U q specifies that q will become true eventually, regardless whether p is ever true (assuming the eventuality condition of the U-operator). Hence, we restrict iterations to those that are followed by another event, e.g.,

path $e_1{}^*$; e_2 **end**

3.6.1. Definitions and Assumptions

We introduce the notion of a binary Kleene star operation:

α^* ; β

meaning that α occurs zero or more times, followed by a single occurrence of β. We restrict the use of the Kleene star in simple path expressions to the binary Kleene star.

Our transformation from path expressions to state formulas requires two steps:

1. Transform a restricted, simple path expression to an event expression (an event-based temporal formula).
2. Transform an event expression to a state formula (a state-based temporal formula in PTL).

Before presenting an algorithm for transforming path expressions into state formulas, we define events explicitly and formalize their relationship to states. In the event-specification languages that we have reviewed, the definition of an event depends on the underlying concurrency model (shared memory versus message passing). For example, recall that in path expressions (Section 3.5.3) each operation on a shared object is an event. On the other hand, in Vogt's event specifications (Section 3.5.2) each interprocess I/O operation is an event.

In our transformation algorithm we assume that path expressions and event expressions can involve events of either class: operations on a shared object or interprocess I/O operations. In transforming a path expression into an event expression, the class of each event remains unchanged.

In transforming an event expression into a state formula, we need to consider the relationship between states and events of *both* event classes. Recall that EPTL required the following single event condition: Each state transition corresponds to one interprocess I/O operation. Similarly, we require the following single event condition: Each state transition corresponds to a single event, which is either a single interprocess I/O operation or an operation on a shared object.

For Ada, this single event condition requires that each state transition correspond to either of the following operations:

- Definition of a shared variable (i.e., assigning a value to a shared variable, for example, by execution of an assignment statement or by execution of an input statement).
- Rendezvous, where at least one parameter is passed between the tasks.

A second difficulty in the transformation process is that sequencing of events in path expressions, e.g.,

path e_1 ; e_2 **end**

differs from sequencing of states in PTL formulas:

$p \Rightarrow Oq$

These specifications differ because path expressions allow intervening events, but PTL formulas specify successive states. (The property of immediate responsiveness cannot be expressed in path expressions.)

Thus, we are forced to choose between the following restrictions:

1. No intervening events are allowed in path expressions.
2. The "next" operator in state formulas allows intervening states.

We have chosen to assume the first restriction (which allows for the property of immediate responsiveness). This approach is more practical because we can always filter out the intervening events before matching a path expression against an event sequence.

3.6.2. Transforming Path Expressions into Event Expressions

Theorem 2: Let ϕ be a restricted, simple path expression involving a finite set of events, $E=\{e_1,...,e_k\}$. There is a mapping, ρ, from ϕ to a finite, well-formed event expression, ψ, such that Σ, the set of event sequences *allowed* by ϕ, is equivalent to the set of event sequences *specified* by ψ.

Proof: We use induction on the number of operators in ϕ.

Basis: If ϕ has no operators, then $\phi \in E$ and $\rho(\phi)=\psi$.

Induction: In the following let α and β be either events or restricted, simple path expressions.

We consider the form of ϕ by cases:

1. **path** ... **end** (implicit Kleene star)

   $\rho(\textbf{path}\ \alpha\ \textbf{end}) = (\Box\rho(\alpha))$

2. Next

   $\rho(\alpha\ ;\ \beta) = (\rho(\alpha) \Rightarrow O\rho(\beta))$

3. Exclusive Or (mutual exclusion)

   $\rho(\alpha + \beta) = (\rho(\alpha) \vee \rho(\beta))$

4. Binary Kleene star

   $\rho(\alpha^*\ ;\ \beta) = (\rho(\alpha)\ U\ \rho(\beta))$

   If α occurs zero times then $\rho(\alpha) =$ false.

5. Parentheses

   $\rho((\alpha)) = (\rho(\alpha))$

For example:

path (a+b) ; c **end**

$\downarrow^1$

$\Box$((a+b) ; c)

$\downarrow^2$

$\Box$((a+b) $\Rightarrow$ Oc)

$\downarrow^3$

$\Box$((avb) $\Rightarrow$ Oc)

3.6.3. Transforming Event Expressions into State Formulas

In EPTL, computations are restricted to those in which only one atomic proposition is true in any state: the proposition that corresponds to the event immediately preceding that state. (See Section 3.4.4.) We place a similar restriction on the control predicate **after** in each state, although we allow any other propositions to be true in a state.

We make the following assumptions:

1. Each state is obtained from its predecessor in the sequence by the execution of a single operation (an atomic event) in exactly one process.
2. Only finite-state programs are considered (to ensure that all predicates on program variables can be expressed as Boolean variables).
3. Only one control predicate of the form **after** e_i is true in any state.

We will say that an event, e_i, *corresponds* to a state, s_i, if the control predicate **after** e_i is true in s_i. Let ψ be a well-formed event expression involving a finite set of events, $E=\{e_1,...,e_k\}$. A transformation from ψ to a wfsf, ξ, maps each event, e_i, in the expression ψ to its corresponding state, s_i.

3.7. Interval Temporal Logics

Temporal-logic operators ($\Box$, $\Diamond$, U, O) are always interpreted as extending from the current state through the remainder of the computation. (This has been called the "tail-sequence" property of temporal logic (Ref. 70).) Yet, one often wants to assert that a condition *become* true before some point in the execution and not just before the end of the computation. For example, an Ada entry call requires a response before the called task terminates, which can happen long before the program terminates. Safety assertions on bounded intervals are more appropriate than eventualities for reasoning about processes that are expected to terminate.

As discussed previously (Section 3.1.1), another consequence of the tail-sequence property of temporal operators is that *short-term fairness* can be expressed in neither linear-time nor branching-time logic. That is, there is no finite, well-formed, state-based temporal formula for expressing the

requirement:

$$p \wedge \bigcirc p \wedge \bigcirc\bigcirc p \wedge \ldots \wedge \bigcirc^N p \Rightarrow \Diamond q$$

for some fixed integer N, where $\bigcirc^N p$ is defined inductively as follows:

$\bigcirc^N p \equiv p$ for k=0

$\bigcirc^N p \equiv \bigcirc(\bigcirc^{N-1} p)$ for k > 0

Unlike temporal-logic operators, *history variables* allow specifying properties that hold for a bounded sequence of future states. History variables have been used for stating that a property *remains* true continuously over a bounded interval, but not forever.

Yet, as noted by Schwartz and Melliar-Smith, the introduction of history variables simplifies temporal formulas at the expense of increasing the mechanization in the specification. Lamport argued that introducing history variables defeated the whole purpose of using temporal logic, because it returns to reasoning directly about the computation model (Ref. 67). Lamport advocated that a new temporal model be developed to incorporate the semantics of history variables. *Interval logic* addresses these issues. Lamport introduced a "Timeset language" for defining properties over intervals.

Interval logic has been introduced for specifying program properties that hold over a bounded interval, which may extend over a sequence of states. It has been observed that the technique of using interval logic approximates history variables, but is more convenient (Ref. 73). Interval logic offers several advantages over (a point-based) temporal logic:

- Eliminates need for history variables.
- Allows reasoning about delayed processing.
- Allows reasoning about programs that are expected to terminate.
- Allows reasoning about short-term fairness.

In the remainder of this section we review several interval-logic systems.

3.7.1. Interval Formulas

In the interval logic developed by Schwartz and Melliar-Smith, an *interval* is a sequence of states starting with the current state (Ref. 70, 71). An *interval formula* is an assertion that a given property hold for an interval. If the specified interval cannot be found, then the interval formula is *vacuously* satisfied. A star operator (*) is introduced for specifying that the interval must be found for the interval formula to be true. (The star operator differentiates between the two versions of the U-operator; that is, it specifies that the eventuality condition hold.)

Each state in an interval is a *unit* interval. The operations **begin** `I` and **end** `I` denote unit intervals that contain, respectively, the first and last states of interval `I`.

A property, *p*, can hold in the first state of an interval, `I` (the current state):

`[I]`*p*

throughout an interval:

`[I]`□*p*

or sometime during an interval:

`[I]`◊*p*

In this model, an "event" is a predicate on a pair of states. An event occurs at the instant when the truth status of an interval formula changes value. Intervals are derived from primitive intervals, called *event intervals*, which hold over an interval of length two (the interval of change). (See Figure 3-2.)

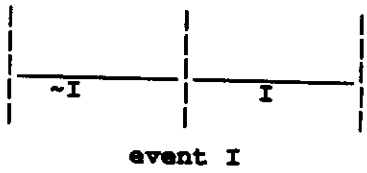

Figure 3-2: Event Interval

Interval terms are either event intervals (primitives) or compound terms composed of the following primitives:

1. Events.
2. Interval operators, including **begin**, **end**, =>, <=, and ∧ (conjunction).

Interval formulas are composed of the following primitives:

1. State predicates, i.e., the propositions described for state formulas.
2. Temporal operators: □ and ◊.
3. The propositional connectives.
4. Interval terms.
5. The star operator (*) for modifying interval formulas.

The interval formula:

`[I=>J]`ψ

means that ψ holds from the end of interval `I` (the instant when event `I` occurs) to the beginning of interval `J` (the instant when event `J` occurs). For example, the formula:

```
[x=y => y=16] □(x > z)
```

specifies that the predicate x > z hold from the time when x is assigned the value of y, until y is assigned the value 16.

The interval formula:

`[I <= J]` ψ

means that ψ holds from the end of interval I to the end of interval J.

3.7.2. An Interval Temporal Logic: ITL

Moszkowski and Manna introduced a program-specification language, called Interval Temporal Logic (ITL), that extends linear-time, predicate temporal logic with bounded intervals (Ref. 72). ITL was intended originally for specifying and reasoning about timing-dependent hardware. It has been applied also in specifying state transitions of programs, including those achieved by assignment, iteration, scoping, and concurrency (via shared variables). A global state is assumed. (Because ITL is an extension of predicate temporal logic, it is undecidable, as noted previously (Ref. 103).)

An ITL interval consists of a sequence of states. That is, successive "subintervals" of an ITL interval correspond to successive states of a computation. An ITL formula specifies allowable sequences of states over an interval. An ITL "variable" represents names of signals (in reasoning about hardware) or names of program variables (in reasoning about programs). A state transition is a relation on the initial and final values of variables over an ITL interval.

Unlike the more formal models previously discussed in this section, ITL defines temporal operators that are application-specific. For example, variable assignment is defined as

```
Y <-- X
```

read: "Variable Y is assigned the initial value of variable X."

An ITL formula for specifying program behavior is constructed from the following primitives:

1. Predicates on program variables.
2. Data structures for program variables, e.g., lists, vectors.
3. Control predicates, including **beg** (the initial state of the interval) and **fin** (the final state of the interval).
4. The connectives and quantifiers of predicate logic.
5. The temporal operators $\Box$ and O, expressed over intervals.
6. Operators for forming regular expressions over interval formulas, including ";" (sequencing), "*" (Kleene star), and α^n (iteration, where n is an arithmetic expression).
7. The assignment operator: <--.

New operators are defined as predicates on already existing operators, e.g., an assignment that is repeated throughout an interval is expressed by the "gets" operator:

```
Y gets X
```

("The current value of X is always equal to the next value of Y.")

An ITL while-loop is a derived operator:

while α_1 **do** α_2 `iff` $(\textbf{beg}(\alpha_1) \wedge \alpha_2)^* \wedge \textbf{fin}(\sim\alpha_1))$

An example of an ITL formula for a while-loop is

beg (I=0) ∧ **while** (I<n) **do** ([I<--I+1])

3.7.3. Quantified Temporal Logic: QTL

Quantified temporal logic (QTL) (Ref. 73) is an interval logic on linear metric time (as described earlier, in Section 3.2.7). QTL is based on a subset of the LAR system (Ref. 104), which extends predicate logic with chronological operators for reasoning about the execution of sequential programs, e.g., branching and looping. QTL has been applied in verifying algorithms for communication protocols in carrier-sense, local area networks.

LAR includes the connectives of propositional logic and the operation $\Box^{\alpha}A$, which asserts that proposition A will become true in α time units from now. The equivalent QTL operation is $O(t)A$, for predicate A, where t may be negative. QTL extends LAR for stating conditions that hold over bounded intervals:

$$[s,u]A \equiv (\forall\, t)\{(s \le t \le u) \Rightarrow O(t)A\}$$

$$<s,u>A \equiv (\exists\, t)\{(s \le t \le u) \wedge O(t)A\}$$

For example, [2,5](A) means that from two units after now to five units after now, A will be true. The expression <0,6>(A) means that some time in the next six units, A will become true. The "box" and "diamond" expressions can be combined, e.g., <1,6>[0,2](A) means that some time in the next six units, A will become true and will remain so for three units.

QTL includes axioms for dealing with metric time, for example:

$O(t)[s,u]A \equiv [t+s,t+u]A$.

QTL assumes the validity of the Barcan formula, for example, the following is a QTL axiom:

$(\forall X)\{[t,s]\phi_X\} \equiv [t,s]\{(\forall X)\phi_X\}$

3.8. A New Interval Logic: System $\mathbb{C}$

In this section we introduce our interval-logic system, $\mathbb{C}$, which is an extension of the basic temporal-logic system, $\mathbb{R}$. An interval in $\mathbb{C}$ is the time between which a pair of events occur.

3.8.1. The Semantics of System $\mathbb{C}$

Let an *interval*, T, be represented as a range of temporal instants with the first endpoint, t^-, not later than the second, t^+. We assume discrete, linear time so that the ordering of temporal instants is isomorphic to that of the non-negative integers. The *length* (or duration) of an interval, denoted |T|, is the integral difference between t^+ and t^-. We permit an interval of duration zero, so that we can speak of statements that hold for an instant, without resorting to system $\mathbb{R}$. (This concept is similar to point

intervals in Allen's model.) Interval T is a *subinterval* of interval S iff all instants in T are contained in S.

Ada syntax provides the compound symbol ".." to denote a range of values, and the pseudo-operator **in** to test for membership within a specified range (including the end values):

$$T \equiv t^- \,..\, t^+, \text{ provided } t^- \le t^+ \tag{14}$$

$$t \text{ in } T \equiv t^- \le t \le t^+ \tag{15}$$

Ada also provides the attribute **succ** to specify the next element in a range, and the attribute **pred** for the previous element in a range.

We define the operation of *temporal realization over intervals*, $C_T(p)$, read: "*p* holds while T," to mean that predicate *p* holds at all discrete times in the interval T:

$$C_T(p) \equiv (\forall t)[t \text{ in } T \Rightarrow R_t(p)] \tag{16}$$

The operator C expresses the "truth value" of a predicate, *p*, as the conjunction of the truth values of *p* at each discrete instant within a specified interval. For example, if time is measured in units of length one, then

$$C_{10..12}(p)$$

asserts that predicate *p* holds at time points 10, 11, and 12.

We also define an operation for "some time," $E_T(p)$, read: "*p* holds at some time in T."* This operation asserts that predicate *p* holds at some instant *t* in the interval T:

$$E_T(p) \equiv (\exists t)\,[t \text{ in } T \wedge R_t(p)] \tag{17}$$

System ℂ includes sentences composed from the following set of primitives:

1. Predicate variables, e.g., A and B, ranging over both temporally definite and indefinite statements.
2. Variables, e.g., T and S, for temporal intervals.
3. Variables, e.g., t^-, t^+, s^-, s^+, for endpoints of intervals, i.e., as either definite or pseudo-dates.
4. Variables, such as *t*, as arbitrary instants.
5. The variable *n* for the pseudo-date "now."
6. The relational operators <, =, and > (earlier/contemporary/later), for comparing variables for temporal instants or the variable *n*.
7. The variables X, Y, Z, ..., ranging over domains other than time.
8. The connectives of propositional logic, including ~ for negation, $\wedge$ for conjunction, $\Rightarrow$ for implication, and $\equiv$ for equivalence.
9. The quantifiers of predicate logic, $\forall$ and $\exists$, ranging over any variables.
10. The operators: .., **in**, **succ**, **pred**, R, E, and C.

---

*Unlike in Schwartz's interval logic, we require that the interval T exist for the interval formula to hold.

We assume the usual principles of predicate logic for sentences in $\mathbb{C}$ not involving C, R, E, or n. In addition, the following axioms are among those that hold over system $\mathbb{C}$:*

$$C_T(A \wedge B) \equiv [C_T(A) \wedge C_T(B)] \tag{18}$$

$$[C_T(A) \wedge (n \text{ in } T)] \equiv R_n(A) \tag{19}$$

$$C_T(A) \wedge (s^- \geq t^-) \wedge (s^+ \leq t^+) \Rightarrow C_S(A) \tag{20}$$

$$C_T(A) \wedge C_S(A) \wedge (\mathbf{pred}(s^-) \leq t^+ \leq s^+) \wedge (t^- \leq s^-) \equiv C_{t^-..s^+}(A) \tag{21}$$

$$C_T(\sim A) \Rightarrow \sim C_T(A) \tag{22}$$

$$C_T[(\forall X)A_X] \equiv (\forall X)C_T[A_X] \tag{23}$$

$$C_T[(\exists X)A_X] \equiv (\forall t \text{ in } T)\,(\exists X)[R_t(A_X)] \tag{24}$$

In Axioms 23 and 24 the validity of the Barcan formulas is assumed.

With the inclusion of the operator E, the following axioms also hold in $\mathbb{C}$:

$$E_T(A \wedge B) \equiv [E_T(A) \wedge E_T(B)] \tag{25}$$

$$E_S(A) \wedge (s^- \geq t^-) \wedge (s^+ \leq t^+) \Rightarrow E_T(A) \tag{26}$$

$$\sim E_T(A) \equiv C_T(\sim A) \tag{27}$$

$$E_T[(\forall X)A_X] \Rightarrow (\forall X)E_T(A_X) \tag{28}$$

$$E_T[(\forall X)A_X] \Rightarrow (\exists t \text{ in } T)\,\{R_T[(\forall X)A_X]\} \tag{29}$$

$$E_T[(\exists X)A_X] \equiv (\exists X)E_T(A_X) \tag{30}$$

Interval relations can also be defined, e.g.:

S **during** T iff $(s^- \geq t^-) \wedge (s^+ \leq t^+)$ ("S is a subinterval of interval T.")

T **meets** S iff $t^+ = s^-$ ("T ends where S begins.")

S **next** T iff $\mathbf{succ}(t^+) = s^-$ ("S is the next interval adjacent to T.")

Our **next** operator is similar to one that Lamport (Ref. 67) introduced for specifying adjacent states: $(\forall i)\ s_{i+1}$ **next** s_i.

Axiom 20 states that if an arbitrary predicate, p, holds in T, then p holds in any subinterval of T. T is a *subinterval* of interval S if all instants in T are contained in S. Axiom 20 can be restated with the operator

---

*We assume that a program state represents a homogeneous activity; i.e., we assume that propositions that are true at the beginning of a state are true throughout the state. For example, the implication $C_T(A) \wedge (n \text{ in } T) \Rightarrow R_n(A)$ (from Axiom 19) fails otherwise. Also the implication $C_T(A) \Rightarrow \sim C_T{\sim}A$ holds for homogeneous and majoritative activities, but not for occasional ones.

during:

$$C_T A \Rightarrow (\forall S)\,[S \textbf{ during } T \Rightarrow C_S(A)]. \tag{31}$$

Axiom 21 implies that whenever an arbitrary predicate, p, holds over two intervals that meet, then p holds over their combined duration:

$$C_T(A) \wedge C_S(A) \wedge (T \textbf{ meets } S) \Rightarrow C_{t^-..s^+}(A) \tag{32}$$

Also, if predicate p holds over adjacent intervals T and S, then p holds over the combined duration of T and S:

$$C_T(A) \wedge C_S(A) \wedge (S \textbf{ next } T) \Rightarrow C_{t^-..s^+}(A) \tag{33}$$

Axiom 26 implies that if an arbitrary predicate, p, is true at some time in some subinterval S of T, then p is true at some time in T:

$$E_S(A) \wedge S \textbf{ during } T \Rightarrow E_T(A)$$

System $\mathbb{C}$ provides the mechanics for defining states as predicates on events. If e_i and e_j represent events that, respectively, trigger the transition into and out of state s_i, and $t^- < t^+$, then

$$\{R_{t^-}(e_i) \wedge R_{t^+}(e_j) \wedge \sim(\exists t)[R_t(e_j) \wedge (t^- < t < t^+)]\} \Rightarrow C_{t^-..\textbf{pred}(t^+)}(s_i) \tag{34}$$

3.8.2. Comparison with Other Interval Logics

Because the C-calculus is an extension of the basic temporal-logic system, it is easily compared with other interval logics. For example, in Schwartz's system an interval formula has the form:

$[I]\Box\psi$ or $[I]\Diamond\psi$

This is equivalent to stating

$C_I(\psi)$ or $E_I(\psi)$

in our system. In Chapter Four we show that ITL-constructs can be expressed in $\mathbb{C}$, as well.

In QTL an interval formula has the form:

$[s,u]A$ or $\langle s,u\rangle A$

This is equivalent to stating

$C_{s..u}(A)$ or $E_{s..u}(A)$

in our system.

Given the relationship that we have shown between path expressions and temporal formulas, we can express regular properties in $\mathbb{C}$. Thus, $\mathbb{C}$ is not only a basis for all other interval logics, but also encompasses simple path expressions.

3.9. Summary and Conclusions

A temporal model is a formal system for representing knowledge about the timing relationships of events or states that can take place in time. Temporal reasoning has been applied in proving properties of concurrent programs because other proof techniques are inadequate for reasoning about concurrent

behavior and delayed processing.

We have reviewed temporal models for specifying and verifying both concurrent and distributed program behavior. These models formalize constraints on the execution paths of programs:

- Temporal formulas are useful for reasoning about eventualities, such as absence of individual starvation.
- Path expressions are useful for reasoning about the ordering of execution, e.g., concurrency restrictions on access to shared data.
- Petri nets are useful for graphically displaying all possible execution paths.
- Interval formulas are useful for reasoning about properties that hold over a bounded interval, e.g., short-term fairness.

Each model relies on assumptions about the underlying concurrency paradigm, and, in particular, each assumes a language in which either shared variables or message passing is allowed, but not both.

Although the merits of any of these approaches can be argued from expressibility and elegance, the choice is often one of convenience. In particular, an arbitrary path expression can be translated into a Petri net. We have shown that a restricted class of path expressions can be translated into temporal formulas.

This chapter provides a foundation for applying temporal models in debugging. In the following chapter we discuss drawbacks of program verification and introduce temporal models for debugging. We present a debugging tool that uses both the interval-logic system $\mathbb{C}$ and path expressions. Path expressions have recently been applied in debugging, e.g., for assertion monitoring, that is, for triggering breakpoints on the occurrence of a specified sequence of events (Ref. 105, 106, 107). Harter has proposed a distributed debugger, called IDD, that would support interval formulas for assertion monitoring (Ref. 37). Also, Petri net graphs are being investigated for displaying a graphical analysis of a program's execution (Ref. 64).

4. A TRACE ANALYSIS APPROACH TO DEBUGGING

There are two ways to write error-free programs;
only the third one works.

-- Perlis (*Epigrams on Programming*, 1982)

In the preceding chapter we discussed the importance of time in specifying and verifying a concurrent program's behavior. We described and compared four temporal models (temporal formulas, path expressions, Petri nets, and interval formulas) that are widely used for proving properties of concurrent programs. We examined the expressiveness of these models and investigated their relationship to one another. We introduced $\mathbb{C}$, an interval-logic system that extends $\mathbb{R}$, the basic temporal-logic system.

Time is an important concept in debugging. Understanding a program's past behavior requires reasoning about causality, change, and invariance. An event causes a program's state to change. One event necessarily follows another, e.g., message receipt follows message transmission. A program variable retains its current value until it is redefined. The ordering of events is crucial to debugging.

Tracing captures the time-dependent relationships of events and states that occur in the execution of a program. A trace is the basic tool of debugging. It specifies a program solely by its behavior. A trace represents one possible execution path.

Although several tools are available for tracing concurrent programs, few of these maintain a trace history throughout the program's execution. Some monitors simply display information as it is captured, while others retain a partial history of the execution (e.g., to "replay" preceding events). Limited memory is the primary reason for discarding trace data; however, as the cost of memory decreases, saving traces becomes more practical.

Collecting trace data has some drawbacks. For example, monitoring may interfere with the analysis of timing problems, as discussed in Chapter One. Monitoring distributed software adds to the difficulties of capturing simultaneous events and of synchronizing *distributed clocks*. In a distributed system we cannot assume a global (universal) clock. Each processor has a different clock; the clocks that lag behind must be advanced periodically. Several approaches have been proposed for synchronizing distributed clocks, e.g., the use of logical clocks (Ref. 108, 109, 110, 111). Algorithms to synchronize clocks can approximate a global clock, but a complete ordering of events is not always possible.

A trace can generate voluminous data, but the programmer needs to extract data pertinent to the error that has occurred. Tools are needed for retrieving important events from a trace history and for "abstracting" from events that are retrieved. The process of *abstraction* involves recognizing similarities and relationships among collections of objects. Abstractions are fundamental to debugging. *Trace analysis* consists of examining a trace of a program's execution and extracting information from the trace.

We can limit the amount of trace data to be analyzed by restricting the selection of events when

collecting data. Analysis of a trace history can be aided with several automated techniques: graphical display, database access, and knowledge-based feature analysis of trace data.

In this chapter we develop new techniques for automating trace analysis. Our approach is to apply temporal-specification techniques in comparing a program's expected behavior with its observed behavior. These techniques allow programmers to examine a trace history and to *test assertions* against the trace history, e.g., "If a cycle of entry calls occurred, then the program failed to satisfy freedom from deadlock."

In adapting temporal-specification techniques to trace analysis, we investigated the following issues:

- How does diagnosing a program error relate to verifying a program's behavior?
- How is assertion testing different from assertion writing?
- In what way can specifying the intended behavior of a program help to diagnose an observed error?
- What are the essential requirements of trace analysis?
- What tools are needed for automating trace analysis?
- How can path expressions be applied in trace analysis?
- How can temporal logic be applied in testing program properties?

This chapter is organized as follows. In Section 4.1 we show that trace analysis has practical advantages over program verification for improving program reliability. In Section 4.2 we investigate previous approaches for specifying abstractions on a trace history. Section 4.3 introduces our approach to trace analysis. Section 4.4 formalizes sequences and operations on sequences. Section 4.5 implements interval logic for expressing queries on sequences. Section 4.6 presents our approach to applying path expressions in trace analysis. Section 4.7 formalizes events, states, and traces. Section 4.8 compares our debugging approach with those described in Section 4.2. We summarize our results in Section 4.9.

4.1. Trace Analysis versus Verification

Trace analysis departs from program verification in several respects. The purpose of trace analysis is to detect where the program failed to conform to a specification or expectation, e.g., a concurrency restriction. On the other hand, the purpose of program verification is to prove that a program is correct.

The benefit of verification is that program properties are specified formally. Yet, verification is often either undecidable or computationally intractable (Ref. 112). While not an alternative to verification, trace analysis is a *practical* tool for isolating errors that are known to exist.

4.1.1. Disadvantages of Verification

In verification, decidability is important because each possible execution path must be considered. Recall from Chapter Three that predicate temporal logic is undecidable; thus, temporal formulas are restricted to propositional temporal logic (e.g., PTL and EPTL).

Often verification fails to meet its goal of proving correctness. In analyzing case examples of "verified" programs, Gerhart and Yelowitz (Ref. 113) observed that modern programming methods, including formal specifications, verification, and structured programming, are still fallible in spite of the application of mathematical reasoning to programming.

Verification has some serious drawbacks:

- Writing a specification is at least as difficult as writing the program and, thus, is prone to errors.
- Correctness proofs rely on assumptions about the program's implementation and environment.
- Verification is difficult to automate, even for simple communication protocols.
- Verification is impractical for large and complex programs.
- As the number of possible states (or events) increases, specifying all interactions becomes impractical.
- Invariants are difficult to specify and to understand because all possible computations must be considered.

4.1.2. Practical Advantages of Trace Analysis

Trace analysis is concerned with isolating errors as opposed to proving their absence. Not all possible errors are reported by a trace. In trace analysis, decidability is not an issue.

Trace analysis allows programmers to test arbitrary program properties. In analyzing a trace, we examine the program's (recorded) past. We can always place ourselves at the beginning or end of a trace, or at any point within a trace. Thus, we can regard a trace as both a program's past and its (predetermined) future.

Although we cannot require that any given event eventually occur, we can ask whether it has occurred before "now" or whether a sequence of past events will prevent its future occurrence. In analyzing the execution of an Ada program, for example, if we observe that a cycle of entry calls has been executed, then we can conclude that the program is in a deadlock state. Consequently, we can infer that the last entry call in the cycle will never be accepted for rendezvous.

4.2. Previous Approaches to Trace Analysis

In this section we discuss two previous debugging approaches that define abstractions on a trace history. Some work has been done also on defining abstractions on event sequences in a general model of temporal semantics (Ref. 100, 87).

4.2.1. The Behavioral Abstraction Approach

Bates proposed the Behavioral Abstraction approach (Ref. 114, 115, 116) to debugging distributed systems. In this approach, program activity is represented as a stream of occurrences of "primitive" events (e.g., process creation, page faults, message transmission and reception). To express abstractions on an event stream, two techniques are provided: *clustering* and *filtering*. *Clustering* combines an ordered sequence of primitive events into a single "higher-level" event. *Filtering* removes selected event instances from consideration in forming a higher-level event.

The Event Definition Language (EDL) is a tool for implementing the Behavioral Abstraction approach. Clustering is achieved by the "catenation" operator for expressing sequencing between two events in an event expression. Events are filtered from an event stream by specifying their class and by specifying relationships (equality/inequality) between attributes of events. A monitor collects an event stream and attempts to recognize EDL-defined (high-level) events as the program executes.

The major disadvantage of EDL is that it provides a limited set of temporal operators. Only the sequencing operator is defined. EDL allows neither interval-logic operators nor iteration operators (e.g., the Kleene star).

4.2.2. A Temporal Query Language: TQuel

The Temporal Query Language (TQuel) is a query language for retrieving information about program behavior (Ref. 55, 117). TQuel is an extension of the query language Quel, which is used in the Ingres relational database management system. Quel was augmented syntactically and semantically to include temporal relations.

Unlike in EDL, intervals are introduced in TQuel. A TQuel database contains *event* relations and *period* (state) relations. An event relation includes a time domain that ranges over instants. A period relation includes a time domain that ranges over intervals. In TQuel, as in other conventional relational database management systems, a *tuple* is a collection of facts about a single object, an entity, or a relationship. A tuple in an event relation describes a state transition that occurs at an instant. A tuple in a period relation specifies a relationship that holds over an interval.

TQuel extends the **retrieve** statement of Quel by adding separate clauses for specifying temporal constraints. These include the **when** clause and the **at** clause. For example, these clauses are used in the following TQuel query:

Example 1:

```
range of R is RunningOn
retrieve StartRunning(R.Process)
where R.processor = Processor1
when "3:00pm" ; R
at R.start
```

In this example, the period relation RunningOn(Process,Processor) contains data about the successive states of processes that are continually started, stopped, and restarted. Tuples are selected from the RunningOn relation and stored into the event relation StartRunning(Process). The *temporal expression* <"3:00pm" ; R> is a path expression that evaluates to a Boolean expression. The tuples selected in this query are those of starting a process after "3:00pm." For each tuple selected, the *event expression* <**at** R.**start**> is evaluated to the time at which the process started running. (That is, this event expression determines the time domain of the relation StartRunning.)

When α and β are temporal expressions, TQuel's temporal operators are defined as follows:

α.**start**	specifies the earlier endpoint of an interval.
α.**stop**	specifies the later endpoint of an interval.
α.**time**	specifies both endpoints of an interval.
α ; β	specifies that α precedes β (sequencing operator).
α | β	specifies the value of either α or β (selection operator).
α , β	specifies that α and β overlap (parallel operator).

When α and β are event expressions, the temporal operators are defined as follows:

α.**start**	selects the time when α starts.
α.**stop**	selects the time when β stops.
α ; β	selects the interval between the time when α starts and β stops.
α , β	selects the interval during which α and β overlap (or if α and β both evaluate to instants, then this operation selects the interval between the first occurrence of α and the next occurrence of β).

TQuel does not allow the selection operator ("|") in event expressions because it could be ambiguous. For example, the following event expression is invalid:

```
(a ; (b|c)).stop
```

If the sequence "abc" occurred, then either the event associated with "b" or the event associated with "c" could be returned.

Another example of a TQuel query is as follows:

Example 2:

```
range of A is Iteration
range of B is Iteration
retrieve Catch
where A.Process=P1 and B.Process=P2
        and A.Internum=B.Internum
when A.start ; B.start
at B.start
```

In this example, the event relation Catch is derived from the period relation Iteration(Process,Internum). The events selected are those of starting an iteration of process P2 during an iteration of process P1.

Snodgrass had difficulty in extending Quel aggregates to handle time. His problem was in whether to quantify over time or over states that overlap (and over simultaneous events). This problem can be partly attributed to Quel, but also to the controversial Barcan formula, as discussed previously in Chapter Three. (See Section 3.2.9.) For period relations, TQuel supports two versions of the operator **count**:

1. All tuples existing at time *t*.
2. All tuples existing up to or including time *t* (cumulative over time).

Aggregates on event relations in TQuel are cumulative, assuming simultaneous events are either unlikely or of little interest in aggregate operations.

4.2.3. Discussion

A major problem with both EDL and TQuel is that many useful temporal operators are missing, but no *new* temporal operators can be defined, without changing the syntax and semantics of these languages. Even if these languages were changed, some temporal operators could never be defined, e.g., iteration of event sequences cannot be specified in TQuel because of the limitations of conventional relational query languages. (We cannot express universal quantification in TQuel or in Quel.) Another problem in TQuel is confusion over expressing quantification over time (resulting in two representations, only one of which assumes the Barcan formula).

4.3. YODA's Approach to Trace Analysis

The temporal operators provided by EDL and TQuel are a subset of those used in restricted, simple path expressions. Recall that, in Chapter Three, we showed that path expressions of this class can be expressed as temporal formulas. There are many formalisms for expressing program specifications; however, we showed that interval formulas offer advantages over (point-based) temporal formulas. We also showed that all interval formulas can be expressed in the interval-logic system $\mathbb{C}$.

In our approach, we express assertions about program behavior as interval-logic formulas in system $\mathbb{C}$. We use queries to test assertions about program behavior. That is, if the specification of a program implies a property, ψ, then analysis of the program asks the question:

"Does ψ hold over the interval `I`?"

Although we can express all assertions with interval formulas, we also implement path expressions, mainly for convenience. We analyze a trace by asking if the observed computation satisfies a given $\mathbb{C}$-formula, ψ, or if a given path expression, ϕ, accepts the observed sequence of events.

4.3.1. Trace Queries

Queries motivate a retrospective approach to debugging. We assume that information about the program's execution history has been captured as an event stream into a *trace database*. We view a trace database as an historical database containing a collection of events ordered by their occurrence.

An *historical database* is a static collection of time-dependent facts that are organized in such a way that one can ask questions about the temporal relationships of facts in the database. To keep track of when the facts in a database are true, we need to record a clock time, or "timestamp." *Timestamps* are positive numbers that provide a temporal ordering over facts in an historical database.

Although we maintain all program activity as events, we support queries over abstractions on events. Processing temporal queries on a *trace database* involves defining abstractions on events and testing assertions against the abstractions. We view events as primitives, and states as predicates on events. Trace queries are supported with interval relations similar to those presented by Allen (Ref. 100), e.g., "during," "before," and "after."

Queries can take many different forms, e.g., one may want to ask

- Did all the expected events occur?
- Is the sequence of events that occurred allowed by the specification?
- During which intervals did a specified program variable have a certain value?
- What aspect of the program's behavior led to a certain error?

4.3.2. Requirements for Trace Analysis

Desirable features for trace analysis include the following kinds of support:

- collecting a trace of a program's execution,
- selecting the events that are collected,
- maintaining trace data in an historical database,
- processing temporal queries on a trace database, and
- providing a convenient user interface for expressing trace queries.

As in the temporal models surveyed in Chapter Three, we needed to simplify our representation of trace analysis by choosing only essential features and making certain assumptions. We chose an approach that is amenable not only to formalizing trace analysis, but also to automating it. The essential features of our approach are *minimality* of primitives and *expressive power* of queries. We give a minimal

set of operators and show that new operators can be defined with this basic set.

We *ignore* various features. These include the selection of events and a *syntax* for a trace query language. Features that can be included are a user interface for simplifying the expression of queries and extensions for debugging distributed programs.

In this chapter we illustrate our debugging approach with only a few classes of program events to give the flavor of trace analysis, without specifying the events to be selected. The programming language and errors under consideration motivate the level of abstraction. In later chapters we elaborate on the details of trace-analysis techniques for Ada, including monitoring techniques, the selection of events, and relationships among them.

This chapter presents the major concepts of trace analysis. Implementation issues (for example, time and space requirements for monitoring programs, trade-offs in efficiency of queries, and the user interface) are issues that we are continuing to investigate.

Instead of developing the syntax of a query language, we simply define predicates using Prolog, which allows a logical basis for reasoning about time and provides flexibility in defining temporal operators. We implement system ℂ and path expressions in Prolog. That is, we define interval-logic formulas and path expressions as Prolog predicates. We capitalize on Prolog's extensibility for defining new operators.

We emphasize that this presentation may differ from the query language seen by the user. Features that are not supported directly by Prolog can be defined in the user interface. For example, Hornsby and Leung (Ref. 118) are building a relational query language that interfaces with Prolog. This language is to provide the numeric aggregate operators that are supported by Quel, such as **maximum**, **minimum**, **average**, **sum**, and **count**. Temporal operators can be defined in a natural-language query language that interfaces with Prolog.

We present our approach to trace analysis in three layers:

1. defining sequences and abstractions on sequences,
2. expressing ℂ using sequences and
3. program traces using ℂ.

4.4. Sequences

In this section we define sequences and operations on sequences.

Definition 1: A *sequence* is an ordered collection of objects.

Each object X in a sequence S is called a *member* of S, denoted $X \in S$. Each member of a sequence has an *index* giving its position or order in the sequence.

Definition 2: If A and B are sequences, then A is a *subsequence* of B, denoted $A \lessdot B$, iff every member of A is a member of B, and A preserves the ordering of B.

Let A be an arbitrary sequence of objects of class X ordered by index I. Let N be a fixed, positive integer, and let P be an arbitrary predicate on objects of class X. Let J be a secondary index for ordering objects of class X. We define the following operations on sequences:

slice(P, A, B)	B is a subsequence of A, and all members of B satisfy predicate P.
first(N, P, A, B)	B is a subsequence of A, and for each member X of B, X has index I such that $I \leq N$ and X satisfies P.
last(N, P, A, B)	B is a subsequence of A, and for each member X of B, X has index I such that $I \geq N$ and X satisfies P.
order(P, J, A, B)	Sequence B is ordered by index J, and X is a member of B iff X is a member of A and X satisfies P.

4.5. Implementing ℂ in Prolog

Moszkowski and Manna (Ref. 72) have argued that interval formulas, such as those of ITL (defined in Chapter Three), cannot be expressed directly in Prolog. In particular, they give the following arguments: Prolog has no sense of time, there is no analog in Prolog to ITL assignments, and there is no analog in Prolog to ITL while-loops.

On the contrary, we show that ℂ-formulas can be expressed in Prolog. In Section 4.8.1 we show that ITL assignments can be expressed in Prolog, and we outline an approach to expressing ITL while-loops in Prolog.

4.5.1. Prolog Semantics

Prolog is based on *Horn clausal logic*, which is a subset of predicate logic. Briefly, Prolog programs contain databases of rules and facts, called *clauses*. Appendix A gives a detailed introduction to Prolog, including a description of its syntax. All predicate-logic formulas can be normalized to the *clausal form* of logic. (The details of this normalization process have been described previously, e.g., by Kowalski (Ref. 119).) Translation into clausal form removes all equivalences and explicit quantifiers.

A proposition in clausal form consists of a collection of *clauses*. A clause is an expression of the form:

$$A_1,...,A_n \texttt{<--} B_1,...,B_m$$

where "<--" is a logical connective meaning "if," or "implies," $A_1,...,A_n$ are called *conclusions*, and $B_1,...,B_m$ are called *conditions*. Conclusions are unnegated atomic propositions separated by disjunction (";"). Conditions are atomic propositions separated by conjunction (","). Collectively, conclusions and conditions are called *literals*.

In Prolog, "not" (meaning, not provable) is expressed as \+ P. This operation means that if the goal P has a solution, fail; otherwise, succeed:

```
\+(P) :- P, !, fail.
\+(_).
```

A clause can have several alternate conclusions, at least one of which must hold, and several

conditions, each of which must hold. A *Horn clause* is a clause with at most one conclusion. Prolog clauses correspond directly to Horn clauses.

The execution of a Prolog program can be viewed as processing via goal reductions, where unifiable clauses for subgoals are searched with a top-down, depth-first strategy. The search involves resolution, i.e., matching a condition of one clause with a conclusion of another clause. Goal reduction through resolution allows Prolog programs to behave as both recognizers and generators. For example, a Prolog-based parser can generate as well as recognize legal strings in the language it parses.

In implementing ℂ, we use the following Prolog predicates:

member(X,L)	For object X and list L, succeeds if X is a member of L.
findall(X,P,L)	Collects all objects X that satisfy goal P into the list L. (If no objects satisfy P then L is the empty list.)
forall(X,P)	For each success of goal X, goal P is executed.* (The **forall** predicate always succeeds.)

We introduce the **testall/2** predicate: "For each success of goal X, goal P is executed, but **testall** fails if any individual test fails." A simple implementation of **testall/2** is as follows:

```
testall(G,P) :- \+(G,\+ test(P)), testpassed.

test(P) :-      P, !.
test(_) :- assert(testfailure).

testpassed :-   \+ testfailure, !.
testpassed :-   retractall(testfailure), fail.
```

4.5.2. Implementing ℂ using Sequences

We use sequences in applying ℂ-operators to Prolog databases. Recall that ℂ includes primitives for specifying intervals, endpoints of intervals, and the pseudo-date "now." We can define the endpoints of ℂ as single timestamps and the intervals of ℂ as pairs of timestamps.

Let the variables $t_1,...,t_k$ be consecutive timestamps and let the variables $T_1,...,T_k$ be (closed, discrete) intervals between any pair of timestamps. As in Clifford's historical databases (Ref. 89), we always interpret the variable "now" as the latest state of the database, i.e., "now" is evaluated to t_k, the timestamp of the most recent fact in the database.

We implement sequences as *lists* in Prolog. To create a sequence, we need to select clauses from a Prolog database and append them to a list in chronological order. Let P be an arbitrary predicate and let A be a variable. For creating a sequence, A, we could define the following Prolog predicate:**

---

* The **forall** predicate is defined in Prolog as `forall(X,P) :- \+ (X, \+ P).`

** In this implementation of **slicedb/2**, the entire predicate P becomes a member of the sequence. Other implementations can be defined to collect only specified arguments of P or different event types matching P. If P is a disjunctive clause, then the resulting sequence will be out of order and will need to be re-ordered by timestamps.

```
slicedb(P,A) :- findall(P,P,A).
```

That is, P is a member of sequence A iff X is a clause (in a Prolog database) that satisfies P.

Slices improve the efficiency of processing universal queries; however, database lookup is preferable for existential queries. We can always examine slices, but we need database lookup to obtain the slices (e.g., by applying the predicate **slicedb/2**).

The primitives of system **C** are easily mapped to Prolog:

1. Predicate variables: P and Q.
2. Variables for (closed) temporal intervals: T and S.
3. Variables for timestamps: Tminus, Tplus, Sminus, and Splus.
4. The variables Ti and Tj for arbitrary instants.
5. The predicate **now** for *n* (the pseudo-date "now"). This predicate has a single argument, which ranges over instants.
6. System-defined relational operators: **<**, **=**, and **>**, for comparing instants or "now."
7. The variables X,Y,Z, ..., ranging over domains other than time.
8. The connectives of propositional logic, including the system predicates: **not** for negation and ',' for conjunction.

   The system predicate for implication is ':-'; however, this predicate cannot be expressed in a query. We define a new operator for implication:

```
            :- op(240,xfx,=>).
P=>Q        :- (not P ; Q), !.
```

   We define an operator for equivalence:

```
            :- op(240,xfx,<=>).
P<=>Q       :- P=>Q, Q=>P.
```

9. Converting to Horn clauses eliminates the need for explicit quantifiers.
10. The primitives '..' and **in** are declared as Prolog operators:

```
                    :- op(220,xfx,'..').
                    :- op(230,xfx,in).
Ti in T             :- T=Tminus..Tplus,
                       Tminus=<Ti, Ti=<Tplus.
```

    The operator **successor** is defined by the granularity of timestamps, and **predecessor** is defined as follows:*

```
predecessor(Tj,Ti) :- successor(Ti,Tj).
```

    The temporal operators R,E, and C are defined with arguments having a temporal domain.

For system **C**, we defined the interval relations: **during**, **meets**, and **next**. These temporal relations are easily expressed in Prolog. Recall (from Chapter Three) that although Allen defined a basic set of 13 temporal relations between intervals (such as "during," "contains," "before," "overlap," "meets," and

---

*Because **succ** and **pred** are system-defined Prolog predicates, we need to define new predicates: **successor/2** and **predecessor/2**.

"equal"), he showed that the only interval relation required is "meets," because the other relations can be defined with the "meets" relation, the existential quantifier, and propositional connectives (Ref. 88). System ℂ provides the mechanics for declaring any relation between temporal intervals, as required by the user, e.g.:

```
:- op(240,xfx,during).
:- op(240,xfx,meets).
:- op(240,xfx,next).
:- op(240,xfx,before).
:- op(240,xfx,overlaps).
:- op(240,xfx,contains).
:- op(240,xfx,after).

during(T,S)      :- T=Tminus..Tplus, S=Sminus..Splus,
                    Tminus >= Sminus, Tplus =< Splus.

meets(T,S)       :- T=_..Tplus, S=Sminus.._,
                    Tplus = Sminus.

next(T,S)        :- T=Tminus.._, S=_..Splus,
                    successor(Splus,Tminus).

before(T,S)      :- T=_..Tplus, S=Sminus.._,
                    Tplus < Sminus.

overlaps(T,S)    :- T=Tminus..Tplus, S=Sminus..Splus,
                    Tminus < Sminus, Sminus < Tplus,
                    Tplus  < Splus.

contains(T,S)    :- during(S,T).

after(T,S)       :- before(S,T).
```

4.5.3. Temporal Completeness

We will say that a query language is *temporally complete* with respect to ℂ if any assertion that can be *specified* via a formula in the C-calculus can be *tested* via a statement in the query language.

Theorem 3: Any assertion in system ℂ can be expressed as a query in Prolog.

Proof: We use induction on the number of operators in an expression.

Basis: Zero Operators

Let p be an arbitrary predicate with a list of attributes, X.

$R_t(p_X)$ is expressed as ?- p(X,Tminus).

$C_T(p_X)$ is expressed as ?- p(X,T).

$E_T(p_X)$ is expressed as ?- p(X,S), S during T.

Induction: Let α and β be formulas in ℂ, involving no more than k-1 operators. These interval formulas can be expressed in Prolog as $V(\alpha)$ and $V(\beta)$, respectively, by the induction hypothesis.

Case(1) Negation: $\sim\alpha$ is expressed as ?- \+V(α).

Case(2) Conjunction: $\alpha \wedge \beta$ is expressed as ?- V(α), V(β).

Case(3) Quantification over Attributes:

Let **typeX/1** be a predicate that generates (via backtracking) all objects X implied by the quantification ∀X. (Objects can be quantified over attributes, e.g., over all times up to **now** or over all variable names.)

$(\forall X)\alpha_X$ is expressed as `?- testall(typeX(X), V(α(X))).`

$(\exists X)\alpha_X$ is expressed as `?- V(α(X)).`

Case(5) Implication: $\alpha \Rightarrow \beta$ is expressed as `?- V(α) => V(β).`

Case(6) Equivalence: $\alpha \equiv \beta$ is expressed as `?- V(α) <=> V(β).`

4.6. Expressing Path Expressions in Prolog

We have shown that we can implement system **C** in Prolog and that we can implement path expressions in **C** (in Chapter Three). Yet, path expressions are widely used in specifying program behavior; hence, we show that we can implement them directly in Prolog. Recall that a path expression can be identified with the set of strings it accepts. We match path expressions against an historical database to confirm that a particular sequence has occurred. We verify path expressions against slices.

We propose using Definite Clause Grammar forms (DCGs) (Ref. 120) for implementing path expressions. (See Appendix Section A.9 for a description of DCGs.) Each path expression is translated into the DCG form of its equivalent grammar. For specifying program properties, DCGs provide a *much* richer formalism than path expressions.

The slice is "parsed" to determine if it can be generated by the DCGs representing the given path expression. If parsing of the slice fails, then the program has failed to conform to the specification expressed by the path expression. Appendix B gives the Prolog program for transforming a path expression into a set of DCGs.

For example, the following path expression:

path `a;b;c` **end**

generates the following DCGs:

```
node1 --> [].
node1 --> [a], node2.
node2 --> [b], node3.
node3 --> [c], node1.
```

4.7. Events, States, and Execution Histories

We have shown that we can express temporal operations on an historical database. We now apply temporal operators on a trace database. Trace analysis deals with events, states, and execution histories. In Chapter Three we discussed the definitions of events and states for a general theory of

temporal logic and, in particular, for program verification. In this section we give formal definitions for *events*, *traces*, *slices*, and *states*.

4.7.1. Assumptions

In developing trace-analysis techniques, we make the following assumptions:

- Time is linear, discrete, and absolute.
- A global clock exists (and is accessible to all processes generating events). Timestamps are positive integers. The clock is incremented by one each time an event occurs. That is, timestamps have no relationship with real time.
- A finite number of events occurs at any instant.
- Concurrency is achieved via both shared memory and message passing.
- All traces are of finite length.

By assuming a global clock, we can always order distinct events by their relative temporal position. To ensure that the length of a trace be finite, we assume that either the execution of the program eventually terminates, or it is (temporarily) halted, for example, by an interactive breakpoint (i.e., all tasks are halted).

4.7.2. Events

Definition 3: An *event* is a program action that occurs at an instant.

An event is denoted by its *class* and can be parameterized by one or more *attributes*. Examples of event classes are variable assignment, entry call, and the start of a rendezvous. The event class distinguishes events by their characteristics. Event attributes distinguish occurrences of events within the same class. We will use "event" to mean either an action or its description.

The number of attributes and their domains can differ for each event class; however, each event has a temporal attribute, which consists of a single timestamp. For example, in addition to a timestamp, the event of "variable assignment" can take the following attributes:

- the name of the variable whose value has changed,
- the new value of the variable, and
- the program location where the assignment has occurred.

The event of "accepting an entry call" can include the following attributes:

- the name of the calling task,
- the name of the called task, and
- the name of the called task entry.

The same timestamp is assigned to occurrences of *event aliases*, i.e., primitive actions of an *atomic* event. For example, the event of beginning a rendezvous requires several primitive actions, such as starting the execution of an Ada accept statement and removing a task name from an entry queue. The

relative ordering of these individual actions is transparent to the programmer and unimportant for debugging. If both actions are recorded, then they are assigned the same timestamp, so that each rendezvous can be associated with its corresponding queue update.

4.7.3. Event Instances

Definition 4: An *instance* of an event is a single occurrence of an event, having the form:

$e(\alpha_1,...,\alpha_m,t_i)$

where e is an arbitrary event class, $\alpha_1,...,\alpha_m$ are values of attributes, and t_i is an arbitrary timestamp.

Each instance of an event has a single timestamp and a value for each attribute in its class. For example, consider an instance of the event "variable assignment":

```
variable_assignment(index,5,main,10)
```

Here, "index" is the identifier of the variable to which a value is assigned, "5" becomes the value of "index" immediately at the assignment, "main" is the name of the procedure in which the assignment occurs, and "10" is a timestamp.

4.7.4. Traces

Each event instance represents a fact about a program's execution history. We will write e_i to denote the event occurring at time t_i, i.e.,

$R_{t_i}(e) \equiv e_i$

(Here, we use the R-operator of the basic temporal-logic system, described in Chapter Three.)

Definition 5: A *trace* of a program is a finite sequence of event instances, $E = <e_1,e_2,...,e_k>$, where k is the *length* of the trace (the number of event instances that it contains). The first event in the trace is *program activation*, and the last event is *program termination* (or the current breakpoint).

For example, consider the following fragment of Ada code:

```
TEST:
  for MY_INDEX in 8..10 loop
    null;
  end loop TEST;
```

If assignment to the program variable MY_INDEX* were monitored during execution of the above code, then the trace would contain three event instances:

```
variable_assignment(my_index,8,test,1)
variable_assignment(my_index,9,test,2)
variable_assignment(my_index,10,test,3)
```

Here, 1, 2, and 3 are consecutive timestamps.

---

*In "standard" Ada notation, variable names are in uppercase letters. In collecting traces, we use the names of Ada program variables as constants. In Prolog, constants must begin with lowercase letters.

4.7.5. Slices

A *slice* is a subsequence of program activity. More formally,

Definition 6: A *slice* is a subsequence of a trace, i.e., X is a slice of trace E iff $X \blacktriangleleft E$.

Membership in a slice is determined by a predicate on event attributes and event classes. The following are examples of slices:

- all assignments to a specified program variable,
- all definitions and uses of a specified variable, and
- all calls from a specified task to a specified task entry.

4.7.6. States and State Instances

Recall from Chapter Three that events and states are duals. Given that a state is an encoding of the sequence of prior events, we can view a state as a *predicate on events*. States can be characterized declaratively as sets of constraints on events. For example, an Ada task is "callable" after task activation and before task completion or termination. This representation of states is convenient for expressing the absence of events (invariance), time outs (cancellation events), and delays (intervals between events).

Each *instance* of a state is associated with two timestamps, one for each endpoint of the interval it spans. For a *final state*, the second endpoint is the timestamp of the final event in the trace (i.e., program termination or a breakpoint). We use the notation of system ℂ (defined in Chapter Three) for specifying the endpoints of intervals, i.e., $T \equiv t^{-}..\ t^{+}$. We will write s_I to denote the state holding over interval T_I, i.e.,

$$C_{T_I}(s) \equiv s_I$$

(Here, we use the C-operator of system ℂ.)

Definition 7: A *state instance* is a predicate succeeding over a specified trace or slice, written in the form:

$$s(\alpha_1,...,\alpha_m,T_I)$$

where s is an arbitrary state class, $\alpha_1,...,\alpha_m$ are values of attributes, and T_I is an arbitrary closed interval.

For example, in Ada, a deadlock state can arise from a single event (a task calling itself) or from a sequence of events (a cycle of entry calls). A state can be a predicate on an ordered pair of events (e.g., a start and stop event):

s_I such that I=i..**predecessor**(j) iff e_i and e_j

States attain their attributes from the events over which they are defined. We assume a "persistence of facts," e.g., a variable retains its current value until it is redefined, and the length of an entry queue is unchanged until the next entry call, rendezvous, or cancellation occurs. A state can be defined over events that are triggered by different processes, e.g., by processes that access a shared variable.

The representation that we have chosen for states fits Prolog well. For example, we can define the

state **variable_state** as a predicate on two successive occurrences of the event **variable_assignment**:

```
variable_state(X,Value,T):-
     now(Now),
     variable_assignment(X,Value,Location,Tminus),
     nextvalue(X,Tminus,Now,Tplus),
     predecessor(Tplus, Ti),
     T=Tminus..Ti.

nextvalue(X,Tminus,Now,Tplus):-
     variable_assignment(X,_,_,Tplus),
     Tplus in Tminus..Now.

nextvalue(X,Tminus,Now,Now):-
     not((variable_assignment(X,_,_,Tplus),
     Tplus in Tminus..Now)).
```

4.7.7. Event and State Relationships

Having formally defined events and states, we now define relationships among them.

Let s_i and s_j be state instances. We say that s_i "holds before" s_j iff T_i is before T_j:

s_i **holds_before** s_j iff T_i **before** T_j

Temporal relations can be defined over states for any interval relation, for example, **holds_after** and **holds_during**.

Let e_i be an event instance and let s_j be a state instance. We say that e_i "occurs during" s_j iff t_i is contained in T_j:

e_i **occurs_during** s_j iff t_i **in** T_j

Other temporal relations can be defined between events and states by considering the instant at which an event occurs as a point interval, e.g.,

e_i **occurs_before** s_j iff $t_i..t_i$ **before** T_j

4.7.8. Temporal Views of Trace Databases

The trace database contains events captured from the time program activation occurs until the program terminates (or is halted). A query with no temporal constraints selects events occurring up to and including "now"; thus, the Barcan formulas of Section 3.2.9 hold for trace queries. (See discussion in Section 4.2.2 on aggregate operators.)

Our approach supports *temporal views* of trace databases, such that a query retrieves only those events that have occurred during a specified bounded interval. The programmer can obtain a temporal view of the trace database by appending temporal constraints to queries. Let **eventP/2** and **stateP/2** represent, respectively, an arbitrary event instance and an arbitrary state instance in the following queries:

```
?- stateP(Attribute, Stime), Stime during 30..40.

?- eventP(Attribute, Stime), Stime > 10.

?- stateP(Attribute, Sminus.._),Sminus>50.
```

4.8. Comparison with Previous Work

In this section we compare trace queries with specifications in ITL and with queries in TQuel.

4.8.1. Comparison with ITL

Recall that ITL variable assignment has the following form:

```
A<--B+2
```

This is equivalent to asking in Prolog:

```
?-assignment(a,b+2).
```

where the predicate **assignment/2** is defined as follows:

```
assignment(X,Y) :-
        variable_assignment(X,XValue,_,Sminus),
        exp_eval(Y,XValue,Sminus).

exp_eval(XValue,XValue,_) :-  /* constant */
        number(XValue).

exp_eval(Y,YValue,Sminus) :-        /* variable */
        variable(Y),
        variable_state(Y,YValue,Tminus..Tplus),
        successor(Tplus,Ti), Ti >= Sminus.

exp_eval(A+B,XValue,Sminus) :-      /* addition */
        exp_eval(A,AValue,Sminus),
        exp_eval(B,BValue,Sminus),
        XValue is AValue + BValue.
```

The predicate **exp_eval/3** can be further defined, e.g., for subtraction and multiplication.

Recall that the ITL formula for a while-loop has the following form:

beg(I=0) ∧ **while**(I<n) **do** ([I<--I+1])

This can be expressed in Prolog by taking a slice of assignments to variable `I` and comparing every two elements of the slice successively:

```
while(I,N)      :-
  findall(Value,
          variable_assignment(I,Value,_,_),
          A),
  increasing_sequence(A,0,N).

increasing_sequence([N],_,N) :- !.

increasing_sequence([M|L],M,N) :-
        successor(M,M1),
        increasing_sequence(L,M1,N).
```

4.8.2. Comparison with TQuel

Prolog queries are more general and more flexible than those of Quel or of TQuel. Conventional query languages, such as Quel, cannot deal with universal quantifiers. For example, recall the ambiguity problem that Snodgrass had with the selection operator. Prolog can get *both* answers, because of unification.

Example 1 from TQuel can be expressed in Prolog as follows:

```
start_running(Process, Start)       :-
        running_on(Process, processor1, Start..._),
        tick("3:00pm", CTime), Start > CTime.
```

Example 2 from TQuel can be expressed in Prolog as follows:

```
catch(Start2)    :-
        iteration(p1,Internum,Time1),
        iteration(p2,Internum,Start2..._),
        Start2 in Time1.
```

4.9. Summary

The concept of time is crucial to debugging. A system that reasons about time-varying data must incorporate a representation of knowledge about time. Debugging requires specifying and examining program behavior to isolate the differences between expected behavior and observed behavior.

We have used formal mechanisms in analyzing the behavior of concurrent programs. Temporal models motivate a *trace* approach to debugging. We have adapted temporal-specification techniques to the task of providing database access to a trace history.

The trace-database approach to debugging is an abstraction and representation of a program's execution. In this approach, program activity is captured as an event stream and maintained in an historical database. We use the following abstractions on a trace of program events: states, slices, and interval relations.

We have extended Prolog with temporal-logic primitives for expressing time-dependent relationships among events and abstractions on events. One consequence of using Prolog is that knowledge about time is easily expressed in trace queries. The interval-logic system $\mathbb{C}$ and path expressions provide the formalism for defining temporal operators in Prolog. We could implement branching-time (multiple traces); however, in practice, it would be costly to retain and query multiple traces of a complex program.

By querying a trace database, a programmer can test assertions about program properties. The programmer can pose queries on the ordering of states, on the sequencing of events, and on the simultaneity of a given set of conditions. Practicality is the major advantage that trace analysis offers over verification.

From their investigation of debugging techniques for Ada tasking programs, Helmbold and Luckham

concluded (Ref. 48):

> Because history is often important in determining the cause of a deadness error, the programmer should be able to query the past tasking states of relevant history.

In the next chapter we explore implementation issues dealing with the selection and representation of trace events in Ada.

5. YODA: AN ADA PROTOTYPE DEBUGGER

Is thy face like thy Mother's, my fair child!
Ada! sole daughter of my house and heart?
When last I saw thy young blue eyes they smiled,
And then we parted, not as now we part,
But with hope.

--Lord Byron (*Childe Harold's Pilgrimage*)

Chapter Four introduced a debugging approach in which a program's execution is captured as an event stream into an historical database. The semantics of the underlying programming language determine the events to be traced. This chapter addresses event selection for Ada programs and presents the details of our implementation for collecting a trace database.

5.1. Ada Terminology

Development of the Ada language began in the mid-70's and an ANSI standard was approved in 1983. Ada was designed for the U.S. DoD by Cii Honeywell Bull, with Jean Ichbiah as the principal designer. The language was named for Lady Augusta Ada Byron (1815-1852), the Countess Lovelace (daughter of the poet Lord Byron), in recognition for her contributions as the world's first programmer (Ref. 121). The ALRM is known also as Military Standard 1815A (MIL-STD 1815A), for the year in which Ada Lovelace was born.

Since January 1984, Ada has been mandated by the U.S. DoD as the standard language for mission-critical software (Ref. 122, 123). Interest in Ada has spread to other continents, for example, the United Kingdom has announced plans to mandate Ada as the Ministry of Defense's standard language on real-time operational systems as of 1987.

5.1.1. Modularity

The major components of Ada programs are *subprograms*, *packages*, and *tasks*. A main program can be any subprogram. The specification of a main program is arbitrary; thus, linkage may require that the main program be designated, if ambiguous. Generally, it is treated as a procedure that is called by an implicitly declared main task.

5.1.2. Names and Program Objects

The Ada language defines many different entities, including identifiers, numeric literals, exceptions, single entries, entry families, formal parameters, and compilation units. Each *identifier* is used as either a *name* or an *object*. A name may be the name of a pragma, a number, a type, a subtype, a label, a loop, an enumeration literal, or a block. An object is an entity that contains a value of a specified type. Ada permits overloading of identifiers and of operators. (Unlike Prolog, no new operators can be defined.)

A type has a *class*, a range of values, and a set of operations. The main classes of types are scalar (including enumeration, integer, and real types), composite types (including records and arrays), access types, and private types. The package STANDARD encloses all system-defined library units (such as TEXT_IO for handling input and output). Types that are predefined in the package STANDARD include Boolean (an enumeration type) and duration (a real type).

5.1.3. Strong Typing and Scoping

All identifiers in an Ada program must be declared explicitly, except loop names, block names, and statement labels. A section of code that can contain declarations is called a *declarative region*. Entities that can contain declarative regions are

- the specification and body of a subprogram,
- the specification (both the visible and private sections) of a package and a package body,
- the specification and body of a task,
- the specification and body of a generic package or of a generic subprogram,
- an entry declaration and its corresponding accept statement,
- a generic parameter declaration,
- a record type declaration,
- a renaming declaration, and
- a block statement or loop.

The first declarative region around an entity is its "immediate" scope. Some names are visible outside their immediate scope, for example, entry names and formal parameter names.

The usage and scope of an Ada identifier is determined statically by its declaration. In the visible part of a package specification, private types can be declared as incomplete types, but their declaration must be completed in the private section of the package.

5.2. Implementation of YODA

YODA is a stand-alone system that captures events by inserting probes into the program to be debugged. YODA parses an Ada program, generates a symbol table, and embeds diagnostic output statements into a copy of the source program. When the annotated program is compiled and executed, the diagnostic statements invoke a program monitor to capture trace data.

YODA's parser generates an abstract syntax tree. This parser is based on the LALR(1) grammar for Ada '82 (Ref. 124). (We eliminated left recursion and factored out common prefixes of alternate rules.) The parser fails if the input program has syntactic errors; thus, YODA requires that the program to be debugged have compiled successfully. Appendix C shows the implementation of YODA's top-down parser.

Semantic analysis, annotations, and pretty-printing are performed while traversing the abstract-syntax tree. Semantic analysis generates a symbol table and selects the critical points at which annotations are required. Pretty-printing is required for saving the annotated program.

Semantic analysis of an Ada program generates two new programs: a Prolog database containing the symbol table of the Ada program and an annotated version of the original Ada program. We will refer to the original Ada program as the "program unit under analysis" and the annotated version as the "annotated program unit."

5.2.1. The Symbol Table

The *immediate* symbol table, that is, the symbol table of the program unit under analysis, is passed as an argument during traversal of the syntax tree. It is organized as a binary tree to provide efficient table lookup and to allow for maintaining information that may be incomplete.

Separate compilation is supported by saving symbol tables of parent program units. When the semantic analysis of a program unit is completed, the immediate symbol table is written to a file, after conversion from binary-tree format to a Prolog database (one fact for each symbol declared).

To process a program subunit, the user provides an ordered list of the filenames containing the symbol tables of parent program units.* Although symbol tables of parent units are stored as external Prolog databases, they are converted to *recorded* databases (internal, indexed databases) before processing the subunit. A recorded database provides quicker access than the binary-tree format. The symbol table of the predefined library package STANDARD is maintained as a recorded database that is saved with the YODA program.

Symbol-table entries can be parameterized. We include parameters for the following entities: type names, subtype names, and identifiers used as objects (e.g., number_constant, named_component). The symbol table gives the main class of each type name and each subtype name. For example, Boolean is a type name that is in the type class enumerated. Objects that are declared implicitly are given a single argument, their base type. For example, loop_parameter_name is declared implicitly as scalar.

The symbol table contains the name, base type (e.g., enumerated type), usage (e.g., Boolean), and "declarative context" (e.g., system-defined) of objects that are declared explicitly in the program. Declarative context refers to the hierarchy of an identifier's enclosing declarative region(s), as opposed to its scope (ALRM 8.2). When names for loops and blocks are not declared explicitly, YODA generates names: *loop_name1*, *block_name1*, etc. Scoping rules control symbol-table lookup.

---

*This can be a nuisance, but the alternative is to map filenames to program unit names, and then get the parent unit names from the Ada **with** clauses. This would ensure that the ordering of library units conform to Ada specifications, but would require implementing a library-management facility, which is usually provided by the Ada environment.

Since the focus of our research was on showing feasibility of our debugging approach, we kept our naming conventions minimal. For example, overloading was not supported. Also, we do not create new names for objects that are created dynamically, e.g., each time a procedure is invoked, any task defined in the procedure is always given the same name.

The type name and main class of the type of each object are saved so that they can be used in queries, e.g., "list all objects of type COLOR." The class of an identifier declared as an incomplete type is left uninstantiated until the declaration is completed. Incomplete information can be maintained in the symbol table, because Prolog allows uninstantiated variables.

Table 5-1 shows the structure of the symbol table. Appendix F shows symbol tables for the program example presented in Appendix E.

Declarative context is represented as a Prolog list, such that the most deeply nested declarative region is at the head of the list. While traversing the syntax tree, YODA maintains the names of successive declarative regions in the Declarative Context List (DCL).

When the semantic analyzer reaches a node that defines a declarative region (e.g., a subprogram body or loop), it appends the name of the region to the head of the DCL. The head of the DCL is removed (implicitly) on popping the stack of Prolog goals. Thus, only the descendants of the current node will use the new DCL. On encountering a subunit, the semantic analyzer appends the expanded name of the parent unit to the DCL. For example, the declarative context of `x` is `'[t,c,b,a]'` in the following subunit:

```
separate (A.B.C)
task body T is
   X:     INTEGER;
begin
  null;
end T;
```

YODA ignores some declarative regions to simplify symbol-table lookup. These omissions are in the definition of package specifications and bodies. Names are appended to the DCL when the semantic analyzer reaches any of the following nodes in the syntax tree:

ALRM Section Number	Node
5.5	loop_statement
5.6	block_statement
6.1	subprogram_body
6.3	task_declaration and body
9.1	entry_declaration and its corresponding accept statement
10.2	subunit
12.1	generic_declaration of a subprogram

All separately compiled, user-defined packages are viewed as program units enclosing the main

Table 5-1: Usage Categories of Symbol Table

ALRM Section #	Usage Category
2.8	pragma_name
3.2	pragma_argument
	object_name(Class,TypeName,TypeRegion)
	constant_object_name(Class,TypeName,TypeRegion)
3.3.1	number_constant(scalar)
3.3.2	type_name(Class)
3.5.1	subtype_name(Class)
3.7	enumeration_literal(scalar)
3.7.1	named_component(Class,TypeName,TypeRegion)
4.1.4	discriminant_name(Class,TypeName,TypeRegion)
5.1	attribute
5.5	label_name
	loop_name
	loop_parameter_name(scalar)
5.6	block_name
5.8	subprogram_name
6.1	formal_parameter(Class,TypeName,TypeRegion)
7.1	package_name
7.4	type_name(private)
9.1	task_type
	object_name(task_type,anonymous,TypeRegion)
9.5	entry_name
11.1	exception_name
12.1	generic_package_name
	generic_subprogram_name
	generic_type_definition
	generic_parameter_object
	generic_package_instantiation
	generic_function_instantiation
	generic_subprogram_instantiation

program. That is, the package name is omitted from the declarative context of any identifier declared within the package. For example, consider the following fragment of Ada code:

```
package P is
  procedure S(X: INTEGER);
end P;
-------------------------
with P; use P;
procedure MAIN is
begin
  S(X=>1);
end MAIN;
```

The declarative context of `x` is `'[s]'`, not `'[s,p]'`, and the declarative context of `s` is `'[ ]'`, not `'[p]'`.

5.2.2. Operations on the Symbol Table

The following symbol-table operations are provided:

1. Storing and updating information on *identifiers* (names that are not yet in the symbol table or that are specified incompletely).
2. Accessing information on *simple_names* (names that already must have been declared when encountered).

Originally, symbol-table lookup was performed on all identifiers and simple names in the program unit under analysis. The overhead was costly, in light of our assumption that the program was semantically correct. In the current version of YODA, symbol-table lookup is performed for identifiers only. If an identifier is encountered that is not yet in the symbol table, it is added to the table. If an identifier is encountered that is already in the symbol table, but with incomplete information (i.e., an incomplete type), the new information is added to the symbol table.

Symbol-table *search* occurs only when information is needed to complete an annotation. For example, to capture the trace of a variable update, the symbol table must be searched to determine the DCL and usage of this variable. Because we assume that the program has been compiled successfully, we can expect also that each simple name encountered has been declared previously and, thus, must appear in the symbol table.

When accessing a simple name, YODA searches first in the immediate symbol table. If the simple name has not been declared in the current DCL, then the search continues through successive outer layers of declarative regions. For example, if YODA is searching for the variable `x` and the DCL of the current node is `'[t1,main]'`, then YODA searches first for the variable `x` declared in `'[t1,main]'`, and next for the variable `x` declared in `'[main]'`. If the simple name is not found in the immediate symbol table, then the predefined symbol tables are searched in the order in which they have been listed by the user. Finally, the predefined symbol table of the library package STANDARD is searched.

If a simple name is not found in any symbol table, then it is added to the immediate symbol table as an identifier with a usage class of "undeclared." The undeclared option will be encountered only if a predefined symbol table has been omitted from the list.

5.2.3. Annotations and the Trace Database

YODA monitors the following events: variable definition and use, task synchronization, and changes in task status. Each of these events is parameterized with attributes, as described in Chapter Four. Each attribute specifying a program object is qualified by its declarative context. Table 5-2 shows the structures representing these events in Prolog notation.

YODA embeds calls to the program monitor to generate trace-database entries wherever they are syntactically and semantically appropriate. For example, entry calls are detected before their execution.

Table 5-2: Trace Database Events

entry_called(Caller,Callee,Entry,Time).
call_canceled(Caller,Callee,Entry,Time).
entry_queue_lengthened(Caller,Callee,Entry,Time).
entry_queue_shortened(Caller,Callee,Entry,Time).
rendezvous_started(Caller,Callee,Entry,Time).
rendezvous_completed(Caller,Callee,Entry,Time).
var_read(Variable,ProgramLocation,Value,Time).
var_write(Variable,ProgramLocation,Value,Time).
entry_parm_set(Caller,Callee,Entry,IO_Mode,Parm,Value,Time).
task_activated(Task,Time).
task_completed(Task,Time).
ready_to_terminate(Task,Time).
abnormal_termination(Task,Time).
program_ended(Program,Time).

Rendezvous start and completion are detected in the accept statement of the called task. An annotation for variable use precedes its occurrence, whereas an annotation for variable definition follows its occurrence. Initializations in object declarations cannot be annotated with a monitor call, owing to the separation of body and declarative region in Ada. In Chapter Six, Figure 6-3 shows the annotated program for the Ada program in Figure 6-1.

To insert an annotation *before* the current node, the pretty-printer is called directly to write the annotation to the output file for the annotated source. To attach an annotation *following* the current node, the annotator is called to either replace or decorate the syntax tree (i.e., to modify the descendants of the current node). YODA maintains the original version of the syntax tree, such that, on completing all goals, the original tree is preserved and can be displayed.

Each variable definition and use includes the name of the program unit and of the subunit in which it occurs (instead of a statement number). Because Ada is a procedure-oriented language that promotes modularization, information about the context of a statement is more appropriate than its statement number. Also, the temporal ordering of events is more relevant to concurrency than their spatial ordering. For example, it makes more sense to ask:

"List all updates to variable x that occurred while executing task t1."

than to ask:

"List all updates to variable x that occurred between statement numbers 20 and 25."

The formal part of each entry declaration and accept statement is annotated to capture the name and DCL of the calling task. Thus, the name of the calling task can be recorded in task synchronization events, e.g., in entry call, in rendezvous start, and in rendezvous completion. On encountering an entry call, the annotator examines the current DCL to determine the most deeply-nested task body enclosing the entry call. If no tasks are on the DCL, it is assumed that the caller is the implicitly defined main task.

The body of each accept statement is annotated to capture the events of rendezvous start and completion. If the body of the accept statement is empty, a **do-end** pair is added. Since each entry can be associated with more than one accept statement, it would be useful to identify the accept statement that is executing. That is, on encountering an accept statement, the DCL could be updated to reflect not only the name of the entry, but also the ordering of the accept statement. Changes in the length of entry queues are recorded on detecting an entry call, starting a rendezvous, or timing out.

Monitoring of variables is restricted to scalars, in keeping with our emphasis on showing feasibility. Monitoring can be further restricted, e.g., according to variable usage or declarative region, by adding a few simple rules to the annotator. Extending monitoring to composite types would require modifying the program monitor.

Tracing of formal parameters is treated separately from tracing of other variables. Formal parameters of mode **in** are constants; thus, their values need to be captured only once -- immediately after entering a program unit. The value of an **in out** parameter is captured before exiting a program unit. Formal parameters of mode **out** cannot be traced, owing to Ada's restriction against reading these parameters.

Although a formal part can contain many parameters, the association of each parameter with a value is treated as a separate event. Because each formal parameter is traced separately, the programmer can specify the parameters to be traced. Furthermore, a set of parameters can always be specified in a query, e.g., "List the values of all parameters in entry E of task T at time t_1."

We considered tracing all parameter associations of a call as a single event, but concluded that it would be both costly and inconvenient. For example, we considered maintaining all entry parameters of a call as a Prolog list, e.g.,

```
entry_called(caller(taxi,[main]),
        callee(customer,[main]),entry(call),
        [parm1(car_code,"54"), parm2(pay,10)],25).
```

Lists are a natural data structure in Prolog, and list-processing facilities can be written in Ada. Nonetheless, annotating an Ada program to capture these lists would be difficult. Ada lists must be allocated dynamically and Ada's requirement for strong typing places a considerable overhead on maintaining lists of heterogeneous objects (e.g., the difficulty of storage reclamation). The expense of

grouping parameter associations into a single event fails to justify the convenience, which is questionable.

Monitoring the status of each task object requires tracing task activation, completion, and termination. It is difficult to annotate an Ada program in such a way that a task termination event is recorded. Tracing of normal task termination can be replaced by recording the time when a task is "ready to terminate," e.g., on each execution of a select statement with a terminate alternative. Abnormal task termination can be detected by annotating each task body with an exception handler for the exception choice **others** (where there is no explicit others exception).

5.2.4. Program Monitor

The Ada program monitor converts trace data into Prolog clauses and updates the trace database. Also, the monitor controls the global clock, which guarantees that time is represented as a linear series. The clock is implemented as an Ada task with one entry, such that each recorded event triggers one "tick". This approach guarantees a FIFO ordering of the event trace. Thus, the timestamps reflect the order in which events occur in the system.

Recall that in Chapter Four we described event aliases. In YODA, updating the length of an entry queue does not advance the clock, but is synchronized with an entry call, rendezvous start, or cancellation of an entry call. Chapter Six shows several examples of events recorded in sample traces, e.g., in Figure 6-8.

5.2.5. The Trace Query Processor

All queries are expressed in Prolog. To give the flavor of trace queries, we present some examples in Chapter Six. To understand how queries are answered, the programmer can invoke Prolog's debugging facility to show the logical inferences drawn in the solution process. This facility can help to explain how YODA works.

Table 5-3 shows the Prolog rules for converting the program events shown in Table 5-2 to states.

5.3. Conclusions

We have described the implementation of YODA, focusing on the difficulties that we encountered in representing the behavior of Ada programs and on our solutions to these problems. In Chapter Six we use examples to illustrate trace-analysis techniques.

Table 5-3: Translation from YODA Events to States

```
variable(Name,Location,Value,T):-
        var_write(Name,Location,Value,Tminus),
        nextvalue(Name,Tminus,Tplus),
        predecessor(Tplus,Ti),
        T = Tminus..Ti.

nextvalue(Name,Tminus,Tplus):-
        var_write(Name,_,_,Tplus),
        Tplus>Tminus.

nextvalue(Name,Tminus,Now):-
        now(Now),
        not((var_write(Name,_,_,Tplus),
        Tplus>Tminus)).

task_callable(TaskName,T)  :-
        task_activated(TaskName,Tminus),
        task_completed(TaskName,Tplus),
        Tplus>Tminus,
        predecessor(Tplus,Ti),
        T = Tminus..Ti.

task_callable(TaskName,Tminus..Now) :-
        now(Now),
        task_activated(TaskName,Tminus),
        not((task_completed(TaskName,Tplus))).

entrycall(Caller,Callee,Entry,T):-
        entry_called(Caller,Callee,Entry,Tminus),
        successor(Tminus,Ti),
        T = Tminus..Ti.

rendezvous(Caller,Callee,Entry,T):-
        rendezvous_started(Caller,Callee,Entry,Tminus),
        rendezvous_completed(Caller,Callee,Entry,Tplus),
        Tplus>Tminus,
        T = Tminus..Tplus.

rendezvous(Caller,Callee,Entry,Tminus..Now) :- now(Now),
        rendezvous_started(Caller,Callee,Entry,Tminus),
        not(rendezvous_completed(Caller,Callee,Entry,_)).
```

6. DEBUGGING WITH YODA: CASE EXAMPLES

Of all my programming bugs, 80 percent are syntax errors.
Of the remaining 20 percent, 80 percent are trivial logical errors.
Of the remaining 4 percent, 80 percent are pointer errors.
And the remaining 0.8 percent are hard.

-- Marc Donner (Programming Pearls, *CACM*, 1985)

We can divide diagnosis of a program error into two phases: *diagnostic reasoning* (generating candidate diagnoses) and *database reasoning* (evaluating candidate diagnoses).

To evaluate diagnoses, YODA uses the following kinds of information:

1. A trace database and symbol tables.
2. Temporal predicates for specifying relationships between events and abstractions on events.
3. Knowledge of Ada semantics, such as visibility of variables.
4. Knowledge about debugging strategies for some common classes of errors, e.g., cyclic deadlock and misuse of shared data.

Diagnostic reasoning is a *classification* problem. Clancey (Ref. 125) divides the general classification problem into three phases:

1. data abstraction (data about the entity to be classified),
2. heuristic mapping onto a hierarchy of pre-enumerated solutions, and
3. refinement within the hierarchy.

In this chapter we consider the problem of defining an *error hierarchy* to aid in controlling the *selection of tests* for diagnosing an error.

Diagnostic reasoning can be performed by an exhaustive search of all pre-enumerated candidate diagnoses. A better approach is to provide control over diagnostic reasoning via an error hierarchy. Errors can be classified in various ways, e.g., by their recognizable symptoms or by the expected outcome of testing assertions on the program's behavior. Controlled diagnostic reasoning requires imperative knowledge about what tests can either eliminate or promote a candidate diagnosis. These are rules of "good judgment" to govern the debugging strategy.

Tests can be characterized by their role in evaluating hypotheses: *exclusionary*, *confirming*, or *restrictive*. Exclusionary, or negative, tests rule out the possibility of an error or a class of errors. Restrictive tests determine that a class of errors has occurred. Confirming, or positive, tests refine the diagnosis.

This chapter addresses the following topics:

- What is an appropriate model for categorizing program errors?

- What categories of errors can be diagnosed using trace queries?
- Which program events are associated with which errors?
- What expert-system techniques can provide user control over diagnostic reasoning (i.e., how can YODA be incorporated into an expert system)?
- What techniques can improve the efficiency of diagnostic reasoning?

We present several case examples to illustrate trace-analysis techniques for diagnosing run-time errors associated with Ada concurrency. All example Ada programs were written in ANSI Ada and translated and executed using the validated New York University Ada translator and interpreter (Ada/ED ANSI Version 1.1), running under the UNIX 4.1 operating system on a DEC VAX 11/780 computer.

The first three sections of this chapter focus on bug categories. Section 6.1 examines various approaches to categorizing errors. In Section 6.2 we enumerate errors associated with Ada tasking. (This is not intended to be an exhaustive list.) Section 6.3 defines a functional hierarchy for organizing errors.

The remaining sections of this chapter present Ada program examples:

1. The "Deadlock" program exhibits cyclic deadlock. Confirmation of this error relies on YODA's knowledge base for a description of events leading to a cyclic deadlock.
2. The "Lost Update" program exhibits inappropriate use of shared data. This error is diagnosed by testing for *non-serializability*.
3. The "Simplified Stenning Protocol" program is included to show the benefit of slices as event abstractions. This program simulates a communications protocol in which the sender continues to duplicate a message until an acknowledgment is received.
4. The "Taxi Service" program is included to show the benefits of a database approach to organizing information about the structure of a large program. This program was designed using the resource-control paradigm (agent-consumer-producer) discussed in Chapter Three.

6.1. Error Taxonomies: A Survey of Existing Work

We define an *error model* to be a classification scheme for program errors, so that some general principles, or similarity measures, can be applied in partitioning errors into classes. Most of these principles are *descriptive*, i.e., errors are characterized by their features. An error model determines the features that are shared by errors within a class.

Most error models choose from the following principles in characterizing errors:

1. *Semantics* — Group errors by the development phase during which the error can be detected, e.g., during design, compilation, or execution.
2. *Symptoms* — Group errors by their manifestation, e.g., the error message or system response.
3. *Structure* — Group errors by the language construct in which the error occurs, e.g., the statement type or the language feature.

4. *Behavior*	Group errors by the proof technique that can show their absence, e.g., invariance or eventualities.
5. *Difficulty*	Group errors by the level of difficulty in detecting or diagnosing the error, such as the error's reproducibility, its subtlety, and the length of time and expertise required to isolate the error.
6. *Frequency*	Group errors by their reported frequencies in empirical studies of program errors.
7. *Diagnosis*	Group errors by their "cure," e.g., whether code is missing, misplaced, or incorrect.

In the remainder of this section, we review some error hierarchies for several of these principles.

6.1.1. Semantic Error Model

Most language standards organize errors into a few classes associated with the phase of development in which the error can be detected. We present two similar models: a Pascal description and the Ada standard.

In describing errors that can occur in Pascal programs, Eggert (Ref. 126) divides program errors into *logical* and *physical* errors. Recall that a logical error is one that need not be detected at compilation or execution, but can prevent the program from conforming with its specifications.

Physical errors are divided into two categories:

- *Portability errors* -- Need not be detected in compilation or in execution, but the effect of the construct is unpredictable because it relies on information outside the language standard.
- *Language errors* -- Violate a rule contained in the language standard and should be detected during compilation or execution, e.g., exceeding array bounds. These are partitioned into:
  - *Compile-time errors* -- Can be detected before execution.
  - *Run-time errors* -- Cannot be detected before execution.

The ALRM divides program errors into four categories:

- *Compile-time errors* -- Must be detected by Ada compiler before execution.
- *Run-time errors* -- Must be detected during execution of Ada programs.
- *Erroneous constructs* -- Need not be detected in compilation or execution, but the effect of the construct is unpredictable.
- *Incorrect order dependencies* -- Need not be detected in compilation or execution, but the effect of the construct is different, if execution is in a different order. One example of an incorrect order dependency is reliance on a particular algorithm for scheduling evaluation of open accept statements in a selective wait.

An important difference between these error models is the treatment of portability errors. Ada defines two types of errors that can arise when porting software: erroneous constructs and incorrect order dependencies. Examples of erroneous constructs are relying on a particular mechanism for parameter

passing and relying on a particular scheme for storing arrays. Incorrect order dependencies can occur only in programs that use tasking.

6.1.2. Error Modeling by Symptoms

Language manuals and user guides often provide a dictionary or index of error messages and codes. These dictionaries map an error symptom (e.g., illegal operation, fatal error, data-typing constraint) to either the type of statement in which the error may have occurred or the programming feature that may have been used inappropriately. Because the manifestation of an error is determined by both the programming language and the translator for that language, each implementation provides its own error model for symptoms. Binder (Ref. 127) presents a symptom taxonomy as a tutorial aid to debugging programs executing in the MVS operating system environment.

A difficulty in classifying errors by symptoms is that more than one error may yield the same symptoms. For example, the ALRM requires that a "subscript out of range" error raise the language-predefined exception Constraint_Error. Many other errors in Ada will raise this same exception. Thus, the manifestation of a "subscript out of range" error in Ada is inconclusive for diagnosing the error. Some translators may provide more specific error messages, and more information can be obtained by exception handling.

6.1.3. Structural Taxonomy

In a *structural taxonomy*, program errors are grouped by language constructs, e.g., I/O handling, file management, loops, and data-typing constraints. A taxonomy based on structural features helps to localize an error by associating it with a particular type of statement in the program being debugged. It relies on a categorization that is well-defined by the syntax and semantics of the programming language.

For example, Eggert gives a feature-based categorization of program errors in Pascal, demonstrating the effect of the semantics of a programming language on determining the program errors that can be detected during execution (Ref. 126).

6.1.4. Behavioral Taxonomy

Another approach is to categorize program errors by proof techniques, such as those defined by temporal logic in Chapter Three. Recall that temporal formulas map program errors onto a well-defined categorization of program properties: liveness, safety, and precedence. Similarly, program errors can be categorized by the program properties that guarantee their absence.

6.1.5. A Taxonomy based on Difficulty

The length of time it takes to find an error depends not only on the difficulty of finding the error, e.g., an intermittent bug, but also on the experience of the programmer and their level of familiarity with the application. The later the phase of development in which the bug is discovered, the more difficult it is to locate the bug (Ref. 128).

6.2. Enumeration of Ada Program Errors

The programming language determines the errors that can occur. For example, Ada prevents activating the same task object more than once. On the other hand, Ada does not guarantee freedom from deadness errors. The following errors are associated with multitasking:

1. dependence on a particular scheduling algorithm (unfairness)
2. deadness errors, e.g., cyclic deadlock
3. non-serializability
4. individual starvation
5. faulty error recovery
6. incorrect guard on accept statement (e.g., always "false")
7. missing terminate alternative in select statement
8. incorrect end-of-file handling ("eof" not passed on to other tasks)
9. calling a task in a package before the package is elaborated
10. blocking

6.3. YODA's Error Model

We propose organizing errors into a hierarchy by the queries that confirm their presence. Each query may confirm the presence of a wide class of bugs or of a single bug. Some queries can be regarded as subgoals in evaluating a candidate bug.

Formulating debugging strategies appropriate to all errors is not practical, because it is difficult to define assertions to characterize all possible errors. This difficulty arises both from the complexity of programs and from the complexity of writing assertions. One difficulty is that the type of error influences the usefulness of a particular abstraction of the data. Queries can be characterized by their level of complexity:

- Simple queries.
- Queries involving the abstraction of slices.

6.3.1. Knowledge Representation of Errors

As programmers develop expertise in debugging, they develop effective techniques for troubleshooting as well as a database of knowledge about the manifestation of specific errors. In addition, programmers incorporate into the debugging process their knowledge about the programming language, the application, the compiler or interpreter, and the reliability of the various components of the software system they are debugging.

Short of having expert programmers available as consultants at debugging sessions, *knowledge-based systems* offer a promising mechanism for incorporating into an automated debugging tool some of those factors that guide an expert programmer while debugging. Knowledge-based systems have been effectively applied in diagnosis for several specific problem domains, such as medical consulting (Ref. 129) and hardware fault diagnosis (Ref. 130, 131).

We propose a functional error model that aids in selecting the order in which candidate diagnoses should be investigated. This model should provide a description of program errors that are historically common. For each error defined, there should be a body of knowledge about the symptoms that manifest that error and appropriate debugging strategies for confirming the error.

6.3.2. Performance Issues

Diagnostic reasoning involves some tradeoffs in performance. Performance can be improved by suggesting easy tests to perform early to eliminate some bugs from consideration or to promote promising candidates. For example, to confirm an absence of cyclic deadlock, it may be easier to determine (1) that all entry queues were empty at program termination than to confirm (2) that no task in the program has a cycle of entry calls. If the first test fails, then a deadness error has occurred; the second test refines the diagnosis.

A functional error hierarchy could guide the application of diagnostic tests to ensure that the simplest and most general tests are applied first. For example, before testing for cyclic deadlock, a test should be made to confirm that a deadness error has occurred. Within each level of a hierarchy, errors should be ordered by the complexity of the test and the likelihood of the error.

6.3.3. Examples

Examples of rules used in YODA for generating and testing diagnoses are paraphrased in English as follows:

Inheritance Properties:

Program unit P1 is contained in program unit P2 if
P2 is a member of the declarative context of P1.

X is a candidate diagnosis for program unit P1 if
P1 is contained in P2 and
X is a candidate diagnosis for program unit P2.

Semantic Properties:

A program is concurrent if
at least one task is declared.

The main program is an (undeclared) task.

Program object X is visible in P if
P is in the declarative context of X.

Program object X is visible in P if
X is an entry name,
X is declared in task T,
and T is visible in P.

X is a shared variable if
X is accessed by more than one task.

Diagnostic Tests:

Tasking error is a candidate diagnosis if
the program is concurrent.

A deadness error is a candidate diagnosis if
a tasking error is a candidate diagnosis,
the program has abnormally terminated,
the error message is "system inactive,"
at least one task has not terminated, and
at least one entry queue is non-empty.

Cyclic deadlock is a candidate diagnosis if
a deadness error is a candidate diagnosis, and
there is at least one entry
that has a cyclic path of entry calls.

Non-serializability is a candidate diagnosis if
a tasking error is a candidate diagnosis,
the program terminated normally
with incorrect results, and
there is at least one shared variable.

6.4. Cyclic Deadlock

Recall that a deadness error occurs when tasks of a concurrent program reach a state from which execution cannot continue, although the tasks have not yet terminated. A cyclic deadlock occurs when a cyclic path of entry calls is executed. (See Chapter One.)

6.4.1. Implementation

Figure 6-1 shows a simple Ada program that exhibits cyclic deadlock. Output generated by this program is shown in Figure 6-2.

6.4.2. Trace Database

Figure 6-3 shows the annotated version of this program. Figure 6-4 shows a trace database generated by this annotated program.

6.4.3. Trace Analysis

To determine that a cyclic deadlock has occurred, we examine the history of task entry calls and *incomplete rendezvous*. A rendezvous is considered incomplete if it began executing, but did not complete before program termination. Figure 6-5 shows a Prolog program for diagnosing cyclic deadlock.

6.4.4. Discussion

Deadlock will occur also with the following version of task body CINDY:

```
task body CINDY is
begin
        AL.MEETING;
        accept MEETING;
end CINDY;
```

Although reciprocal entry calls are executed in the following version of task body CINDY, cyclic deadlock is avoided:

```
task body CINDY is
begin
        accept MEETING;
        AL.MEETING;
end CINDY;
```

6.5. Lost Update

A *serializability* error occurs when there is no sequential execution of the tasks in a program that could produce the same effect as the interleaved execution. This error is an example of misusing shared data.

```
with TEXT_IO; use TEXT_IO;
procedure DEADLOCK is
  task AL is
    entry MEETING;
  end AL;
  task BOB is
    entry MEETING;
  end BOB;
  task CINDY is
    entry MEETING;
  end CINDY;

  task body AL is
  begin
    put_line("AL is calling BOB.");
    BOB.MEETING;
    put_line("AL has had a rendezvous with BOB.");
    accept MEETING do
      put_line("AL is at a meeting.");
    end MEETING;
  end AL;
  task body BOB is
  begin
    accept MEETING do
      put_line("BOB is calling CINDY.");
      CINDY.MEETING;
      put_line("BOB has had a rendezvous with CINDY.");
    end MEETING;
  end BOB;
  task body CINDY is
  begin
    accept MEETING do
      put_line("CINDY is calling AL.");
      AL.MEETING;
      put_line("CINDY has had a rendezvous with AL.");
    end MEETING;
  end CINDY;
--
begin
  null;
end DEADLOCK;
```

Figure 6-1: Ada Program Exhibiting Cyclic Deadlock

```
  Begin Ada execution

AL is calling BOB.
BOB is calling CINDY.
CINDY is calling AL.
  System inactive


THE FOLLOWING TASKS ARE WAITING FOR ACCESS TO ENTRIES :
-------------------------------------------------------------------

TASK DEADLOCK.CINDY IS QUEUED ON ENTRY MEETING #1 OF TASK
                                                        DEADLOCK.AL

THE FOLLOWING TASKS ARE WAITING FOR CALL ON ENTRIES :
-------------------------------------------------------------------

( NONE )

  Execution complete
  Execution time: 13 seconds
  I-code statements executed: 50
```

Figure 6-2: Execution of Cyclic Deadlock Program

```
with text_io;
use text_io;
with trace;
procedure yoda is
package my_debugger is
  new trace(filename=>"deadlock.trace");
use my_debugger;
procedure deadlock is separate ;
begin
  debug_option:=true;
  open_db;
  deadlock;
  close_db("deadlock");
end yoda;

separate (yoda)
procedure deadlock is
  task al is
    entry meeting(who_called_id:string;
                  who_called_region:string);
  end al;
  task bob is
    entry meeting(who_called_id:string;
                  who_called_region:string);
  end bob;
  task cindy is
     entry meeting(who_called_id:string;
                   who_called_region:string);
  end cindy;

  task  body al is
  begin
    task_activated("al","[deadlock]");
    put_line("al is calling bob.");
    ecalled("al","[deadlock]" , "bob","[deadlock]","meeting");
    bob.meeting("al","[deadlock]");
    put_line("al has had a rendezvous with bob.");
    accept meeting(who_called_id:string;
        who_called_region:string) do
      rendezvous_started(who_called_id ,
        who_called_region , "al","[deadlock]","meeting");
      put_line("al is at a meeting.");
      rendezvous_completed(who_called_id ,
        who_called_region , "al","[deadlock]","meeting");
    end meeting;
    task_completed("al","[deadlock]");
  exception
     when others => abnormal_termination("al","[deadlock]");
  end al;
```

Figure 6-3: Annotation of Cyclic Deadlock Program

```
task body bob is
begin
  task_activated("bob","[deadlock]");
  accept meeting(who_called_id:string;
      who_called_region:string) do
    rendezvous_started(who_called_id ,
      who_called_region , "bob","[deadlock]","meeting");
    put_line("bob is calling cindy.");
    ecalled("bob","[deadlock]" , "cindy","[deadlock]","meeting");
    cindy.meeting("bob","[deadlock]");
    put_line("bob has had a rendezvous with cindy.");
    rendezvous_completed(who_called_id ,
      who_called_region , "bob","[deadlock]","meeting");
  end meeting;
  task_completed("bob","[deadlock]");
exception
   when others => abnormal_termination("bob","[deadlock]");
end bob;

task body cindy is
begin
  task_activated("cindy","[deadlock]");
  accept meeting(who_called_id:string,
    who_called_region:string) do
  rendezvous_started(who_called_id ,
    who_called_region , "cindy","[deadlock]","meeting");
  put_line("cindy is calling al.");
  ecalled("cindy","[deadlock]" , "al","[deadlock]","meeting");
  al.meeting("cindy","[deadlock]");
  put_line("cindy has had a rendezvous with al.");
  rendezvous_completed(who_called_id ,
    who_called_region , "cindy","[deadlock]","meeting");
  end meeting;
  task_completed("cindy","[deadlock]");
  exception
    when  others => abnormal_termination("cindy","[deadlock]");
end cindy;
begin
null ;
end deadlock;
```

Figure 6-3: Annotation of Cyclic Deadlock Program, concluded

```
then(0).

task_activated(task(cindy,[deadlock]), 1).

task_activated(task(bob,[deadlock]), 2).

task_activated(task(al,[deadlock]), 3).

entry_called(caller(al,[deadlock]),
        callee(bob,[deadlock]),meeting, 4).

rendezvous_started(caller(al,[deadlock]),
        callee(bob,[deadlock]),meeting, 5).

entry_called(caller(bob,[deadlock]),
        callee(cindy,[deadlock]),meeting, 6).

rendezvous_started(caller(bob,[deadlock]),
        callee(cindy,[deadlock]),meeting, 7).

entry_called(caller(cindy,[deadlock]),
        callee(al,[deadlock]),meeting, 8).

program_ended(deadlock,9).

now(9).
```

Figure 6-4: Trace Database of Cyclic Deadlock Program

```
/*********************************************************/
/* Usage: cyclic(T).                                     */
/* where T is a functor:        task(Name,ContextList)   */
/* T need not be instantiated.                           */
/* 1. A task that calls itself deadlocks.                */
/* 2. A cyclic path of calls deadlocks.                  */
/* ......................................................*/
/* Note: If a rendezvous never reached completion,       */
/* then it "ended" at program termination.               */
/*********************************************************/
cyclic(X)       :-
        then(Then),
        now(Now),
        transcalls(X,X,Then..Now,Now).

/*****************************************************************/
/* X made an entry call to Z during time interval T1.            */
/* If X=Z then  no rendezvous can have occurred, i.e.,           */
/* Z was still "busy."  Thus, there is no need to check for an   */
/* "absence" of rendezvous.                                      */
/*****************************************************************/

transcalls(task(X,RX),task(Z,RZ),T1,_)      :-
        entrycall(caller(X,RX),callee(Z,RZ),_,_,Etime),
        Etime during T1,!.

/*****************************************************************/
/* X made an entry call to Y during T1, and a rendezvous         */
/* between X and Y started during time interval T2,              */
/* but was never completed.                                      */
/* There is a  path of entry calls from Y to Z.                  */
/*****************************************************************/

transcalls(task(X,RX),task(Z,RZ),T1,Now)        :-
        entrycall(caller(X,RX),callee(Y,RY),E1,_,Etime),
        rendezvous(caller(X,RX),callee(Y,RY),E1,_,T2),
        T2=_..Now, T2 after Etime,
        Etime during T1,
        transcalls(task(Y,RY),task(Z,RZ),T2,Now).
```

Figure 6-5: Prolog Rules for Detecting Cyclic Deadlock

6.5.1. Implementation

The sample program in Figure 6-6 shows a simple example of a serializability error. In this example, a shared variable, X, is assigned an incorrect final value because task T1 reads X at the start of a rendezvous, although X is updated by task T2 before T1 updates X and completes its rendezvous. If tasks T1 and T2 were executed in sequence (e.g., both called from a single task) this error would not occur. In the database field, this error is also called the "lost update" problem. Figure 6-7 shows YODA's symbol table for this program.

6.5.2. Trace Database

Figure 6-8 shows a trace database that was generated by the Lost Update program.

6.5.3. Trace Analysis

We have chosen to diagnose non-serializability by applying concurrency control concepts from the database field (Ref. 132, 133, 134). If we make the assumption that whenever a task updates a shared variable, X_i, it has previously read X_i, then the following query detects a serializability error:

"Is there a cycle of tasks, $T_0..T_{n-1}$, such that
for each i
T_i accesses (either reads or writes) some shared variable, X_i,
before $T_{i+1(\text{mod } n)}$ writes on X_i."

6.6. The Stenning Protocol

In this section we present a problem in communication protocols. This problem is drawn from Stenning's simplified data-transfer protocol. Hailpern and Owicki verified the liveness properties of this protocol by using temporal formulas and history variables (Ref. 78).

The Stenning protocol requires input messages to be received by a producer process and delivered to a consumer process in the same order in which they are received. We assume that messages can be reordered, duplicated, or lost. Although the interprocess communication is unreliable, the protocol ensures that messages are delivered in the correct order. Two buffer processes synchronize communication: a message buffer and an acknowledgment buffer. A sequence number is attached to each message to maintain its ordering. The message is repeated until an acknowledgment is received.

6.6.1. Implementation

We implemented the simplified Stenning's protocol using one task for each of the four processes. Figure 6-9 shows the Ada code for the body of the producer task. Figure 6-10 shows the Ada code for the consumer task body. Figure 6-11 shows the specification of the buffer tasks. Each buffer queues messages received. The queuing operations are in a package that is hidden from the consumer and producer tasks.

```
with TEXT_IO; use TEXT_IO;
procedure MAIN is
  X : INTEGER:= 0;
  task T1 is
    entry E(A_WHILE: DURATION);
  end T1;
  task T2  is
    entry E;
  end T2;
  task C1;  task C2;
  task body T1 is
  begin
    loop
      select
        accept E (A_WHILE: DURATION) do
            if X < 1 then
              delay A_WHILE;
              X := X + 1;
              put_line("X = " & INTEGER'IMAGE(X));
            end if;
        end E;
      or
        terminate;
      end select;
    end loop;
  end T1;
  task body T2 is
  begin
    accept E do
      if X < 1 then
        X := X + 1;
        put_line("X = " & INTEGER'IMAGE(X));
      end if;
    end E;
  end T2;
  task body C1 is
  begin
    T1.E(50.0);
  end C1;
  task body C2 is
  begin
    T2.E;
  end C2;
begin
  null;
end MAIN;
-------------------------------------------------------

  Begin Ada execution

X =  1
X =  2
  Execution complete
```

Figure 6-6: Ada Program Exhibiting Lost Update

symbol(Name, Declarative_Context, Usage).

```
symbol(main,[],subprogram_name).

symbol(e,[t1,main],entry_name).

symbol(c1,[main],object_name(task_type,anonymous,[main])).

symbol(a_while,[e,loop_name1,t1,main],
        object_name(real_type_definition,duration,[])).

symbol(c2,[main],object_name(task_type,anonymous,[main])).

symbol(e,[t2,main],entry_name).

symbol(loop_name1,[t1,main],loop_name).

symbol(t1,[main],object_name(task_type,anonymous,[main])).

symbol(t2,[main],object_name(task_type,anonymous,[main])).

symbol(x,[main],
        object_name(integer_type_definition,integer,[])).
```

Figure 6-7: Symbol Table of Lost Update Program

```
then(0).
task_activated(task(c2,[main]),1).
task_activated(task(t2,[main]),2).
task_activated(task(t1,[main]),3).
task_activated(task(c1,[main]),4).
entry_called(caller(c2,[main]),callee(t2,[main]),e,5).
ready_to_terminate(task(t1,[main]),6).
entry_queue_lengthened(caller(c2,[main]),callee(t2,[main]),e,5).
entry_called(caller(c1,[main]),callee(t1,[main]),e,7).
entry_queue_lengthened(caller(c1,[main]),callee(t1,[main]),e,7).
rendezvous_started(caller(c2,[main]),callee(t2,[main]),e,8).
entry_queue_shortened(caller(c2,[main]),callee(t2,[main]),e,8).
rendezvous_started(caller(c1,[main]),callee(t1,[main]),e,9).
entry_queue_shortened(caller(c1,[main]),callee(t1,[main]),e,9).
var_read(variable(x,[main]),[e,t2,main],0,10).
var_read(variable(x,[main]),[e,loop_name1,t1,main],0,11).
var_read(variable(x,[main]),[e,t2,main],0,12).
var_updated(variable(x,[main]),[e,t2,main],1,13).
rendezvous_completed(caller(c2,[main]),callee(t2,[main]),e,14).
var_read(variable(x,[main]),[e,loop_name1,t1,main],1,15).
task_completed(task(c2,[main]),16).
task_completed(task(t2,[main]),17).
var_updated(variable(x,[main]),[e,loop_name1,t1,main],2,18).
rendezvous_completed(caller(c1,[main]),callee(t1,[main]),e,19).
task_completed(task(c1,[main]),20).
ready_to_terminate(task(t1,[main]),21).
program_ended(main,22).
now(22).
```

Figure 6-8: Trace Database of Lost Update Program

Loss of messages is simulated using a timed entry call. Although Ada permits specifying a section of code to be executed following a canceled entry call, our implementation assumes that the producer is not notified of "lost" messages. Instead, the producer repeats a message until its acknowledgment is received.

If message confirmation is not received within some arbitrary amount of time, the producer will send a duplicate message. The waiting time depends on the scheduling algorithm of the implementation.

This simplified protocol *prevents* reordering of messages because a second message is *delayed* until the first one is received and acknowledged. The *full* Stenning protocol allows messages to be reordered, i.e., a second message can be sent before the first one is acknowledged. We have not addressed the full Stenning protocol because reordering of messages is difficult to simulate in Ada, given the automatic FIFO queuing of entry calls. It can be simulated by having several consumer processes access the same input stream; however, this approach would alter the problem, because Stenning assumes that reordering may occur on *receiving* messages instead of on sending them.

6.6.2. Trace Database

Hailpern and Owicki assumed that the buffer media are "black boxes," and they applied verification to the consumer and producer processes. All communication between processes, however, is visible. To capture interprocess communication, we traced all processes.

6.6.3. Trace Analysis

Table 6-1 shows the Prolog rules that we use for extracting slices from the trace database generated by this program. In these rules we use the predicate **pairlist/3**:

pairlist(L1,L2,L3) The list [X, Y] is the *ith* member of list L3 if X is the *ith* member of list L1 and Y is the *ith* member of list L2.

For example:

```
pairlist([a,b,c], [d,e,f], [[a,d],[b,e],[c,f]]).
```

The input history of the producer and the output history of the consumer both consist of message items. The input/output history of the message buffer consists of sequence numbers and message items. The input/output history of the acknowledgment buffer consists of sequence numbers and the acknowledgment message "ack."

6.7. Taxi Service

This section presents sample trace queries for an Ada program simulating a taxi service. This program was inspired by a tutorial example suggested by Booch (Ref. 135).

```
  task body PRODUCER is
    ACK,
    MSG                    : ITEM;
    MSG_AVAILABLE,
    TIMED_OUT                : BOOLEAN;
    ACKNO,
    WAITING_FOR_ACK,
    HIGHEST_SENT    : INTEGER;
  begin
    TIMED_OUT := FALSE;
    WAITING_FOR_ACK := 1;
    HIGHEST_SENT := 0;
L1: loop
      if HIGHEST_SENT < WAITING_FOR_ACK then
        HIGHEST_SENT := HIGHEST_SENT + 1;
L2:     loop
          begin
            put_line("Enter 3-character message.");
            get(MSG);
            put_line("The message entered was " & MSG);
            exit L2;          -- valid input received
          exception
            when END_ERROR => exit L1;      -- eof
            when DATA_ERROR =>
                put_line("Invalid input.  Try Again.");
          end;
        end loop L2;
        select
          MSG_BUFFER.SEND(HIGHEST_SENT, MSG);
        or
          delay 15.5;
        end select;
        TIMED_OUT := TRUE;
      end if;
```

Figure 6-9: Stenning Protocol: Producer Task

```
    ACK_BUFFER.EXISTS(MSG_AVAILABLE);
    if MSG_AVAILABLE then
      ACK_BUFFER.RECEIVE(ACKNO, ACK);
        if ACKNO = WAITING_FOR_ACK then
          TIMED_OUT := FALSE;
          WAITING_FOR_ACK := ACKNO +1;
        end if;
    end if;
    if TIMED_OUT then
      select
        MSG_BUFFER.SEND(HIGHEST_SENT, MSG);
      or
        delay 20.5;
      end select;
    end if;
  end loop L1;
exception
  when BUFFER_EMPTY =>
      put_line("Acknowledgement buffer empty.");
  when BUFFER_OVERFLOW =>
      put_line("Message buffer overflowed.");
end PRODUCER;
```

Figure 6-9: Producer Task, concluded

```
task body CONSUMER is
  MSG                  : ITEM;
  MESSNO,
  NEXT_REQUIRED,
  LAST_REQUIRED   : INTEGER;
begin
  NEXT_REQUIRED := 1;
  loop
    MSG_BUFFER.RECEIVE(MESSNO,MSG);
    if MESSNO = NEXT_REQUIRED then
      put_line("The last message received was: ");
      put(MSG);
      NEXT_REQUIRED := NEXT_REQUIRED + 1;
    end if;
    LAST_REQUIRED := NEXT_REQUIRED -1;
    ACK_BUFFER.SEND(LAST_REQUIRED, "ack");
    if MSG="eof" then -- assumes last msg sent is "eof"
      exit;
    end if;
  end loop;

exception
  when BUFFER_EMPTY => put_line("Message buffer empty.");
  when BUFFER_OVERFLOW =>
      put_line("Acknowledgement buffer overflowed.");

end CONSUMER;
```

Figure 6-10: Stenning Protocol: Consumer Task

```
generic

   type DATA is private;

package BUFFER_PACKAGE is

   task MSG_BUFFER is
      entry SEND    (I : in NATURAL;  D: in DATA);
      entry RECEIVE (I : out NATURAL; D: out DATA);
   end MSG_BUFFER;

   task ACK_BUFFER is
      entry SEND    (I : in NATURAL;  D: in DATA);
      entry RECEIVE (I : out NATURAL; D: out DATA);
      entry EXISTS  (AVAIL: out BOOLEAN);
   end ACK_BUFFER;

   BUFFER_OVERFLOW,
   BUFFER_EMPTY      : exception;

end BUFFER_PACKAGE;
```

Figure 6-11: Stenning Protocol: Buffer Tasks

Table 6-1: Stenning Protocol: Slices of the Trace Database

English Description	Prolog Description
Input history of the producer	`bagof(V, var_write(variable(msg,[producer,stenning]),_, V,_),X).`
Output history of the consumer	`bagof(V, var_read(variable(msg,[consumer,stenning]),_, V,_),Y).`
Input history of the message buffer	`bagof(V,entry_parm_set(_, callee(msg_buffer,[]),send,i,_,V,_),I), bagof(V,entry_parm_set(_, callee(msg_buffer,[]),send,d,_,V,_),D).`
Output history of the message buffer	`bagof(V,entry_parm_set(_, callee(msg_buffer,[]),receive,i,_,V,_),I), bagof(V,entry_parm_set(_, callee(msg_buffer,[]),receive,d,_,V,_),D), pairlist(I,D,Beta).`
Input history of the acknowledgment buffer	`bagof(V,entry_parm_set(_, callee(ack_buffer,[]),send,i,_,V,_),I), bagof(V,entry_parm_set(_, callee(ack_buffer,[]),send,a,_,V,_),A), pairlist(I,A,Delta).`
Output history of the acknowledgment buffer	`bagof(V,entry_parm_set(_, callee(ack_buffer,[]),receive,i,_,V,_),I), bagof(V,entry_parm_set(_, callee(ack_buffer,[]),receive,A,_,V,_),A), pairlist(I,A,Gamma).`

6.7.1. Implementation

Appendix E shows the source listings for the taxi service program. Table F-1 in Appendix F shows the symbol tables generated by YODA for this program.

6.7.2. Trace Queries

The following queries are based on the Ada taxi service program:

1. **Simple query.** Get the first task that executed calls to entry CONNECT in task SWITCH_BOARD of MAIN.

```
?- entry_called(Caller,
     callee(switch_board,[main]),
     connect,_,_).
```

2. **Query on ordering of events.** Did a rendezvous complete between a CUSTOMER and the SWITCH_BOARD before any DISPATCHER executed an entry call to a TAXI?

```
?- rendezvous_completed(caller(customer,_),
       callee(switch_board,_),_,Rtime),
     entry_called(caller(dispatcher,_),
       callee(taxi,_),_,Etime),
     Rtime > Etime.
```

7. SUMMARY AND EVALUATION

7.1. Conclusions

This dissertation has presented the design and implementation of a knowledge-based debugging system for use with concurrent programs written in the Ada programming language. Trace queries have been introduced to provide a high-level abstraction of program behavior based on an event stream.

We have extended Prolog to form queries about the dynamic behavior of Ada tasking programs. It has been argued that Prolog provides a more appropriate formalism for trace information than the relational database model. An interval-logic system, $\mathbb{C}$, was introduced to express temporal relations over intervals. Both system $\mathbb{C}$ and path expressions were implemented as Prolog predicates. We proposed DCGs as an approach to implementing path expressions.

High-level tools are needed for debugging high-level, concurrent software. The difficulties of debugging in a concurrent environment have been described. A retrospective approach to debugging can obviate some of the problems associated with dynamic debugging of concurrent software. Often the only way to find program errors that disappear when a debugger is turned on is to use a history log or simulation.

The flexibility and effectiveness of trace analysis has been demonstrated with a stand-alone, prototype debugger for Ada, called YODA, that captures trace data by embedding the source with diagnostic output statements to aid in debugging. Accumulation of trace data into an historical database can provide a resource for both monitoring and replaying program execution.

Evaluating candidate diagnoses was described as the process of testing assertions about integrity constraints and consistency constraints on a database. YODA has been used successfully for retrospective analysis of Ada programs to expose run-time errors associated with concurrency.

A taxonomy of program errors has been presented and suggested as a starting point for applying expert-system technology to debugging. Generating candidate diagnoses was presented as the process of interrogating a knowledge base of error descriptions and debugging strategies.

Dynamic analysis may overlook program errors because it is not possible to force all synchronization or execution paths during execution. When bugs do make themselves known, YODA supports efficient debugging by providing flexible and controlled access to useful information about a program's structure and execution history.

7.2. Open Problems

YODA can be integrated with a specific Ada programming support environment. Falis (Ref. 136) describes some of the requirements for an integrated Ada debugger, such as providing access to the run-time task supervisor to query task priority and task status. One possibility in an integrated environment would be to rely on the trace database for triggering conditional breakpoints, e.g., to set a breakpoint when a cyclic deadlock has occurred.

This work opens many other avenues for research:

1. The need to capture trace information without perturbing program behavior opens areas for future research in considering the options available in various computer architectures and options for implementing Ada tasking on various operating systems. An interesting research area is how to design mechanisms that minimize interference with timing-dependent behavior.
2. The approach we have described rests on the use of Prolog for expressing queries; however, natural language question-answering systems have been designed in Prolog (Ref. 24).
3. A trace database can be a resource for animation of program execution. The multi-level views provided by a trace database could support a hierarchical view of program visualization, allowing the user to view program behavior at an abstract level and to "zoom in" on details of program execution and program structure. By filtering information captured during execution, the trace database could support the visual display of abstractions of the program behavior, such as the queuing of entry calls to Ada tasks.
4. Trace queries can be extended to incorporate *fuzziness* into queries. For example, in the question: "Was task T1 terminated *when* task T2 called an entry in E1?", "when" can mean an approximate time (within a certain interval.) On the other hand, *ambiguity* in queries requires an understanding of the user. For example, "when" can mean immediately before, immediately after, sometime before, or sometime after (Ref. 90).

REFERENCES

[1] Brown, A.R. and Sampson, W.A.
Program Debugging: The Prevention and Cure of Program Errors.
Macdonald, London, 1973.

[2] Neuman, P.G.
Letter from the Editor.
Software Engineering Notes 8(3):2-6, July, 1983.

[3] Neuman, P.G.
Letter from the Editor: Risks to the Public.
Software Engineering Notes 10(2):4-11, April, 1985.

[4] Model, M.L.
Monitoring System Behavior In a Complex Computational Environment.
PhD thesis, Stanford University, January, 1979.
(Published as Technical Report CSL-79-1, Xerox PARC).

[5] Johnson, Mark Scott (editor).
Proceedings of the Software Engineering Symposium on High-Level Debugging.
ACM SIGSOFT/SIGPLAN, Pacific Grove, California, 1983.
(Published as SIGPLAN Notices 18(8), August, 1983).

[6] Zellwegger, P.T.
Interactive Source-Level Debugging of Optimized Programs.
PhD thesis, UC Berkeley, 1984.
(Published as Technical Report No. CSL-84-5, Xerox PARC).

[7] Andrews, G.R. and Schneider, F.B.
Concepts and Notations for Concurrent Programming.
Computing Surveys 15(1):3-44, March, 1983.

[8] MacLaren, L.
Evolving Toward Ada in Real Time Systems.
In *Proceedings of the Symposium on the Ada Programming Language*, pages 146-155. ACM-SIGPLAN , Boston, MA, December 9-11, 1980.
(Published as SIGPLAN Notices, 15(11), November, 1980).

[9] Garcia-Molina, H., Germano, F. Jr., and Kohler, W.H.
Debugging a Distributed Computing System.
IEEE Trans. on Software Engineering SE-10(2):210-219, 1984.

[10] Hoare, C.A.R.
Communicating Sequential Processes.
Communications of the ACM 21(8):666-677, August, 1978.

[11] ANSI/MIL-STD 1815A.
Reference Manual for the Ada Programming Language
U.S. Department of Defense, 1983.

[12] Barnes, J.G.P.
Programming in Ada.
Addison-Wesley Publishers Limited, London, 1982.

[13] Booch, G.
Software Engineering with Ada.
Benjamin-Cummings Publishing Company, 1983.

[14] Goodenough, J.B.
The Ada Compiler Validation Capability.
In *Proceedings of the Symposium on the Ada Programming Language*, pages 1-8. ACM-SIGPLAN , Boston, MA, December 9-11, 1980.
(Published as SIGPLAN Notices, 15(11), November, 1980).

[15] Verdix Corporation.
Advertisement.
IEEE Software 2(4):81, July, 1985.

[16] Taylor, R.N. & Osterweil, L.J.
Anomaly Detection in Concurrent Software by Static Data Flow Analysis.
IEEE Trans. on Software Engineering SE-6(3):265-277, May, 1980.

[17] Fairley, R.E.
Debugging and Testing Support Environments.
SIGPLAN Notices 8:16-25, November, 1980.

[18] Gait, J.
A Debugger for Concurrent Programs.
Software -- Practice and Experience 15(6):539-554, June, 1985.

[19] Clocksin, W.F. & Mellish, C.S.
Programming in Prolog.
Springer-Verlag, 1981.

[20] Pereira, F. (Ed.).
C-Prolog User's Manual
Version 1.5 edition, Edinburgh Computer Aided Architectural Design, Department of Architecture, University of Edinburgh, 1984.

[21] Colmerauer, A.
Metamorphosis Grammars.
In Bloc, L. (editor), *Lecture Notes in Computer Science*. Volume 63: *Natural Language Communication with Computers*, pages 133-189. Springer-Verlag, 1978.

[22] Gallaire, H.
Impacts of Logic on Databases.
In *Proceedings of the 7th VLDB Conference*, pages 248-259. VLDB , Cannes, France, September, 1981.

[23] Kowalski, R.
Logic for Problem Solving.
North Holland, New York, 1979.

[24] Warren, D.H.D.
Efficient Processing of Interactive Relational Database Queries Expressed in Logic.
In *Proceedings of the 7th VLDB Conference*, pages 272-282. VLDB , Cannes, France, September, 1981.

[25] Kowalski, R.
Logic for Data Description.
In Gallaire, H. and Minker, J. (editor), *Logic and Databases*, pages 77-107. Plenum Press, New York, 1978.
(Symposium on Logic and Data Bases, Centre d'Etudes et de Recherches de Toulouse, 1977).

[26] Futo, I., Darvas, F., and Szeredi, P.
The Application of PROLOG to the Development of QA and DBM Systems.
In Gallaire, H. and Minker, J. (editor), *Logic and Databases*, pages 346-376. Plenum Press, New York, 1978.
(Symposium on Logic and Data Bases, Centre d'Etudes et de Recherches de Toulouse, 1977).

[27] Dahl, V.
On Database Systems Development Through Logic.
ACM Trans. Database Syst. 7(1):102-123, March, 1982.

[28] LeDoux, C.H. and Parker, D.S. Jr.
Saving Traces for Ada Debugging.
In Barnes, J.G.P. and Fisher, G.A. Jr. (editor), *Proc. Ada Int'nl. Conference 1985: Ada in Use*, pages 97-108. Cambridge University Press, Paris, France, May 14-16, 1985.
(Published as *Ada Letters*, Volume V, Number 2, September, October 1985).

[29] Adrion, W.R., Branstad, M.A., and Cherniavsky, J.C.
Validation, Verification, and Testing of Computer Software.
Computing Surveys 14(2):159-192, June, 1982.

[30] Gauss, E.J.
The "Wolf Fence" Algorithm for Debugging.
Communications of the ACM 25(11):780, November, 1982.

[31] Taylor, R.N.
A General Purpose Algorithm for Analyzing Concurrent Programs.
Communications of the ACM 26(5):362-376, May, 1983.

[32] Yemini, S. and Berry, D.M.
A Modular Verifiable Exception-Handling Mechanism.
ACM Trans. on Prog. Lang. and Systems 7(2):214-243, April, 1985.

[33] *OS PL/I Optimizing Compiler: Programmer's Guide*
IBM, 1974.

[34] Johnson, M.S.
A Software Debugging Glossary.
SIGPLAN Notices 17(2):53-70, February, 1982.

[35] Medina-Mora, R. and Feiler, P.H.
An Incremental Programming Environment.
IEEE Trans. on Software Engineering SE-7(5):472-482, September, 1981.

[36] *Reference Manual for the Ada Programming Language*
U.S. Department of Defense, 1980.
Proposed Standard Document.

[37] Harter, P.K. Jr., Heimbigner, D.M., and King, R.
IDD: An Interactive Distributed Debugger.
In *Proc. Fifth Int'nl. Conf. on Distributed Computing Systems*, pages 498-506. IEEE Computer Society, Denver, CO, May 13-17, 1985.

[38] Satterthwaite, E.H. Jr.
Source Language Debugging Tools.
PhD thesis, Computer Science Department, Stanford University, May, 1975.

[39] Smith, E.T.
Debugging Techniques for Communicating, Loosely-Coupled Processes.
PhD thesis, University of Rochester, December, 1981.
(Published as Technical Report Number 100, University of Rochester, Department of Computer Science).

[40] Reiser, J.F.
BAIL: A Debugger for SAIL.
Technical Report STAN-CS-75-523, Stanford University Computer Science Department, October, 1975.

[41] Weber, J.C.
Interactive Debugging of Concurrent Programs.
In *Proceedings of the Software Engineering Symposium on High-Level Debugging*, pages 112-113. ACM SIGSOFT/SIGPLAN, Pacific Grove, California, March, 1983.
(Published as SIGPLAN Notices 18(8), August, 1983).

[42] Gramlich, W.C.
Checkpoint Debugging.
In *Proceedings of the Software Engineering Symposium on High-Level Debugging*. ACM SIGSOFT/SIGPLAN, March, 1983.
Appeared as a position paper in Preliminary Proceedings (not in press).

[43] *Ada Source Code Debugger Reference Manual*
Rolm Corporation, 1983.

[44] Standish, T.A. and Taylor, R.N.
Arcturus: a Prototype Advanced Ada Programming Language.
In *Proceedings of the Software Engineering Symposium on Practical Software Development Environments*, pages 57-64. ACM SIGSOFT/SIGPLAN, Pittsburgh, PA, April 23-25, 1984.
(Published as SIGPLAN Notices 19(5), May, 1984).

[45] Brindle, A.F., Taylor, R.N., and Martin, D.F.
A Debugger for Ada Tasking.
Aerospace Report ATR-85(8033)-1, The Aerospace Corporation, El Segundo, CA, September, 1985.

[46] Luckham, D.C., Larsen, H.J., Stevenson, D.R., and von Henke, F.W.
ADAM - An Ada based Language for Multi-Processing.
Technical Report STAN-CS-81-867, Stanford University Department of Computer Science, July, 1981.

[47] German, S.M., Helmbold, D.P., and Luckham, D.C.
Monitoring for Deadlocks in Ada Tasking.
In *Proceedings of the AdaTEC Conference on Ada*, pages 10-25. ACM SIGPLAN , October, 1982.

[48] Helmbold, D. and Luckham, D.
Debugging Ada Tasking Programs.
In *Proc. of the 1984 Conf. on Ada Applications and Environments*, pages 96-105. IEEE Computer Society , October, 1984.

[49] German, S.M.
Monitoring for Deadlock and Blocking in Ada Tasking.
IEEE Trans. on Software Engineering SE-10(6):764-777, 1984.

[50] Sheil, B.A.
The Psychological Study of Programming.
Computing Surveys 13(1):101-120, March, 1981.

[51] Shneiderman, B.
Exploratory Experiments in Programmer Behavior.
Int. J. Computer Inf. Sci. 5:123-143, 1976.

[52] Rich, C. and Waters, R.C.
Abstraction, Inspection and Debugging in Programming.
Technical Report 634, M.I.T. A.I. Laboratory, June, 1981.

[53] Soloway, E., Ehrlich, K., Bonar, J., and Greenspan, J.
What Do Novices Know About Programming?.
Research Report 218, Yale University Department of Computer Science, January, 1982.

[54] Ehrlich, K. and Soloway, E.
An Empirical Investigation of the Tacit Plan Knowledge in Programming.
Research Report 236, Yale University Department of Computer Science, April, 1982.

[55] Snodgrass, R.
Monitoring Distributed Systems: A Relational Approach.
PhD thesis, Carnegie-Mellon University, December, 1982.
(Published as Technical Report CMU-CS-82-154, Department of Computer Science, Carnegie-Mellon University).

[56] Powell, M.L. and Linton, M.A.
A Database Model of Debugging.
In *Proceedings of the Software Engineering Symposium on High-Level Debugging*, pages 67-70. ACM SIGSOFT/SIGPLAN , Pacific Grove, California, March, 1983.
(Published as SIGPLAN Notices 18(8), August, 1983).

[57] Linton, M.A.
Implementing Relational Views of Programs.
In *Proceedings of the Software Engineering Symposium on Practical Software Development Environments*, pages 132-140. ACM SIGSOFT/SIGPLAN , Pittsburgh, PA, April 23-25, 1984.
(Published as SIGPLAN Notices 19(5), May, 1984).

[58] Shapiro, E.Y.
Algorithmic Program Debugging.
PhD thesis, Yale University, May, 1982.
(Published as Technical Report Number 237, Yale University Department of Computer Science.

[59] Shapiro, D.G.
Sniffer: a System that Understands Bugs.
Master's thesis, M.I.T., June, 1981.
(Published as Technical Report Number 638, M.I.T. A.I. Lab).

[60] Sedlmeyer, R.L., Thompson, W.B., and Johnson, P.E.
Knowledge-Based Fault Localization in Debugging.
In *Proceedings of the Software Engineering Symposium on High-Level Debugging*, pages 25-31. ACM SIGSOFT/SIGPLAN, Pacific Grove, California, March, 1983.
(Published as SIGPLAN Notices 18(8), August, 1983).

[61] Hanson, S.J. and Rosinski, R.R.
Programmer Perceptions of Productivity and Programming Tools.
Communications of the ACM 28(2):180-189, February, 1985.

[62] Gould, J.D.
Some Psychological Evidence on How People Debug Computer Programs.
Int. J. Man-Machine Studies 7:151-182, 1975.

[63] Ostrand, T.J. and Weyuker, E.
Collecting and Categorizing Software Error Data in an Industrial Environment.
Technical Report 47, New York University Courant Institute, 1982.

[64] Garcia, M.E. and Berman, W.J.
An Approach to Concurrent Systems Debugging.
In *Proc. Fifth Int'nl. Conf. on Distributed Computing Systems*, pages 507-514. IEEE Computer Society, Denver, CO, May 13-17, 1985.

[65] Sajkowski, M.
Protocol Verification Techniques: Status Quo and Perspectives.
In Yemini, Y., Strom, R., and Yemini, S. (editor), *Proc. Fourth Int'nl Workshop on Protocol, Specification, Testing, and Verification*, pages 697-720. IFIP, Elsevier Science Publishers B.V. (North-Holland Publishing Company), Skytop Lodge, Pennsylvania, 1985.

[66] Pnueli, A.
The Temporal Semantics of Concurrent Programs.
In Goos, G. and Hartmanis, J. (editor), *Proc. International Symposium on Semantics of Concurrent Computation*, pages 1-20. Springer-Verlag, Evian, France, July 2-4, 1979.
(Published as *Lecture Notes in Computer Science*, New York).

[67] Lamport, L.
"Sometime" is Sometimes "Not Never": On the Temporal Logic of Programs.
In *Conf. Record of the 7th Annual ACM Symp. on Principles of Programming Languages*, pages 174-185. ACM, Las Vegas, Nevada, January, 1980.

[68] Chen, B. and Yeh, R.T.
Formal Specification and Verification of Distributed Systems.
IEEE Trans. on Software Engineering SE-9(6):710-721, November, 1983.

[69] Pnueli, A.
The Temporal Logic of Programs.
In *Proc. of the 18th Annual Symposium on Foundations of Computer Science*, pages 46-57. IEEE Computer Society, Providence, Rhode Island, October 31 - November 2, 1977.

[70] Schwartz, R.L. and Melliar-Smith, P.M.
From State Machines to Temporal Logic: Specification Methods for Protocol Standards.
In Sunshine, C. (editor), *Proc. Second Int'nl Workshop on Protocol, Specification, Testing, and Verification*, pages 3-19. IFIP, North-Holland Publishing Company, Idyllwild, California, May 17-20, 1982.

[71] Schwartz, R.L., Melliar-Smith, P.M., and Vogt, F.H.
An Interval-Based Temporal Logic.
In Clarke, E. and Kozen, D. (editor), *Proc. Logics of Programs*, pages 443-457. Springer-Verlag, 1983.
(Published as *Lecture Notes in Computer Science*, New York).

[72] Moszkowski, B. and Manna, Z.
Reasoning in Interval Temporal Logic.
Research Report STAN-CS-83-969, Stanford University, Department of Computer Science, July, 1983.

[73] Shasha, D.E., Pnueli, A., and Ewald, W.
Temporal Verification of Carrier-Sense Local Area Network Protocols.
In *Conf. Record of the 11th Annual ACM Symposium on Principles of Programming Languages*, pages 54-65. ACM, Salt Lake City, Utah, Jan. 15-18, 1984.

[74] Campbell, R.H. and Habermann, A.N.
The Specification of Process Synchronization by Path Expressions.
In Goos, G. and Hartmanis, J. (editor), *Lecture Notes in Computer Science*, pages 89-102. Springer Verlag, New York, 1974.

[75] Habermann, A.N.
Path Expressions.
Technical Report, Carnegie-Mellon University, June, 1975.

[76] Peterson, J.L. .
Petri Net Theory and the Modeling of Systems.
Prentice-Hall, Inc., Englewood Cliffs, N.J., 1981.

[77] Burstall, R.M.
Formal Description of Program Structure and Semantics of First Order Logic.
In Meltzer, B. and Michie, D. (editor), *Machine Intelligence*, pages 79-98. Edinburgh, 1970.

[78] Hailpern, B.T. and Owicki, S.
Modular Verification of Computer Communication Protocols.
Research Report RC 8726 (#38174), IBM Thomas J. Watson Research Center, March, 1981.

[79] Wolper, P.
Specification and Synthesis of Communicating Processes Using an Extended Temporal Logic (Preliminary Version).
In *Conf. Record of the 9th Annual ACM Symposium on Principles of Programming Languages*, pages 20-33. ACM, Albuquerque, New Mexico, January 25-27, 1982.

[80] Manna, Z. and Wolper, P.
Synthesis of Communicating Processes from Temporal Logic Specifications.
ACM Trans. on Programming Languages and Systems 6(1):68-93, January, 1984.

[81] Karp, R.A.
Proving Failure-Free Properties of Concurrent Systems Using Temporal Logic.
ACM Trans. on Programming Languages and Systems 6(2):239-253, 1984.

[82] Schwabe, D. and Cavalli, A.R.
Temporal Logic Specification of a Virtual Ring LAN Access Protocol.
In Yemini, Y., Strom, R., and Yemini, S. (editor), *Proc. Fourth Int'nl Workshop on Protocol, Specification, Testing, and Verification*, pages 79-91. IFIP, Elsevier Science Publishers B.V. (North-Holland Publishing Company), Skytop Lodge, Pennsylvania, 1985.

[83] Emerson, E.A. and Sistla, A.P.
Deciding Branching Time Logic: A Triple Exponential Decision Procedure for CTL*.
In Clarke, E. and Kozen, D. (editor), *Proc. Logics of Programs*, pages 176-191. Springer-Verlag, 1983.
(Published as *Lecture Notes in Computer Science*, New York).

[84] Emerson, E.A. and Lei, C.
Modalities for Model Checking: Branching Time Strikes Back.
In *Conf. Record of the 12th Annual ACM Symp. on Principles of Programming Languages*, pages 84-96. ACM, January, 1985.

[85] McLean, J.
A Complete System of Temporal Logic for Specification Schemata.
In Clarke, E. and Kozen, D. (editor), *Proc. Logics of Programs*, pages 360-370. Springer-Verlag, 1983.
(Published as *Lecture Notes in Computer Science*, New York).

[86] McDermott, D.
A Temporal Logic for Reasoning About Processes and Plans.
Cognitive Science 6:101-155, 1982.

[87] Allen, J.F.
Maintaining Knowledge About Temporal Intervals.
Communications of the ACM 26(11):832-843, November, 1983.

[88] Allen, J.F. and Hayes, P.J.
A Common-Sense Theory of Time.
In *Proc. Ninth Int'nl Joint Conf. on A.I.*, pages 528-531. IJCAI, Los Angeles, California, 18-23 August, 1985.

[89] Clifford, J. and Warren, D.S.
Formal Semantics for Time in Databases.
ACM Trans. on Database Systems 8(2):214-254, June, 1983.

[90] Yip, K.M.
Tense, Aspect and the Cognitive Representation of Time.
In *Proc. Ninth Int'nl Joint Conf. on A.I.*, pages 806-814. IJCAI, Los Angeles, California, 18-23 August, 1985.

[91] Turner, R.
Temporal Logic in Artificial Intelligence.
Logics for Artificial Intelligence.
Ellis Horwood Limited, University of Essex, 1985, Chapter 6.

[92] Lauer, P.E., and Campbell, R.H.
Formal Semantics of a Class of High-Level Primitives for Coordinating Concurrent Processes.
Acta Informatica 5:297-332, 1975.

[93] Gabbay, D., Pnueli, A., Shelah, S., and Stavi, J.
On the Temporal Analysis of Fairness.
In *Conf. Record of the 7th Annual ACM Symposium on Principles of Programming Languages*, pages 163-173. ACM, Las Vegas, Nevada, January 28-30, 1980.

[94] Vogt, F.H.
Event-Based Temporal Logic Specifications of Services and Protocols.
In Sunshine, C. (editor), *Proc. Second Int'nl Workshop on Protocol, Specification, Testing, and Verification*, pages 63-73. IFIP, North-Holland Publishing Company, Idyllwild, California, May 17-20, 1982.

[95] Rescher, N. and Urquhart, A.
Temporal Logic.
Springer-Verlag, New York, 1971.

[96] Kamp, H.
On Tense Logic and the Theory of Order.
PhD thesis, University of California, Los Angeles, 1968.

[97] Hughes, G.E. and Cresswell, M.J.
An Introduction to Modal Logic.
Methuen and Co. LTD, London, 1968.

[98] Nguyen, V., Gries, D, and Owicki, S.
A Model and Temporal Proof System for Networks of Processes.
In *Conf. Record of the 12th Annual ACM Symposium on Principles of Programming Languages*, pages 121-130. ACM, Las Vegas, Nevada, January, 1985.

[99] Ramamritham, K. and Keller, R.M.
Specification of Synchronizing Processes.
IEEE Trans. on Software Engineering SE-9(6):722-733, November, 1983.

[100] Allen, J.F.
A General Model of Action and Time.
Technical Report TR 97, Department of Computer Science, University of Rochester, November, 1981.

[101] Holt, A.W. and Commoner.
Events and Conditions.
In *Record of the Project MAC Conference on Concurrent Systems and Parallel Computation*, pages 1-52. ACM, Woods Hole, Massachusetts, June 2-5, 1970.

[102] Andler, S.
Predicate Path Expressions.
In *Conference Record of the Sixth Annual Symposium on Principles of Programming Languages*, pages 226-236. ACM, January, 1979.

[103] Plaisted, D. A.
A Low Level Language for Obtaining Decision Procedures for Classes of Temporal Logics.
In Clarke, E. and Kozen, D. (editor), *Proc. Logics of Programs*, pages 403-420. Springer-Verlag, 1983.
(Published as *Lecture Notes in Computer Science*, New York).

[104] Kroeger, F.
LAR: A Logic of Algorithm Reasoning.
Acta Informatica 8:243-266, 1977.

[105] Lantz, K.A., Gradischnig, K.D., Feldman, J.A., and Rashid, R.F.
Rochester's Intelligent Gateway.
Computer 15(10):54-68, October, 1982.

[106] Bruegge, B. and Hibbard, P.
Generalized Path Expressions: A High Level Debugging Mechanism.
In *Proceedings of the Software Engineering Symposium on High-Level Debugging*. ACM SIGSOFT/SIGPLAN, March, 1983.
Appeared as a position paper in Preliminary Proceedings (not in press).

[107] Bruegge, B.
Debugging Ada.
Technical Report CMU-CS-85-127, Department of Computer Science, Carnegie-Mellon University, May, 1985.

[108] Lamport, L.
Time, Clocks, and the Ordering of Events in a Distributed System.
Communications of the ACM 21(7):558-565, July, 1978.

[109] Curtis, R. and Wittie, L.
BugNet: A Distributed Applications Debugging System.
In *Proceedings of the Software Engineering Symposium on High-Level Debugging*. ACM SIGSOFT/SIGPLAN, March, 1983.
Appeared as a position paper in Preliminary Proceedings (not in press).

[110] Shankar, A.U. and Lam, S.S.
Specification and Verification of Time-Dependent Communication Protocols.
In Yemini, Y., Strom, R., and Yemini, S. (editor), *Proc. Fourth Int'nl Workshop on Protocol, Specification, Testing, and Verification*, pages 215-226. IFIP, Elsevier Science Publishers B.V. (North-Holland Publishing Company), Skytop Lodge, Pennsylvania, 1985.

[111] Wittie, L. and Curtis, R.
Time Management for Debugging Distributed Systems.
In *Proc. Fifth Int'nl. Conf. on Distributed Computing Systems*, pages 549-550. IEEE Computer Society, Denver, CO, May 13-17, 1985.

[112] DeMillo, R.A., Lipton, R.J., and Perlis, A.J.
Social Processes and Proofs of Theorems and Programs.
Communications of the ACM 22(5):271-280, May, 1979.

[113] Gerhart, S.L. and Yelowitz, L.
Observations of Fallibility in Applications of Modern Programming Methodologies.
IEEE Trans. on Software Engineering SE-2(3):195-207, September, 1976.

[114] Bates, P.C., Wileden, J.C., and Lesser, V.R.
A Debugging Tool for Distributed Systems.
Technical Report 82-34, Computer & Information Science Department, University of Massachusetts at Amherst, December, 1982.

[115] Bates, P.C. and Wileden, J.C.
An Approach to High-Level Debugging of Distributed Systems.
Technical Report 82-35, Computer & Information Science Department, University of Massachusetts at Amherst, December, 1982.

[116] Bates, P.C. and Wileden, J.C.
An Approach to High-Level Debugging of Distributed Systems.
In *Proceedings of the Software Engineering Symposium on High-Level Debugging*, pages 34-44. ACM SIGSOFT/SIGPLAN, Pacific Grove, California, March, 1983.
(Published as SIGPLAN Notices 18(8), August, 1983).

[117] Snodgrass, R.
Monitoring in a Software Development Environment: A Relational Approach.
In *Proceedings of the Software Engineering Symposium on Practical Software Development Environments*, pages 124-131. ACM SIGSOFT/SIGPLAN , Pittsburgh, PA, April 23-25, 1984.
(Published as SIGPLAN Notices 19(5), May, 1984).

[118] Hornsby, C. and Leung, C.H.C.
The Design and Implementation of a Flexible Retrieval Language for a Prolog Database System.
ACM SIGPLAN Notices 20(9):43-51, September, 1985.

[119] Kowalski, R.
Logic and Semantic Networks.
Communications of the ACM 22(3):184-192, March, 1979.

[120] Pereira, C.N. and Warren, D.H.D.
Definite Clause Grammars for Language Analysis -- A Survey of the Formalism and a Comparison with Augmented Transition Networks.
Artificial Intelligence 13:231-278, 1980.

[121] Kean, D.
The Computer and the Countess.
Datamation :60-63, May, 1973.

[122] DeLauer, R.
Interim DoD Policy on Computer Programming Languages.
Memorandum. Washington, D.C., 1983.

[123] *Higher Order Languages*
Department of Defense, Washington, D.C., 1983.
Draft DoD Directive 3405.1.

[124] Charles, P. and Fisher, G.
A LALR(1) Grammar for '82 Ada.
Ada Letters II(2):34-45, September, October, 1982.

[125] Clancey, W.J. .
Classification Problem Solving.
Research Report STAN-CS-84-1018, Stanford University, Department of Computer Science, July, 1984.

[126] Eggert, P.H.
Detecting Software Errors Before Execution.
PhD thesis, Univ. of California, Los Angeles, April, 1981.
(Published as Technical Report CSD-810402, Univ. of California, Los Angeles, Computer Science Department).

[127] Binder, R.
Application Debugging: An MVS Abend Handbook for COBOL, Assembly, PL/1, and FORTRAN Programmers.
Prentice-Hall, Englewood Cliffs, New Jersey, 1985.

[128] Trachtenberg, M.
Order and Difficulty of Debugging.
IEEE Trans. on Software Engineering SE-9(6):746-747, November, 1983.

[129] Shortliffe, E.H.
Computer-based medical consultations: MYCIN.
American Elsevier, New York, 1976.

[130] Sussman, G.J. and Stallman, R.
Heuristic Techniques in Computer Aided Circuit Analysis.
Technical Report 328, MIT AI Lab, 1975.

[131] Bennett, J.S. and Hollander, R.
DART: An Expert System for Computer Fault Diagnosis.
IJCAI :843-845, 1981.

[132] Ullman, J.D.
Principles of Database Systems.
Computer Science Press, Potomac, Maryland, 1980.

[133] Bernstein, P.A. & Goodman, N.
Concurrency Control in Distributed Database Systems.
ACM Computing Surveys 13(2):185-222, June, 1981.

[134] Date, C.J.
The Systems Programming Series. Volume II: *An Introduction to Database Systems.*
Addison-Wesley Publishing Company, Reading, Massachusetts and Menlo Park, California, 1983.

[135] Booch, G.
Dear Ada.
Ada Letters 2(3):10..13, November, December, 1982.

[136] Falis, E.
Design and Implementation in Ada of a Runtime Task Supervisor.
In *Proceedings of the AdaTEC Conference on Ada*, pages 1-9. ACM-SIGPLAN, October, 1982.

APPENDIX A
INTRODUCTION TO PROLOG

Prolog consists of two syntactical forms, called *clauses*:

facts	Declarations about objects and their relationships, e.g., "The value of a is 5."
rules	Definitions about objects and their relationships, e.g., "For any X, X is positive if X is a number and the value of X is greater than zero."

Facts and rules constitute a Prolog database. A Prolog program is a database. Rules and facts may also be interpreted as *queries*, i.e., questions about objects and their relationships, e.g., "Does there exist an X such that X is positive?"

1. Basic Character Set

The basic character set of Prolog is divided into six categories:

1. uppercase letters
2. lowercase letters
3. digits
4. symbols, consisting of the following special characters:

```
+ - * / \ ^ < > = ' ~ : . ? @ # $ &
```

5. other special characters:

```
! " % ( ) | } ] { [ ` ; ,
```

6. separators, including the space character, tabulations, and the end of the line.

2. Comments

A comment can appear anywhere in a Prolog clause. A comment starts with a `'/*'` and ends with an `'*/'`.

3. Primitives

The primitive objects of Prolog are

1. *variables*	Names denoted by words beginning with an uppercase letter or an underline. In particular, the variable `'_'` is termed an *anonymous variable*.
2. *atoms*	Names denoted by a sequence of characters. The name must be enclosed in single quotes if the first character in the sequence is a digit, an uppercase letter, or a special character in the category "other."
3. *numbers*	Either integers or floating point numbers.

Atoms and numbers are, collectively, called *constants*.

4. Terms

The basic components of Prolog clauses are *terms*. A term can be either a primitive object or a *compound term*. In Prolog, the basic data structure is a compound term, which is also called a *structure*. A compound term is composed of *functors* and *arguments*. An argument is a term. A functor is characterized by a *name*, which is an atom, and an *arity* (i.e., the number of arguments).

The name of the functor is written either as a *predicate*, e.g., `'value(a,5)'` or as an *operator*, e.g., `'a is 5'`.

5. Operators

An operator is either unary or binary. It is defined by its name, *position*, *precedence* class, and *associativity*. The position is infix, prefix, or postfix. The precedence class is a number. The associativity can be "left" or "right."

Precedence and associativity determine the order in which adjacent operators are to be grouped in a compound term. The precedence class of an operator is a number ranging from one to some large number, e.g., 1200. (The range is implementation-dependent.) The *lower* the precedence class of an operator, the *higher* is its precedence. Terms enclosed in parentheses are given a precedence class of zero (the highest possible precedence).

If a term contains two or more adjacent operators having the same precedence, then the grouping is determined by the associativity of the operators. For example, if `'+'` is left-associative (defined as **yfx** or **yf**) then the expression:

```
a+b+c
```

is evaluated as though it were the following expression:

```
(a+b)+c
```

If `'+'` is right-associative (defined as **xfy** or **fy**), then the expression `a+b+c` is evaluated as though it were

```
a+(b+c)
```

Associativity of operators can be specified as invalid. For example, if the associativity of the operator **not** is invalid (defined as **xfx** or **xf**), then the expression:

```
not not X
```

is illegal, although

```
not (not X)
```

is valid.

6. Lists

A *list* is an important data structure in Prolog. A list is either the atom `'[]'`, denoting the empty list, or a compound term such that its first argument is the head of the list and its second argument is the tail of the list. Lists can be written in several ways:

1. dotted notation, e.g., `'.(a,.(b,[]))'`,
2. list notation, e.g., `'[a,b]'`, or
3. a list notation that separates the head and tail of a list with a vertical bar, e.g., `'[a|b]'`.

Strings are lists of ASCII character codes enclosed in double quotes, for example:

```
"hello"
```

7. Variables

Variables are names that can be *instantiated* to particular objects. The instantiation occurs when Prolog matches a fact to a question. For example, suppose the database contains the following fact:

```
value(a,5).
```

read, "The value of a is 5."

On answering the question:

```
?- value(X,5).
```

read, "Does there exist an X such that its value is 5?", the variable X will become instantiated to `a`.

Facts and rules can also contain variables, e.g.,

```
positive(X) :- value(X,N), integer(N), N>0.
```

This rule can be read as "For any X and N, if the value of X is N, N is an integer, and N is greater than zero, then X is positive." The scope of a variable is limited to a single clause. If a variable becomes instantiated, all occurrences of the variable in the clause become instantiated to the same object.

8. Backtracking

When a question is asked, Prolog will search through its database to match facts to the goals in the question. The database is searched top-down (i.e., in the order in which clauses were entered). If a match is found to the first goal, then Prolog sets a marker to that goal's place in the database and attempts to satisfy each successive goal. A goal *fails* (is not satisfied) if the information in the database is insufficient to show the fact is true. If all the goals can be satisfied, Prolog will respond "yes"; otherwise, it will respond "no."

For each goal, Prolog starts its search at the beginning of the database and sets a place marker if the goal becomes satisfied. If a goal fails, then Prolog attempts to *re-satisfy* the previous goal, by resuming its search from the previous goal's place marker. This behavior of re-satisfying goals is termed *backtracking*.

9. Defining Predicates and Operators

In Prolog, programmers can define new functors -- both predicates and operators. Any legal atom is a valid name for a functor. The system defines a standard set of commonly-used functors. A few examples are:

read, write, is, not, `+, *, =, ',', and ';'`

The `','` is an infix operator separating conjunction of goals; the `';'` is an infix operator separating disjunction of goals.

Redefining system predicates is prohibited, but redefining system operators is permitted. The **op** functor is a special predicate used in defining operators, e.g.,

```
:-op(240,xfx,newop).
```

Here, '240' specifies the precedence class of **newop**, and **xfx** specifies its associativity.

10. Syntax

We describe Prolog syntax using the Prolog grammar-rule notation, called the Definite Clause Grammar (DCG) form (Ref. [120]). This notation is reminiscent of the Backus-Naur form; i.e., grammar rules consist of a syntactic category (also called a nonterminal), followed by an arrow, followed by sequences of syntactic categories and terminals. DCGs generalize normal context-free grammars; however, since they allow nonterminals to carry variables, they operate essentially as attribute grammars.

In DCGs, the following conventions apply:

1. Lowercase words denote syntactic categories.
2. Square brackets enclose terminals. In particular, the empty list (`'[]'`) denotes the null string.
3. Terminals are denoted by Prolog atoms. (They may be enclosed in single quotes.)
4. A comma separates adjacent items in a sequence.
5. A semicolon separates alternative items and has lower precedence than a comma. We have defined the selection operator `'|'` to replace the semicolon.
6. Each rule ends with a full stop (`'.'`).

The Prolog syntax presented here is drawn from descriptions in the literature (Ref. [19, 20]).

```
program                 -->  sequence_of_clauses.
sequence_of_clauses     -->  clause |
                             clause, sequence_of_clauses.
clause                  -->  query | positive_clause.
positive_clause         -->   fact  | rule.
query                   -->  ['?-'], body, ['.'].
fact                    -->  head.
rule                    -->  head, [':-'], body, ['.'].
head                    -->  literal.
body                    -->  sequence_of_goals |
                             sequence_of_goals, [';'],
                                sequence_of_goals |
```

```
                              ['('], sequence_of_goals, [';']
                                 sequence_of_goals, [')'].
sequence_of_goals       -->   goal |
                              goal,[','],sequence_of_goals |
                              ['('],goal,[','],
                                 sequence_of_goals, [')'].
goal                    -->   literal |
                              ['('], literal, [')'].
literal                 -->   atom | variable | compound term |
                              ['('],literal,[')'].
term                    -->   variable |
                              constant |
                              compound_term |
                              ['('],term,[')'].
constant                -->   atom | number.
compound_term           -->   predicate,['('],
                              sequence_of_arguments,[')'] |
                              argument,infix_operator,argument |
                              prefix_operator, argument |
                              argument, postfix_operator.
infix_operator          -->   operator.
prefix_operator         -->   operator.
postfix_operator        -->   operator.
predicate               -->   functor_name.
operator                -->   functor_name.
functor_name            -->   atom.
sequence_of_arguments   -->   argument |
                              argument,[','],
                                sequence_of_arguments.
argument                -->   term.
```

APPENDIX B
CONVERTING PATH EXPRESSIONS TO DCGS

```
/*********************************************************/
/* usage: path_expression(FileE,S,D,DCGFile).      */
/* FileE is the name of a file containing a              */
/* path expression and                                   */
/* DCGFile is the name of a file to which the            */
/* DCGs will be written.                                 */
/*                                                       */
/* Parses the expression and builds the                  */
/* DCGs for matching against event sequences.            */
/*********************************************************/

path_expression(InFile,Source,DCG,DCGFile) :-
        cat(InFile),
        see(InFile),!,
        pscan(Source),          /* scans a path expression      */
        seen,
        gensym(node,NT),!,      /* get first node for     DCGs  */
        expression(NT,[],OldDCG,Source,[]),
                                /* add rule to head: node1 --> [].*/
        append([rule('',NT,'')],OldDCG,DCG),
        tell(DCGFile),
        inlist(DCG),    /* Saves DCGs for queries */
        told.

cat(File):-     see(File),
                nl,
                repeat,
                  get0(X),
                  (endfile(X);put0(X),fail),
                seen.

put0(X):-       name(A,[X|[]]), /*get0 converts to ascii code   */
                write(A).

expression(NT,OldDCG,DCG) --> path(NT,NT,OldDCG,DCG),['$'].

path(Left,Right,OldDCG,NewDCG) -->
        [path],
        element(E,Left,Right,S,OldDCG,DCG1),
        cycle,
        alternate(A,Left,Right,S,DCG1,DCG2),
        sequence(S,Right,DCG2,DCG3),
        [end],
        {append(DCG3,[rule(E,A,S)],NewDCG)},!.

sequence(NewNT,Right,OldDCG,NewDCG) -->
        [';'],
        {gensym(node,NewNT)},
        element(E,NewNT,Right,S,OldDCG,DCG1),
        cycle,
        alternate(A,NewNT,Right,S,DCG1,DCG2),
        sequence(S,Right,DCG2,DCG3),
        {append([rule(E,A,S)],DCG3,NewDCG)},!.

sequence(NT,NT,DCG,DCG) --> [].
```

```
alternate(_,Left,Right,S,OldDCG,NewDCG)     -->
        [','],
        element(E,Left,Right,S,OldDCG,DCG1),
        cycle,
        alternate(A,Left,Right,S,DCG1,DCG2),
        {append(DCG2,[rule(E,A,S)],NewDCG)},!.

alternate(NT,NT,_,_,DCG,DCG)  --> [].

element(E,Left,Right,S,OldDCG,NewDCG)    -->
        ['('],
        element(E,Left,Right,S,OldDCG,DCG1),
        cycle,
        alternate(_,Left,Right,S,DCG1,DCG2),
        sequence(_,Right,DCG2,NewDCG),
        [')'].

element(E,_,_,_,DCG,DCG) --> [E], {recorded(is_element,E,_)},!.

cycle   -->     ['*'],
                {remove_sym(node,N)},!.

cycle   -->     [].

/*********************************************************/
/* gensym generates unique                               */
/* names for non-terminals                               */
/*********************************************************/
gensym(X,Y)     :-
        getnum(X,N),
        name(X,NameX),
        integer_name(N,NameN),
        append(NameX,NameN,NameY),
        name(Y,NameY).

getnum(X,N)     :-      /*this name encountered before */
        retract(current_num(X,Num1)),!,
        N is Num1 + 1,
        asserta(current_num(X,N)).

getnum(X,1)     :-      /* first time */
        asserta(current_num(X,1)).

remove_sym(X,N)     :-      /*remove a symbol */
        retract(current_num(X,Num1)),!,
        N is Num1 - 1,
        asserta(current_num(X,N)).

remove_sym(X,1).            /* no symbols to remove */

/* convert from integer to list of characters   */
integer_name(I,L)  :- integer_name(I,[],L).
integer_name(I,Sofar,[C|Sofar])     :-
        I<10,!, C is I+48.
integer_name(I,Sofar,L)     :-
        Tophalf is I/10,
        Bothalf is I mod 10,
        C is Bothalf+48,
        integer_name(Tophalf,[C|Sofar],L).

append([],L,L).
append([X|L1],L2,[X|L3]):-
        append(L1,L2,L3).
```

```
/************************************************/
/* inlist(X)                                    */
/* where X is a list of DCGs                    */
/* Writes each rule in either of the            */
/* following forms:                             */
/*      node1 --> [].                           */
/*      node1 --> [a], node2.                   */
/************************************************/
inlist([]).     /* end of list */
inlist([rule(E,A,S)|T]) :-
        write(A),
        write(' --> ['),
        write(E),
        write(']'),
        testNT(S),
        write('.'),
        nl,
        inlist(T).

testNT('').
testNT(S):- write(', '), write(S).
```

APPENDIX C
ADA PARSER

```
/****************************************/
/* Top-Down Parser for ANSI Ada '83     */
/*                                      */
/* An Ada program is arbitrarily        */
/* defined to be a sequence of          */
/* compilation units, followed by an    */
/* eof marker (designated as a '$').    */
/*                                      */
/* Changed some things to avoid         */
/* conflict with Prolog, e.g.,:         */
/*                                      */
/* Uppercase letters are converted to   */
/* lowercase.                           */
/*                                      */
/* Added a rule for an apostrophe       */
/* character.                           */
/*                                      */
/* Changed name of syntactic category:  */
/*  name    =>   a_name                 */
/****************************************/

parse(File,Source,ParseTree) :-
                cat(File),
                see(File),!,
                scan(Source),
                seen,!,
                program(ParseTree,Source,[]).

program(program(C)) --> compilation(C),['$'].

/* apostrophe                           */
an_apostrophe(N)    --> [N],{name(N,[39])}.

/* 2.3                                  */
identifier(identifier(I)) --> [I],
        {recorded(is_identifier,I,_)},!.

/* 2.4                                  */
numeric_literal(numeric_literal(N)) -->
        [N], {recorded(is_numeric_literal,N,_)},!.

/* 2.5                                  */
character_literal(character_literal(N)) -->
        [N], {recorded(is_character_literal,N,_)},!.

/* 2.6                                  */
string_literal(string_literal(S)) -->
        [N],
        {recorded(is_string_literal,N,_),name(S,N)},!.

/* 2.8                                  */
pragma(pragma(pragma,I,R)) -->
        [pragma],identifier(I),rest_pragma(R).

rest_pragma(rest_pragma(';'))          --> [';'].
rest_pragma(rest_pragma('(',A,L,');'))        -->
        ['('],argument_association(A),
        argument_association_list(L),[')',';'].

argument_association_list(argument_association_list('')) --> [].
argument_association_list(argument_association_list(',',A,L)) -->
        [','],argument_association(A),argument_association_list(L).
```

```
argument_association(argument_association(I,'=>',E))  -->
        identifier(I),['=','>'],expression(E).
argument_association(argument_association(E))         -->
         expression(E).

/* 3.1                                      */
basic_declaration(basic_declaration(D))       -->
        object_declaration(D);              number_declaration(D);
        type_declaration(D);                subtype_declaration(D);
        subprogram_declaration(D); package_declaration(D);
        task_declaration(D);                generic_declaration(D);
        exception_declaration(D);  generic_instantiation(D);
        renaming_declaration(D).

/* 3.2                                      */
object_declaration(object_declaration(L,':',C,R))     -->
        identifier_list(L),[':'],constant_option(C),
                rest_object_declaration(R).

rest_object_declaration(rest_object_declaration(T,I,';'))   -->
        (constrained_array_definition(T),
                initialize_option(I),[';']);
        (subtype_indication(T),
                initialize_option(I),[';']).

constant_option(constant_option(constant))    --> [constant].
constant_option(constant_option(''))  --> [].

initialize_option(initialize_option(':=',E))  -->
         [':','='],expression(E).
initialize_option(initialize_option(''))      --> [].

number_declaration(number_declaration(L,': constant :=',E,';')) -->
        identifier_list(L),
        [':',constant,':','='],
        expression(E),[';'].

identifier_list(identifier_list(I,R)) -->
        identifier(I),rest_identifier_list(R).

rest_identifier_list(rest_identifier_list(',',L))   -->
        [','],identifier_list(L).
rest_identifier_list(rest_identifier_list(''))      --> [].

/* 3.3.1                                    */
type_declaration(type_declaration(D)) -->
        full_type_declaration(D);
        incomplete_type_declaration(D);
        private_type_declaration(D).

full_type_declaration(full_type_declaration(type,I,D,is,T,';'))
        -->
        [type],identifier(I),discriminant_part_option(D),
                [is],type_definition(T),[';'].

discriminant_part_option(discriminant_part_option(D)) -->
        discriminant_part(D).
discriminant_part_option(discriminant_part_option('')) --> [].

type_definition(type_definition(D))   -->
        enumeration_type_definition(D);
        integer_type_definition(D);
        real_type_definition(D);
        array_type_definition(D);
        record_type_definition(D);
        access_type_definition(D);
        derived_type_definition(D).
```

```
/* 3.3.2                                    */
subtype_declaration(subtype_declaration(subtype,I,is,S,';')) -->
        [subtype],identifier(I),[is],subtype_indication(S),[';'].

subtype_indication(subtype_indication(T,R))    -->
        type_mark(T),rest_subtype_indication(R).

rest_subtype_indication(rest_subtype_indication(C)) -->
        constraint(C).
rest_subtype_indication(rest_subtype_indication(''))        --> [].

type_mark(type_mark(N))        -->
        type_name(N); subtype_name(N).

type_name(type_name(N))                  --> expanded_name(N).
subtype_name(subtype_name(N)) --> expanded_name(N).

constraint(constraint(C))        -->
        range_constraint(C);
        floating_point_constraint(C);
        fixed_point_constraint(C);
        general_aggregate(C).

/* 3.4                                      */
derived_type_definition(derived_type_definition(new,S))      -->
        [new],subtype_indication(S).

/* 3.5                                      */
range_constraint(range_constraint(range,R))    -->
        [range],range(R).

range(range(S,R))        -->
        simple_expression(S),rest_range(R).

rest_range(rest_range('..',S))      -->
        ['.','.'],simple_expression(S).

rest_range(rest_range(''))          --> [].

/* 3.5.1                                    */
enumeration_type_definition(
        enumeration_type_definition('(',E,L,')'))      -->

        ['('],enumeration_literal_specification(E),
        enumeration_literal_list(L),[')'].

enumeration_literal_specification(
        enumeration_literal_specification(I)) -->

        identifier(I);
        character_literal(I).

enumeration_literal_list(
        enumeration_literal_list(',',E,L))    -->

        [','],enumeration_literal_specification(E),
           enumeration_literal_list(L).

enumeration_literal_list(enumeration_literal_list(''))       --> [].

/* 3.5.4                                    */
integer_type_definition(integer_type_definition(R)) -->
        range_constraint(R).

/* 3.5.6                                    */
real_type_definition(real_type_definition(F))        -->
        floating_point_constraint(F);
        fixed_point_constraint(F).
```

```
/* 3.5.7                                                    */
floating_point_constraint(floating_point_constraint(F,R))    -->
        floating_accuracy_definition(F),range_constraint_option(R).

floating_accuracy_definition(
        floating_accuracy_definition(digits,S))        -->
        [digits],simple_expression(S).

range_constraint_option(range_constraint_option(R)) -->
        range_constraint(R).
range_constraint_option(range_constraint_option('')) --> [].

/* 3.5.9                                                    */
fixed_point_constraint(fixed_point_constraint(F,R)) -->
        fixed_accuracy_definition(F),range_constraint_option(R).

fixed_accuracy_definition(fixed_accuracy_definition(delta,E)) -->
        [delta],simple_expression(E).

/* 3.6                                                      */
array_type_definition(array_type_definition(D))      -->
        unconstrained_array_definition(D);
        constrained_array_definition(D).

unconstrained_array_definition(
   unconstrained_array_definition('array(',I,L,') of',S))      -->
        [array,'('],index_subtype_definition(I),
        index_subtype_definition_list(L),[')',of],
        subtype_indication(S).

index_subtype_definition_list(
        index_subtype_definition_list(',',I,L))  -->
        [','],index_subtype_definition(I),
           index_subtype_definition_list(L).

index_subtype_definition_list(
        index_subtype_definition_list('')) --> [].

constrained_array_definition(
        constrained_array_definition(array,I,of,S)) -->
        [array],index_constraint(I),[of],subtype_indication(S).

index_subtype_definition(index_subtype_definition(N,'range<>'))
        --> a_name(N),[range,'<','>'].

index_constraint(index_constraint('(',D,L,')'))        -->
        ['('],discrete_range(D),discrete_range_list(L),[')'].

discrete_range_list(discrete_range_list(',',D,L))    -->
        [','],discrete_range(D),discrete_range_list(L).
discrete_range_list(discrete_range_list(''))         --> [].

discrete_range(discrete_range(R))      --> range(R).
discrete_range(discrete_range(N,R))    -->
        a_name(N),range_constraint(R).

/* 3.7                                                      */
record_type_definition(record_type_definition(record,L,end,record))
        -->      [record],component_list(L),[end,record].

component_list(component_list(null,';',P))     -->
        [null,';'],pragma_list(P).
```

```
component_list(component_list(P1,X,Y,P2))      -->
        (pragma_list(P1),
                component_declaration(X),
                component_declaration_list(Y),
                pragma_list(P2));
        (pragma_list(P1),
                component_declaration_list(X),
                variant_part(Y),
                pragma_list(P2)).

pragma_list(pragma_list(P,L)) -->
        pragma(P),pragma_list(L).
pragma_list(pragma_list(''))  --> [].

component_declaration_list(component_declaration_list(''))  --> [].
component_declaration_list(component_declaration_list(C,P,L)) -->
        component_declaration(C),pragma_list(P),
                component_declaration_list(L).

component_declaration(component_declaration(L,':',S,I,';'))    -->
        identifier_list(L),[':'],subtype_indication(S),
        initialize_option(I),[';'].

/* 3.7.1                                        */
discriminant_part(discriminant_part('(',D,L,')'))      -->
        ['('],discriminant_specification(D),
           discriminant_specification_list(L),[')'].

discriminant_specification_list(
  discriminant_specification_list(';',D,L)) -->
        [';'],discriminant_specification(D),
         discriminant_specification_list(L).

discriminant_specification_list(discriminant_specification_list(''))
        --> [].

discriminant_specification(discriminant_specification(L,';',T,I))
        -->
        identifier_list(L),[':'],type_mark(T),initialize_option(I).

/* 3.7.2 --See general aggregate        */

/* 3.7.3                                        */
variant_part(variant_part(case,S,is,P,V,L,end,case,';'))       -->
        [case],simple_name(S),[is],pragma_list(P),
           variant(V),variant_list(L),
        [end,case,';'].

variant_list(variant_list(''))          -->     [].
variant_list(variant_list(V,L))         -->
        variant(V),variant_list(L).

variant(variant(when,C,'=>',L)) -->
        [when],choice_list(C),['=','>'],component_list(L).

choice_list(choice_list(C,R))           -->
        choice(C),rest_choice_list(R).
rest_choice_list(rest_choice_list('|',L))    -->
        ['|'],choice_list(L).
rest_choice_list(rest_choice_list(''))                --> [].

choice(choice(others))  -->     [others].
choice(choice(R))       -->     discrete_range(R).

/* 3.8                                          */
access_type_definition(access_type_definition(access,S))     -->
        [access],subtype_indication(S).
```

```
/* 3.8.1                                        */
incomplete_type_declaration(
        incomplete_type_declaration(type,I,D,';')) -->
        [type],identifier(I),discriminant_part_option(D),[';'].
/* 3.9.                                         */
declarative_part(declarative_part(D,R))        -->
        basic_declarative_item_list(D),rest_declarative_part(R).

rest_declarative_part(rest_declarative_part(B,L))    -->
        body(B),later_declarative_item_list(L).
rest_declarative_part(rest_declarative_part(''))     --> [].

basic_declarative_item_list(basic_declarative_item_list(P)) -->
        pragma_list(P).

basic_declarative_item_list(basic_declarative_item_list(D,L,P)) -->
        basic_declarative_item(D),
        basic_declarative_item_list(L),pragma_list(P).

later_declarative_item_list(later_declarative_item_list(D,L,P)) -->
        later_declarative_item(D),
        later_declarative_item_list(L),pragma_list(P).
later_declarative_item_list(later_declarative_item_list(P)) -->
        pragma_list(P).

basic_declarative_item(basic_declarative_item(D))    -->
        basic_declaration(D);
        representation_clause(D);
        use_clause(D).

later_declarative_item(later_declarative_item(D)) -->
        body(D);
        subprogram_declaration(D); package_declaration(D);
        task_declaration(D);                generic_declaration(D);
        use_clause(D);                      generic_instantiation(D).

body(body(B))    -->     proper_body(B);     body_stub(B).

proper_body(proper_body(B))    -->
        subprogram_body(B);
        package_body(B);
        task_body(B).

/* 4.1                                          */
expanded_name(expanded_name(I,'.',E)) -->
        identifier(I),['.'],expanded_name(E).
expanded_name(expanded_name(I))          -->     identifier(I).

a_name(a_name(N,R))    -->
        (character_literal(N),      rest_a_name(R));
        (operator_symbol(N),        rest_a_name(R));
        (simple_name(N),    rest_a_name(R)).

simple_name(simple_name(I))    -->     identifier(I).
/* The syntax rules use the term  identifier for the first      */
/* occurrence of an identifier in  some formal declaration;     */
/*  otherwise, simple_name                                      */

rest_a_name(rest_a_name(''))          --> [].

rest_a_name(rest_a_name(N,R))         -->
        (general_aggregate_list(N),rest_indexed_component(R));
        (selector_list(N),          rest_selected_component(R));
        (attribute_designator_list(N), rest_attribute(R)).

/* 4.1.1                                        */
rest_indexed_component(rest_indexed_component(''))  --> [].
```

```
rest_indexed_component(rest_indexed_component(N,I,R))        -->
        (selector_list(N),
         selector_aggregate_list(I),
         rest_selected_component(R));
        (attribute_designator_list(N),
         attribute_aggregate_list(I),
         rest_attribute(R)).

general_aggregate_list(general_aggregate_list(N,L)) -->
        general_aggregate(N),rest_general_aggregate_list(L).

rest_general_aggregate_list(rest_general_aggregate_list('')) --> [].
rest_general_aggregate_list(rest_general_aggregate_list(I)) -->
        general_aggregate_list(I).

/* 4.1.2        -- see index_component      */

/* 4.1.3                                    */
rest_selected_component(rest_selected_component(''))         --> [].

rest_selected_component(rest_selected_component(N,C))        -->
        general_aggregate_list(N),
        attribute_part(C).

rest_selected_component(rest_selected_component(A,S,R))      -->
        attribute_designator_list(A),
        rest_attribute_list(S),
        rest_attribute(R).

selector_part(selector_part(''))       --> [].

selector_part(selector_part(S,A,R))    -->
        selector_list(S),
        selector_aggregate_list(A),
        rest_general_aggregate_list(R).

selector_list(selector_list('.',N,L)) -->
        ['.'],selector(N),rest_selector_list(L).

rest_selector_list(rest_selector_list(''))  --> [].
rest_selector_list(rest_selector_list(S))   --> selector_list(S).

selector_aggregate_list(selector_aggregate_list(''))         --> [].

selector_aggregate_list(selector_aggregate_list(I,S,L))      -->
        general_aggregate_list(I),
        selector_list(S),
        selector_aggregate_list(L).

selector(selector(all)) -->      [all].
selector(selector(S))   -->
        simple_name(S);              character_literal(S);
        operator_symbol(S).

/* 4.1.4                                    */
attribute_part(attribute_part(''))     --> [].

attribute_part(attribute_part(D,A,E,R))        -->
        attribute_designator_list(D),
        attribute_aggregate_list(A),
        rest_attribute_list(E),
        rest_attribute(R).

rest_attribute(rest_attribute(''))     --> [].
rest_attribute(rest_attribute(A,C))    -->
        (general_aggregate_list(A),selector_part(C));
        (selector_list(A),general_aggregate_list(C)).
```

```
attribute_designator_list(attribute_designator_list(Q,N,L)) -->
        an_apostrophe(Q),
        attribute_designator(N),
        rest_attribute_designator_list(L).

attribute_designator(attribute_designator(digits))     --> [digits].
attribute_designator(attribute_designator(delta))      --> [delta].
attribute_designator(attribute_designator(range))      --> [range].
attribute_designator(attribute_designator(A)) --> simple_name(A).

rest_attribute_designator_list(rest_attribute_designator_list(''))
        --> [].

rest_attribute_designator_list(rest_attribute_designator_list(A))
  -->
        attribute_designator_list(A).

attribute_aggregate_list(attribute_aggregate_list(''))       --> [].
attribute_aggregate_list(attribute_aggregate_list(I,A,L))    -->
        general_aggregate_list(I),
        attribute_designator_list(A),
        attribute_aggregate_list(L).

rest_attribute_list(rest_attribute_list('')) --> [].
rest_attribute_list(rest_attribute_list(A,S,L,X))    -->
        general_aggregate_list(A), selector_list(S),
        selector_aggregate_list(L),attribute_list_one(X).

rest_attribute_list(rest_attribute_list(S,A,X))      -->
        selector_list(S), general_aggregate_list(S),
        attribute_list_two(X).

attribute_list_one(attribute_list_one(A,L)) -->
        attribute_designator_list(A),rest_attribute_list(L).

attribute_list_one(attribute_list_one(G,A,R,L))      -->
        general_aggregate_list(G),
        attribute_designator_list(A),
        attribute_aggregate_list(R),
        rest_attribute_list(L).

attribute_list_two(attribute_list_two(L))    -->
        rest_attribute_list(L).

attribute_list_two(attribute_list_two(A,R,L))        -->
        attribute_designator_list(A),
        attribute_aggregate_list(R),
        rest_attribute_list(L).

/* 4.3                                          */
aggregate(aggregate('(',C,L,')'))               -->
        ['('],component_association(C),
        component_association_list(L),[')'].

component_association_list(component_association_list(',',C,L)) -->
        [','],component_association(C),
        component_association_list(L).

component_association_list(component_association_list('')) --> [].

component_association(component_association(E))        -->
        expression(E).

component_association(component_association(L,'=>',E))         -->
        choice_list(L),['=','>'],expression(E).

general_aggregate(general_aggregate('(',G,L,')'))      -->
        ['('],general_component_association(G),
           general_component_association_list(L),[')'].
```

```
general_component_association_list(
  general_component_association_list(',',G,L))    -->
        [','],general_component_association(G),
        general_component_association_list(L).

general_component_association_list(
        general_component_association_list(''))  --> [].

general_component_association(general_component_association(C)) -->
        component_association(C).

general_component_association(general_component_association(N,R))
        -->  a_name(N),range_constraint(R).

general_component_association(
   general_component_association(S1,'..',S2)) -->
        simple_expression(S1),['.','.'],simple_expression(S2).

/* 4.4                                                  */
expression(expression(R,E)) -->
        relation(R), rest_expression(E).

rest_expression(rest_expression(R)) -->
        and_relation(R);   and_then_relation(R);
        or_relation(R);            or_else_relation(R);
        xor_relation(R).

rest_expression(rest_expression('')) --> [].

and_relation(and_relation(and,R,A)) -->
        [and], relation(R),rest_and_relation(A).

rest_and_relation(rest_and_relation(R))     -->
        and_relation(R).
rest_and_relation(rest_and_relation('')) --> [].

and_then_relation(and_then_relation(and,then,R,A))  -->
        [and,then],relation(R),rest_and_then_relation(A).

rest_and_then_relation(rest_and_then_relation(R)) -->
        and_then_relation(R).
rest_and_then_relation(rest_and_then_relation(''))        --> [].

or_relation(or_relation(or,R,A))      -->
        [or], relation(R),rest_or_relation(A).

rest_or_relation(rest_or_relation(R))       -->
        or_relation(R).
rest_or_relation(rest_or_relation(''))      --> [].

or_else_relation(or_else_relation(or,else,R,A))     -->
        [or,else], relation(R),rest_or_else_relation(A).

rest_or_else_relation(rest_or_else_relation(R))   -->
        or_else_relation(R).
rest_or_else_relation(rest_or_else_relation(''))  --> [].

xor_relation(xor_relation(xor,R,X))   -->
        [xor], relation(R),rest_xor_relation(X).

rest_xor_relation(rest_xor_relation(R))     -->
        xor_relation(R).
rest_xor_relation(rest_xor_relation(''))    --> [].

relation(relation(E,R)) -->
        simple_expression(E),rest_relation(R).
```

```
rest_relation(rest_relation(R,E))      -->
        relational_operator(R),simple_expression(E).
rest_relation(relation(N,in,R))              -->
        not_option(N),[in],range(R).
rest_relation(rest_relation(''))       --> [].

not_option(not_option(not))    --> [not].
not_option(not_option(''))     --> [].

simple_expression(simple_expression(U,T,L))    -->
        unary_adding_operator(U),term(T),
        binary_adding_operator_list(L).

simple_expression(simple_expression(T,L))      -->
        term(T),binary_adding_operator_list(L).

binary_adding_operator_list(binary_adding_operator_list(B,T,L))
  -->   binary_adding_operator(B),
        term(T),binary_adding_operator_list(L).

binary_adding_operator_list(binary_adding_operator_list('')) --> [].

term(term(F,L)) --> factor(F), multiplying_operator_list(L).

multiplying_operator_list(multiplying_operator_list(M,F,L)) -->
        multiplying_operator(M),factor(F),
        multiplying_operator_list(L).

multiplying_operator_list(multiplying_operator_list(''))  --> [].

factor(factor(abs,P))           --> [abs],primary(P).
factor(factor(not,P))           --> [not],primary(P).
factor(factor(P,R))             --> primary(P),rest_factor(R).

rest_factor(rest_factor('**',P)) --> ['*','*'],primary(P).
rest_factor(rest_factor(''))          --> [].

primary(primary(null))  --> [null].
primary(primary(P))     -->
        numeric_literal(P);        allocator(P);
        aggregate(P).
primary(primary(P,R))   -->
        a_name(P), rest_primary(R).

rest_primary(rest_primary(''))        --> [].

rest_primary(rest_primary(P)) -->
        rest_qualified_expression(P).

/* 4.5                                  */
relational_operator(relational_operator('/=')) --> ['/','='].
relational_operator(relational_operator('='))  --> ['='].
relational_operator(relational_operator('<=')) --> ['<','='].
relational_operator(relational_operator('<'))  --> ['<'].
relational_operator(relational_operator('>=')) --> ['>','='].
relational_operator(relational_operator('>'))  -->  ['>'].

binary_adding_operator(binary_adding_operator('+')) --> ['+'].
binary_adding_operator(binary_adding_operator('-')) --> ['-'].
binary_adding_operator(binary_adding_operator('&')) --> ['&'].

unary_adding_operator(unary_adding_operator('+'))   --> ['+'].
unary_adding_operator(unary_adding_operator('-'))   --> ['-'].

multiplying_operator(multiplying_operator('*'))          -->  ['*'].
multiplying_operator(multiplying_operator('/'))          -->  ['/'].
multiplying_operator(multiplying_operator(mod))          -->  [mod].
multiplying_operator(multiplying_operator(rem))          -->  [rem].

/* 4.6 type conversion -- see a_name       */
```

```
/* 4.7                                          */
rest_qualified_expression(qualified_expression(Q,A))      -->
        an_apostrophe(Q),aggregate(A).

/* 4.8                                          */
allocator(allocator(new,T,R))    -->
        [new],type_mark(T),rest_allocator(R).

rest_allocator(rest_allocator(''))     --> [].
rest_allocator(rest_allocator(Q,G))    -->
        an_apostrophe(Q),aggregate(G).

rest_allocator(rest_allocator(G))      -->
        general_aggregate(G).

/* 5.1                                          */
sequence_of_statements(sequence_of_statements(P,S,L))        -->
        pragma_list(P),statement(S),statement_list(L).

statement_list(statement_list(P))      -->
        pragma_list(P).
statement_list(statement_list(S,L,P)) -->
        statement(S),statement_list(L),pragma_list(P).

statement(statement(L,S))        -->
        label_list(L),rest_statement(S).

rest_statement(rest_statement(S))      -->
        simple_statement(S);
        compound_statement(S).

label_list(label_list(LA,LL)) -->
        label(LA),label_list(LL).
label_list(label_list(''))     --> [].

simple_statement(simple_statement(S)) -->
        null_statement(S);          exit_statement(S);
        return_statement(S);                goto_statement(S);
        delay_statement(S);                 abort_statement(S);
        raise_statement(S);                 call_statement(S);
        assignment_statement(S);    code_statement(S).

compound_statement(compound_statement(S))      -->
        if_statement(S);            case_statement(S);
        loop_statement(S);          block_statement(S);
        accept_statement(S);                select_statement(S).

label(label('<<',S,'>>'))        -->
        ['<','<'],simple_name(S),['>','>'].

null_statement(null_statement(null,';'))      -->      [null,';'].

/* 5.2                                          */
assignment_statement(assignment_statement(N,':=',E,';')) -->
        a_name(N),[':','='],expression(E),[';'].

/* 5.3                                          */
if_statement(if_statement(if,C,then,S,I,E,end,if,';'))        -->
        [if],condition(C),[then],
                sequence_of_statements(S),
        elsif_option(I),
        else_option(E),
        [end,if,';'].

elsif_option(elsif_option(elsif,C,then,S,I))  -->
        [elsif],condition(C),[then],
                sequence_of_statements(S),
                        elsif_option(I).
elsif_option(elsif_option(''))         --> [].
```

```
else_option(else_option(else,S))       -->
        [else],
                sequence_of_statements(S).
else_option(else_option(''))  --> [].

condition(condition(E)) --> expression(E).

/* 5.4                                          */
case_statement(case_statement(case,E,is,P,C,L,end,case,';'))  -->
        [case],expression(E),[is],
                pragma_list(P),case_statement_alternative(C),
                case_statement_alternative_list(L),
        [end,case,';'].

case_statement_alternative_list(case_statement_alternative_list(''))
        --> [].

case_statement_alternative_list(
        case_statement_alternative_list(C,L)) -->

        case_statement_alternative(C),
        case_statement_alternative_list(L).

case_statement_alternative(
        case_statement_alternative(when,C,'=>',S)) -->

        [when],choice_list(C),['=','>'],
                sequence_of_statements(S).

/* 5.5                                          */
loop_statement(loop_statement(L1,I,B,L2,';')) -->
        loop_simple_name_begin(L1),
                iteration_rule_option(I),
                basic_loop(B), loop_simple_name_end(L2),[';'].

loop_simple_name_begin(loop_simple_name_begin(S,':'))      -->
        simple_name(S),[':'].
loop_simple_name_begin(loop_simple_name_begin(''))         --> [].

loop_simple_name_end(loop_simple_name_end(S))      -->
        simple_name_option(S).

simple_name_option(simple_name_option(S))    --> simple_name(S).
simple_name_option(simple_name_option(''))   --> [].

iteration_rule_option(iteration_rule_option(I))      -->
        iteration_rule(I).

iteration_rule_option(iteration_rule_option(''))     --> [].

basic_loop(basic_loop(loop,S,end,loop))        -->
        [loop],
                sequence_of_statements(S),
        [end,loop].

iteration_rule(iteration_rule(while,C))        -->
        [while], condition(C).
iteration_rule(iteration_rule(for,L)) -->
        [for],loop_parameter_specification(L).

loop_parameter_specification(loop_parameter_specification(I,in,R,D))
   -->  identifier(I), [in], reverse_option(R), discrete_range(D).

reverse_option(reverse_option(reverse))        --> [reverse].
reverse_option(reverse_option(''))     --> [].
```

```
/* 5.6                                                    */
block_statement(block_statement(B1,D,begin,S,E,end,B2,';'))    -->
        block_simple_name_begin(B1),
                block_declarative_part_option(D),
                [begin],
                        sequence_of_statements(S),
                exception_option(E),
                [end], block_simple_name_end(B2),[';'].
block_simple_name_begin(block_simple_name_begin(''))       --> [].
block_simple_name_begin(block_simple_name_begin(S,':'))    -->
        simple_name(S),[':'].

block_simple_name_end(block_simple_name_end(S))    -->
        simple_name_option(S).

block_declarative_part_option(
        block_declarative_part_option(declare,D)) -->
         [declare], declarative_part(D).

block_declarative_part_option(block_declarative_part_option(''))
        --> [].

exception_option(exception_option(exception,P,E,L))    -->
        [exception], pragma_list(P), exception_handler(E),
          exception_handler_list(L).
exception_option(exception_option(''))          -->  [].

exception_handler_list(exception_handler_list(E,L)) -->
        exception_handler(E), exception_handler_list(L).
exception_handler_list(exception_handler_list(''))  -->[].
/* 5.7                                                    */
exit_statement(exit_statement(exit,L,W,';'))  -->
        [exit], loop_name_option(L),
        when_condition_option(W), [';'].

loop_name_option(loop_name_option(E))        --> expanded_name(E).
loop_name_option(loop_name_option(''))       --> [].

when_condition_option(when_condition_option(when,C))          -->
        [when], condition(C).
when_condition_option(when_condition_option(''))     --> [].
/* 5.8                                                    */
return_statement(return_statement(return,';'))          -->
        [return,';'].
return_statement(return_statement(return,E,';'))        -->
        [return], expression(E), [';'].
/* 5.9                                                    */
goto_statement(goto_statement(goto,L,';'))     -->
        [goto], label_name(L), [';'].

label_name(label_name(E))      --> expanded_name(E).
/* 6.1                                                    */
subprogram_declaration(subprogram_declaration(S,';')) -->
        subprogram_specification(S), [';'].

subprogram_specification(subprogram_specification(procedure,I,F))
  -->
        [procedure], identifier(I), formal_part_option(F).
subprogram_specification(
        subprogram_specification(function,D,F,return,T)) -->
        [function],designator(D),formal_part_option(F),
        [return], type_mark(T).
```

```
formal_part_option(formal_part_option(F))    --> formal_part(F).
formal_part_option(formal_part_option(''))   --> [].

designator(designator(I)) --> identifier(I); operator_symbol(I).

operator_symbol(operator_symbol(S))    --> string_literal(S).

formal_part(formal_part('(',P,L,')')) -->
        ['('], parameter_specification(P),
        parameter_specification_list(L), [')'].

parameter_specification_list(parameter_specification_list(';',P,L))
        -->

        [';'], parameter_specification(P),
        parameter_specification_list(L).

parameter_specification_list(parameter_specification_list(''))
        --> [].

parameter_specification(parameter_specification(L,':',M,T,I)) -->
        identifier_list(L),[':'], mode(M),
        type_mark(T),initialize_option(I).

mode(mode(out))          --> [out].
mode(mode(in,out))       --> [in, out].
mode(mode(I))            --> in_option(I).
        /* 'in' is the default value */
in_option(in_option(in)) --> [in].
in_option(in_option('')) --> [].

/* 6.3                                        */
subprogram_body(subprogram_body(P,is,D,begin,S,E,end,O,';'))  -->
        subprogram_specification(P), [is],
                declarative_part(D),
        /* declarative_part may be [] */
        [begin],
                sequence_of_statements(S),
        exception_option(E),
        [end], designator_option(O), [';'].

designator_option(designator_option(''))      --> [].
designator_option(designator_option(D))       --> designator(D).

/* 6.4                                          */
call_statement(call_statement(N,';')) -->  a_name(N),[';'].

/* 7.1                                       */
package_declaration(package_declaration(P,';'))       -->
        package_specification(P), [';'].

package_specification(package_specification(package,V,P,end,S)) -->
        [package], visible_part(V),
        private_part_option(P),
        [end], package_simple_name_end(S).

visible_part(visible_part(I,is,D))    -->
        identifier(I), [is],
                basic_declarative_item_list(D).

private_part_option(private_part_option(private,B)) -->
        [private], basic_declarative_item_list(B).
private_part_option(private_part_option(''))        --> [].

package_simple_name_end(package_simple_name_end(N))       -->
        simple_name_option(N).
```

```
package_body(package_body(package,body,N,is,D,R))      -->
        [package,body], simple_name(N), [is],
                declarative_part(D),
                /* declarative part optional */
        rest_package_body(R).

rest_package_body(rest_package_body(end,N,';'))      -->
        [end], package_simple_name_end(N), [';'].

rest_package_body(rest_package_body(begin,S,E,end,N,';')) -->
        [begin],
                sequence_of_statements(S),
        exception_option(E),
        [end], package_simple_name_end(N), [';'].

/* 7.4                                          */
private_type_declaration(
        private_type_declaration(type,I,D,is,L,private,';'))
        -->
        [type],identifier(I),discriminant_part_option(D),
                [is],limited_option(L),[private,';'].

limited_option(limited)     --> [limited].
limited_option(limited_option(''))     --> [].

/* 8.4                                          */
use_clause(use_clause(use,P,L,';'))    -->
        [use], package_name(P),package_name_list(L), [';'].

package_name_list(package_name_list(''))     -->[].
package_name_list(package_name_list(',',P,L))        -->
        [','],package_name(P),package_name_list(L).

package_name(package_name(N)) --> expanded_name(N).

/* 8.5                                          */
renaming_declaration(
  renaming_declaration(I,':',T,renames,N,';'))  -->
        identifier(I),[':'],type_mark(T),
        [renames],a_name(N),[';'].

renaming_declaration(
  renaming_declaration(I,':',exception,renames,E,';'))  -->
        identifier(I),[':'],[exception,renames],
        exception_name(E),[';'].

renaming_declaration(
  renaming_declaration(package,I,renames,N,';')) -->
        [package],identifier(I),  [renames],package_name(N),[';'].

renaming_declaration(
  renaming_declaration(S,renames,N,';')) -->
        subprogram_specification(S),  [renames],a_name(N),[';'].

exception_name(exception_name(E))      --> expanded_name(E).

/* 9.1                                          */
task_declaration(task_declaration(T,';')) -->
        task_specification(T),[';'].

task_specification(task_specification(task,T,I,R)) -->
        [task],type_option(T),identifier(I),
        rest_task_specification(R).

rest_task_specification(rest_task_specification(is,E,R,end,S)) -->
        [is],
                entry_declaration_list(E),
                representation_clause_list(R),
        [end], task_simple_name_end(S).
```

```
rest_task_specification(rest_task_specification('')) --> [].
task_simple_name_end(task_simple_name_end(S))     -->
        simple_name_option(S).
type_option(type)  --> [type].
type_option(type_option(''))  --> [].
entry_declaration_list(entry_declaration_list(P))    -->
        pragma_list(P).
entry_declaration_list(entry_declaration_list(E,L,P))  -->
        entry_declaration(E), entry_declaration_list(L),
        pragma_list(P).
representation_clause_list(representation_clause_list('')) --> [].
representation_clause_list(representation_clause_list(R,L,P)) -->
        representation_clause(R),
        representation_clause_list(L),pragma_list(P).
task_body(task_body(task,body,N1,is,D,begin,S,E,end,S2,';'))  -->
        [task,body], simple_name(N1),[is],
              declarative_part(D),      /* may be [] */
        [begin],
                sequence_of_statements(S),
        exception_option(E),
        [end],task_simple_name_end(S2),[';'].
/* 9.5                                          */
entry_declaration(entry_declaration(entry,I,D,F,';')) -->
        [entry],identifier(I),discrete_range_option(D),
        formal_part_option(F),[';'].
discrete_range_option(discrete_range_option('(',D,')'))      -->
        ['('],discrete_range(D),[')'].
discrete_range_option(discrete_range_option(''))    --> [].
accept_statement(accept_statement(accept,N,I,F,R))     -->
        [accept],simple_name(N),entry_index_option(I),
        formal_part_option(F),rest_accept_statement(R).
rest_accept_statement(rest_accept_statement(';'))   -->
        [';'].
rest_accept_statement(rest_accept_statement(do,S,end,N,';')) -->
        [do],
              sequence_of_statements(S),
        [end],simple_name_option(N),[';'].
entry_index_option(entry_index_option('(',E,')'))   -->
        ['('],expression(E),[')'].
entry_index_option(entry_index_option(''))  --> [].
/* 9.6                                          */
delay_statement(delay_statement(delay,S,';')) -->
        [delay],simple_expression(S),[';'].
/* 9.7                                          */
select_statement(select_statement(select,P1,S))       -->
        [select],
              pragma_list(P1),
        rest_select_statement(S).
rest_select_statement(rest_select_statement(S)) -->
        selective_wait(S).
```

```
rest_select_statement(rest_select_statement(C,X,S)) -->
        call_statement(C),
        sequence_of_statements_option(X),
        rest_conditional_or_timed_entry_call(S).
/* 9.7.1                                                    */
selective_wait(selective_wait(A,L,E,end,select,';')) -->
                select_alternative(A),
                select_alternative_list(L),
                else_option(E),
        [end,select,';'].
select_alternative_list(select_alternative_list(or,P,A,L)) -->
        [or],pragma_list(P),select_alternative(A),
        select_alternative_list(L).
select_alternative_list(select_alternative_list('')) -->[].
select_alternative(select_alternative(G,S)) -->
        guard_option(G),selective_wait_alternative(S).
guard_option(guard_option(when,C,'=>',P)) -->
        [when],condition(C),['=','>'],pragma_list(P).
guard_option(guard_option('')) --> [].
selective_wait_alternative(selective_wait_alternative(A)) -->
        accept_alternative(A);
        delay_alternative(A); terminate_alternative(A).
accept_alternative(accept_alternative(A,S)) -->
        accept_statement(A),
        sequence_of_statements_option(S).
delay_alternative(delay_alternative(D,S)) -->
        delay_statement(D),
        sequence_of_statements_option(S).
sequence_of_statements_option(sequence_of_statements_option(S))
        -->
        pragma_list(S); sequence_of_statements(S).
terminate_alternative(terminate_alternative(terminate,';',P)) -->
        [terminate,';'],pragma_list(P).
/* 9.7.2          cond'l entry call                         */
rest_conditional_or_timed_entry_call(
    rest_conditional_or_timed_entry_call(else,S,end,select,';')) -->
        [else],
                sequence_of_statements(S),
        [end,select,';'].
/* 9.7.3          timed entry call                          */
rest_conditional_or_timed_entry_call(
  rest_conditional_or_timed_entry_call(or,P2,D,end,select,';')) -->
        [or],
                pragma_list(P2),
                delay_alternative(D),
        [end,select,';'].
/* 9.10                                                     */
abort_statement(abort_statement(abort,N,L,';')) -->
        [abort],a_name(N),a_name_list(L),[';'].
a_name_list(a_name_list('')) --> [].
a_name_list(a_name_list(',',N,L)) -->
        [','],a_name(N),a_name_list(L).
```

```
/* 10.1                                                    */
compilation(compilation(P))       -->
        pragma_list(P).
compilation(compilation(U,C,P)) -->
        compilation_unit(U),compilation(C),pragma_list(P).

compilation_unit(compilation_unit(C,L))          -->
        context_clause(C),library_or_secondary_unit(L).

library_or_secondary_unit(library_or_secondary_unit(L))    -->
        subprogram_declaration(L); package_declaration(L);
        generic_declaration(L);                    generic_instantiation(L);
        subprogram_body(L);                        package_body(L);
        subunit(L).

/* 10.1.1                                                  */
context_clause(context_clause(W,U,C)) -->
        with_clause(W),use_clause_list(U),context_clause(C).
context_clause(context_clause(''))     --> [].

with_clause(with_clause(with,N,L,';',P))       -->
        [with],simple_name(N),simple_name_list(L),[';'],
        pragma_list(P).

use_clause_list(use_clause_list(U,L,P))       -->
        use_clause(U),use_clause_list(L),pragma_list(P).
use_clause_list(use_clause_list(''))          --> [].

simple_name_list(simple_name_list(',',N,L)) -->
        [','],simple_name(N),simple_name_list(L).
simple_name_list(simple_name_list(''))        --> [].

/* 10.2                                                    */
body_stub(body_stub(S,is,separate,';'))          -->
        subprogram_specification(S),[is,separate,';'].
body_stub(body_stub(package,body,N,is,separate,';'))  -->
        [package,body],simple_name(N),[is,separate,';'].
body_stub(body_stub(task,body,N,is,separate,';'))      -->
        [task,body],simple_name(N),    [is,separate,';'].

subunit(subunit(separate,'(',N,')',B))  -->
        [separate,'('],parent_unit_name(N),[')'],
        proper_body(B).

parent_unit_name(parent_unit_name(E))          -->
        expanded_name(E).

/* 11.1                                                    */
exception_declaration(exception_declaration(I,':',exception,';'))
        -->
        identifier_list(I),[':',exception,';'].

/* 11.2                                                    */
exception_handler(exception_handler(when,C,L,'=>',S)) -->
        [when],exception_choice(C),
                exception_choice_list(L),['=','>'],
                sequence_of_statements(S).

exception_choice_list(exception_choice_list('|',C,L))          -->
        ['|'],exception_choice(C),exception_choice_list(L).

exception_choice_list(exception_choice_list(''))     --> [].

exception_choice(exception_choice(others))     --> [others].
exception_choice(exception_choice(N))   --> exception_name(N).
```

```
/* 11.3                                                    */
raise_statement(raise_statement(raise,';'))    -->
        [raise,';'].
raise_statement(raise_statement(raise,N,';')) -->
        [raise],exception_name(N),[';'].
/* 12.1                                                    */
generic_declaration(generic_declaration(G,';'))        -->
        generic_specification(G),[';'].

generic_specification(generic_specification(G,R))      -->
        generic_formal_part(G),
        rest_generic_specification(R).

rest_generic_specification(rest_generic_specification(S))  -->
        subprogram_specification(S);
        package_specification(S).

generic_formal_part(generic_formal_part(generic,G)) -->
        [generic],generic_parameter_declaration_list(G).

generic_parameter_declaration_list(
   generic_parameter_declaration_list(G,L))       -->
        generic_parameter_declaration(G),
        generic_parameter_declaration_list(L).

generic_parameter_declaration_list(
   generic_parameter_declaration_list(''))        --> [].

generic_parameter_declaration(
   generic_parameter_declaration(L,':',M,T,I,';'))     -->
        identifier_list(L),[':'],mode_option(M),
        type_mark(T),initialize_option(I),[';'].

generic_parameter_declaration(
   generic_parameter_declaration(type,I,is,G,';'))     -->
        [type],identifier(I),[is],generic_type_definition(I),[';'].
generic_parameter_declaration(generic_parameter_declaration(P)) -->
        private_type_declaration(P).
generic_parameter_declaration(
   generic_parameter_declaration(with,S,I,';'))        -->
        [with],subprogram_specification(S),
        is_name_option(I),[';'].

mode_option(mode_option(in,out))       --> [in,out].
mode_option(mode_option(M))            --> in_option(M).

is_name_option(is_name_option(is,'<>'))      --> [is,'<','>'].
is_name_option(is_name_option(is,N))         --> [is],a_name(N).
is_name_option(is_name_option('')) --> [].

generic_type_definition(generic_type_definition('(<>)')) -->
        ['(','<','>',')'].

generic_type_definition(generic_type_definition(range,'<>')) -->
        [range,'<','>'].
generic_type_definition(generic_type_definition(digits,'<>')) -->
        [digits,'<','>'].

generic_type_definition(generic_type_definition(delta,'<>')) -->
        [delta,'<','>'].

generic_type_definition(generic_type_definition(A)) -->
        array_type_definition(A);
        access_type_definition(A).
```

```
/* 12.3                                                    */
generic_instantiation(
  generic_instantiation(package,I,is,new,N,G,';')) -->
        [package],identifier(I),[is],
           [new],generic_package_name(N),
                generic_actual_part_option(G),[';'].
generic_instantiation(
        generic_instantiation(function,D,is,new,N,G,';')) -->
        [function],designator(D),[is],
           [new],generic_function_name(N),
                 generic_actual_part_option(G),[';'].
generic_instantiation(generic_instantiation(S,is,new,N,G,';')) -->
        subprogram_specification(S),[is],
           [new],generic_procedure_name(N),
           generic_actual_part_option(G),[';'].
generic_actual_part_option(generic_actual_part_option(G)) -->
         generic_actual_part(G).
generic_actual_part_option(generic_actual_part_option('')) --> [].
generic_package_name(generic_package_name(X))        -->
        expanded_name(X).
generic_function_name(generic_function_name(X))      -->
        expanded_name(X).
generic_procedure_name(generic_procedure_name(X)) -->
         expanded_name(X).
generic_actual_part(generic_actual_part('(',G,L,')'))        -->
        ['('],generic_association(G),
        generic_association_list(L),[')'].
generic_association_list(generic_association_list(G,',',L)) -->
        [','],generic_association(G),generic_association_list(L).
generic_association_list(generic_association_list(''))       --> [].
generic_association(generic_association(G))   -->
        generic_actual_parameter(G).
generic_association(generic_association(F,'=>',A))     -->
        generic_formal_parameter(F),['=','>'],
        generic_actual_parameter(A).
generic_formal_parameter(generic_formal_parameter(X))        -->
        simple_name(X);
        operator_symbol(X).
generic_actual_parameter(generic_actual_parameter(X))        -->
        expression(X).
/* 13.1                                                    */
representation_clause(representation_clause(X))        -->
        length_clause(X);  enumeration_representation_clause(X);
        address_clause(X); record_representation_clause(X).
/* 13.2                                                    */
length_clause(length_clause(for,A,use,E,';')) -->
        [for],attribute(A),[use],simple_expression(E),[';'].
/* 13.3                                                    */
enumeration_representation_clause(
   enumeration_representation_clause(for,N,use,A,';'))         -->
        [for],simple_name(N),[use],aggregate(A),[';'].
```

```
/* 13.4                                        */
record_representation_clause(
  record_representation_clause(for,N,use,record,A,C,end,record,';')
    -->
        [for],simple_name(N),[use],
                [record],alignment_clause_option(A),
                        component_clause_list(C),
                [end,record,';'].

alignment_clause_option(alignment_clause_option(P)) -->
        pragma_list(P).
alignment_clause_option(alignment_clause_option(A,P)) -->
        alignment_clause(A),pragma_list(P).

component_clause_list(component_clause_list(C,L,P)) -->
        component_clause(C),component_clause_list(L),
        pragma_list(P).
component_clause_list(component_clause_list('')) --> [].

component_clause(component_clause(N,at,E,range,R,';'))          -->
        a_name(N),[at],simple_expression(E),[range],range(R),[';'].

alignment_clause(alignment_clause(at,mod,E,';'))        -->
        [at,mod],simple_expression(E),[';'].

/* 13.5                                        */
address_clause(address_clause(for,N,use,at,E,';'))      -->
        [for],simple_name(N),[use,at],simple_expression(E),[';'].

/* 13.8                                                  */
code_statement(code_statement(Q,A,';'))         -->
        a_name(N),an_apostrophe(Q),aggregate(A),[';'].
```

APPENDIX D
YODA USER'S GUIDE

The following instructions explain how to use the prototype debugger YODA.

1. Getting Started

YODA's lexical scanner, parser, semantic analyzer, annotator, and pretty printer are saved as the Prolog program **yodafile**. We will refer to the program unit to be annotated as the "program under analysis."

To generate the symbol table and the YODA-annotated program, call the Prolog predicate **yoda**:

```
?-yoda(SourceFile,PredefFile,Tokens,
        ParseTree,SymbolTable,SymbolFile).
```

Here, the parameters SourceFile, PredefFile, and SymbolFile must be instantiated on the call.

These parameters are used as follows:

SourceFile	The name of the Ada source file containing the program under analysis.
PredefFile	A (possibly empty) list of predefined symbol tables, e.g., symbol tables for with'ed packages and parent program units. The symbol table of the library package STANDARD is implicitly included and, thus, should be omitted from this list.
Tokens	The list of tokens generated by lexical analysis.
ParseTree	The parse tree generated by the parser.
SymbolTable	The binary-tree format of the symbol table generated by the parser for the program under analysis.
SymbolFile	The name of the file that will contain the symbol table generated for the program under analysis.

2. Compilation of Annotated Program

If the program unit under analysis is a main program, then in the annotated version this program unit will be declared as a subprogram called by the main program YODA. Since the specification of an Ada main program is arbitrary, linkage requires that the main program be designated, if ambiguous. Thus, to link the compilation of the annotated program, designate the main program as "YODA" instead of the name of the program under analysis.

When the annotated program is executed, the trace is saved in the file "<program>.trace" where <program> is the name of the main program of the unit under analysis. If a subunit is to be traced, its parent unit must also be traced.

3. Changing Things

If any changes are made to either the prototype debugger or the symbol table for the library package STANDARD, then **yoda** must be initialized before being saved again as a Prolog program. To initialize **yoda**, use the predicate **init**, which has zero arguments. This predicate records the symbol table for the library package STANDARD.

4. Known Bugs

When YODA annotates an Ada program, it converts fixed-point numbers with zero fractions to integers. For example, the Ada assignment statement:

```
duration := 10.0;
```

becomes

```
duration := 10;
```

The resulting annotated program will raise a constraint error during compilation because of unmatched types. This error occurs during lexical analysis because the source program is read as ASCII characters and converted to strings by the **name** predicate. The **name** predicate in C-Prolog converts whole floating-point numbers to integers.

APPENDIX E
ADA TAXI SERVICE PROGRAMS

1. Main Program

```
-------------------------------------------------------------------------
--          Main Task of Taxi Service Program                          --
--The time distances between taxi stops are arbitrary.                  --
--This procedure activates all tasks                                    --
--and sets initial values for taxis and customers.                      --
-------------------------------------------------------------------------
with TEXT_IO; use TEXT_IO;
procedure MAIN is
  type DRIVER_NAME is       (Beth, Byron, Carol);
  type PLACE_NAME is     (LAX, BONAVENTURE, UCLA, USC,
                           DISNEYLAND, COLISEUM, PASADENA, HOME);
  type CAR_CODE is       (CAR_1, CAR_23, CAR_54);
  type DISPATCHER_NAME is (Stott, Ada);
  type CUSTOMER_NAME is     (Anne, Tom, Shiang, Deborah, Jim, Dave);
  package DRIVER_IO is new ENUMERATION_IO (ENUM=>DRIVER_NAME);
  use DRIVER_IO;
  package PLACE_IO is new ENUMERATION_IO (ENUM=>PLACE_NAME);
  use PLACE_IO;
  package CAR_CODE_IO is new ENUMERATION_IO (ENUM=>CAR_CODE);
  use CAR_CODE_IO;
  package DISPATCHER_IO is new ENUMERATION_IO (ENUM=>DISPATCHER_NAME);
  use DISPATCHER_IO;
  package CUSTOMER_IO is new ENUMERATION_IO (ENUM => CUSTOMER_NAME);
  use CUSTOMER_IO;
  POSITION : INTEGER;                    -- used as an index
  subtype MONEY is DURATION delta 0.01; --same range as DURATION
  PAY    : array(DRIVER_NAME) of MONEY :=
          (DRIVER_NAME'FIRST .. DRIVER_NAME'LAST=>0.00); --shared variable

  DISTANCE: constant array(PLACE_NAME, PLACE_NAME) of DURATION:=
                (LAX =>          (LAX | HOME              => 0.0,
                                  DISNEYLAND              => 20.0,
                                  BONAVENTURE | PASADENA |
                                  USC | COLISEUM | UCLA   => 10.0),

                 BONAVENTURE =>  (BONAVENTURE | HOME      => 0.0,
                                  DISNEYLAND              => 20.0,
                                  UCLA | LAX              => 10.0,
                                  USC | COLISEUM |
                                  PASADENA                => 5.0),

                 UCLA =>         (UCLA | HOME             => 0.0,
                                  DISNEYLAND              => 20.0,
                                  LAX | BONAVENTURE | USC |
                                  COLISEUM | PASADENA     => 10.0),

                 USC =>          (USC | HOME              => 0.0,
                                  UCLA | LAX              => 10.0,
                                  DISNEYLAND              => 20.0,
                                  BONAVENTURE | COLISEUM |
                                  PASADENA                => 5.0),

                 DISNEYLAND =>   (DISNEYLAND | HOME       => 0.0,
                                  LAX | BONAVENTURE |
                                  UCLA | PASADENA |
                                  COLISEUM | USC          => 20.0),

                 COLISEUM    =>  (COLISEUM | HOME=>  0.0,
                                  UCLA | LAX      => 10.0,
                                  DISNEYLAND      => 20.0,
                                  BONAVENTURE | USC |
                                  PASADENA        =>  5.0),
```

```
                PASADENA    =>  (PASADENA | HOME          =>  0.0,
                                 UCLA | LAX               => 10.0,
                                 DISNEYLAND               => 20.0,
                                 BONAVENTURE | USC |
                                 COLISEUM                 =>  5.0),
                HOME        =>  (BONAVENTURE | HOME | UCLA |
                                 COLISEUM | PASADENA |
                                 DISNEYLAND | USC | LAX =>  0.0)
              );

  task type TAXI is
    entry DRIVER_IS              (NAME              : out DRIVER_NAME);
    entry LOCATION_IS            (PLACE             : out PLACE_NAME);
    entry SET_DRIVER             (NAME              : in DRIVER_NAME;
                                    AUTHORITY         : in DISPATCHER_NAME);
    entry SET_SERIAL_NUMBER         (IDENTITY          : in CAR_CODE);
    entry SET_LOCATION           (PLACE             : in PLACE_NAME);
    entry TAKE_RIDER_FROM (CUSTOMER_LOCATION : in PLACE_NAME;
                                    NAME              : in CUSTOMER_NAME;
                                    PLACE             : in PLACE_NAME);
    entry FARE_PAID              (AMOUNT            : in MONEY);
  end TAXI;
  type FLEET is array (CAR_CODE) of TAXI;
  YELLOW_CAB: FLEET;

  task type DISPATCHER is
    entry SET_BOSS_ID      (NAME    : in DISPATCHER_NAME);
  end DISPATCHER;
  type FLEET_DISPATCHER is array (DISPATCHER_NAME) of DISPATCHER;
  YELLOW_CAB_DISPATCHER : FLEET_DISPATCHER;

  task SWITCH_BOARD is
    entry RECEIVE_CALL (NAME: in CUSTOMER_NAME;  PLACE: in PLACE_NAME);
    entry CONNECT    (NAME: out CUSTOMER_NAME; PLACE: out PLACE_NAME);
    entry STOP_RECEIVING;
  end SWITCH_BOARD;

  task type CUSTOMER is
    entry SET_IDENTITY           (NAME              : in CUSTOMER_NAME);
    entry SET_LOCATION           (PLACE             : in PLACE_NAME);
    entry TAKE_CAB               (CODE              : in CAR_CODE);
  end CUSTOMER;
  type CAB_RIDERS is array(CUSTOMER_NAME) of CUSTOMER;
  YELLOW_CAB_CUSTOMER : CAB_RIDERS;

  task ASK is
    entry NEXT_DESTINATION      (IDENTITY        : in CUSTOMER_NAME;
                                   LOCATION        : in PLACE_NAME;
                                   DESTINATION     : out PLACE_NAME);
  end ASK;

  task body TAXI is separate;    --compile task bodies separately
  task body DISPATCHER is separate;
  task body SWITCH_BOARD is separate;
  task body CUSTOMER is separate;
  task body ASK is separate;

begin                                   --start execution, activate tasks!
  for INDEX in CAR_CODE loop
    YELLOW_CAB(INDEX).SET_SERIAL_NUMBER(INDEX);
    YELLOW_CAB(INDEX).SET_LOCATION(LAX);
  end loop;
  for INDEX in CUSTOMER_NAME loop
    YELLOW_CAB_CUSTOMER(INDEX).SET_IDENTITY(INDEX);
    POSITION := CUSTOMER_NAME'POS(INDEX) mod PLACE_NAME'POS(HOME);
    YELLOW_CAB_CUSTOMER(INDEX).SET_LOCATION
                (PLACE_NAME'VAL(POSITION));
  end loop;
end MAIN;
```

2. Swithboard Task

```
-------------------------------------------------------------------------
--              body of task SWITCH_BOARD                              --
--The switchboard acts as an "agent" task to connect a "customer"       --
--with a "dispatcher."  The customer (user) need know nothing           --
--about the dispatcher(server).  The switchboard is "available" to      --
--accept the next entry call from a customer as soon as a dispatcher    --
--has been assigned to the previous customer.                           --
--The switchboard also handles waking up dispatchers, sending them home,--
--and publishing the cab drivers' intake.                               --
-------------------------------------------------------------------------
  separate(MAIN)
  task body SWITCH_BOARD is
    package MONEY_IO is new FIXED_IO (NUM => MONEY);
    use MONEY_IO;
    CUSTOMER_WAITING       : BOOLEAN :=FALSE;
    PASSENGER              : CUSTOMER_NAME;
    LOCATION               : PLACE_NAME;
    NO_MORE_CUSTOMERS : BOOLEAN := FALSE;

  begin
    put_line("Yellow Cab Taxi at your service.");
    new_line;
    for INDEX in DISPATCHER_NAME loop  --assign dispatchers
      YELLOW_CAB_DISPATCHER(INDEX).SET_BOSS_ID(INDEX);
    end loop;
    --
    loop
      select

        when not CUSTOMER_WAITING =>
          accept RECEIVE_CALL  (NAME   : in CUSTOMER_NAME;
                                PLACE  : in PLACE_NAME) do
            PASSENGER := NAME;
            LOCATION  := PLACE;
            CUSTOMER_WAITING := TRUE;
            put_line ("The switchboard has received a call from " &
                      CUSTOMER_NAME'IMAGE(PASSENGER)     &
                      " at " & PLACE_NAME'IMAGE(LOCATION) & ".");
          end RECEIVE_CALL;

      or

        when CUSTOMER_WAITING  =>
          accept CONNECT     (NAME   : out CUSTOMER_NAME;
                              PLACE  : out PLACE_NAME) do
            NAME         := PASSENGER;
            PLACE        := LOCATION;
            CUSTOMER_WAITING := FALSE;
          end CONNECT;

      or

          accept STOP_RECEIVING do  --publish payroll
            NO_MORE_CUSTOMERS := TRUE;
            new_line;
            for I in DRIVER_NAME loop
              put_line(DRIVER_NAME'IMAGE(I) & " has earned $");
              put(PAY(I));
            end loop;
          end STOP_RECEIVING;

      or
```

```
      when NO_MORE_CUSTOMERS =>  --send dispatchers home
        accept CONNECT     (NAME    : out CUSTOMER_NAME;
                            PLACE   : out PLACE_NAME) do
          NAME          := PASSENGER;
          PLACE         := HOME;
        end CONNECT;

    or

        terminate;

    end select;
  end loop;
end SWITCH_BOARD;
```

3. Dispatcher Task

```
-------------------------------------------------------------------------------
--              body of dispatcher task                                      --
--The dispatcher tells the customer the code of the taxi that has been       --
--dispatched, and then services the next customer.                           --
--Initially, one dispatcher assigns all the cab drivers to cars,            --
--while the other dispatchers are kept waiting.                             --
--The algorithm for mapping cabs to customers is as follows:                --
--       Take the first cab within "5 seconds" from                          --
--          the customer's location.                                         --
--       If none that close, take Cab #1.                                    --
--On returning from the entry call CONNECT,                                  --
--if all customers are home, then terminate dispatcher                       --
-------------------------------------------------------------------------------
  separate(MAIN)
  task body DISPATCHER is
    ID                              : CUSTOMER_NAME;
    CUSTOMER_LOCATION                 : PLACE_NAME;
    CAB_LOCATION                 : PLACE_NAME;
    CAR_CALLED                        : CAR_CODE;
    BOSS_ID                           : DISPATCHER_NAME;
  begin
    accept SET_BOSS_ID (NAME : in DISPATCHER_NAME) do
      BOSS_ID:=NAME;
    end SET_BOSS_ID;

    if BOSS_ID = DISPATCHER_NAME'FIRST then
      for INDEX in CAR_CODE loop
        YELLOW_CAB(INDEX).SET_DRIVER
                  (NAME      => DRIVER_NAME'VAL(CAR_CODE'POS(INDEX)),
                   AUTHORITY => BOSS_ID);
      end loop;
    else
      delay 50.0;  --keep other dispatchers waiting
    end if;

    loop
      SWITCH_BOARD.CONNECT (ID, CUSTOMER_LOCATION);
      exit when CUSTOMER_LOCATION = HOME;
      CAR_CALLED:=CAR_CODE'FIRST;
      for INDEX in CAR_CODE loop
          YELLOW_CAB(INDEX).LOCATION_IS(CAB_LOCATION);
          if DISTANCE(CAB_LOCATION, CUSTOMER_LOCATION) <= 5.00 then
            CAR_CALLED := INDEX;
            exit;
          end if;
      end loop;

      YELLOW_CAB_CUSTOMER(ID).TAKE_CAB (CODE => CAR_CALLED);
      put_line(DISPATCHER_NAME'IMAGE(BOSS_ID) &
                " is dispatching " & CAR_CODE'IMAGE(CAR_CALLED)&
                " for " & CUSTOMER_NAME'IMAGE(ID) & " at " &
                PLACE_NAME'IMAGE(CUSTOMER_LOCATION) & ".");
  end loop;
end DISPATCHER;
```

4. Customer Task

```
-------------------------------------------------------------------------
--               customer task body                                    --
--The customer calls for a cab, waits for cab, rides to the destination --
--and pays the cab.                                                     --
--The cost of the trip is computed from the time distance.              --
-------------------------------------------------------------------------
  separate (MAIN)
  task body CUSTOMER is
    IDENTITY             : CUSTOMER_NAME;
    LOCATION             : PLACE_NAME;
    DESTINATION          : PLACE_NAME := HOME;
    CAR                  : CAR_CODE;
    CAB_DRIVER              : DRIVER_NAME;
    CASH                 : MONEY;         --Rate for this trip

  begin
    --Initialize name of customer and his/her starting location
    accept SET_IDENTITY (NAME: in CUSTOMER_NAME) do
      IDENTITY :=NAME;
    end SET_IDENTITY;
    accept SET_LOCATION (PLACE : in PLACE_NAME) do
      LOCATION := PLACE;
    end SET_LOCATION;

    loop
      ASK.NEXT_DESTINATION(IDENTITY, LOCATION, DESTINATION);
      exit when DESTINATION=HOME;
      put_line (CUSTOMER_NAME'IMAGE(IDENTITY)&
                " is calling a taxi to go to "  &
                PLACE_NAME'IMAGE(DESTINATION) & ".  ");
      SWITCH_BOARD.RECEIVE_CALL        (NAME => IDENTITY, PLACE => LOCATION);
      accept TAKE_CAB (CODE: in CAR_CODE) do
        CAR := CODE;
      end TAKE_CAB;

      YELLOW_CAB(CAR).TAKE_RIDER_FROM
                        (CUSTOMER_LOCATION => LOCATION,
                         NAME                    => IDENTITY,
                         PLACE                   => DESTINATION);
      put_line (CUSTOMER_NAME'IMAGE(IDENTITY) & " has arrived at "
                & PLACE_NAME'IMAGE(DESTINATION) & ".");
      YELLOW_CAB(CAR).DRIVER_IS(NAME    => CAB_DRIVER);
      CASH:=DISTANCE(LOCATION,DESTINATION) * 2;
      YELLOW_CAB(CAR).FARE_PAID(AMOUNT => CASH);
      put_line (CUSTOMER_NAME'IMAGE(IDENTITY) & " has paid " &
                DRIVER_NAME'IMAGE(CAB_DRIVER) & " for the cab ride.");
      LOCATION := DESTINATION;
    end loop;
  end CUSTOMER;
```

5. Ask Task (Customer Request for Service)

```
--------------------------------------------------------------------------
--                  the task body for ASK                                --
-- Receives calls from CUSTOMER tasks for customer's next destination    --
--A single task is used for I/O of all CUSTOMER tasks to ensure          --
--that the prompt for input immediately precedes a read.                --
--An exception is raised for invalid DESTINATION  or                    --
--if DESTINATION = LOCATION                                             --
--If invalid data is entered on three successive attempts,              --
--the DESTINATION is set to HOME.                                       --
--When all customer tasks will be terminated (i.e. all have a          --
--destination=HOME) ASK notifies the switchboard and terminates.        --
--------------------------------------------------------------------------
separate (MAIN)
task body ASK is
   NUM_ERRORS              : INTEGER := 0; --# errors
   NUMBER_HOME             : INTEGER := 0; --# CUSTOMER tasks terminated
   NOT_MOVING              : exception;
   PLACE_REQUESTED         : PLACE_NAME;
begin
   loop
      select

         accept NEXT_DESTINATION(IDENTITY:      in CUSTOMER_NAME;
                                 LOCATION:        in PLACE_NAME;
                                 DESTINATION:     out PLACE_NAME) do
            NEW_LINE;
            put_line(CUSTOMER_NAME'IMAGE(IDENTITY) &
               " is at " & PLACE_NAME'IMAGE(LOCATION) & ".");
   ENTER:loop
            begin
               put_line ("Enter HOME (to stop) or next destination for "
                 & CUSTOMER_NAME'IMAGE(IDENTITY) & ":  ");
               GET (PLACE_REQUESTED);
               put (PLACE_NAME'IMAGE(PLACE_REQUESTED));
               if PLACE_REQUESTED = LOCATION then
                  raise NOT_MOVING;
               end if;
               DESTINATION := PLACE_REQUESTED;
               exit ENTER;
            exception
               when NOT_MOVING =>
                  put_line (CUSTOMER_NAME'IMAGE(IDENTITY)&" is already at "
                        & PLACE_NAME'IMAGE(LOCATION) & ".  Try again.");
               when DATA_ERROR =>
                  NUM_ERRORS := NUM_ERRORS+1;
                  if NUM_ERRORS > 3 then
                     DESTINATION := HOME;
                     exit ENTER;
                  else
                     put_line ("Garbage!  Try another destination for "
                        & CUSTOMER_NAME'IMAGE(IDENTITY) & ".");
                  end if;
            end;
         end loop ENTER;

         if DESTINATION=HOME then
            NUMBER_HOME := NUMBER_HOME + 1;
            put_line(CUSTOMER_NAME'IMAGE(IDENTITY) & " is home.  ");
            if NUMBER_HOME = YELLOW_CAB_CUSTOMER'LENGTH then
               SWITCH_BOARD.STOP_RECEIVING;--all customers home
            end if;
         end if;
      end NEXT_DESTINATION;

   or
```

```
        terminate;
      end select;
    end loop;
end ASK;
```

6. Taxi Task

```
--------------------------------------------------------------------------
--              body of taxi task                                       --
--Taxi receives calls from customer and from dispatcher                  --
--Assumption: taxi can give dispatcher its location while                --
--it is waiting for customer to pay fare, but cannot proceed             --
--to pick up next customer until fare is paid.                           --
--------------------------------------------------------------------------
  separate (MAIN)
  task body TAXI is
    CURRENT_DRIVER          : DRIVER_NAME;
    PERMANENT_SERIAL_NUMBER   : CAR_CODE;
    CURRENT_LOCATION              : PLACE_NAME;
    WAITING_FOR_FARE          : BOOLEAN := FALSE;

  begin
    accept SET_SERIAL_NUMBER(IDENTITY : in CAR_CODE) do
      PERMANENT_SERIAL_NUMBER := IDENTITY;
    end SET_SERIAL_NUMBER;

    accept SET_LOCATION(PLACE : in PLACE_NAME) do
      CURRENT_LOCATION := PLACE;
    end SET_LOCATION;

    accept SET_DRIVER(NAME      : in DRIVER_NAME;
                      AUTHORITY : in DISPATCHER_NAME) do
      CURRENT_DRIVER := NAME;
    end SET_DRIVER;

    put_line(CAR_CODE'IMAGE (PERMANENT_SERIAL_NUMBER) &
      " driven by " & DRIVER_NAME'IMAGE(CURRENT_DRIVER) &
      " is waiting at " & PLACE_NAME'IMAGE(CURRENT_LOCATION)&".");
    loop

      select

        when not WAITING_FOR_FARE =>
        accept TAKE_RIDER_FROM     (CUSTOMER_LOCATION : in PLACE_NAME;
                                    NAME              : in CUSTOMER_NAME;
                                    PLACE             : in PLACE_NAME) do
          if CUSTOMER_LOCATION /= CURRENT_LOCATION then
             put_line (CAR_CODE'IMAGE (PERMANENT_SERIAL_NUMBER) &
             " is en route to pick up "&CUSTOMER_NAME'IMAGE(NAME)&".");
             delay DISTANCE(CURRENT_LOCATION,CUSTOMER_LOCATION);
             CURRENT_LOCATION := CUSTOMER_LOCATION;
          end if;
          delay DISTANCE(CUSTOMER_LOCATION,PLACE);
          CURRENT_LOCATION := PLACE;
          WAITING_FOR_FARE := TRUE;
        end TAKE_RIDER_FROM;

      or

        accept DRIVER_IS(NAME : out DRIVER_NAME) do
          NAME := CURRENT_DRIVER;
        end DRIVER_IS;

      or

        accept LOCATION_IS(PLACE : out PLACE_NAME) do
          PLACE := CURRENT_LOCATION;
        end LOCATION_IS;

      or

        when WAITING_FOR_FARE =>
        accept FARE_PAID (AMOUNT: in MONEY) do
          PAY(CURRENT_DRIVER) := PAY(CURRENT_DRIVER) + AMOUNT;
          WAITING_FOR_FARE := FALSE;
        end FARE_PAID;
```

```
    or
      terminate;
    end select;
  end loop;
end TAXI;
```

APPENDIX F
SYMBOL TABLES OF ADA TAXI SERVICE PROGRAM

Table F-1: Symbol Table for Taxi Service Program

Main Subprogram

```
symbol(ada,[main],enumeration_literal(scalar)).
symbol(amount,[fare_paid,taxi,main],
    formal_parameter(real_type_definition,money,[main])).
symbol(anne,[main],enumeration_literal(scalar)).
symbol(ask,[main],object_name(task_type,anonymous,[main])).
symbol(authority,[set_driver,taxi,main],
    formal_parameter(enumeration_type_definition,dispatcher_name,[main])).
symbol(beth,[main],enumeration_literal(scalar)).
symbol(bonaventure,[main],enumeration_literal(scalar)).
symbol(byron,[main],enumeration_literal(scalar)).
symbol(cab_riders,[main],type_name(array_type_definition)).
symbol(car_1,[main],enumeration_literal(scalar)).
symbol(car_23,[main],enumeration_literal(scalar)).
symbol(car_54,[main],enumeration_literal(scalar)).
symbol(car_code,[main],type_name(enumeration_type_definition)).
symbol(car_code_io,[main],generic_package_instantiation).
symbol(carol,[main],enumeration_literal(scalar)).
symbol(code,[take_cab,customer,main],
    formal_parameter(enumeration_type_definition,car_code,[main])).
symbol(coliseum,[main],enumeration_literal(scalar)).
symbol(connect,[switch_board,main],entry_name).
symbol(customer,[main],object_name(task_type,anonymous,[main])).
symbol(customer_io,[main],generic_package_instantiation).
symbol(customer_location,[take_rider_from,taxi,main],
    formal_parameter(enumeration_type_definition,place_name,[main])).
symbol(customer_name,[main],type_name(enumeration_type_definition)).
symbol(dave,[main],enumeration_literal(scalar)).
symbol(deborah,[main],enumeration_literal(scalar)).
symbol(destination,[next_destination,ask,main],
    formal_parameter(enumeration_type_definition,place_name,[main])).
symbol(disneyland,[main],enumeration_literal(scalar)).
symbol(dispatcher,[main],object_name(task_type,anonymous,[main])).
symbol(dispatcher_io,[main],generic_package_instantiation).
symbol(dispatcher_name,[main],type_name(enumeration_type_definition)).
symbol(distance,[main],
        constant_object_name(array_type_definition,duration,[])).
```

Table F-1: Symbol Table for Taxi Service Program, continued

Main Subprogram, continued

```
symbol(driver_io,[main],generic_package_instantiation).
symbol(driver_is,[taxi,main],entry_name).
symbol(driver_name,[main],type_name(enumeration_type_definition)).
symbol(fare_paid,[taxi,main],entry_name).
symbol(fleet,[main],type_name(array_type_definition)).
symbol(fleet_dispatcher,[main],type_name(array_type_definition)).
symbol(home,[main],enumeration_literal(scalar)).
symbol(identity,[set_serial_number,taxi,main],
    formal_parameter(enumeration_type_definition,car_code,[main])).
symbol(identity,[next_destination,ask,main],
    formal_parameter(enumeration_type_definition,customer_name,[main])).
symbol(index,[main],loop_parameter_name).
symbol(jim,[main],enumeration_literal(scalar)).
symbol(lax,[main],enumeration_literal(scalar)).
symbol(location,[next_destination,ask,main],
    formal_parameter(enumeration_type_definition,place_name,[main])).
symbol(location_is,[taxi,main],entry_name).
symbol(main,[],subprogram_name).
symbol(money,[main],subtype_name(real_type_definition)).
symbol(name,[driver_is,taxi,main],
   formal_parameter(enumeration_type_definition,driver_name,[main])).
symbol(name,[set_driver,taxi,main],
    formal_parameter(enumeration_type_definition,driver_name,[main])).
symbol(name,[take_rider_from,taxi,main],
    formal_parameter(enumeration_type_definition,customer_name,[main])).
symbol(name,[set_boss_id,dispatcher,main],
    formal_parameter(enumeration_type_definition,dispatcher_name,[main])).
symbol(name,[receive_call,switch_board,main],
    formal_parameter(enumeration_type_definition,customer_name,[main])).
symbol(name,[connect,switch_board,main],
    formal_parameter(enumeration_type_definition,customer_name,[main])).
symbol(name,[set_identity,customer,main],
    formal_parameter(enumeration_type_definition,customer_name,[main])).
symbol(next_destination,[ask,main],entry_name).
symbol(pasadena,[main],enumeration_literal(scalar)).
symbol(pay,[main],object_name(array_type_definition,money,[])).
symbol(place,[location_is,taxi,main],
    formal_parameter(enumeration_type_definition,place_name,[main])).
symbol(place,[set_location,taxi,main],
    formal_parameter(enumeration_type_definition,place_name,[main])).
symbol(place,[take_rider_from,taxi,main],
    formal_parameter(enumeration_type_definition,place_name,[main])).
symbol(place,[receive_call,switch_board,main],
    formal_parameter(enumeration_type_definition,place_name,[main])).
symbol(place,[connect,switch_board,main],
    formal_parameter(enumeration_type_definition,place_name,[main])).
symbol(place,[set_location,customer,main],
    formal_parameter(enumeration_type_definition,place_name,[main])).
```

Table F-1: Symbol Table for Taxi Service Program, continued

Main Subprogram, continued

```
symbol(place_io,[main],generic_package_instantiation).
symbol(place_name,[main],type_name(enumeration_type_definition)).
symbol(position,[main],object_name(integer_type_definition,integer,[])).
symbol(receive_call,[switch_board,main],entry_name).
symbol(set_boss_id,[dispatcher,main],entry_name).
symbol(set_driver,[taxi,main],entry_name).
symbol(set_identity,[customer,main],entry_name).
symbol(set_location,[taxi,main],entry_name).
symbol(set_location,[customer,main],entry_name).
symbol(set_serial_number,[taxi,main],entry_name).
symbol(shiang,[main],enumeration_literal(scalar)).
symbol(stop_receiving,[switch_board,main],entry_name).
symbol(stott,[main],enumeration_literal(scalar)).
symbol(switch_board,[main],object_name(task_type,anonymous,[main])).
symbol(take_cab,[customer,main],entry_name).
symbol(take_rider_from,[taxi,main],entry_name).
symbol(taxi,[main],object_name(task_type,anonymous,[main])).
symbol(tom,[main],enumeration_literal(scalar)).
symbol(ucla,[main],enumeration_literal(scalar)).
symbol(usc,[main],enumeration_literal(scalar)).
symbol(yellow_cab,[main],object_name(array_type_definition,fleet,[main])).
symbol(yellow_cab_customer,[main],
        object_name(array_type_definition,cab_riders,[main])).
symbol(yellow_cab_dispatcher,[main],
        object_name(array_type_definition,fleet_dispatcher,[main])).
```

Body of Switchboard Task

```
symbol(customer_waiting,[switch_board,main],
        object_name(enumeration_type_definition,boolean,[])).
symbol(i,[stop_receiving,switch_board,main],loop_parameter_name).
symbol(index,[switch_board,main],loop_parameter_name).
symbol(location,[switch_board,main],
        object_name(enumeration_type_definition,place_name,[main])).
symbol(money_io,[switch_board,main],generic_package_instantiation).
symbol(name,[receive_call,switch_board,main],
        formal_parameter(enumeration_type_definition,customer_name,[main])).
symbol(name,[connect,switch_board,main],
        formal_parameter(enumeration_type_definition,customer_name,[main])).
symbol(no_more_customers,[switch_board,main],
        object_name(enumeration_type_definition,boolean,[])).
symbol(passenger,[switch_board,main],
        object_name(enumeration_type_definition,customer_name,[main])).
symbol(place,[receive_call,switch_board,main],
        formal_parameter(enumeration_type_definition,place_name,[main])).
symbol(place,[connect,switch_board,main],
        formal_parameter(enumeration_type_definition,place_name,[main])).
```

Table F-1: Symbol Table for Taxi Service Program, continued

Body of Dispatcher Task

```
symbol(boss_id,[dispatcher,main],
        object_name(enumeration_type_definition,dispatcher_name,[main])).
symbol(cab_location,[dispatcher,main],
        object_name(enumeration_type_definition,place_name,[main])).
symbol(car_called,[dispatcher,main],
        object_name(enumeration_type_definition,car_code,[main])).
symbol(customer_location,[dispatcher,main],
        object_name(enumeration_type_definition,place_name,[main])).
symbol(id,[dispatcher,main],
        object_name(enumeration_type_definition,customer_name,[main])).
symbol(index,[dispatcher,main],loop_parameter_name).
symbol(name,[set_boss_id,dispatcher,main],
    formal_parameter(enumeration_type_definition,dispatcher_name,[main])).
```

Body of Ask Task

```
symbol(block_name1,[next_destination,ask,main],block_name).
symbol(destination,[next_destination,ask,main],
        formal_parameter(enumeration_type_definition,place_name,[main])).
symbol(identity,[next_destination,ask,main],
        formal_parameter(enumeration_type_definition,customer_name,[main])).
symbol(location,[next_destination,ask,main],
        formal_parameter(enumeration_type_definition,place_name,[main])).
symbol(not_moving,[ask,main],exception_name).
symbol(num_errors,[ask,main],
        object_name(integer_type_definition,integer,[])).
symbol(number_home,[ask,main],
        object_name(integer_type_definition,integer,[])).
symbol(place_requested,[ask,main],
        object_name(enumeration_type_definition,place_name,[main])).
```

Table F-1: Symbol Table for Taxi Service Program, concluded

Body of Customer Task

```
symbol(cab_driver,[customer,main],
        object_name(enumeration_type_definition,driver_name,[main])).
symbol(car,[customer,main],
        object_name(enumeration_type_definition,car_code,[main])).
symbol(cash,[customer,main],
        object_name(real_type_definition,money,[main])).
symbol(code,[take_cab,customer,main],
        formal_parameter(enumeration_type_definition,car_code,[main])).
symbol(destination,[customer,main],
        object_name(enumeration_type_definition,place_name,[main])).
symbol(identity,[customer,main],
        object_name(enumeration_type_definition,customer_name,[main])).
symbol(location,[customer,main],
        object_name(enumeration_type_definition,place_name,[main])).
symbol(name,[set_identity,customer,main],
        formal_parameter(enumeration_type_definition,customer_name,[main])).
symbol(place,[set_location,customer,main],
        formal_parameter(enumeration_type_definition,place_name,[main])).
```

Body of Taxi Task

```
symbol(amount,[fare_paid,taxi,main],
    formal_parameter(real_type_definition,money,[main])).
symbol(authority,[set_driver,taxi,main],
    formal_parameter(enumeration_type_definition,dispatcher_name,[main])).
symbol(current_driver,[taxi,main],
        object_name(enumeration_type_definition,driver_name,[main])).
symbol(current_location,[taxi,main],
        object_name(enumeration_type_definition,place_name,[main])).
symbol(customer_location,[take_rider_from,taxi,main],
    formal_parameter(enumeration_type_definition,place_name,[main])).
symbol(identity,[set_serial_number,taxi,main],
    formal_parameter(enumeration_type_definition,car_code,[main])).
symbol(name,[set_driver,taxi,main],
    formal_parameter(enumeration_type_definition,driver_name,[main])).
symbol(name,[take_rider_from,taxi,main],
    formal_parameter(enumeration_type_definition,customer_name,[main])).
symbol(name,[driver_is,taxi,main],
    formal_parameter(enumeration_type_definition,driver_name,[main])).
symbol(permanent_serial_number,[taxi,main],
        object_name(enumeration_type_definition,car_code,[main])).
symbol(place,[set_location,taxi,main],
    formal_parameter(enumeration_type_definition,place_name,[main])).
symbol(place,[take_rider_from,taxi,main],
    formal_parameter(enumeration_type_definition,place_name,[main])).
symbol(place,[location_is,taxi,main],
    formal_parameter(enumeration_type_definition,place_name,[main])).
symbol(waiting_for_fare,[taxi,main],
        object_name(enumeration_type_definition,boolean,[])).
```