AdaTAD - A Debugger for the Ada Multi-Task Environment

by

Robert Gaffney Fainter

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy
in
Computer Science and Applications

APPROVED:

Timothy E. Lindquist, Chairman	George W. Gorsline
John A. N. Lee	Ezra A. Brown
Joel S. Greenstein	Robert C. Williges

July, 1985

Blacksburg, Virginia

AdaTAD - A Debugger for the Ada Multi-Task Environment

by

Robert Gaffney Fainter

Timothy E. Lindquist, Chairman

Computer Science and Applications

(ABSTRACT)

In a society that is increasingly dependent upon computing machinery, the issues associated with the correct functioning of that machinery are of crucial interest. The consequences of erroneous behavior of computers are dire with the worst case scenario being, conceivably, global thermonuclear war. Therefore, development of procedures and tools which can be used to increase the confidence of the correctness of the software that controls the world's computers is of vital importance.

The Department of Defense (DoD) is in the process of adopting a standard computer language for the development of software. This language is called Ada[1]. One of the major features of Ada is that it supports concurrent programming via its "task" compilation unit. There are not, however, any automated tools to aid in locating errors in the tasks.

---

[1] Ada is a registered trademark of the Department of Defense - Ada Joint Program Office

The design for such a tool is presented. The tool is named AdaTAD and is a debugger for programs written in Ada. The features of AdaTAD are specific to the problems of concurrent programming.

The requirements of AdaTAD are derived from the literature. AdaTAD is, however, a unique tool designed using Ada as a program description language. When AdaTAD is implemented in Ada it becomes portable among all environments which support the Ada language. This offers the advantage that a single debugger is portable to many different machine architectures. Therefore, separate debuggers are not necessary for each implementation of Ada.

Moreover, since AdaTAD is designed to allow debugging of tasks, AdaTAD will also support debugging in a distributed environment. That means that, if the tasks of a user's program are running on different computers in a distributed environment, the user is still able to use AdaTAD to debug the tasks as a single program. This feature is unique among automated debuggers.

After the design is presented, several examples are offered to explain the operation of AdaTAD and to show that AdaTAD is useful in revealing the location of errors specific to concurrent programming.

ACKNOWLEDGEMENTS

In The American Adventure pavilion at Walt Disney's Epcot Center, there is a quote from Wilma Rudolph, a former Olympic track star. Her message is "Nothing of any value is ever accomplished alone. There is always someone who has helped us." It is now my pleasure to thank those who have helped me in this project, both directly and indirectly.

The most important person to me in this or any other effort is my devoted and loving wife, . Quite simply, her encouragement is the reason that this project reached fruition. If she had not supported me, I would have quit this project before it ever truly got started.

My parents, , sacrificed a great deal to start my education. I want to thank them for their patience and encouragement. But mostly, I want to thank them for being my Mom and Dad.

Drs. Tim Lindquist, George Gorsline, JAN Lee, Bud Brown, Joel Greenstein and Bob Williges are my advisory committee. I would like to thank them for their technical and educational expertise. Their patience and continuing good nature is also greatly appreciated. My special thanks go to Tim Lindquist and JAN Lee who put their plans for a Commercial Pilot license on hold while I completed this project. Joel Greenstein also deserves special thanks because his joining

the committee late in the work caused him to have to catch up.

The faculty of the Human Factors Lab of the Industrial Engineering and Operations Research Department, Dr. Bob Williges, and Dr. Joel Greenstein, played an important role in this project. They employed me for the last two years of this work and though there was much work to be done in their lab, they made sure that I had time to work on this project each day.

and were my colleagues for the development of the GENIE project (described later in this document). That was the project which sparked my interest in the problems of program concurrency. This dissertation grew from that interest.

, who is the system manager for the DEC VAX 11/780 on which the Ada programming was done, was most helpful in providing me with the necessary computing resources. He kept the Ada compiler running when no one but myself was using it.

Finally, there are people who, while they did not aid directly in this endeavor, provided friendship and, in general, made life more pleasant and stimulating. I list some of them here because I consider them my friends and because they deserve recognition. There is no particular order to this list. ,

, ,

TABLE OF CONTENTS

1.0 Introduction - The Problem of Correctness 1

1.1 Aspects of Achieving Correctness 2

1.2 Problem Statement and a Proposed Solution 4

2.0 Review of Pertinent Literature 7

2.1 The Literature of Program Correctness 7

2.2 Program Testing 9

2.3 Proof versus Testing 10

2.4 Temporal Logic 11

2.5 Automatic Program Debugging 16

2.6 Program Concurrency and Ada Tasks 17

2.6.1 Concurrency In Ada 18

3.0 Requirements of a Task Debugger 28

3.1 A Basis in the Literature 28

3.2 User Interface 31

3.3 Temporal Logic 32

4.0 AdaTAD - A Multi-Task Debugger 36

4.1 The AdaTAD System 37

4.1.1 The Compiler 37

4.1.2 Object Module Format 38

4.1.2.1 Statement Prologue 40

4.1.2.2 Argument Descriptor List 44
4.1.2.3 Code Modification 45
4.1.3 The Linker 48
4.2 The Overall Structure of AdaTAD 51
4.3 User Interface 54
4.3.1 The Command Language 55
4.3.1.1 Description of AdaTAD commands 59
4.3.2 The Display 65
4.3.2.1 Execution Information Pane 70
4.3.2.2 Data Information Pane 74
4.3.2.3 Source Code Context Pane 74
4.3.2.4 Task Output Pane 75
4.3.2.5 Zooming 75
4.3.3 A Sample Debugging Session 77
4.4 Major Module Functionality 89
4.4.1 Logical Processors 89
4.4.1.1 Entry Receive_user_command 91
4.4.1.2 Entry Receive_rendezvous request 93
4.4.1.3 Entry Receive_rendezvous_completion . . . 94
4.4.1.4 Task Executor 94
4.4.1.5 Task Transmitter 95
4.4.1.6 Task Execution_area_monitor 95
4.4.2 Coordinator 97
4.4.2.1 Rendezvous_request 99
4.4.2.2 Rendezvous_begin 99
4.4.2.3 Rendezvous_completion 100

4.4.2.4 OS_request 100
4.4.2.5 IO_driver_interrupt 101
4.4.2.6 Command_in 101
4.4.2.7 DB_update 101
4.4.3 Data Base Monitor 102
4.4.3.1 Hold 103
4.4.3.2 Release 103
4.4.3.3 Done 103
4.4.4 Terminal Communication 103
4.4.4.1 From_terminal 105
4.4.4.2 From_coord 106
4.4.4.3 From_monitor 107
4.4.4.4 From_cmd_int 107
4.4.4.5 Done 108
4.4.5 Command Interpreter 108
4.4.5.1 Parse 109
4.4.6 Terminal Drivers 109
4.4.6.1 Identify 109
4.4.6.2 Output 110
4.4.6.3 Input 110
4.4.6.4 Done 110
4.4.6.5 Terminal_watcher 110
4.4.7 Other I/O Device Drivers 111
4.5 Inter-task Communication 111
4.5.1 Rendezvous Request 112
4.5.2 Acceptance of a Rendezvous Request 115

5.0 Result of AdaTAD Research 118

5.1 AdaTAD Fulfils Requirements 119

5.1.1 Sequential 119

5.1.2 Execution State 120

5.1.3 Synchronization 120

5.1.4 Execution Context 121

5.1.5 Execution Speed 121

5.1.6 Execution Order 122

5.2 Importance of Design in Ada 122

5.2.1 Transportability of AdaTAD 123

5.2.2 Distributed Debugging 123

5.3 Temporal Logic 124

6.0 Discussion and Conclusion 125

6.1 Significance 125

6.2 Future Work 126

7.0 Bibliography and References 128

Appendix A. AdaTAD Source Code 144

Appendix B. Producer-Consumer Source Code 197

Vita 201

LIST OF ILLUSTRATIONS

Figure 1. Syntax of a Simple Ada Task 21

Figure 2. Example of a Selective Wait with Else . . . 23

Figure 3. Timed Entry Call 25

Figure 4. Conditional Entry Call 26

Figure 5. Object Module Format 39

Figure 6. Statement Prologue Algorithm 42

Figure 7. Parameter Descriptor List 46

Figure 8. Location of User Task in AdaTAD 50

Figure 9. Overall Structure of AdaTAD 53

Figure 10. AdaTAD Command Language 60

Figure 11. AdaTAD Logical Processor Command Language . 63

Figure 12. A Possible AdaTAD Screen 68

Figure 13. Task Window 69

Figure 14. Synchronization lists while TASKE executes 72

Figure 15. Commands to Debug Producer-Consumer Program 79

Figure 16. Producer's Screen after PC1 80

Figure 17. Consumer's Screen after PC1 81

Figure 18. Buffer_Control's Screen after PC1 82

Figure 19. Producer's Screen after PC2 83

Figure 20. Consumer's Screen after PC2 84

Figure 21. Buffer_Control's Screen after PC2 85

Figure 22. Producer's Screen after PC3 86

Figure 23. Consumer's Screen after PC3 87

Figure 24. Buffer_Control's Screen after PC3 88

Figure 25. Diagram of Logical Processors 90
Figure 26. Diagram of the Coordinator 98
Figure 27. Diagram of the Terminal Communicator . . 104
Figure 28. Example of User Task Synchronization . . 113

1.0 INTRODUCTION - THE PROBLEM OF CORRECTNESS

One of the most critical issues that face software engineers today is that of program correctness. Indeed, there is little disagreement that if a program does not work as specified, then characteristics such as efficiency, readability or elegance are of little value. In the 1980 Turing Lecture, C. A. R. Hoare [Hoar81] alluded to some very frightening aspects of unreliable software. It is not at all inconceivable that incorrect software could precipitate a world disaster. Incorrectly processed radar signals could set off retaliation against a nuclear strike which had, in fact, never been launched. On a smaller scale, but no less important to those people involved, incorrect software in the computers controlling the nation's Air Traffic Control system could cause mid-air collisions dwarfing the most spectacular accidents of recent years. At it's least harmful, incorrect software has caused extensive delays in the Air Traffic Control system.

The examples cited above are two of the more spectacular results of incorrect software. Less dramatic, but real nonetheless, are incorrect bank statements, impossible class schedules, bills for negative amounts and bouncing income tax refund checks. The list of these contretemps is virtually endless. Ignoring the problems of maliciousness and computer

crime, all of the above-mentioned problems could be caused by incorrect software. Clearly, there is a need to insure the correctness of software.

The remaining sections of this chapter:

1. describe the several aspects of achieving program correctness,

2. extract a single problem from the domain of problems in correctness,

3. propose a solution to the chosen problem,

4. show that current methods are weak in dealing with the problem, and

5. show how the proposed solution will solve the problem.

The organization of this chapter is recapitulated in greater detail in the remainder of the dissertation.

1.1 ASPECTS OF ACHIEVING CORRECTNESS

In the early days of any computer scientist's career, program correctness meant satisfying some mentor. Correctness was achieved by 1) compiling a program without any com-

piler errors and 2) running that program with some (possibly highly contrived) set of test data which yielded an answer that satisfied the mentor. Little more was asked of a two-page FORTRAN program than that.

With the sophistication of modern problems as well as modern hardware and software products, correctness is somewhat more difficult to attain. A software engineer may choose a two-pronged attack on the problem of producing correct software. Firstly, efforts are made to prevent the occurrence of errors and secondly, when errors do enter the program, they must be detected, located and removed.

Prevention of the occurrence of errors, sometimes called "anti-bugging", begins as the software engineer formulates the problem. After the problem is formulated, anti-bugging efforts should continue throughout the development of the problem solution and the production of the program's source code. Anti-bugging techniques include structured design, walk-throughs, program proving and structured programming.

Efforts toward the detection, location and removal of errors should begin almost as early in the development of the software as anti-bugging techniques. This helps minimize the cost of errors that do enter the program despite a person's best efforts to prevent them. Program testing is a technique often used to detect the presence of errors. Once the presence of errors is confirmed, an automated debugging tool is often employed to locate the errors. Once an error is lo-

cated and the necessary correction is decided upon, modification of the source code removes the error in question. Of course, there is no guarantee that the correction itself does not introduce some further error. The program remains suspect until further testing or proving indicate that the program has reached the desired state of correctness. Clearly, program correctness is best achieved by continued application of several techniques.

1.2 PROBLEM STATEMENT AND A PROPOSED SOLUTION

The problem to be addressed in this work is that of locating errors in the interface between two or more concurrent tasks. When errors are known to exist in such an interface, locating them is difficult because the time independence of tasks causes a given error to manifest itself in different ways on different runs of a program. On some runs, the error may not occur at all; on other runs, it may appear to occur as different sections of code are executed.

For many years, the importance of temporal considerations has been recognized. But due to difficulties in describing the time relationships among cooperating software modules, relatively little has been done to provide tools to help in preventing and correcting program errors involving time and temporal relationships. This dissertation suggests a design for such a software tool.

The proposed solution to the problem of locating errors in the interface of concurrent tasks is to specify and design an automated debugger which has features that address the special problems of locating errors in concurrent tasks. The research addresses specifically the interface of concurrent tasks written in the Ada programming language. Very little work has been done on testing Ada tasks and locating errors in Ada tasks.

The dissertation discusses aspects of debugging Ada tasks. A software tool for locating errors is designed. This tool is an automated debugger which functions at the source code level and aids the user in locating errors known to exist in the interface of two or more concurrent tasks. This debugger incorporates features specific to the problems of debugging Ada tasks. The features that make this debugger unique are as follows:

1. The user can control the interleaving and order of execution of tasks. This allows the user to locate undesirable time dependencies among tasks.

2. The user can control the relative speed of execution of tasks.

3. The user may set breakpoints. This allows the user to examine the state of the tasks at a designated point.

4. The user can view lists of synchronized tasks.

5. The user can view lists of variable aliases.

The features listed above and the fact that the debugger operates on Ada tasks sets it apart from other debuggers.

In summary, the purpose of the research reported herein is to demonstrate the feasibility of an automated debugger for concurrent programs by presenting the design of such a debugger. The debugger has the features of modern sequential debuggers and also has features necessary for debugging concurrent tasks.

Most importantly, the debugger is designed in Ada. Since such a debugger would run on any system which runs Ada, the debugger is able to execute on any architecture that supports Ada. This means that concurrent, distributed systems may be debugged on the actual environment of execution. In other words, the debugger presented herein allows distributed debugging. The tasks of a single program may actually be running on several different machines. A thorough review of the literature has revealed no other debugger with this characteristic.

2.0 REVIEW OF PERTINENT LITERATURE

In this chapter, the literature pertinent to the definition of program correctness and to techniques for achieving program correctness will be reviewed and discussed. The organization of the chapter will be as follows.

1. The literature of program correctness

2. The literature of program testing

3. The literature of automatic program debugging

4. The literature of program concurrency, with special emphasis on Ada tasks.

This review will form a compendium upon which the research of the dissertation is based.

2.1 THE LITERATURE OF PROGRAM CORRECTNESS

Obviously, people who write programs are interested in producing correct results. One of the earliest uses of automatic electronic computing machines was the calculation of artillery trajectories. If the program was incorrect, then,

at best, the artillery shell missed the target; at worst, it hit the gun emplacement that fired it! As early as 1950, work was being done on determining the correctness of programs. A. M. Turing [Turi50] published an early article on correctness called "Checking a Large Routine." It was not, however, until the early 1960s that a concerted effort was made to establish a mathematical foundation for software [McCa62]. This was the first step in formalizing the notion of program correctness. The dearth of published literature over the ten years following McCarthy's [McCa62] paper is indicative not of a lack of research but of the difficulty of the mathematical formalisms involved in software correctness. By the late 1960s, some theoretical results on the proof of program correctness had begun to appear [Alle68, Burs68, Burs69A, Burs69B, Coop67, Flor68, Floy67A, Floy67B, Hoar69, Mann68, Mann69, Naur66, Pain67]. This early work dealt, in the main, with proving a program correct by some formal, theoretical method. In general, the program to be proven and its specifications were stated as a theorem and an automatic theorem prover was employed to prove the theorem. If the theorem prover concluded that the statement of the program and its specifications was a theorem, then the program was considered correct. Recent language facilities such as Ada's exceptions, generics and tasks have not received adequate treatment by the correctness literature.

2.2 PROGRAM TESTING

By the early-to-mid 1970s, work was being done in the area of program validation via testing, Dijkstra's [Dijk76] caveat about being unable to prove a program correct by testing notwithstanding. In 1972, in Chapel Hill, North Carolina, a conference was held to explore program test methods. The work of this conference was reported by Hetzel [Hetz73] and formed the basis of further research into testing for the next several years.

William Howden [Howd75, Howd76, Howd77, Howd77A, Howd78] has studied the necessity of testing each path in a program and the symbolic execution of programs. Huang [Huan75] also studied path testing.

Goodenough and Gerhart [Good75] developed a theory which allows a program to be verified by testing. Their method requires the existence of a set of criteria for producing the necessary test data. Ramamoorthy [Rama73, Rama75, Rama75A, Rama76] has developed a method of automated generation of test data.

One aspect of testing that has remained virtually untouched in the literature is that of testing the time behavior of concurrent programs. There have been several reports relating to formal verification of concurrent programs [Owic76, Kell76, Lamp77], but nothing relating the theory of testing to concurrent programs.

2.3 PROOF VERSUS TESTING

The modern software engineer has access to several tools and techniques for increasing the confidence level that a software module works as specified. A partial list includes structured programming techniques, code walk-throughs, formal proof techniques and testing methodologies. The former two items in the list are typically employed during the coding of a system, while the latter two are generally applied to code that is sufficiently complete to actually run on a machine. The work described herein will deal mainly with the latter two.

Proponents of formal proof techniques contend that the techniques virtually assure that a software module will work as specified. However, De Millo, et al., [DeMi79] pointed out some of the weaknesses of formal theorem proving. When properly applied, formal proof techniques can greatly increase the confidence that a program is correct. However, the application of formal proof techniques to production software can be quite difficult and can result in large expenditures of programmer and machine time. In many instances, formal proving is not economically feasible. Some software, however, is of sufficiently great importance to warrant formal proof regardless of the cost. Examples are nuclear missile control software and microprograms.

Practicing programmers rely more heavily on testing methodologies than on proving. Testing methods which imply partial program correctness share one major shortcoming: they require either prohibitively large sets of test data or the generation of small sets of data is prohibitively difficult. Therefore, the trend recently has been toward using testing as a tool for increasing the confidence in the program. No doubt this remains the most viable method for producing reliable programs available today to the programmer producing non-critical software.

There is, of course, a weakness in this theory. As it has been pointed out by Hamlet [Haml81], proving that a set of data is reliable (or in the philosophy of Goodenough and Gerhart, [Good75], that a data selection criterion is reliable) is a non-trivial problem which is probably equal in difficulty to the proof of the correctness of the program. Therefore, testing is not a panacea for program correctness. It is merely one more tool at the software engineer's disposal for increasing the confidence in the correctness of the program.

2.4 TEMPORAL LOGIC

In recent years, Manna and Pneuli [Mann81, Mann83] have reported work on temporal logic. Since temporal relationships are a major characteristic of concurrent programming,

a proof system involving time relationships is particularly intriguing to those dealing with concurrency. That work is discussed at some length here.

Manna and Pneuli [MANN81] point out that propositional or predicate expressions are, temporally, instantaneous. These expressions are either true or false for a certain instant in time and they make no statement about conditions prior to that instant or after that instant. The temporal logic, however, allows a person to write an expression which may be evaluated for some interval of time. A temporal expression, therefore, describes the truth of a propositional or predicate expression over time.

Manna and Pneuli [MANN81] defined four modal operators. These are

1. always,

2. sometimes,

3. next and

4. until.

In the paper [MANN81], these operators were represented by, respectively, a small square, a diamond, a circle and an upper case "u", U. For the purpose of this dissertation, these

operators are represented by the words ALWAYS, SOMETIMES, NEXT and UNTIL. These operators are taken to be single valued functions mapping expressions onto the boolean values of true and false. The first three operators are unary functions which map propositional, predicate or temporal expressions into the boolean space. The fourth is a binary function mapping two of the aforementioned expressions into the boolean space.

The operators have the following meanings. The function

ALWAYS (<expression>)

yields a value of true if it can be shown that the expression is currently true and will remain true forevermore. The function

SOMETIMES (<expression>)

yields a value of true if it can be shown that the expression will become true at some future instant (possibly the present instant). The SOMETIMES operator does not require that the expression remain true; only that it becomes true at some point. The function

NEXT (<expression>)

yields a value of true if it can be shown that the expression will be true in the next instant. The function

UNTIL (<expression1>, <expression2>)

will be true if it can be shown that at some future time <expression2> will become true and that <expression1> will be continuously true until that time. Temporal expressions are formed by combining the temporal functions with the regular boolean connectives (and, or, implies, not) and the quantifiers "for all" and "there exists". The boolean connectives are represented by the symbols &, |, => and ¬, respectively. The quantifiers are represented by the quoted words, as above. The "expressions" in each function may be propositional expressions, predicate expressions or other temporal expressions.

The following example is paraphrased from Manna and Pneuli [MANN81] and illustrates the relationship of propositional, predicate and temporal expressions. Consider the statement "It rains today". There are at least two parameters to this statement: one is the location at which it rains, the other is the time frame of the rain. If the date and location are specified, say l0 and t0, then the statement "It rains at l0 on t0" is propositional. It is fully specified and is either true or false. However, if the location and time are variable, the statement becomes predicate in

nature. "It rains at l on t" by itself cannot be evaluated. More information must be supplied; specifically, the location and date. The notation "rain(l,t)" will be used to write such a predicate expression. If the time is considered to be the variable factor, this expression may be written as "rain(l)", meaning that, given a date, it rains at location l.

Temporal character is given to the predicate expression by including that expression into one of the temporal functions. For example,

$$\text{SOMETIMES} (\text{rain} (l))$$

is interpreted to mean that at some point in time (possibly right now), it will rain at location l. This, of course, allows more complex statements to be written. The notion that it will stop raining can be expressed as

$$\text{rain}(l) \Rightarrow \text{SOMETIMES} (\neg \text{rain} (l))$$

The expression

$$\text{rain}(l) \Rightarrow \text{ALWAYS} (\text{rain}(l))$$

claims that once it starts raining at location 1, it will never stop. The reader is directed to the literature [MANN81, MANN83] for further examples of temporal logic.

2.5 AUTOMATIC PROGRAM DEBUGGING

There are a number of automated tools for debugging currently available. A few of the more sophisticated will be mentioned.

Digital Equipment Corporation's VAX DEBUG is production quality debugger which is included with the VAX/VMS operating system. With Version 4 of this operating system, which was announced in November, 1984 and became available in January, 1985, DEBUG was greatly enhanced over its predecessor versions. DEBUG is a source level, symbolic debugger which utilizes windowing in its user interface. DEBUG allows debugging of concurrent processes but does nothing to ease the task. Concurrency on the VAX architecture requires that the concurrent modules be run in separate VAX sub-processes and communication is done with global sections or mailboxes. Each concurrent module must be linked separately into different copies of the debugger and then each debug-linked image must be run in a separate sub-process. As a result, the user using DEBUG for debugging "n" concurrent processes is dealing with "n" communicating DEBUG sessions with no special facilities for observing or controlling sub-process inter-

action. Other debuggers in common use include DELTA, a product of the Honeywell Corporation, and BLIT, a C language debugger developed by Bell Laboratories, now Bell Communications Research.

2.6 PROGRAM CONCURRENCY AND ADA TASKS

Program concurrency is a well-known concept. that existed as early as the ENIAC machine [Lee85]. However, a short review here may refresh the reader's mind. Two (or more) programs are considered to be concurrent if they execute simultaneously. True simultaneity of programs is, of course, unattainable on a single-processor machine because such a machine may execute only one instruction at a time. Therefore, to allow for concurrency on a single-processor machine, the definition of concurrency is generalized to the concept that programs are concurrent if their executions overlap in time. This definition was presented by Per Brinch Hansen [HANS73] and will suffice as a definition of concurrency for the research reported herein. Notice that this definition does not require that the programs execute simultaneously, but it does allow simultaneity. Therefore, concurrent programs may be executed on a single-processor machine by interleaving the instructions during execution. This interleaving may occur in any order and the user of concurrent programs on a single-processor machine may not

depend upon an particular order. Concurrent programs may also execute on multi-processor or distributed processor machines (a network). Two or more concurrent programs will be considered correct only if they exhibit correctness in all three types of execution environments. This implies that the programs must be consistent in all scenarios of concurrent environments.

Data interaction in concurrent programs may be complex. When concurrent programs attempt to access a common variable, there is not a built in assurance that conflict does not occur. If program A tries to read a variable common to itself and program B just as program B changes the value of that variable, then program A may read an inconsistent value from that variable. In most instances, some synchronizing mechanism is used to insure that such inconsistencies do not occur. Examples of such mechanisms are critical regions, [DIJK65] and semaphores, [DIJK65]. With these mechanisms, a program must wait while another program accesses the variable. In effect, these mechanisms cause concurrent programs to become sequential programs for that time when they could conflict.

2.6.1 Concurrency In Ada

Concurrency is a prominent feature in the Ada programming language. Since many of the applications that DoD ex-

pects to be developed in Ada entail concurrency, it is included as a language construct. Concurrent programs in Ada are called tasks. The task is a compilation unit that obeys essentially the same rules as does the procedure [Ada83]. Tasks may have parameters with the modes IN, OUT and IN OUT, allowing the tasks to communicate with other compilation units. Tasks may also share variables with other tasks as long as the rules of visibility allow the shared variable to be known to both tasks.

Tasks are activated after elaboration of the declarative part that defines the task object. When a task is activated, its declarative part is elaborated and then its execution continues in parallel with the compilation unit in which it was declared. If a compilation unit includes several tasks, then all of the tasks execute in parallel with the declaring compilation unit. The execution of the declaring unit does not begin until all of the tasks that it declares are fully activated. Therefore, a unit declaring a task and that task are synchronized at the time of activation of the task.

Tasks communicate via the rendezvous. When a task desires service from another task, it issues an entry call, to the servicing task. Upon issuing the entry call, the calling task enters a suspended state. If the called task responds immediately, then the rendezvous begins and the called task executes the section of its code that the calling task requested. If the called task is busy when the rendezvous is

requested, then the calling task's request is queued on a first in, first out basis to await service by the called task. When the called task has completed the requested service, the rendezvous is complete and the calling and called tasks proceed in parallel. The two tasks are synchronized during the rendezvous.

Tasks may also communicate via shared variables. Shared variables may be read and updated while tasks are synchronized. If tasks are not synchronized, then the user must insure that conflict does not occur. If the user must require access of shared variables, the language provides a pragma, called "shared", which insures that no conflict can occur. If the user does not insure against conflict, then the program is erroneous [Ada83].

Figure 1 on page 21 depicts the syntax of a simple Ada task. The notation used in the figure is the same as that used in the Ada Language Reference Manual [Ada83]. Each task may have several entries. This allows a task to provide a set of services that might be utilized by several other tasks. These entries are denoted in the declarative part by the reserved word entry and in the task body by the statement accept. When a task with entries reaches an accept statement, its execution is suspended until such time as another task makes an entry call to the accept statement at which the servicing task waits. At that point, a rendezvous is com-

```
task <task name> [is
<entry declarations>
end <task name>];

task body <task name> is
[<declaration part>]
begin
<sequence of statements>
end <task name>;
```

Figure 1. Syntax of a Simple Ada Task

plete. The rendezvous remains in effect until the servicing task

1. reaches the logical end of its execution,

2. encounters another accept statement or

3. encounters the end of a selective wait clause.

When the rendezvous begins, any parameters with mode IN or IN OUT will receive the values of arguments in the entry call statement. When the rendezvous completes, values associated with the parameters with mode OUT or IN OUT are associated with the arguments in the entry call statement.

Tasks waiting to perform a service may wait non-deterministically. By using the selective wait statement with "or" clauses, the user may designate several different alternatives for waiting. A selective wait must contain one or more accept statements. It may also contain one terminate alternative, one or more delay alternatives or an else part [Ada83]. These three possibilities are mutually exclusive. A sequence of statements may follow each alternative. Figure 2 on page 23 is an example of a selective wait with an else alternative. When this statement is executed, the appropriate accept alternative is taken if an entry call is

```
select
  <accept alternative>
or
  <accept alternative>
or
  <accept alternative>
else
  <else alternative>
end select;
```

Figure 2. Example of a Selective Wait with Else

outstanding to the accept. If no such call is outstanding, the else alternative is taken.

When a selective wait containing only accept statements is encountered, the task waits until an entry call is made to one (or more) of these alternatives. If entry calls to two or more accept statements arrive simultaneously, then one accept statement will be chosen arbitrarily. This choice is non-deterministic. Different executions of the program may produce different choices and the user may make no assumptions about the choice.

If a selective wait contains several accept statements and a delay alternative, then when the wait is encountered, execution will be suspended pending arrival of an entry call or expiration of the time specified in the delay alternative. If the delay alternative expires, then the sequence of statements following it are executed, if any.

If the selective wait contains several accept statements and a terminate alternative, then when the wait is encountered with no outstanding entry calls, the task terminates. If there are outstanding entry calls, the oldest one is executed.

If the selective wait contains several accept statements and an else alternative, then when the wait is encountered with no outstanding entry calls, the else clause is executed immediately. If there are outstanding entry calls, then the oldest one is executed.

```
select
  <entry call>
or
  <delay alternative>
end select;
```

Figure 3. Timed Entry Call

```
select
  <entry call>
else
  <sequence of statements>
end select;
```

Figure 4. Conditional Entry Call

A task may also make conditional or timed entry calls. The select statement is used in both cases. In the case of the conditional entry call, the entry call itself is in the first portion of the select statement. Following this must be an else part containing a sequence of statements. When the select is encountered, the entry call to another task is made if it can be made immediately. Otherwise, the else clause is executed. Figure 4 on page 26 shows a conditional entry call.

In the case of the timed entry call, an "or" clause follows the entry call in the first part of the select statement. In this "or" clause is a delay alternative. Upon encountering this construct, the entry call is made if it can be made before the delay alternative expires. If not, no entry call is made and execution continues beyond the select statement. Figure 3 on page 25 depicts a timed entry call.

With the exception of reports by committees designing Ada and the Ada Language Reference Manual [Ada83], relatively little literature exists on Ada. This is due, no doubt, to the newness of Ada and to the lack of a compiler until recently. Pyle [Pyle83] published a text covering Ada. There have been several letters in the ACM's SIGPLAN Notices and SIGSOFT's Software Engineering Notes alternately praising and reviling the language. However, the Language Reference Manual [Ada83] remains the definitive source of information on the language.

3.0 REQUIREMENTS OF A TASK DEBUGGER

The purpose of this chapter is to outline the requirements of a debugger for programs with tasks and to justify, through literature citation, argument and example, the requirements for which the debugger is designed.

3.1 A BASIS IN THE LITERATURE

There is little information in the literature to guide the designer of a task debugger. A thorough search of the literature yielded only the proceedings of a joint SIGSOFT/SIGPLAN conference held in late March, 1983. This conference was a Software Engineering Symposium on High-Level Debugging and one session was devoted to Distributed Debugging. Two papers presented in this session alluded to the requirements of distributed debugging.

The first, by Baiardi, et al., [Baia83], lists three characteristics of the execution of concurrent programs. Several requirements are easily derivable from these characteristics. The characteristics are

1. synchronization points exist among processes,

2. non-deterministic choices and

3. low visibility, which means that it is impossible to deduce the execution order of statements of separate tasks.

The second paper of interest is by Weber [Webe83]. This paper lists characteristics common to all debuggers. Specifically included are

1. the ability to examine and modify variables,

2. to set trace areas,

3. to set conditional breakpoints and

4. to invoke source level stepwise execution.

Additionally, she [Webe83] indicated that "The features of more specific interest to the programmer of concurrent systems are those that involve process monitoring." These two papers appear to cover the literature characterizing requirements for task debuggers.

A coalescing of the two lists yields the following, which is used to produce a list of requirements.

1. Which tasks are synchronized with which other tasks

2. Which tasks are running (and, by implication, stopped)

3. What statement is to be executed next in each task

4. The debugger must not interfere with the temporal relationships of the tasks being debugged.

5. It is impossible to deduce the order of execution of statements in separate tasks.

The following set of specific requirements is deduced from the above list. A debugger for tasks must have the following capabilities:

1. It must have the characteristics of current technology single program debuggers.

2. It must display the execution state of each task.

3. It must display the state of task synchronization.

4. It must display the "execution context" in the source language of each task.

5. It must allow the user to control the relative execution speeds of the tasks so that the debugger will not interfere with the temporal relationships of the tasks.

6. It must allow the user to control the order in which statements of each task are executed so that the user may experiment to find any undesired temporal dependencies.

It is clear that each of these requirements is necessary for a task debugger. If any one is removed from the list, then some execution characteristic of concurrent tasks will become unmonitored and/or uncontrolled.

It is not clear that this list is sufficient or complete. It is possible that other requirements and capabilities should be on the list. However, a list of necessary capabilities is needed to design any sort of task debugger. Then, once such a debugger exists, it can be used to test the list of capabilities.

3.2 USER INTERFACE

Though not specifically listed as a requirement of the task debugger, a well-engineered user interface must be used. Some effort must be spent in designing the user interface because a large volume of information is available from the debugger and this information must be presented in a coherent manner if the user is to avoid information overload. Though no attempt is made to discover new truths in human-computer interface design, modern tenets of interface design are applied to the user interface. The user interface consists of

a command language to allow the user to communicate with AdaTAD and a window-based display system for the user to view the actions of AdaTAD and the program being debugged. Moreover, the design of the interface is modular so that other interfaces may be used.

3.3 TEMPORAL LOGIC

Since time independence is the one characteristic that separates tasks from other program units, it is highly desirable to give a user the ability to compute expressions based upon temporal relationships. Proper construction of such expressions can allow the user to determine if tasks are in fact time independent.

A syntax of temporal logic expressions is presented in such a way that they could be parsed and executed by Ada. The semantics of the temporal expressions is given in the work of Manna and Pneuli [MANN81].

Clearly, temporal logic is based upon the concept of eternity. Any concrete design or implementation of temporal expressions must have some facility for defining what is meant by eternity. Any implementation of temporal logic will require an approximation of eternity. This concept of approximation is analogous to the approximation of real numbers in computing machine. It is quite common to approximate real

numbers with integers. In the design of AdaTAD, it was assumed that "eternity" could be defined in one of three ways.

1. Eternity will occur "n" seconds (or any time unit) in the future.

2. Eternity will occur after "n" program statements have been executed.

3. Eternity will occur when machine state repetition occurs.

Either of the first two is straightforward to implement. The third is not so straightforward. The current design of AdaTAD incorporates a command allowing the user to specify eternity as some number of seconds from the present.

The NEXT operator requires the definition of the next instant. For the purpose of designing AdaTAD, the next instant is defined as the next single unit of whatever unit is being used to define eternity. Reference the definitions of eternity in the previous paragraph. Then the next instant will be, respectively,

1. one second from now,

2. the execution of one statement from now or

3. the next machine state from now.

Using one of these definitions of eternity, three of the operators can be implemented. The necessity (ALWAYS), the possibility (POSSIBLE) and the until (UNTIL) are the three. The definition of the next instant allows implementation of the NEXT operator.

Temporal assertions are established by the SET ASSERTION command in the logical processor command language. The assertion is boolean in nature. When a temporal assertion is encountered, its truth value is computed with respect to the eternity defined on the program. If, within that eternity, it can be shown that the temporal expression is true, then the assertion is true. If, within the eternity, it cannot be shown that the expression is true, then the assertion is false.

When an assertion is encountered in the program, the current state of the program is saved. Once the state has been saved, the program is restarted and the elements of the temporal expression are monitored along with the program's progress to eternity. If it can be shown that the temporal expression is true when eternity is reached, then the state of the program at the time and place of the assertion is restored and the program's execution is resumed with the value of the assertion being true. If the temporal expression is false or indeterminate when eternity is reached, then a mes-

sage is issued and execution of the program halts. The user will have an opportunity to make desired changes and restart the program at some point.

If the assertion is false, the user is provided with information on the temporal expression. For example, the user is informed if the expression was ever true before eternity arrived. It is conceivable that an expression could become true for a while and then become false again and remain so until eternity. Indeed the transition might occur several times. The state of the program when these transitions took place provides useful information. The user is also provided with a history of the execution of other tasks in the progress toward eternity. It is conceivable that some supposedly active tasks may become quiescent as eternity approaches or vice versa.

It is realized that the implementation of temporal assertions could become memory intensive. However, time efficiency is of relatively little importance at the development stage of software. The implementation can be optimized after it exists.

4.0 ADATAD - A MULTI-TASK DEBUGGER

This chapter describes the design, in increasing levels of detail, of a task oriented debugger. Firstly, the support system needed by the debugger is described. Then, the overall structure of the debugger is presented. Next, a description of the user's interface with the debugger is presented. The functionality of each major module of the debugger forms the next section. The details of the communication among the debugger's major modules conclude the description of the design of AdaTAD. Examples describing user task synchronization, user task output, effect of user commands and a sample debugging session are presented at appropriate places.

The debugger whose design is presented herein is named AdaTAD. This is an acronym for Ada TAsk Debugger. AdaTAD is designed to be an integral part of an Ada Programming Support Environment (APSE).

An important feature of AdaTAD is that Ada itself is used as the program description language (PDL) for the design. This PDL design is to form the basis of the actual source code for AdaTAD. Since the implementation is in Ada, AdaTAD runs on any system supporting Ada. This allows transportability of AdaTAD among Ada systems and also allows true distributed debugging of Ada tasks.

4.1 THE ADATAD SYSTEM

The AdaTAD debugger requires support from other portions of the APSE. Specifically, the Ada compiler must be capable of generating object modules of a specific format and the linker must be able to link the user's program into the debugger. The following two sections describe the characteristics that the compiler and linker must have in order to generate an executable image of the debugger and the user's program. This material is presented first because understanding it facilitates understanding the AdaTAD debugger.

4.1.1 The Compiler

The compiler portion of the AdaTAD system accepts as input a source program written in Ada. If the program is not a legal Ada program, the compiler reacts according to the Ada language specifications. If the program is a legal Ada program, the compiler generates object code for the target machine in a form specified in "Object Module Format" on page 38.

The AdaTAD compiler generates separate object modules for each task body declared in a program. Each task object declared in a program is bound to an object module at compile time. The linker, described in "The Linker" on page 48, binds these modules into an executable image. For task ob-

jects created by evaluation of an allocator, the "new" procedure creates and links the necessary object code. Each object module contains the object code for the user's source code, the symbol table for the task, a prologue for each statement in the task, the local data area and the object code for each assertion in the task.

The compiler must produce a readable version of the symbol table to exist at execution time. This is so that AdaTAD can display the contents of a given variable by name as well as by address. In addition, the symbol table contains the locations of any labels in the program. The source code labels are used in setting breakpoints and logical assertions when AdaTAD is active. The module implementing the user's main program also contains the runtime symbol table. The symbol table is organized as a general tree. The root of the tree contains the program name while the root nodes of its subtrees contain the identifiers declared in the main program.

4.1.2 Object Module Format

The compiler itself is not implemented here, but the form of the object module is described in detail. The object module is described here using Ada as the description tool.

The object code produced by the compiler consists of a series of distinct modules, each of which implements one user

```
Header
Prologue for statement 1
Object code for statement 1
Prologue for statement 2
Object code for statement 2
.
.
.
Prologue for statement n
Object code for statement n
Module epilogue
Task entry procedure (see below)
Source code for statement 1
Source code for statement 2
.
.
.
Source code for statement n
```

Figure 5. Object Module Format

task body. Figure 5 on page 39 depicts the format of the object module. Header information, which identifies each separate module to the linker, is included. The user's main program is treated as an entry-less task.

Each module, implementing one user task body, consists of the object code to implement the user's logic and of the source code of the task. The source code and the object code are keyed together with the statement numbers of the source code. Since AdaTAD is a source code debugger, it is necessary to include the source code somewhere and the object module is a natural choice. Of course, the source code is delimited from the object code.

The object code itself consists of the instructions needed to implement each source statement. The code for each source statement is preceded by a prologue. The code of this prologue is that which is necessary to implement some of the AdaTAD features.

4.1.2.1 Statement Prologue

At any arbitrary point in time, the execution of a given source statement may be enabled or disabled. If enabled, the code implementing the statement is executed; if disabled, the prologue waits until execution is enabled. Execution is disabled when any one of the following conditions hold.

1. The current execution state is "wait". This would occur when the user issues a "wait" command or when a breakpoint interruption occurs. Execution would be enabled again only when the user explicitly changes the state.

2. If the execution state is "timed" and the time to execute the statement has not yet arrived. Execution would be enabled when the execution time arrives.

3. If the execution state is "single step" and the release character has not been pressed. Execution becomes enabled when the release character is pressed.

4. An unconditional breakpoint has been encountered.

5. An assertion breakpoint has been evaluated to a value of false or indeterminate.

The statement prologue code examines the execution mode as each statement is encountered and determines if execution is enabled or disabled. The algorithm for the statement prologue is shown in Figure 6 on page 42. The statement prologues perform some computation before determining whether or not execution is enabled. The first action performed by the prologue is to place in the execution data base the num-

```
current_statement := this_statement_number;
If breakpoint_state = enabled Then
  If breakpoint_set Then
    If breakpoint_type = uncond Then
      execution_mode := wait;
    Else
      assertion_value := evaluate_assertion (assertion);
      If Not assertion_value Then
        execution_mode := wait;
      End if;
    End if;
  End if;
End if;
Loop
  Case execution_mode Is
    When wait        =>
      execution_enable := false;
    When normal      =>
      execution_enable := true;
    When timed       =>
      Loop
        now := time_of_day; -- Function "time_of_day" returns
                            -- current time.
        Exit when now >= release_time;
      End loop;
      release_time := time_of_day + execution_rate;
      execution_enable := true;
    When singlestep =>
      Loop
        c := arrival (release_character);
        Exit when c;
      End loop;
      execution_enable := true;
  End case;
  Exit when execution_enable;
End loop;
```

Figure 6. Statement Prologue Algorithm

ber of the source statement that the following object code implements. This action is always taken.

The next action of the prologue is to check to see if breakpoint checking is currently enabled. If breakpoint checking is currently disabled, then control skips directly to the code that checks the current state of the "execution enabled" variable. If breakpoint checking is enabled, then the next part of the prologue checks to see if this statement has been designated as a breakpoint. If not, then control skips directly to the code that checks the current state of the "execution enabled" variable. If this statement has been designated as a breakpoint, the prologue checks to see if an assertion is present. If there is no assertion, the prologue sets the execution state to "wait" and informs AdaTAD that a breakpoint has been encountered. If an assertion is present, then the code which implements the assertion is executed immediately. If the result of the assertion code is "true", then control passes to the code which checks the current execution state. If the result of the assertion is "false" or, in the case of a temporal assertion with user-defined eternity, indeterminate, then the action taken is that of an unconditional breakpoint. The last action performed by the prologue is a check of the "execution enabled" variable. If execution is currently disabled, then the prologue enters a busy wait until execution is enabled or the task is terminated from elsewhere.

After checking the breakpoint state, the prologue code examines the execution mode. If the mode is "wait", execution is disabled. Otherwise, execution is enabled. If the mode is "timed" or "singlestep", execution is enabled only after the appropriate criteria are met. The checking of the execution mode continues until execution is enabled.

4.1.2.2 Argument Descriptor List

When the user passes arguments between two tasks, the compiler generates a list of descriptors and the address of the head element of this list is what is actually passed. In this way, AdaTAD can obtain information from the list. Also, using the "address of descriptor" mechanism, it is not necessary to know the type of the user arguments a priori. The descriptor list consists of one head node and several argument nodes, one node for each argument.

The head node consists of the name of the called task and entry, the name of the calling task and a pointer to the first argument node. Notice that the task names itself in the header of the argument list. This is necessary to allow AdaTAD to identify the calling task as rendezvous is made.

Each argument descriptor node is an element in a doubly linked list and contains the following information.

1. The offset of the argument in the defining module.

2. The total length of the argument.

3. The length of each element of the argument, if it is an aggregate.

4. The type of the argument (through a reference to the symbol table). If this is a user-defined type, the actual type format is in the symbol table.

5. The mode of the argument (IN, OUT or IN OUT). This information is available at compile time because the definition of the entry is known.

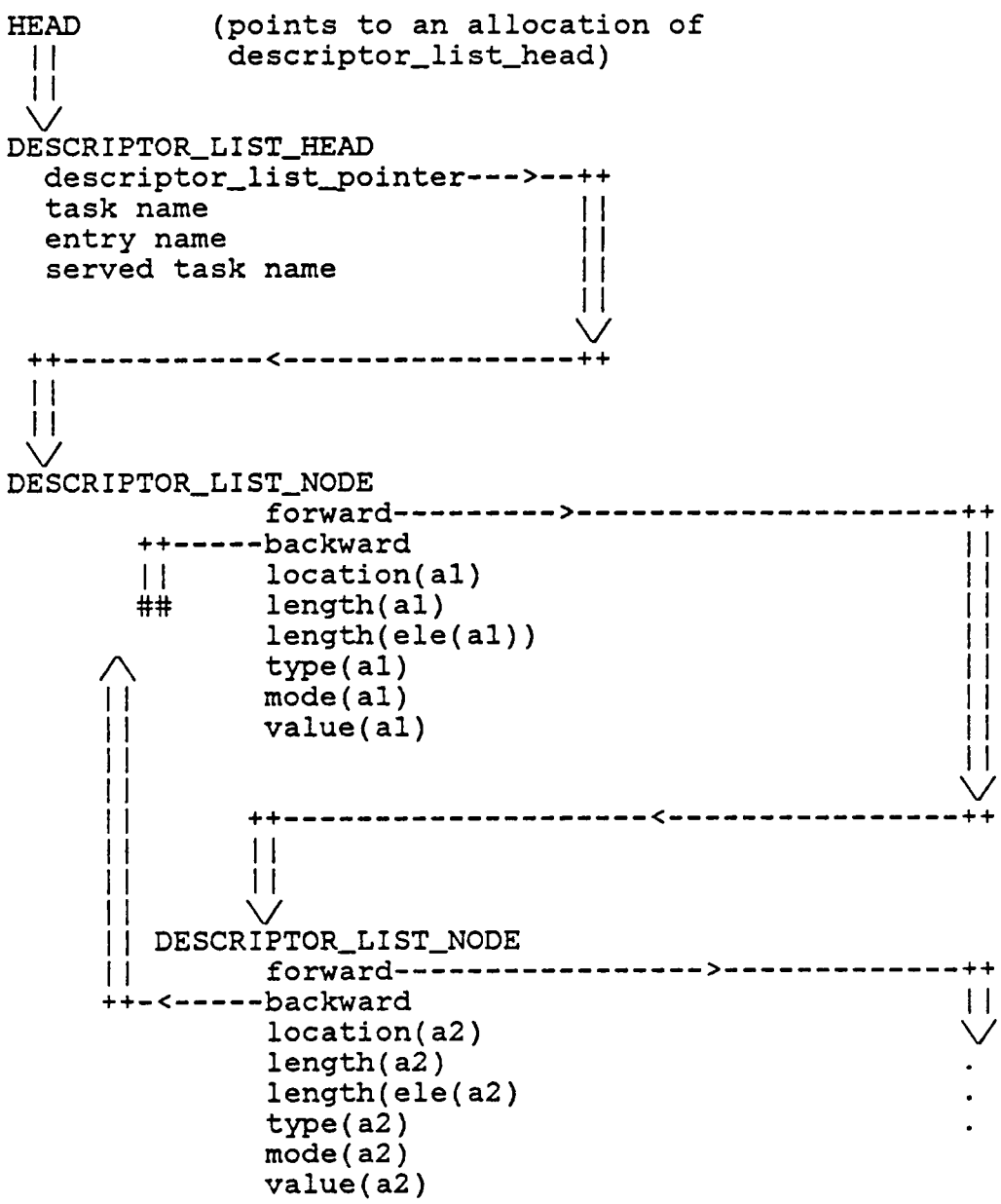
6. The value of the argument if the mode is IN or IN OUT.

Additionally, of course, each node has forward and backward pointers to implement the doubly linked list structure.

Figure 7 on page 46 depicts the format of the argument descriptor list.

4.1.2.3 Code Modification

Because AdaTAD intervenes to a large degree in the user's code, the compiler must make some changes in the way certain statements are compiled. The affected statements are



(## denotes a null pointer.)

Figure 7. Parameter Descriptor List

1. task entries (task entries and selective waits),

2. entry calls and

3. input/output statements.

The changes that the compiler makes in these statements is now discussed.

Each entry within the user's task has two statements added to it. Immediately after the "accept" statement is generated an entry call that informs AdaTAD that a rendezvous has in fact begun. Immediately before a rendezvous completes (just before the "end" statement of an "accept ... do" statement or immediately after an "accept" statement) is generated an entry call that informs AdaTAD that the rendezvous is now over. The actual mechanisms for this are detailed below in the section on task synchronization.

Additionally, the compiler generates, for each user task, a procedure for handling incoming entry calls. This procedure appears in Figure 5 on page 39 as "Task entry procedure", and it consists of a case statement where the choice for each case statement alternative is the name of an entry. There is one case statement alternative for each entry in the users task. The code within each case statement alternative decodes the argument descriptor list that is passed to the corresponding entry at runtime. It also makes the actual

entry call to the user's task. The use of this procedure is described in the section on task synchronization, below.

Each user entry call causes the generation of code to handle several functions. Firstly, the compiler generates code that creates the argument list in a region of AdaTAD's storage reserved for argument lists. Secondly, code is generated that initializes this storage at runtime. Finally, the user's entry call is changed to an entry call to AdaTAD's logical processor transmitter (see description below), passing the location of the argument list as a single argument to the transmitter.

Since AdaTAD takes control of all input/output devices, user input/output must be mediated by AdaTAD. Therefore, all user I/O statements are changed to entry calls to AdaTAD itself. The task wishing to perform I/O constructs an argument list similar to that used for rendezvous calls. This argument list includes the value to be output or storage to receive input. The compiler then generates code to make an entry call to AdaTAD, passing the argument list and an indicator of what is to be done. The actual mechanism is described below.

4.1.3 The Linker

The linker portion of AdaTAD accepts as input a series of object modules output by the compiler and the object mod-

ules for AdaTAD's logical processors. The linker's job is to incorporate the user's tasks object code into the AdaTAD debugger. To do this, the linker has a template for AdaTAD's logical processor task. At link time, each user task is linked into a copy of the template. The user's task is linked into the EXECUTOR task of an AdaTAD logical processor. Figure 8 on page 50 illustrates the location of the user's task. It is the linker's task to establish the proper environment for the user's task with respect to the AdaTAD logical processor. The result of this link is an object module that is now linked with a coordinator task. The final result is an executable image in which the user's entire program runs under the control of AdaTAD. Neither the AdaTAD debugger nor the user program compiled with the AdaTAD compiler can execute alone. The link step creates the AdaTAD debugger specifically for the given user program.

Once the linker has created the debugger with the user program, the only remaining input to the debugger are the commands that the user can enter interactively. These are the commands that allow the user to control interactively the execution of AdaTAD during the debugging session. These commands implement the functionality described in "User Interface" on page 54.

As a user task finally begins execution, two actions must take place.

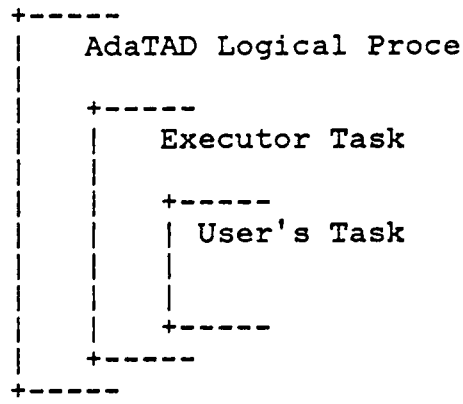


Figure 8. Location of User Task in AdaTAD

1. The logical processor's activation record must have been augmented by the activation record of the user's task.

2. Execution of the image must begin.

Regardless of whether the task of interest is declared immediately, is a task type with which another object is declared or is an allocated task created dynamically at runtime, both of the above actions must occur. For immediately declared tasks and for objects declared from a task type, the linker performs the first action while the runtime support system performs the second. If the task is allocated at runtime, then the "new" procedure must perform both jobs when evaluating the allocator.

The above linking results in an executable image that is named "AdaTAD" and which runs the user's program under the aegis of AdaTAD. The remainder of this section describes in detail how the AdaTAD debugger is constructed. AdaTAD's algorithm and data structures are defined in the design code in Appendix A.

4.2 THE OVERALL STRUCTURE OF ADATAD

The two overriding goals in the design of AdaTAD are to implement correctly the requirements as described in chapter three and to minimize the impact of the debugger on the exe-

cution of the user's program. AdaTAD is designed to achieve these goals in that order. The first level of detail of AdaTAD's design consists of seven task bodies. These are

1. Logical Processor,

2. Coordinator,

3. Terminal Communicator,

4. Command Interpreter,

5. Data Base Monitor,

6. Terminal Driver, and

7. I/O Device Driver.

Two of these bodies, Logical Processor and Terminal Driver, are used to define arrays of tasks. The Logical Processor and the Terminal Driver each declare local tasks of their own.

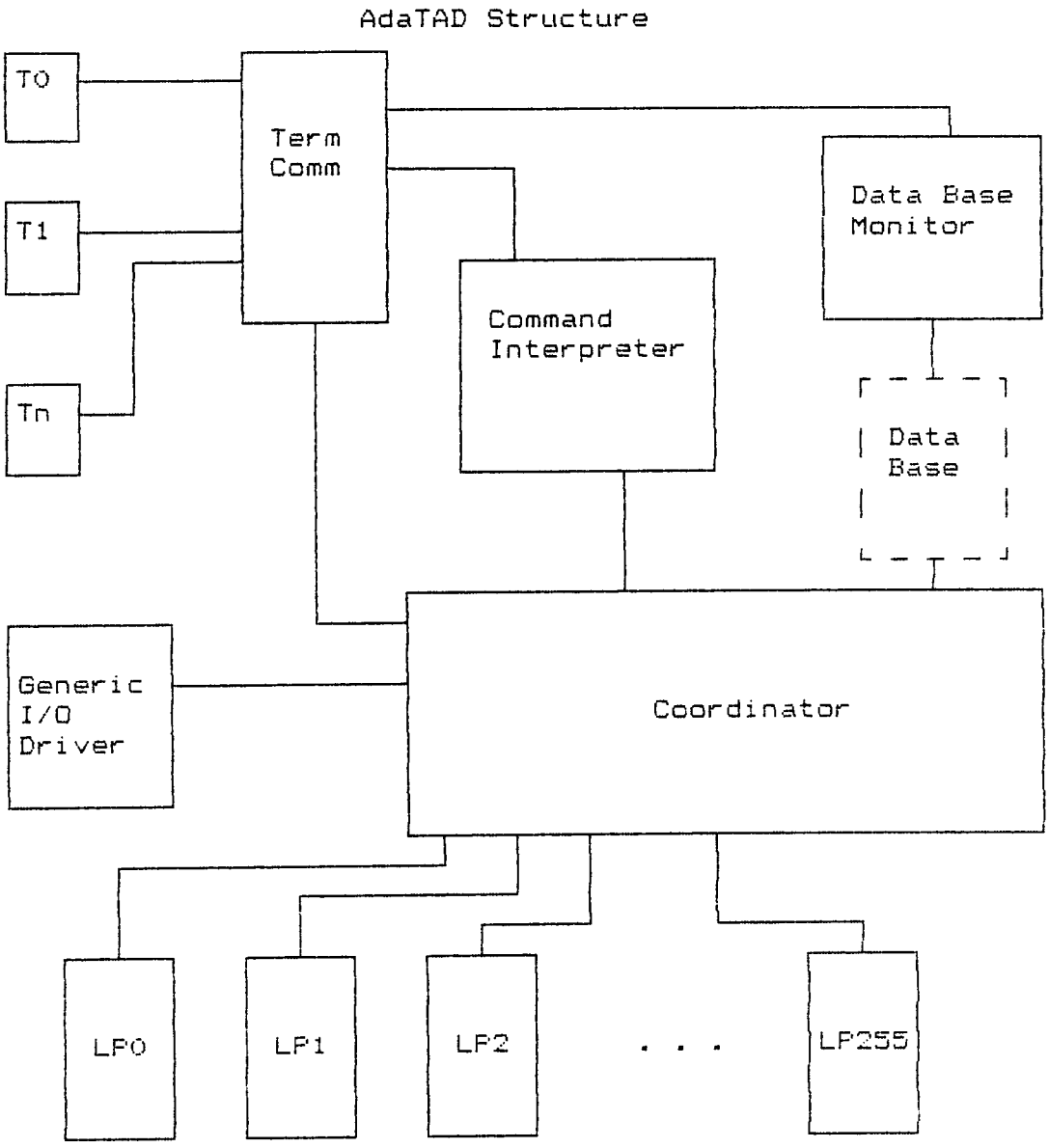


Figure 9. Overall Structure of AdaTAD

4.3 USER INTERFACE

The design of AdaTAD's user interface is motivated by the tenets of human factors engineering of human/computer interfaces. AdaTAD's interface design makes no attempt to uncover new truths in the field of human/computer interface design. The design conservatively follows current technology in human/computer interface design.

Efforts are made to insure that AdaTAD's human interface is well engineered. Shneiderman [Shne83] lists five measurable quantities that can be used to gauge the effectiveness of human engineered software. They are

1. time to learn,

2. speed of performance,

3. rate of errors,

4. subjective satisfaction and

5. retention over time.

Theoretically, if these quantities are optimized, a well engineered interface will result. Since the research reported herein is a design and not an implementation, it will be im-

possible to actually measure these quantities. However, steps are taken that, in the author's view, help to optimize these factors. When AdaTAD is finally implemented, these factors can be measured.

The difference between "well engineered" and "user friendly" should be noted. Stevens [Stev83] has criticized the use of the term "user friendly" because it contains undefined notions. The term "well engineered", on the other hand, refers to quantifiable, scientific measures which support use of certain techniques and principles in interface design.

The user interface consists of two parts:

1. The method by which the user directs the system, described in "The Command Language", and

2. The method by which the system displays information to the user, described in "The Display" on page 65.

4.3.1 The Command Language

Research recently reported by Whiteside, et al., [Whit85] shows that there are, in general, three human-computer interface schema in use today. They are

1. the command system,

2. the menu system and

3. the icon system.

Whiteside [Whit85] compares the usability of each type of system with two different types of users. One conclusion of the research is that the type of interface has less impact on the usability than does the quality of design of the interface. In other words, menu driven systems and icon systems are not necessarily better than command systems. Another conclusion of the research is that the experience level of the user has little impact on the usability of the system.

This last conclusion is counterintuitive. It has long been felt, at least by this author, that experienced users fared better with a command driven system than they did with a menu system and that inexperienced users were better off with a menu or icon system than with a command system. Whiteside [Whit85] indicates, however, that an experienced user will do better with a carefully designed menu or icon system than with a poorly designed command system. Conversely, the inexperienced user will accomplish more with a well-designed command system than with a poorly designed menu system.

AdaTAD uses a command language system. Use of this system is consistent with the findings Whiteside, et al..

Use of a command system does not preclude a menu or iconic interface. AdaTAD is designed in a modular fashion so that minimal changes would be necessary to include a different interface.

The actual form of the commands used to invoke AdaTAD and its support software depend on the APSE being used. To describe a possible scenario for invoking AdaTAD and its support software, the syntax of the APSE for Digital Equipment Corporation's Ada system. The compiler is invoked with the command

```
$ ada <filespec>
```

where <filespec> names the file containing the Ada program. To cause Digital's compiler to generate object modules acceptable to AdaTAD, a qualifier is added to the ada command, so that the command becomes

```
$ ada /adatad <filespec>
```

A similar qualifier is added to the "link" command. The meaning of this qualifier is that the user object modules are to be linked with the AdaTAD debugger.

```
$ link /adatad <filespec>
```

The result of this command is an executable image which runs the user's program under the control of AdaTAD.

Still in the context of Digital's Ada system, the user program is run with the "run" command.

```
$ run <filespec>
```

This command starts execution of the debugger.

The compile and link commands are the only "user direction" commands which affect those portions of AdaTAD. Once the AdaTAD debugger is running, the control functions fall under one of two broad spheres. First of all, there are commands that control AdaTAD itself. These are commands to

1. begin execution of a program or of a task on a specified processor and

2. terminate AdaTAD and return to the APSE level.

Secondly, there are commands for controlling the execution of each individual logical processor. These are commands to

1. Examine the execution state of a given logical processor,

2. Examine contents of memory of a specified logical processor,

3. Change the execution mode of a specified logical processor,

4. Alter the value associated with variables,

5. Place assertions at a given point in a task.

4.3.1.1 Description of AdaTAD commands

This section presents a textual description of the AdaTAD commands and a BNF notation definition of the syntax. Notes on notation: When commands are being discussed, anything contained in square brackets ([]) is optional; angle brackets (<>) indicate that the user is to insert something between the brackets; the vertical bar (|) separating elements means that the user is to select among alternatives. Anything shown in upper case letters is to be entered verbatim. Anything shown in lower case letters is to be replaced by the user.

The command language that is used with AdaTAD breaks down into two sub-languages.

1. the language used to direct AdaTAD itself and

2. the language used to control the logical processors.

```
<AdaTAD_command>     ::=
                         | <execute>
                         | <window definition>
                         | <zoom>
                         | <default task>
                         | <name I/O device>
                         | <assign>
                         | <terminate>
<name>               ::= character string
<execute>            ::= RUN
                         | GO
                         | EXECUTE
                         | START
<default task>       ::= <lp reference>
<lp reference>       ::= <processor no.>
                         | <name>
<window definition>  ::= WINDOW <name>
                             <visibility> [<frame>]
<visibility>         ::= VISIBLE
                         | INVISIBLE
<frame>              ::= <anchor> <extent>
<anchor>             ::= <screen location>
<extent>             ::= <screen location>
<zoom>               ::= ZOOM IN
                         | ZOOM OUT
<screen location>    ::= number number
<name I/O device>    ::= NAME <o/s device name>
                             <I/O device name>
<o/s device name>    ::= Host system dependent name
<I/O device name>    ::= character string
<assign>             ::= ASSIGN <lp reference>
                             TO <I/O device name>
<terminate>          ::= EXIT
```

Figure 10. AdaTAD Command Language

Figure 10 on page 60 displays the BNF description of the language that controls AdaTAD itself. In the figure, each right hand side non-terminal of the productions of the form

<tscommand> ::= <non-terminal>

is a command in the language. These are now defined.

<execute> This command causes the user's program to begin execution with all the tasks in the execution mode to which they are currently set. All of the terminals of the production are synonymous.

<window definition> This command names a window and indicates whether the window is to be displayed. It also optionally defines the size and location of the window. If the size and location are not specified, then the current size and location remain unchanged. Initially, when AdaTAD is invoked, all windows are anchored at the upper left corner of the screen and the extent is zero.

<zoom> The command causes a window to become larger (ZOOM IN) or smaller (ZOOM OUT).

<default task> This command designates one task as the default task. It is to this task that all logical processor - specific commands apply. The user's main program is the default task when the debugger starts.

<name I/O device> This command associates operating system - specific device name to another identifier.

<assign> This command associates a specific I/O device with a task running on the designated logical processor. The designated I/O device is the one used as the task's default I/O device.

<terminate> This command terminates the AdaTAD debugger and returns the user to the APSE command level.

Figure 11 on page 63 is the BNF notation description of the commands that are specific to the logical processors. When these commands are issued, they apply to the task that has been designated as the default task with the appropriate AdaTAD command. If no default task has been designated, then these commands apply to the user's main program. All of these commands begin with the keywords SET or SHOW. This differentiates them from the AdaTAD commands.

```
<logical processor command> ::= <set>
                              | <show>
<show>                      ::= SHOW <variable name>
                              | SHOW ALIASES
                                <variable name>
<set>                       ::= SET <set parameter>
<set parameter>             ::= <execution mode>
                              | <breakpoint>
                              | <variable>
                              | <assertion>
                              | <eternity>
<execution mode>            ::= WAIT
                              | NORMAL
                              | SINGLE STEP RELEASE <char>
                              | TIMED <rate>
<char>                      ::= ASCII character
<rate>                      ::= <real number> ST/SC
                              | <real number> SC/ST
<breakpoint>                ::= BREAKPOINT AT <location>
                              | BREAKPOINT {ON|OFF}
<location>                  ::= statement label
<variable>                  ::= <variable identifier> :=
                                <value>
<assertion>                 ::= ASSERT <exp> AT <location>
<exp>                       ::= <boolean expression>
                              | <temporal expression>
<boolean expression>        ::= expression
                                (see Ada LRM, 4.4)
<temporal expression>       ::= ALWAYS <exp>
                              | SOMETIME <exp>
                              | NEXT <exp>
                              | <exp> UNTIL <exp>
<eternity>                  ::= ETERNITY <number> <unit>
<number>                    ::= integer
<unit>                      ::= STATEMENTS
                              | SECONDS
```

Figure 11. AdaTAD Logical Processor Command Language

SHOW <variable name> This command displays the value associated with the named variable on the task's display area.

SHOW ALIASES <variable name> This command causes the symbol table to be sorted on addresses and the identifiers sharing the same location as the named identifier are displayed.

SET <execution mode> This command places the logical processor of the default task in the specified execution mode.

SET <variable> This associates the computed value with the named variable.

SET <assertion> This establishes an assertion at the specified location. The assertion is to be evaluated when the machine's location counter is the same as the location of the specified label. The assertion is entered as an ASCII string and is to be interpreted by AdaTAD at execution time.

SET <eternity> This command allows the user to specify the amount of time (in either seconds or source state-

ments executed) that the temporal assertion mechanism is to treat as eternity.

4.3.2 The Display

A well engineered display of the data produced by AdaTAD is critical. So much information is made available by AdaTAD that there is danger of "information overload". That is, the user may have difficulty extracting the information pertinent to a particular situation from the large mass of information available. Organization is considered the key to effective information display. The use of "windows" is one method of organizing the information.

The concept of windowing is one which has recently received much attention, particularly in the small computer world. For example, the Microsoft Corporation is marketing a software package called "Windows" which comprises a shell on their MS-DOS operating system, making the operating system interface window oriented. Also, the Borland Company markets a package called "Sidekick" which uses windows to implement utilities such as a calculator and an automated note pad.

Windowing offers flexibility for the display of data on a terminal. In general, windowing allows any process on the system to assume that it has an output device dedicated to it. Assignment of screen geography can vary dynamically during execution of the program. The user can dynamically

assign each window a location on the screen. In fact, if the information displayed in the window for a specified process is not currently of interest, the window can be deleted from the screen. This deletion is transparent to the process writing information to the window. The user can, therefore, design the presentation of information and can, dynamically, display or not display output from a given process. The ability to remove a window from the screen without affecting the execution of the process writing in the window is important in AdaTAD. With windowing, screen geography that would otherwise be wasted can be assigned to tasks whose state is the current focus of attention. Each logical processor has one window assigned to it.

Each logical processor also has a logical keyboard assigned to it. This keyboard is where the task running on the logical processor acquires interactive input. The logical keyboard is assigned to the physical terminal upon which the window of the logical processor is displayed when the particular task is designated as the default task.

Together the window and the logical keyboard form a logical terminal for a task. Each task takes command input from its logical terminal. AdaTAD commands may be entered from any logical terminal. The number of logical terminals that may be used with AdaTAD is limited only the number of logical processors available to AdaTAD (in the design, this limit is 256). Practical limitations on physical devices

may, however, limit the user to a few (or only one) terminals. AdaTAD is designed to work quite well with only one terminal being used for all input/output. The concept of a window is what allows this. Using AdaTAD with one physical terminal is the worst case, most complex scenario.

The user is allowed to specify the actual layout of several windows on a physical terminal. One possible configuration of the screen appears in Figure 12 on page 68. It is emphasized that this is but one possible configuration. The user may configure the screen differently by using the "window definition" command of the AdaTAD command language. With this command, the user names the window, marks it visible (or invisible) and specifies the location of the upper left corner and lower right corner of the window in terms of coordinates on the physical screen.

The user may choose to have any number of these windows displayed simultaneously (within the capability of the screen) and may shuttle back and forth among all, having only a few (or one) on display at a time.

Figure 13 on page 69 is a representation of a task window. Each window has four panes. These panes display, respectively, information about the current execution state of the task, information on designated variables, the source code context and task output.

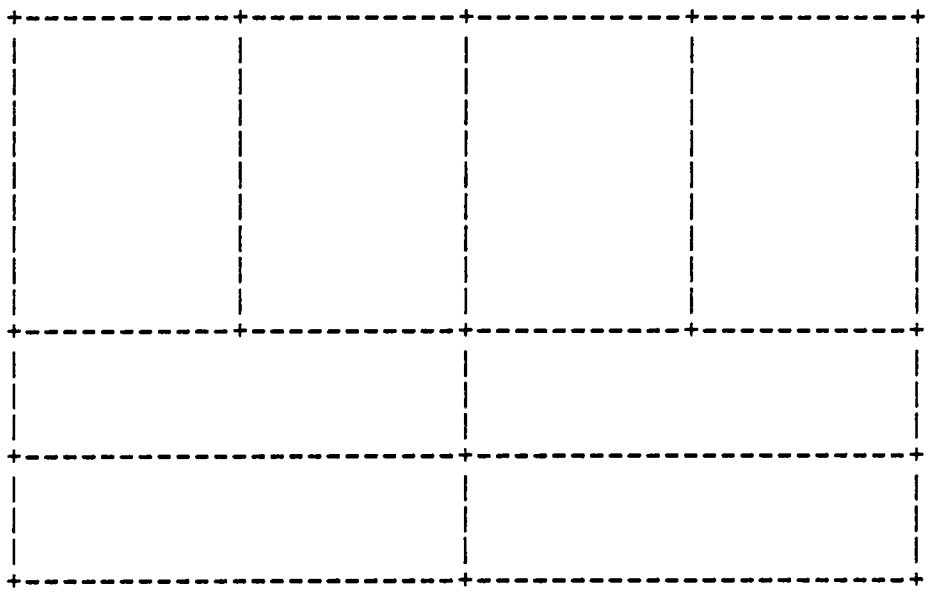


Figure 12. A Possible AdaTAD Screen

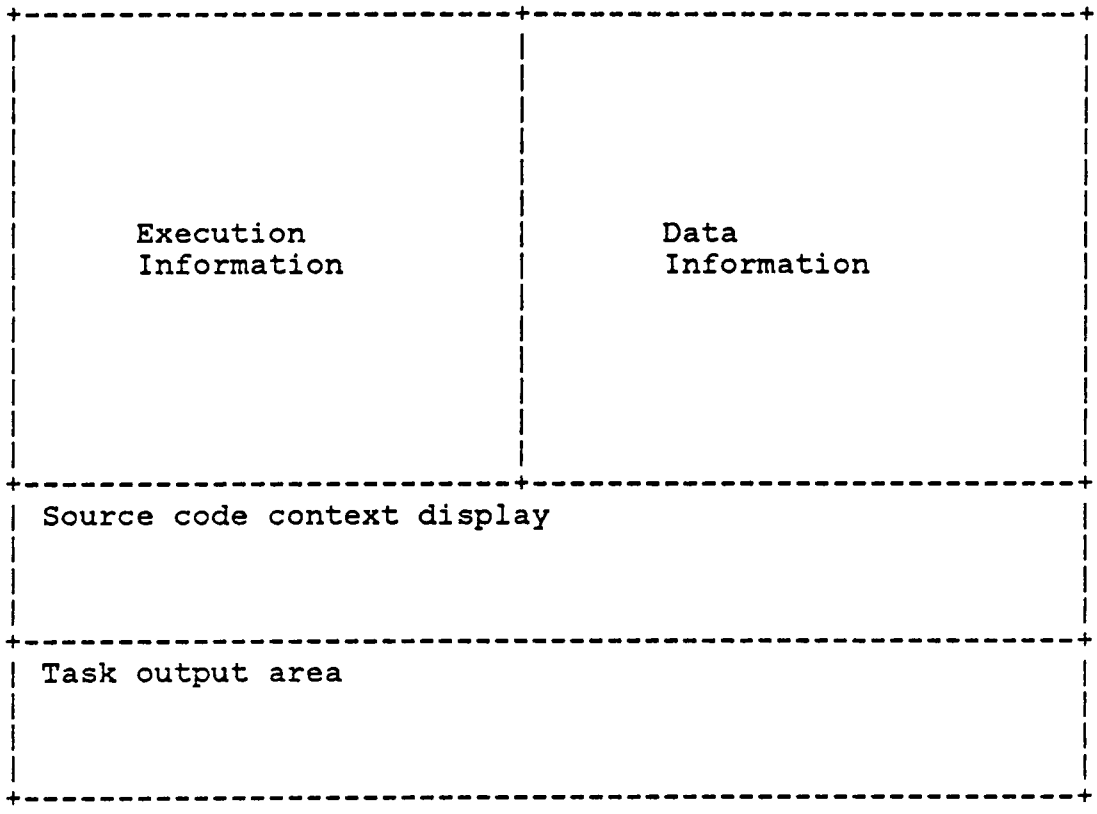


Figure 13. Task Window

4.3.2.1 Execution Information Pane

This pane displays the information regarding the current execution state of the associated task. The included information consists of the following items.

1. The tasks with which this task is synchronized

2. The current execution mode (wait, normal, timed, single step)

3. Breakpoint information

4. Reason for wait

The display of synchronized tasks is broken down into two sub-lists. The first list, called "synchronized at entry", will be those tasks which were synchronized when this task was called by the last task on that list. The other list, called "synchronized caused by", lists those tasks whose synchronization occurred because of an entry call issued by this task. This tells the user the synchronization state when the current task was called and the current synchronization state of the task.

As an example, consider the window for a task named TASKD. Suppose TASKD has accepted an entry call from a task named TASKC. When TASKD accepts the entry call, it is then synchronized with TASKC. Assume further that TASKC was already synchronized with TASKB when TASKC called TASKD. Therefore, when TASKD begins servicing TASKC, all three tasks, TASKB, TASKC and TASKD are synchronized. When TASKD begins service, the "synchronized at entry" list contains TASKB and TASKC.

Suppose that TASKD calls upon TASKE for a service before it completes its service to TASKC. When TASKE begins its service of TASKD, TASKD's "synchronized at entry" list will not change but its "synchronized by" list will contain TASKE. Also, TASKE's "synchronized at entry" list will contain TASKB, TASKC and TASKD. Figure 14 on page 72 illustrates each task's synchronization lists at a point when TASKE is providing the service to TASKD.

If the current execution mode is "timed", then the execution rate will be displayed. If the execution mode is "single step", then the release character will be displayed.

The current breakpoint checking mode is displayed in the breakpoint information area. If breakpoint checking is currently enabled, the area is used to display the status of checking whenever a breakpoint is encountered. This status includes the type of breakpoint (unconditional, logical or temporal) and the value of the associated variables.

	Synchronized at Entry	Synchronized caused by
TASKB	-	TASKC, TASKD, TASKE
TASKC	TASKB	TASKD, TASKE
TASKD	TASKB, TASKC	TASKE
TASKE	TASKB, TASKC, TASKD	-

Figure 14. Synchronization lists while TASKE executes

If the task is executable but waiting (user set execution mode is not "wait" but the task is waiting anyway), the reason for the wait is displayed in the "reason for wait" area. The task may be waiting for one of the following reasons.

1. The task is at an "accept" or "selective wait" statement.

2. The task is waiting for another task to enter into a rendezvous. That is, the current task's entry call has been queued. In this case, the name of the called task is displayed.

3. The task is waiting for another task to complete a service. In this case, the current task is in rendezvous with another task.

4. The task is waiting for an operating system service such as I/O to be performed.

The information included in the execution information pane is used to show the user the dynamics of task execution in the program. It is from this section that the user will glean information about tasks whose execution is being controlled.

4.3.2.2 Data Information Pane

When the user desires to view the value associated with a variable, the "show value" command causes the name and current value of a variable to be displayed in this pane. The user may also specify a parameter or shared variable name. If a parameter is specified, the mode (IN, OUT or IN OUT) is displayed along with the name and value. The value shown with procedural parameters is the name of the procedure currently associated with that parameter.

If the user requests display of aliases of a variable, the run time symbol table is sorted on the current addresses of the variables. In the sorted list all of the identifiers associated with the address of the specified variable are adjacent. All of those identifiers associated with the specified variable are then displayed.

4.3.2.3 Source Code Context Pane

The currently executing source statement is displayed on the middle line of this area. The immediately adjacent source statements are shown above and below the current one. As control moves to the next statement, the pane is scrolled up.

4.3.2.4 Task Output Pane

This pane displays lines of output resulting from "put" statements to the standard output device from within the task.

4.3.2.5 Zooming

The zoom feature allows the user to zoom in on a particular task for closer scrutiny. The purpose of this feature is to allow for better use of the rather limited geography of the CRT screen. If a program with several tasks is running, it is not be feasible to display all pertinent information about all tasks at the same time. Indeed, when the program is running normally, little information need be displayed about the tasks. However, when a given task becomes suspect, more information about that task must be displayed. The programmer then uses the zoom feature to focus in on that task. Information on other tasks is removed from the screen and more information about the suspect task is displayed.

The zoom allows the user to enlarge a window temporarily. It is used when one task begins to execute a critical section of code that the user wants to scrutinize. The user could do this by using the "window definition" command. However, the definition of the windows is considered fairly

static during an AdaTAD session. The zoom allows a temporary change in the use of screen geography without changing the window definition.

If the terminals in use with AdaTAD are high resolution graphics terminals, then a continuous zoom is used. The visual effect in this case is very much like that of a zoom lens on a television or motion picture camera. If the window or pane is being zoomed to a larger size, other visible windows are overlaid. If the window or pane is being zoomed to a smaller size, windows visible when the expansion zoom was made is restored.

If the terminals in use are not high resolution graphics terminals, then zooming is restricted somewhat. A window or pane may be zoomed from its originally defined size to the size of the full screen or vice versa. There are no intermediate sizes for a zoomed window or pane in this case. The zoomed window is always that of the default task. The ZOOM command causes the zoom in or zoom out.

Due to the large amount of information to be displayed and the small amount of screen geography available on most video display terminals, a great deal of pertinent information will probably not be on display at some arbitrary moment. The zooming feature allows the user to select one particular task for scrutiny. When the user signals AdaTAD that a zoom is to be made on a given task, AdaTAD clears the

appropriate parts of the screen and display the increased amount of information on the selected task.

The zoom feature affects the panes of a window differently. The execution information pane is static and the information in it does not change. When a window is zoomed in (made larger), the data information area becomes physically larger to accommodate display of more variables. The source code context pane and the task output area pane also become physically larger, allowing more information to be displayed in them. The zoom feature does not allow the user to change the aspect ratio of a window.

4.3.3 A Sample Debugging Session

The purpose of this section is to present a sample debugging session. The debugging session demonstrates the use of several AdaTAD commands and depicts the displays resulting from the commands. More importantly, the debugging session demonstrates how AdaTAD can be used to find an obscure bug that causes deadlock in a program with tasks.

The program to be debugged is an Ada implementation of the Producer-Consumer program. The source code for this program resides in Appendix B. There are three tasks in the program. One is the producer task, one the consumer task and the other is the buffer control task. The producer task creates objects which it gives to the buffer control task as

long as the buffer control task can take another object. The consumer task takes objects from the buffer control task as long as objects are available. If the buffer control task cannot accept more objects (the buffer is full), the producer must wait until space is available in the buffer before it may produce new objects. If no objects are available from the buffer control task (the buffer is empty), the consumer task must wait until an object is available before it consumes an object.

The source code that appears in Appendix B has an error in it. It is a typographical error that could easily occur. Its presence is easy to detect because the error causes deadlock shortly after the program begins execution. However, the location of the error is not obvious and is best found with the aid of an automated debugger.

Figure 15 on page 79 lists the commands used to debug the Producer-Consumer program. The line numbers included after three of the commands are not part of the AdaTAD command language but are included here for annotation purposes.

Figure 16 on page 80 through Figure 24 on page 88 depict the screens visible to the user after commands PC1, PC2 and PC3, respectively, depicted in Figure 15 on page 79, are executed.

When the program deadlocks with the Producer task waiting rendezvous with the buffer control task, the user views the value associated with the variable CE. Since the value

```
$ run adatad consumer_producer
$$ name txa0: pc_cons
$$ name txa1: bc_cons
$$ window producer visible 0 80 0 24
$$ window consumer visible 0 80 0 24
$$ assign producer pc_cons
$$ assign consumer pc_cons
$$ window buf_control visible 0 80 0 24
$$ assign buf_control bc_cons
$$ producer
$$ set breakpoint on
$$ set breakpoint at <<ck1>>
$$ set normal
$$ consumer
$$ set breakpoint on
$$ set breakpoint at <<ck1>>
$$ set normal
$$ buf_control
$$ set normal
$$ go  -- PC1   begin execution.
$$ producer
$$ set normal
$$ go
$$ consumer
$$ set normal
$$ go  --  PC2 one element has now been produced and consumed.
$$ producer
$$ set normal
$$ go
$$ consumer
$$ set normal
$$ go  --  two elements have now been produced and consumed
$$ producer
$$ set normal
$$ go
$$ consumer
$$ set normal
$$ go  -- three elements have now been produced and consumed
$$ producer
$$ set normal
$$ go  --  system now deadlocks,  producer waiting.
$$ show variable ce  -- PC3    is 0, should be 3
$$ terminate
```

Figure 15. Commands to Debug Producer-Consumer Program

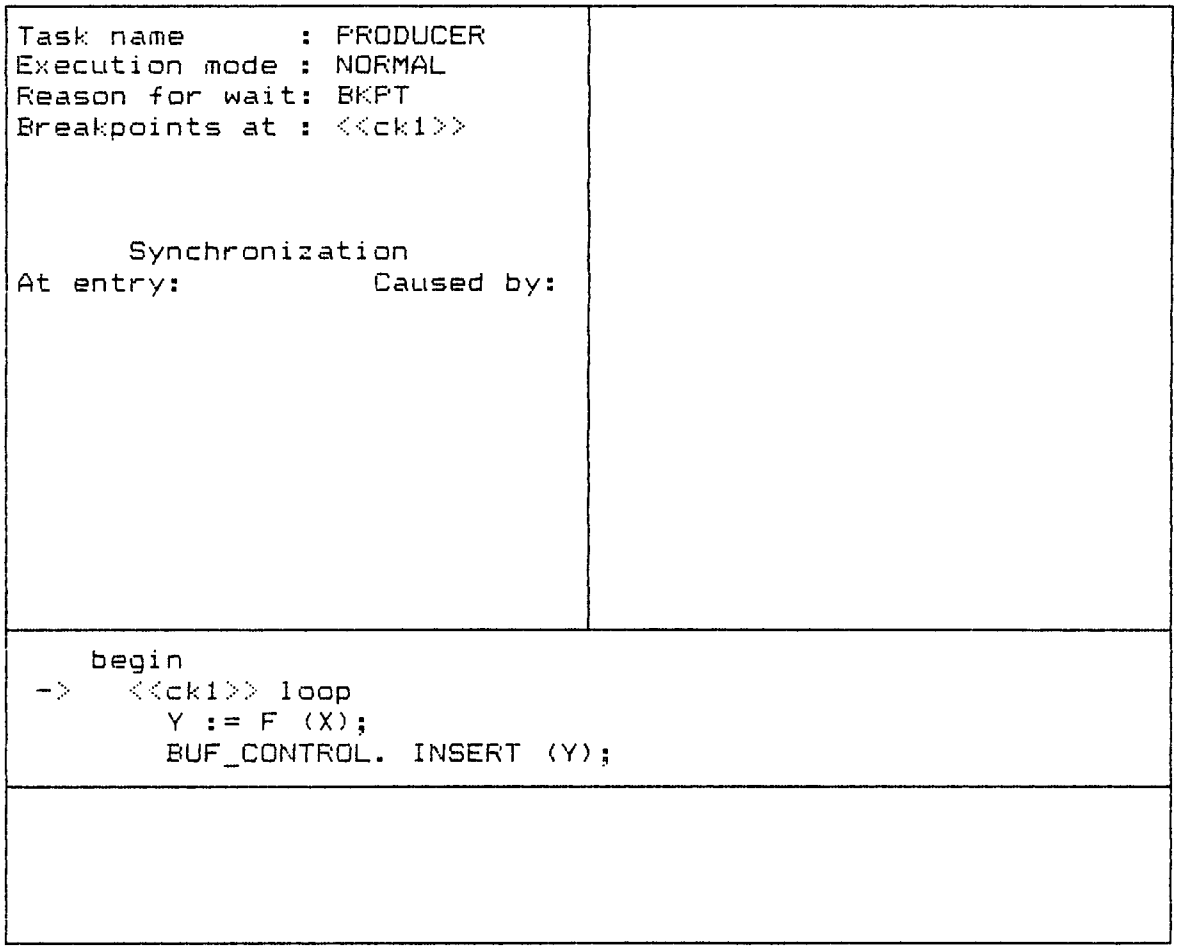


Figure 16. Producer's Screen after PC1

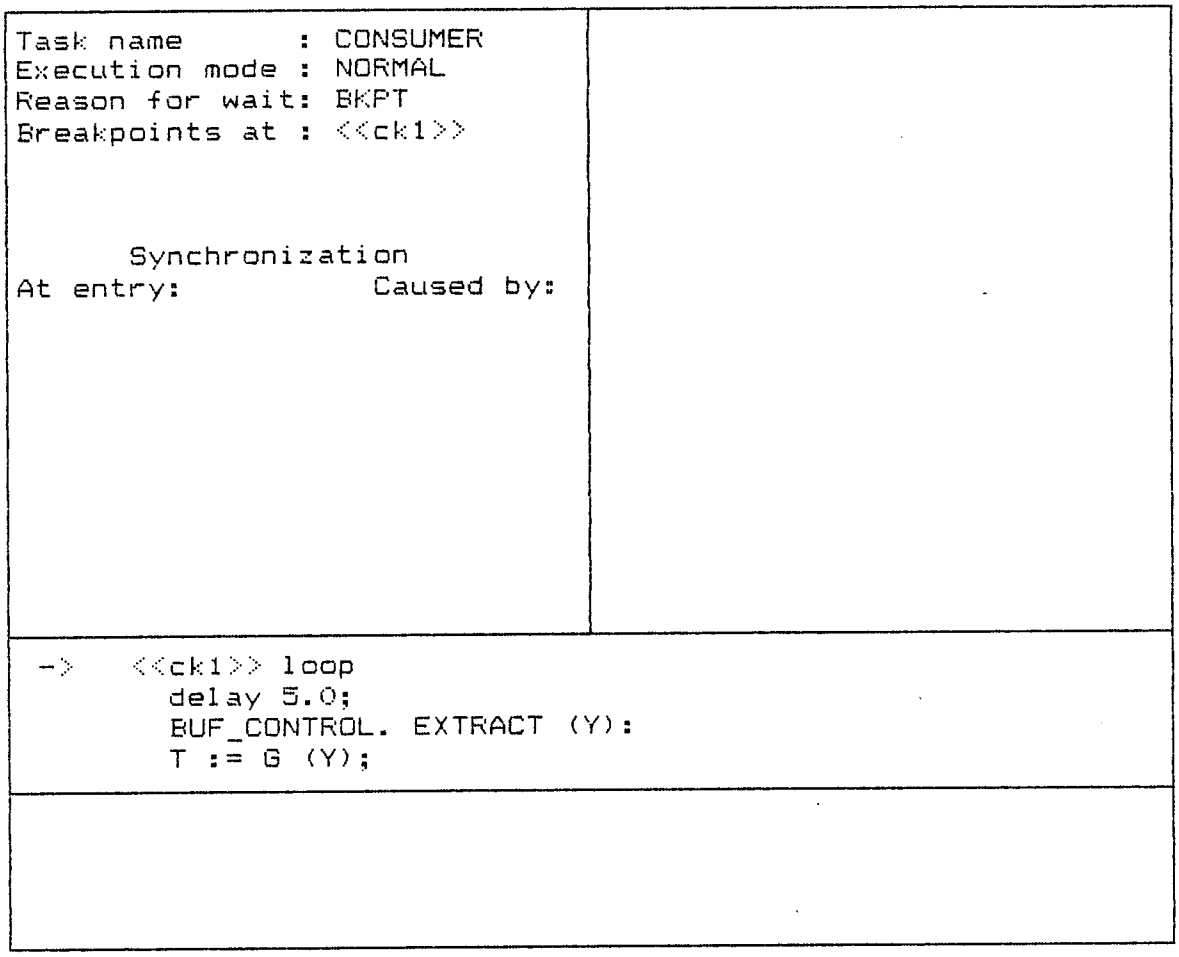


Figure 17. Consumer's Screen after PC1

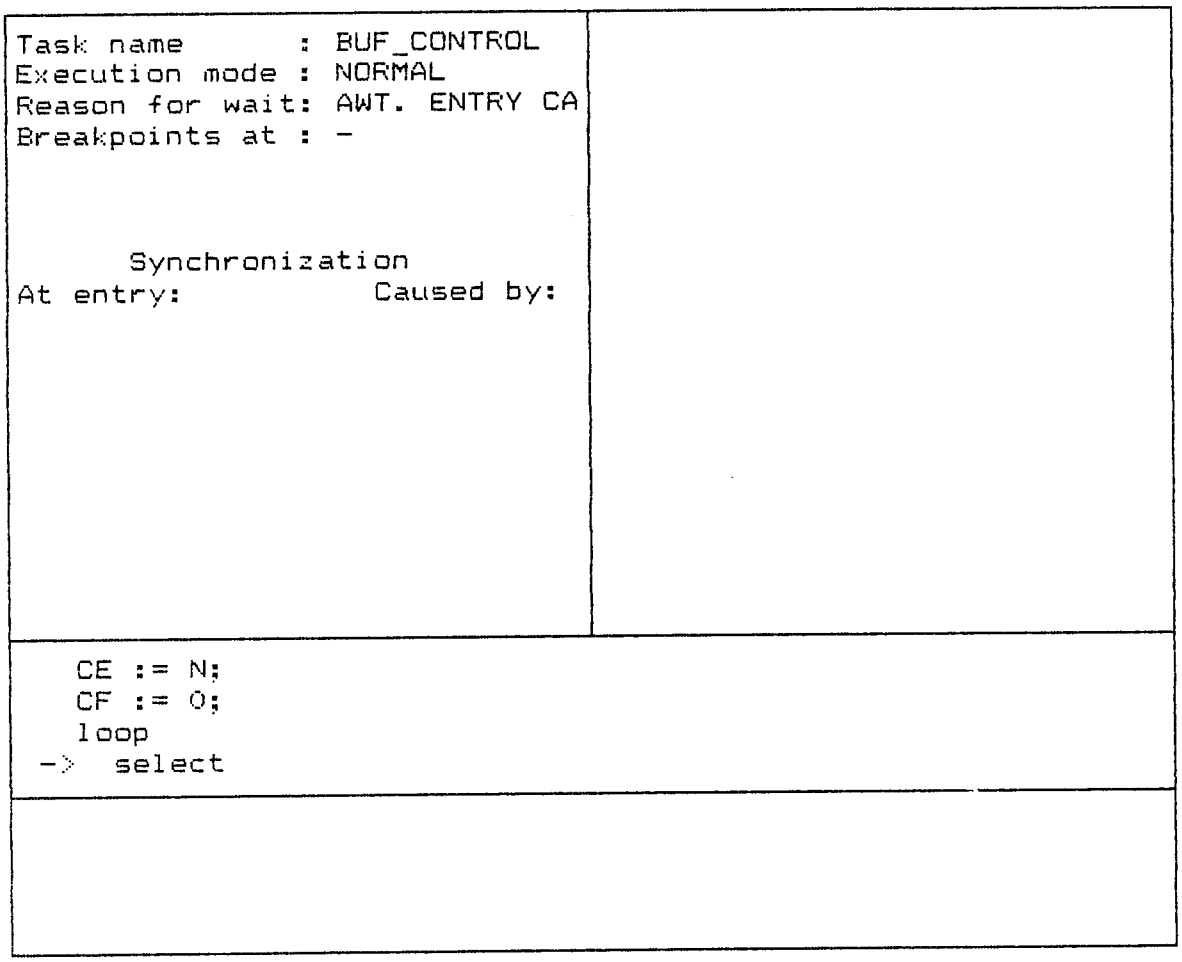


Figure 18. Buffer_Control's Screen after PC1

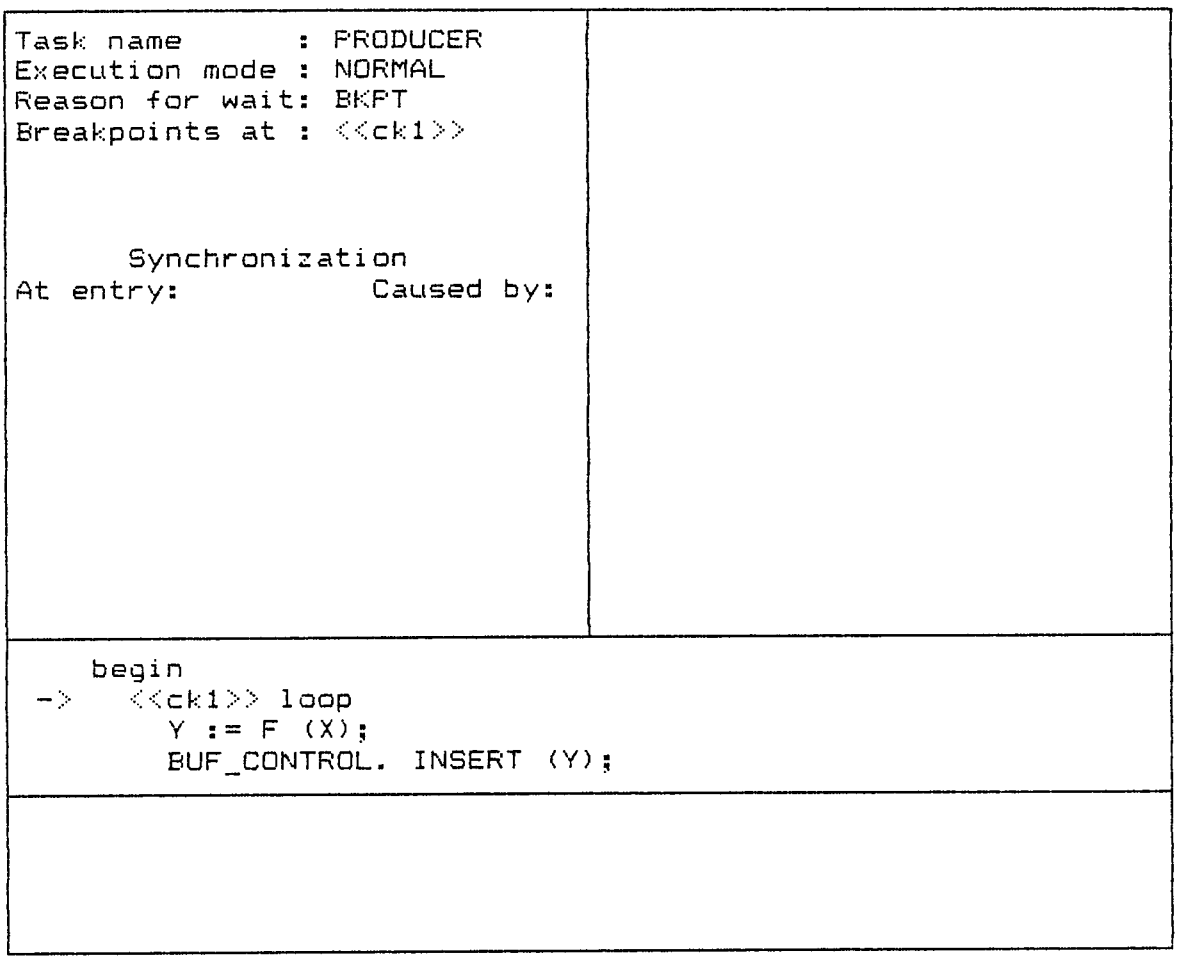


Figure 19. Producer's Screen after PC2

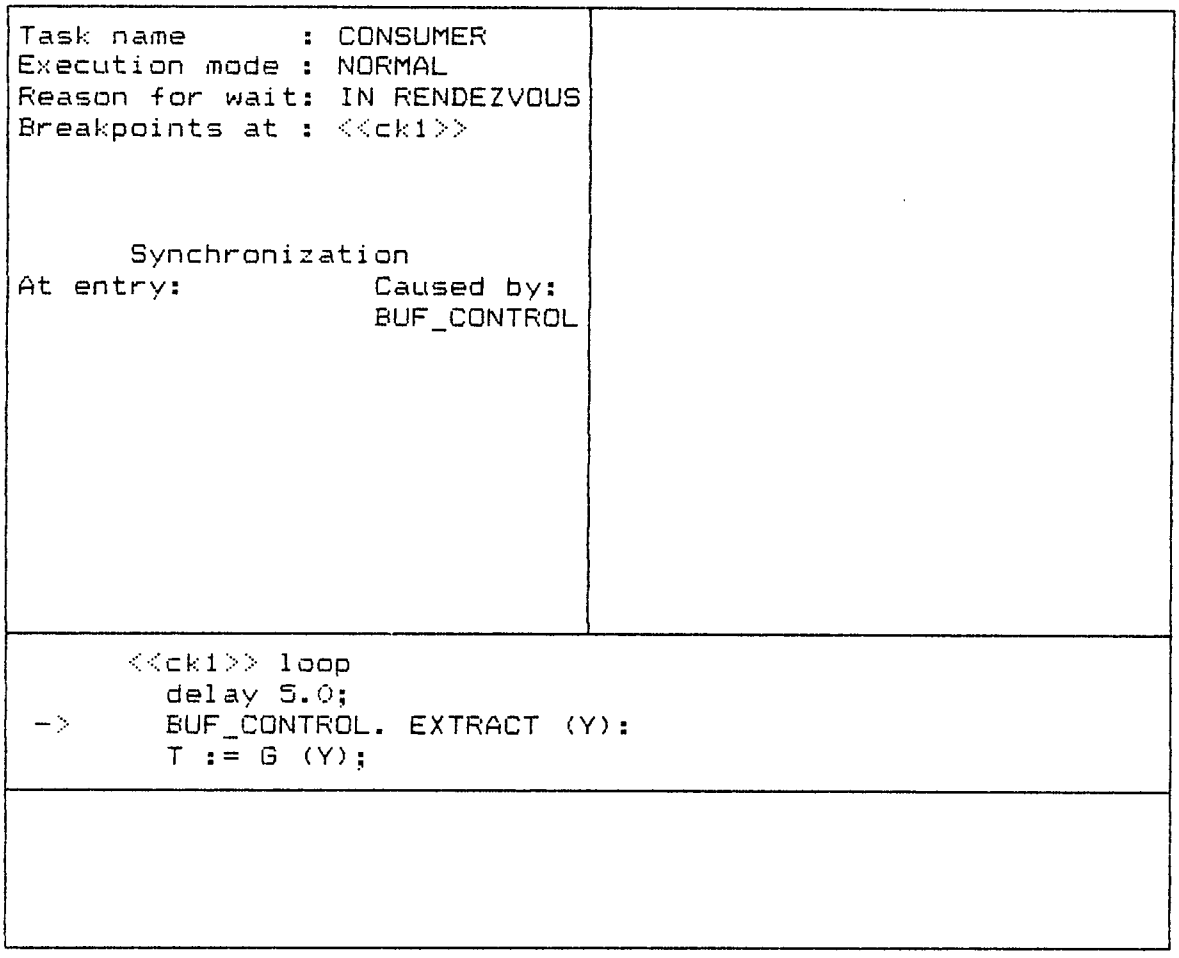


Figure 20. Consumer's Screen after PC2

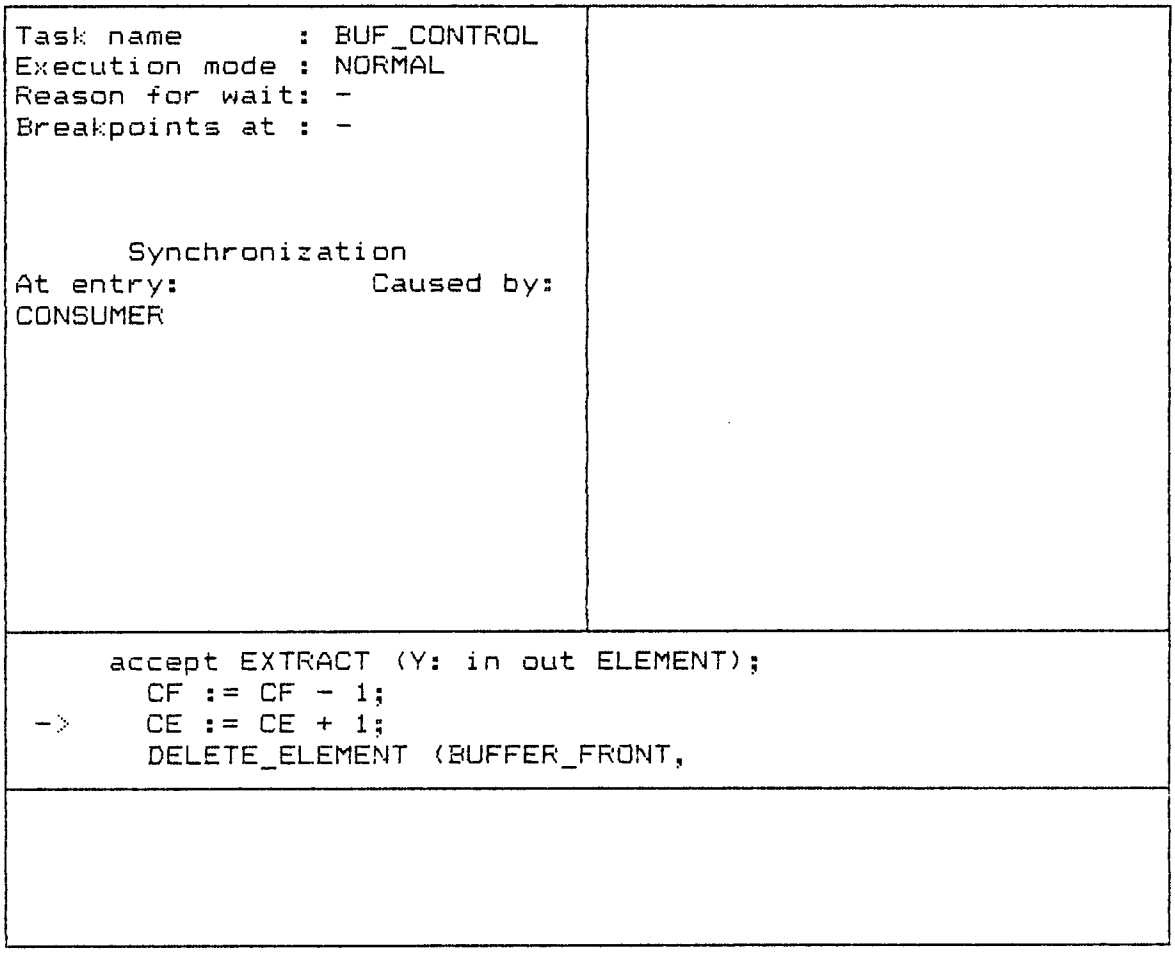


Figure 21. Buffer_Control's Screen after PC2

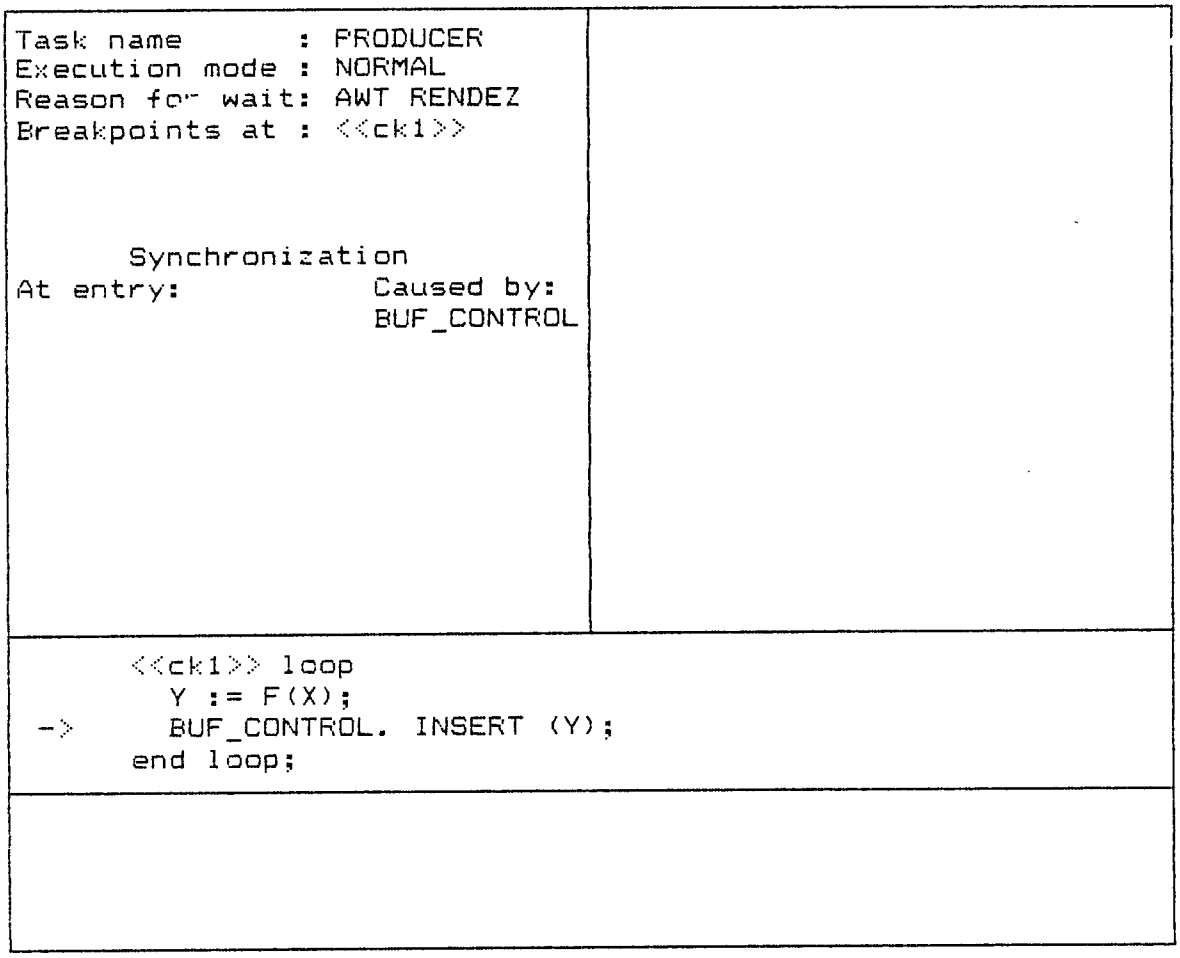


Figure 22. Producer's Screen after PC3

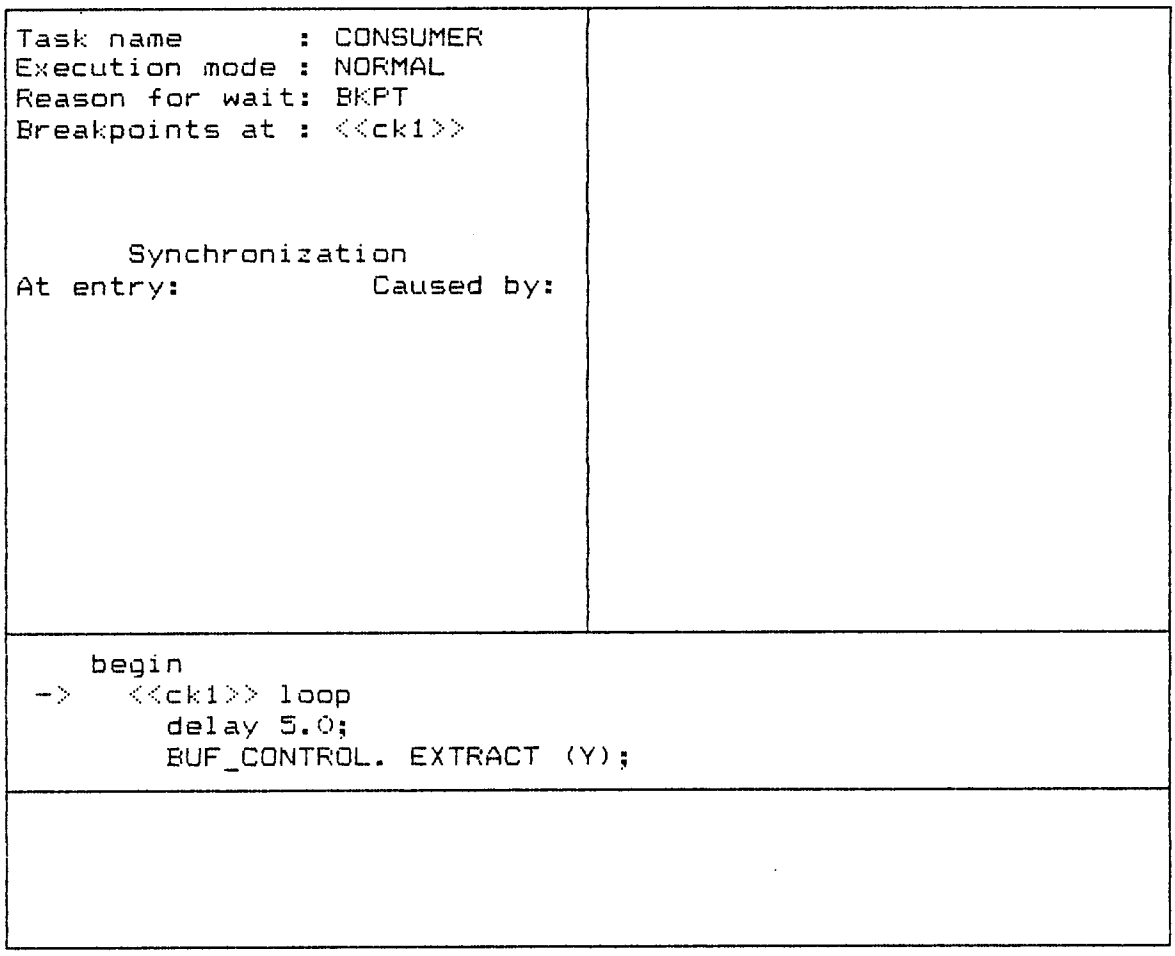


Figure 23. Consumer's Screen after PC3

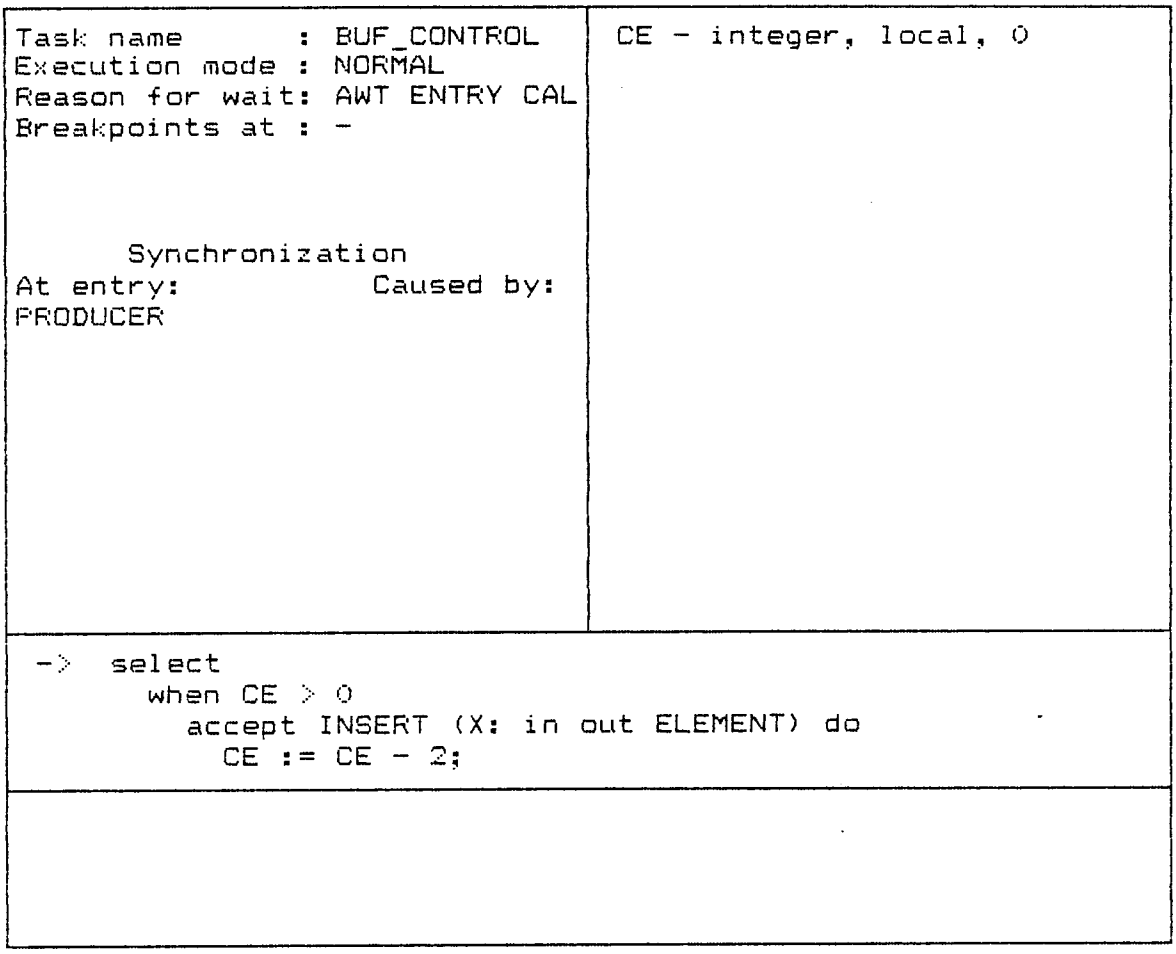


Figure 24. Buffer_Control's Screen after PC3

is zero with the buffer empty, it is clear that an assignment statement has associated an improper value with the variable CE. The user then examines assignment statements whose left hand side is CE. There are three of these in the source code. One assigns CE an initial value and the other two should increment and decrement the value by one, respectively. However, the incrementing statement increments the value by two. Here then, is the error.

4.4 MAJOR MODULE FUNCTIONALITY

This section describes the functionality of each of the modules depicted in Figure 9 on page 53. This section does not deal with the details of inter-task communication. That topic is discussed in "Inter-task Communication" on page 111.

4.4.1 Logical Processors

The logical processor is probably the most complex task in AdaTAD. This is because it has much to do to monitor and control the execution of the user task. Figure 25 on page 90 is a block diagram of the logical processor. It uses numerous rendezvous requests to communicate the current state of execution to its internal data base and to the data base that the AdaTAD coordinator maintains. This section explains

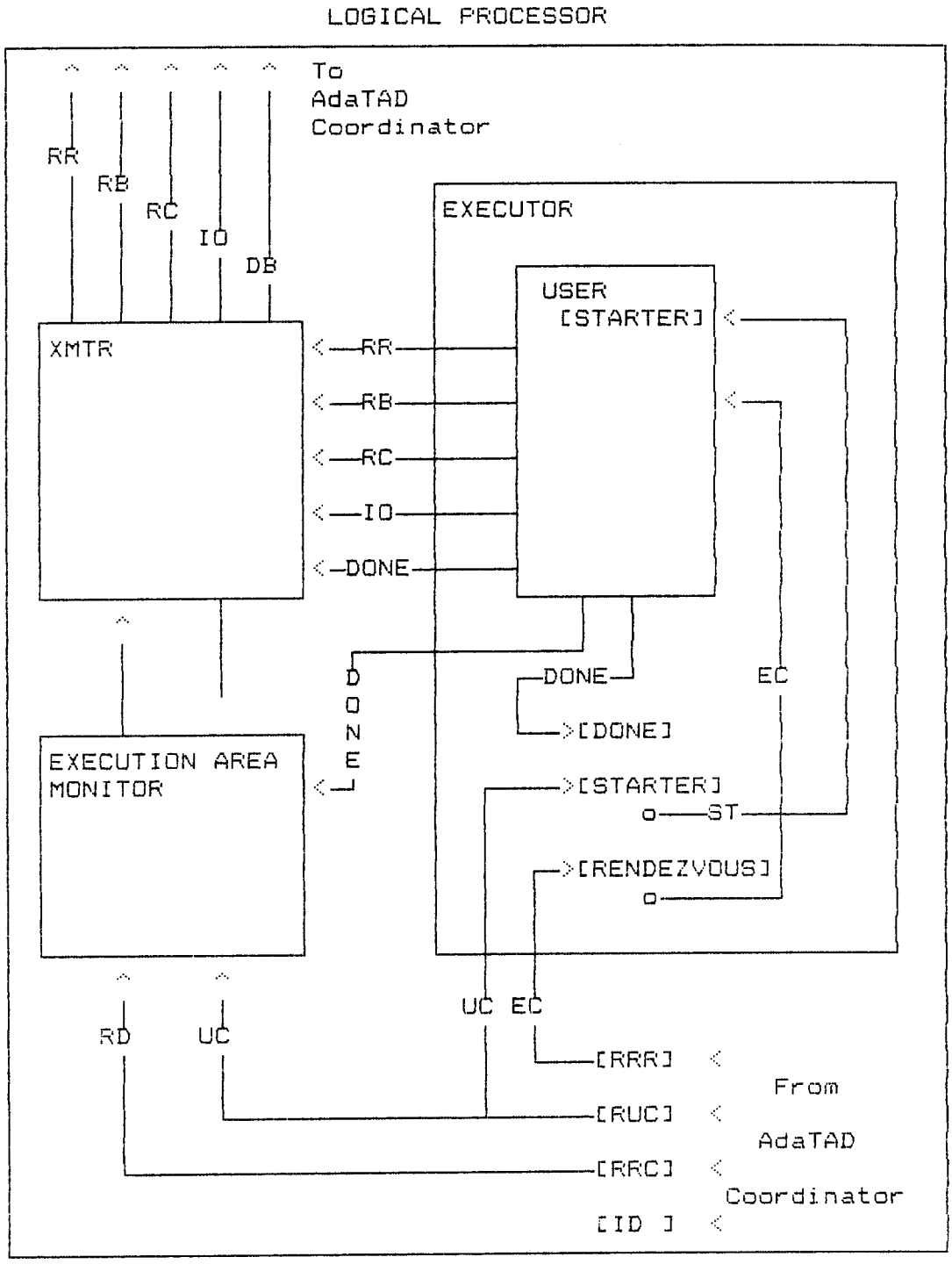


Figure 25. Diagram of Logical Processors

the use of each entry in the task. The "Synchronization of User Tasks" section describes the interactions involved in the communication of user tasks.

The logical processor itself has four entries. They receive commands from the command interpreter, requests for rendezvous from the user tasks, notification of rendezvous completion from servicing tasks and notification of task termination from the logical processors. It declares three tasks of its own. The EXECUTOR task is the controlling environment for the user task. The TRANSMITTER task serves as a funnel through which all messages bound for the coordinator are sent. The EXECUTION_AREA_MONITOR maintains the variables which reflect the current execution state of the user task. This task utilizes the TRANSMITTER task to send the updated execution state to the coordinator. The presence of the three tasks allows for maximal parallelism in the execution of the logical processor. This effect minimizes the time spent by the user task in synchronization with the logical processor and so the execution of the user task more closely approaches the execution without AdaTAD. This minimizes the Heisenberg Uncertainty effect mentioned at the beginning of the chapter. The entries to these are also discussed.

4.4.1.1 Entry Receive_user_command

This entry is called by the AdaTAD coordinator when a command is to be executed by the logical processor. The command itself is received in an integer parameter; any ancillary information such as command parameters are received through a character string parameter. A case statement in this entry selects the proper code to implement the command. With only two exceptions, the implementation of the commands at this level involve setting values in the execution data base. For example, if the user wants to change the execution state, the command is issued and the execution state is changed in the data base.

The two breakpoint commands do not involve the execution data base. When the user issues the command to define a hard breakpoint, the coordinator sends the label to the logical processor. The logical processor parses the label and then looks the label up in the symbol table. This gives the location of the label in the object code. Then the location in the executable image is computed by getting the entry point of the task from the activation record and adding this value to the label offset. This computation is machine-dependent and this overly simplified description is presented here as an example. The address computation yields the location of the small data area in the prologue to the statement for which the breakpoint is to be defined. The logical

processor then sets a flag there indicating that a hard breakpoint has been established for that statement. If the command defined a soft, or assertion, breakpoint, then the same process just outlined occurs but a flag is set in the prologue indicating that the breakpoint is an assertion breakpoint. Additionally, the "message" parameter to the entry contains an ASCII string holding the expression to be evaluated as the assertion. The logical processor invokes a translator to compile the assertion expression into executable code. This code is then appended to the user task's executable image and the location of the code is placed in the prologue to the statement for which the breakpoint is set. When the statement is executed, the prologue indicates that an assertion breakpoint has been defined for this statement and also causes a branch to the code that evaluates the expression. The result of the assertion computation is placed in a location in the statement prologue and the assertion code returns to the statement prologue that invoked it.

4.4.1.2 Entry Receive_rendezvous request

This entry is called by the AdaTAD coordinator when another task desires rendezvous with the task running on this logical processor. The parameter is the location of the argument list. All that this entry does is receive the argument list and then call the Executor's rendezvous entry. The

Executor is a task local to the logical processor which directly controls a user task's execution. The Executor is explained below.

4.4.1.3 Entry Receive_rendezvous_completion

This entry is called by the AdaTAD coordinator when a rendezvous that was requested by the task running on this logical processor has been completed. The entry updates the local data base so that this task can continue execution. Any arguments which were changed by the servicing task now exist in the argument list that was constructed by this task before the rendezvous request was made.

4.4.1.4 Task Executor

This task is the direct controller of the user's task. The compiler and linker set up the user's task so that it is a task local to the Executor. The user's task begins running when the Executor begins running. The Executor has one entry which is called by the logical processor's "receive_rendezvous_request" entry when the coordinator has sent an entry call. After being called, the executor extracts the name of the calling task and places it in the global variable "caller". Then, the entry name is extracted and the procedure which the compiler generated to actually

do the user task entry call is invoked. The executor repetitively performs these activities.

4.4.1.5 Task Transmitter

The purpose of the transmitter is to send messages to the AdaTAD coordinator. It is called by the user's task for the purpose of requesting a rendezvous, requesting an input/output service, informing the coordinator that a rendezvous is beginning or has completed. It is called by the execution area monitor to send the current state of the data base to the coordinator.

4.4.1.6 Task Execution_area_monitor

Since the execution data base is a shared variable that must be accessed by the Executor, the Transmitter and the Logical Processor itself, the task EXECUTION_AREA_MONITOR is the only task with access to that data. Any other task requesting information from that data base must get the information by making an entry call to the monitor. The task loops on a selective wait with an else clause. The entries are as follows.

Sing_step_rel: This entry is called by the logical processor after the coordinator has signaled that the user has pressed

a key to cause execution of the next statement in single step mode. The entry sets the execution enable flag to true so that the next statement may be executed.

Set_bk_state: This entry is called by the logical processor when a command has been entered enabling or disabling breakpoint checking. The data base is updated accordingly.

Set_ex_md: This entry is called by the logical processor whenever the execution mode is to be changed. This may occur upon receipt of a command to do so or when a breakpoint is evaluated to a value other than true.

Set_ex_rt: This entry is called by the logical processor when a command to set the rate for timed execution.

Set_ex_un: This entry is called by the logical processor as part of the command to enter timed execution mode. The value set in the data base by this entry indicates whether the number in the execution rate variable is in statements per second or seconds per statement.

Examine_exe: This entry is called by the statement prologue to see if the next user task statement can be executed. It has an OUT parameter by which it returns the current value of the execution enable flag.

The else clause: The else clause is executed if there are no outstanding entry calls to the monitor. This clause sets the execution enable flag based upon the current execution mode and other parameters affecting the execution state. The clause determines the current execution mode and then takes appropriate action. If the execution mode is TIMED, it determines if it is now time to execute the next statement. If it is, it sets the execution enable flag to true and computes the time of execution for the next statement. If it is not time to execute the next statement, the clause insures that the execution enable flag is false. If the mode is single step, then the clause insures that the execution enable flag is false if it was true, indicating that a statement has just been executed. If the mode is NORMAL, the clause insures that the execution enable flag is true. Otherwise (mode is WAIT), the execution enable flag is set to false.

4.4.2 Coordinator

The AdaTAD coordinator is a task whose function is to mediate the communication of among other AdaTAD tasks (and ultimately the user's tasks). Figure 26 on page 98 is a block diagram of the coordinator. The coordinator loops on a selective wait. It, therefore, accepts entry calls to any of its entries in the order in which they come. The coordi-

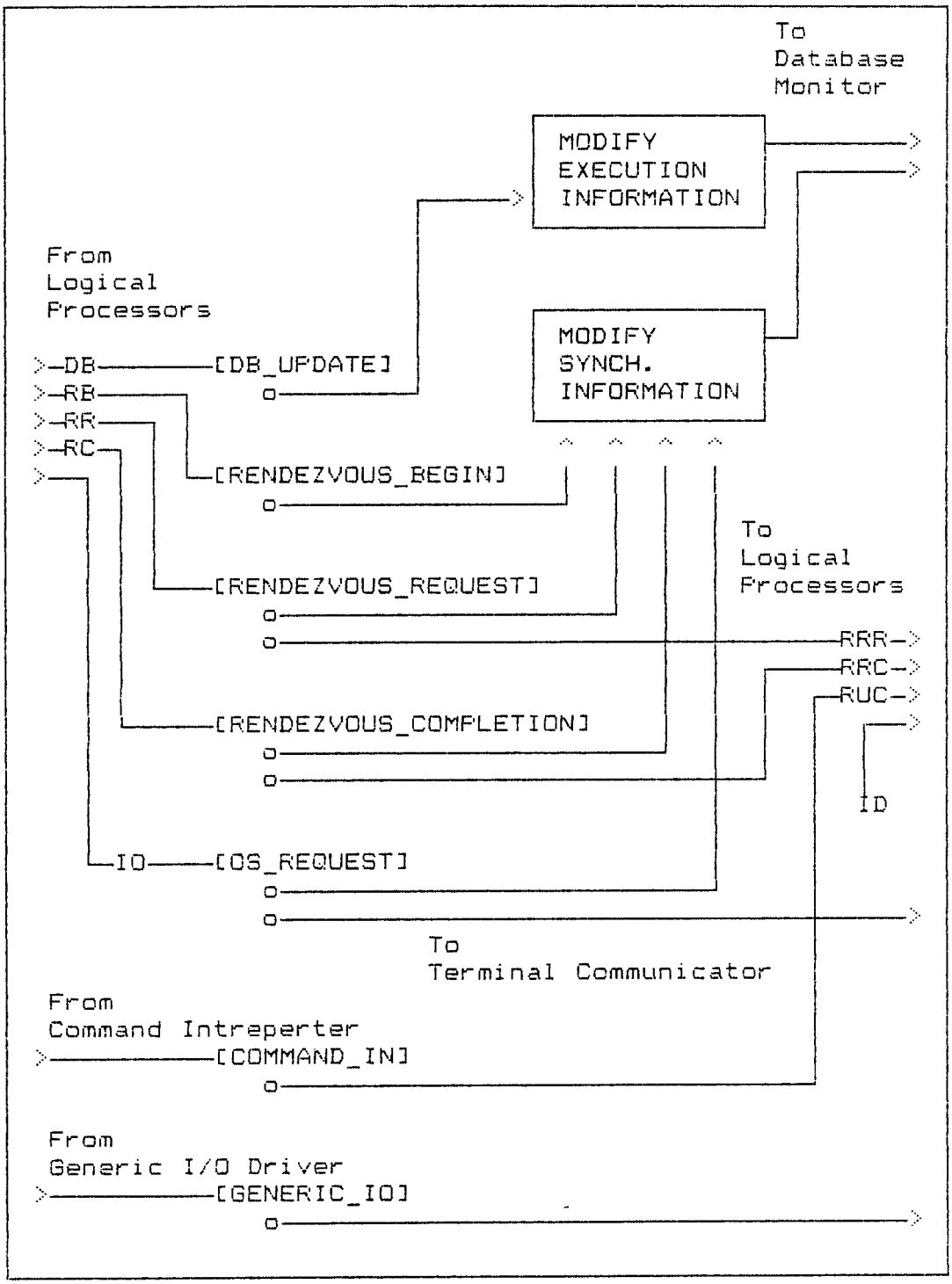


Figure 26. Diagram of the Coordinator

nator is described in terms of the functionality of its entries. The following sub-sections describe each entry.

4.4.2.1 Rendezvous_request

Whenever a task running in a logical processor requests a rendezvous with a task running in another logical processor, the logical processors control the rendezvous through the mediation of the coordinator. Mediation occurs by the calling logical processor making an entry call to this entry of the coordinator. The accept statement code first looks up, in the headers of the execution data base, the logical processor running the called task. Then the awaiting synchronization bit in this task's data base entry is set, indicating that the task has requested a rendezvous, and finally, the coordinator makes an entry call to the logical processor task running the desired task. The parameter for this entry is the location of a descriptor list. The name of the called task is taken from the first node of the descriptor.

4.4.2.2 Rendezvous_begin

The servicing, or called, task makes calls to this entry. It is called with the names of the two synchronized tasks when the requested rendezvous begins. This entry up-

dates the synchronization information for the two tasks. It clears the "waiting" bit, sets the "is_synchronized" bit and places the names of the called and calling tasks into their proper locations in the synchronization data base.

4.4.2.3 Rendezvous_completion

When the servicing task has completed its service, it causes its logical processor to call this entry. This occurs when the servicing task either terminates or encounters the "end" statement of an "accept do" statement or completes executing an "accept" statement. The entry updates the synchronization data base to reflect the completed rendezvous and makes an entry call to the logical processor running the served task so that that task may now continue its execution. The single parameter for this entry is the name of the task which has been served.

4.4.2.4 OS_request

Any terminal related input/output that the user's task must perform is mediated by this entry. The entry has two parameters; the first is a value indicating whether the request if for input or output. The second is a parameter list in the same format that is passed between user tasks at rendezvous. The entry looks up the name of the task's I/O device

and passes that, along with the other parameters to the terminal communicator.

4.4.2.5 IO_driver_interrupt

This entry is called by the task which handles non-video terminal I/O devices. Because of the large number and diversity of these devices, no attempt is made here to define specifically the interactions that take place.

4.4.2.6 Command_in

The command processor task calls this entry when it has successfully parsed a user's command. The two parameters are, respectively, an integer indicating which command was entered by the user and character string containing any arguments to the command. This entry actually defines the semantics of the user's command language. The case statement in this entry chooses the appropriate code to be executed depending upon what the command was.

4.4.2.7 DB_update

Each logical processor has execution time data local to it. It is that data that actually control the execution of the user's tasks. However, when that data change, AdaTAD

must be informed of the change. This entry is called by each logical processor periodically to update the execution time data base that AdaTAD keeps for display purposes. The parameters are the current execution state and the task name to whom the execution state applies. The entry looks up the index of the logical processor running the task and then uses this to index into the data base. The appropriate portion of the data base is then updated. The entry uses a local procedure to perform the update. The local procedure synchronizes with the data base monitor task, blocking it from looking at the data base (which is a shared variable) while the data are being updated. At the completion of the update, the local procedure unblocks the monitor and lets it continue to scan the data base.

4.4.3 Data Base Monitor

The purpose of the data base monitor is to send user task information to the terminal communicator for display. This task loops on a selective wait with else clause. If none of the three entries have been called when the selective wait is encountered, then the else clause is executed. This clause transmits the current state of the data base to the terminal communicator. The entire data base is sent before the monitor may allow update of the data base. Therefore, no inconsistent data base is ever transmitted.

4.4.3.1 Hold

This entry is called by the coordinator when it wants to update the data base. Hold waits for the coordinator to call the entry "release" and then completes. The synchronization with the coordinator does not include waiting for the "accept release" to be called. If it did, then the coordinator and data base monitor would deadlock.

4.4.3.2 Release

This entry is called by the coordinator after it has completed updating the data base. The entry is solely for synchronization purposes. This entry is only accepted after a previous call has been made to the "hold" entry.

4.4.3.3 Done

This entry is called by the coordinator when AdaTAD is about to terminate.

4.4.4 Terminal Communication

A task's terminal input/output is controlled by the logical processor, through the mediation of the terminal communicator. The terminal communicator also provides the

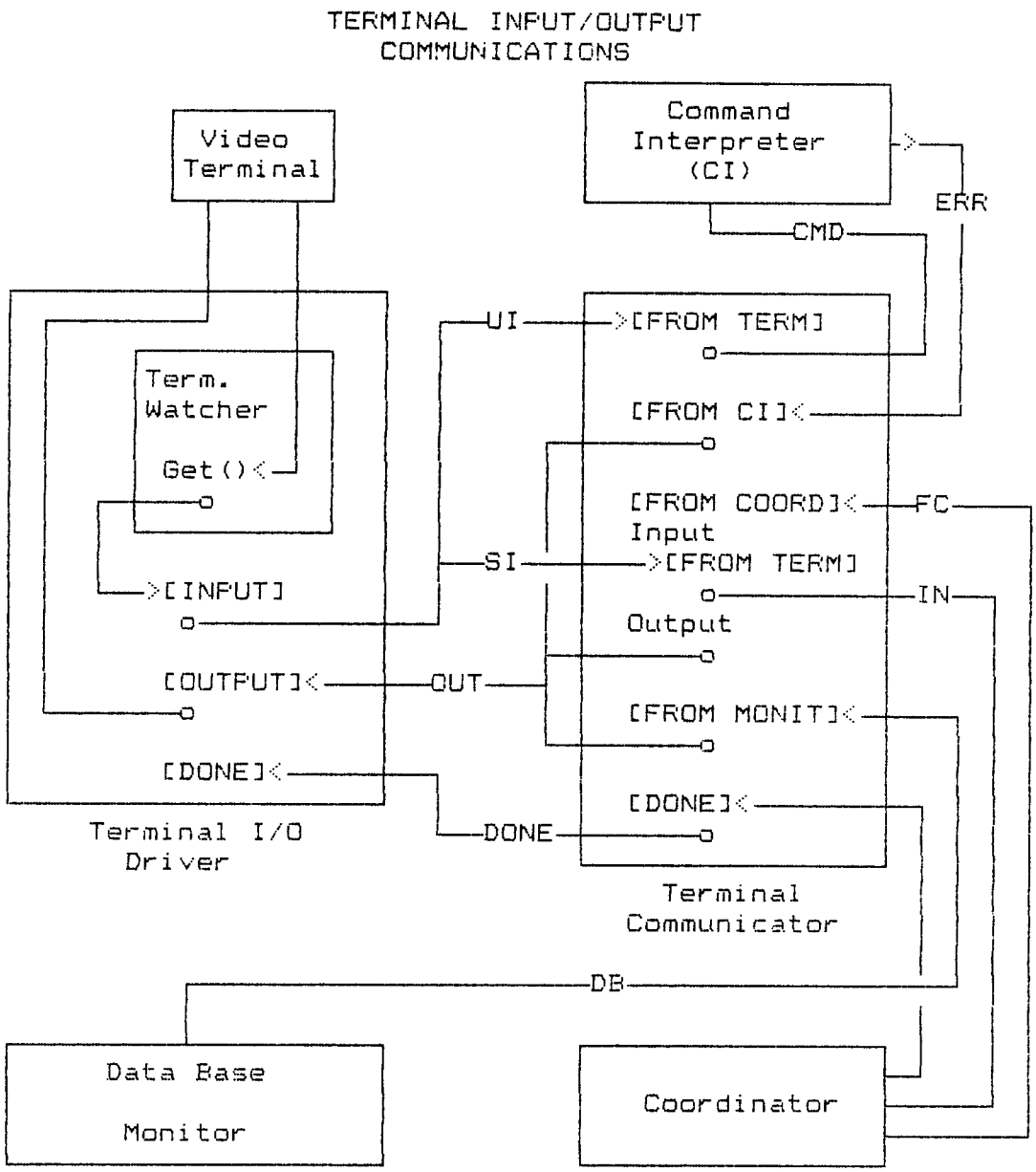


Figure 27. Diagram of the Terminal Communicator

intelligence for display of the AdaTAD data base. The terminal communicator manages the windowing capability of AdaTAD. The five entries in this task receive information from the coordinator, the terminals, the data base monitor and the command interpreter. Figure 27 on page 104 is a block diagram of the terminal communicator.

4.4.4.1 From_terminal

The terminal communicator task has two accept statements for the "from_terminal" entry. The first handles unsolicited input from a terminal. Assuming that unsolicited input is a command, the first accept receives an information string and passes that string along to the command processor task. For example, when the user enters the string "SET WAIT", the terminal communicator assumes that this is a command and sends it to the command processor.

The second accept for the "from_terminal" entry is used for input of solicited information. "From_terminal" is accepted after accepting the "from_coord" entry described below. This scenario occurs when a user task has requested terminal input. The string entered at the terminal is returned in the argument descriptor sent by the coordinator. It does not go through the command interpreter.

4.4.4.2 From_coord

This entry is called by the coordinator when a user task desires input or output. There are three parameters. The first is the number of the logical processor that is running the task requesting I/O service. The second is a code indicating whether the action is to be input or output (a value of 1 implies input, a value of 2 implies output). The third is a pointer to an argument descriptor list the same as that used for user-task rendezvous. The argument list contains information on the variables whose value is to be written, in the case of output, or whose value is to be read from the terminal, in the case of input.

If the user is requesting output, then the "then" branch of an "if" statement is executed. Within this code, the values to be written are converted to character strings and sent to the appropriate terminal.

If the user task is requesting input, then the second "from_terminal" accept statement is encountered. At that time, the communicator awaits input from the specified terminal. When obtained, the string is converted to the specified data type and returned to the requesting task.

4.4.4.3 From_monitor

The monitor calls this entry to cause the current information about task execution and synchronization to be displayed. The terminal communicator moves the information from parameters into the appropriate window data structure and then displays all of the visible windows on the appropriate terminal.

The windows are stored in a doubly linked list. When the user requests creation of a window, a new structure is created and placed on the list. Information stored in this list includes the name of the task assigned to the window, the terminal upon which the window is to be displayed, a boolean indicator as to whether the window is currently visible, the location of the window on the terminal, the boundaries of the window and all the values of the task associated with the window. Only one task is allowed per window.

4.4.4.4 From_cmd_int

This entry is called by the command processor when it has detected an error in a user command. This entry displays the error message on the terminal from which as the command was entered.

4.4.4.5 Done

The "done" entry is called by the AdaTAD coordinator when AdaTAD is about to terminate. This entry shuts down all the device drivers and then stops the terminal communicator.

4.4.5 Command Interpreter

The command processor (also called the command interpreter) has the job of analyzing the user commands. When a command is successfully parsed, it is dispatched, along with its parameters, to the AdaTAD coordinator for execution. Even commands which affect information display are executed by the coordinator. If a command is erroneous, nothing is sent to the coordinator. An error message is sent directly back to the terminal communicator.

The internal procedure "analyze_command" actually does the lexical and syntactic analysis of the command. Details of this procedure are not included here or in the source code because this analysis is a common, well-known technique. The semantic analysis of the commands is done where the commands are used. Therefore, the semantics of commands destined for the coordinator are defined there while semantics of the commands dealing with display of data are defined in the terminal communicator.

4.4.5.1 Parse

"Parse" is the only entry into the command processor. "Parse" is called by the terminal communicator when unsolicited input occurs on a terminal. The command processor awaits input to the accept for "parse".

4.4.6 Terminal Drivers

The terminal drivers are an array of tasks. They handle the transmission of data between the physical terminals and the terminal communicator. Since they are all alike, the remainder of this discussion speaks in the singular while meaning the plural. The terminal driver has four entries and one internal task which has no entries.

4.4.6.1 Identify

This entry is called by the terminal communicator as soon as the driver begins execution. The terminal communicator assigns the driver a number by which the communicator knows the driver. This is done once when AdaTAD starts and the communicator and driver remember the identification number.

4.4.6.2 Output

This entry is called by the terminal communicator whenever a string must be written on a terminal. The terminal communicator converts the data to be output into an ASCII string and determines which terminal is to receive the output. It then calls the "output" entry in the terminal driver. This entry accepts that string and then writes it on the device to which it is physically attached.

4.4.6.3 Input

This entry is called by the internal task, terminal_watcher and is passed a character string. The entry calls the terminal communicator with the string and the driver's identification number.

4.4.6.4 Done

This entry is called by the terminal communicator when AdaTAD is about to terminate.

4.4.6.5 Terminal_watcher

This is a task whose sole job is the wait for an input string from the terminal. When a string has been received

(terminated by a terminator character, usually a carriage return), the task makes an entry call to the terminal driver's "input" entry, passing the string.

4.4.7 Other I/O Device Drivers

This task is dependent upon the devices connected to the hardware running AdaTAD. Basically, all this task does is shuttle characters between the AdaTAD coordinator and the actual device driver.

4.5 INTER-TASK COMMUNICATION

This section describes the communication among AdaTAD Logical Processor tasks and Coordinator task. The discussion is couched in terms of an example of user tasks in rendezvous.

The control of the synchronization of user tasks is undoubtedly one of the most complex actions that AdaTAD performs. That is because AdaTAD must intervene when a rendezvous request is made, when the rendezvous begins and again when the rendezvous ends. In order to keep track of this, the compiler, as noted above, converts user entry calls to entry calls to AdaTAD. In this way, AdaTAD knows of all user entry calls and can log their occurrence. The compiler also generates code to inform AdaTAD of when the rendezvous

actually starts and when it completes. Of course, all these actions occur whenever a rendezvous request is made, but they are normally transparent to the user. The following subsections describe what occurs in each case of AdaTAD intervention.

Figure 28 on page 113 is a representation of the synchronization of two user tasks. Assume that task A wants to make an entry call to task B's entry named E1. Since this section deals with task synchronization, assume that no data are passed during the rendezvous. Assume further that task A is running in logical processor one and that task B is running in logical processor two.

4.5.1 Rendezvous Request

Task A has an entry call statement of the form B.E1. For this call, the compiler generates code to produce an empty argument list, "alist", which consists only of the head node. This node is necessary because it names the calling task, the called task and the called entry, which is information AdaTAD needs. The compiler converts the statement B.E1 into

```
TRANSMITTER. SEND_RENDEZVOUS_REQUEST (alist);
```

```
task body A is
  begin
   -- A calls B here
   B. e1;
  end A;

task body B is
  begin
   Loop
        .
        .
   -- B accepts entry calls here
   Accept e1 Do;
        .
        .
   -- and completes the rendezvous here.
   End e1;
        .
        .
   End loop;
  end B;
```

Figure 28. Example of User Task Synchronization

Therefore, the first action that takes place at execution time when task A is ready to make this rendezvous is that the transmitter is invoked.

The transmitter's send_rendezvous_request entry accepts the call and immediately sets task A's execution mode to "wait". Then, the transmitter makes an entry call to the coordinator, passing the argument list along unchanged.

The request for rendezvous arrives at the coordinator's "rendezvous_request" entry. The coordinator looks in the head node of the argument list, gets the name of the called task (not the entry), in this example, B, and gets the number of the logical processor that is running the called task. The coordinator then looks up the called entry name in the task data base. In this example, the entry is E1. The coordinator uses the number to index the array of tasks which implement the logical processors. Next, the coordinator makes an entry call to the "receive_rendezvous_request" entry of the appropriate logical processor. At this point, the synchronization information on the calling task will be updated to reflect that the task is waiting for a rendezvous.

When the logical processor accepts the rendezvous request, it passes the argument list to the executor running the servicing task. At this point, AdaTAD knows that a rendezvous request has been made and that the calling task is in a wait state for that rendezvous. Further, the user notices on the display that the calling task has entered a wait

state awaiting a rendezvous. The display also indicates the task being called, the state of the calling task and any other tasks awaiting rendezvous.

When the executor receives the request at its "rendezvous" entry, it extracts the name of the calling task and the name of the called entry from the head of the argument list. The name of the calling task is used later to tell the coordinator that the rendezvous is in progress. The name of the entry allows the executor to request the proper entry into the user's task. The executor calls the procedure written by the compiler for the receiving task. This procedure decodes the argument list and executes the entry call into the user's task.

Assume that the called user task (B) is waiting at the entry that is being called. In that case, the executor's entry call is answered immediately and the user's task begins execution.

4.5.2 Acceptance of a Rendezvous Request

The first thing that the user task's accept statement for E1 does is make an entry call to the transmitter with the name of the calling task. This entry call is to the "send rendezvous beginning" entry. The "send rendezvous beginning" entry sets the called task's execution data base to reflect that the called task is now running, and then the coordinator

is informed that the rendezvous is beginning. The coordinator acts on this information by updating its synchronization information data base. The user would now see that the rendezvous is in process.

After the user's task indicates that the rendezvous has been accepted, AdaTAD does not intervene. This guarantees that the user task is consistent with the rules of Ada and that effects of the Heisenberg Uncertainty Principle are minimized. A user observing the synchronized behavior of the tasks would see that they obey the rules of synchronization prescribed by Ada.

When the rendezvous between A and B is complete, the servicing task, B, encounters a call to the transmitter's entry "send rendezvous completion". The servicing task remains in a running state until it reaches a point where it must wait for another rendezvous. The transmitter sends a message to the coordinator that the rendezvous is complete. (If data had been communicated between the user tasks, then any parameter with OUT mode would have had the new value computed by the servicing task copied into the storage of the parameter list.)

As far as the servicing task's logical processor is concerned, the rendezvous is now over. However, there is still work for the coordinator to do. Upon receiving notification of the termination of the rendezvous, the coordinator updates its synchronization data base to reflect the

end of the rendezvous. As far as the coordinator is concerned, the rendezvous is now over and it sends a message so indicating to the "receive rendezvous completion" entry in the logical processor running the calling task. When the calling task's logical processor receives this message, the calling task's execution mode is set to "run" so that it can proceed.

5.0 RESULT OF ADATAD RESEARCH

In actuality, AdaTAD itself is the result of this research. The literature indicated a need for a task oriented debugging tool [Ston81] and a lack of such a tool. A set of requirements for the tool were located [Baia83, Webe83]. Though not specifically among the requirements, inclusion of some form of Manna and Pneuli's [Mann81] temporal logic aided in satisfying the requirements. No work had been reported on designing a debugger with the requirements, so that work was proposed and carried out. The results are reported here.

The design described in this document possesses several important characteristics. They are:

1. It fulfils the requirements stated in "Requirements of a Task Debugger" on page 28 for a task debugger.

2. AdaTAD's design is entirely in Ada, a characteristic which provides two important features.

    a. It provides transportable debugging capability among different machines.

b. It provides distributed debugging capability for user programs whose tasks are running on different processors in a distributed environment.

3. Though inclusion of temporal logic was only carried to a high level in the design, this provides the basis for inclusion in an implementation.

5.1 ADATAD FULFILS REQUIREMENTS

The list of requirements from "Requirements of a Task Debugger" on page 28form the basis for the sections "Sequential" through "Execution Order" on page 122. Each section mentions a requirement and explains briefly how AdaTAD fulfills that requirement.

5.1.1 Sequential

Requirement: The debugger must have the characteristics of current technology single program debuggers.

These features are not detailed in the design of AdaTAD presented because that design is specific to those features needed for concurrency. The sequential features are not new or innovative and would merely clutter the issues described in this work. They are to be included in the implementation of AdaTAD.

5.1.2 Execution State

Requirement: The debugger must display the execution state of each task.

In the examples presented in Chapter 4, the windows associated with each task always display a line indicating whether or not the task is running or waiting. If it is waiting, the reason for the wait is displayed. AdaTAD performs this action by monitoring the execution state of each task. The data base monitor is the module that performs the monitoring. The data base itself is maintained by the coordinator under direction of each logical processor. When the execution state of any user task changes, the logical processor notifies the coordinator and the coordinator in turn updates the data base.

5.1.3 Synchronization

Requirement: The debugger must display the state of task synchronization.

This requirement is fulfilled in much the same way as the requirement to monitor execution state. Whenever a user task desires synchronization with another user task, the request is routed through the coordinator. When the synchronization actually occurs and when it terminates, the coordinator is informed. The coordinator maintains the data

base with this information. A detailed description of how this is done is in "Inter-task Communication" on page 111.

5.1.4 Execution Context

Requirement: The debugger must display the "execution context" in the source language of each task.

In each task window there is a pane called the execution context pane. This is described in "Execution Information Pane" on page 70. The window always displays the currently executing source statement and several statements surrounding that statement. This is accomplished by keying the executing code to the source code. The exact mechanism for doing this is described in "Object Module Format" on page 38.

5.1.5 Execution Speed

Requirement: The debugger must allow the user to control the relative execution speeds of the tasks so that the debugger will not interfere with the temporal relationships of the tasks.

In each task window is a line displaying the current execution speed in statements per second or seconds per statement. This informs the user of the currently set execution speed. The user may change the execution speed using the "set execution mode" command described in "Description

of AdaTAD commands" on page 59. Allowing the user to control speed of execution is accomplished with the "statement prologue", described in "Statement Prologue" on page 40.

5.1.6 Execution Order

Requirement: The debugger must allow the user to control the order in which statements of each task are executed so that the user may experiment to find any undesired temporal dependencies.

Setting of breakpoints and placing tasks in a wait state are the two tools provided by AdaTAD to fulfil this requirement. The commands to do this are described in "Description of AdaTAD commands" on page 59. The statement prologue, "Statement Prologue" on page 40, is the entity that actually carries out the control of interleaving.

5.2 IMPORTANCE OF DESIGN IN ADA

No part of the design of AdaTAD relies upon a particular machine dependent feature. It depends upon no service which cannot be provided at the APSE or KAPSE level. AdaTAD is designed entirely in Ada and can be implemented entirely in Ada. The importance of this cannot be overemphasized because this is what allows AdaTAD to provide truly transportable, distributed debugging.

5.2.1 Transportability of AdaTAD

AdaTAD's design in Ada guarantees transportability among all implementations of Ada. The executable code is obviously not transportable but the source code can be moved to any machine which runs an Ada environment. Once on that target machine, that source code is compiled with the Ada compiler already on the machine. The result is an executable version of AdaTAD for the target machine.

5.2.2 Distributed Debugging

Given the appropriate support for a distributed environment, AdaTAD has the capability to perform debugging operations on that distributed system. Appropriate support means that the standard Ada environment must exist on the distributed system.

On such a system, the user simply compiles and links the tasks on the systems on which they will run. A version of AdaTAD exists on each system and the user task is linked with AdaTAD on the appropriate system. Once the link is completed, the distributed nature of the debugging is as transparent to the user as the distributed nature of the execution of the program. This gives the user the advantage of debugging in the actual execution environment that the program uses once it is in production.

5.3 TEMPORAL LOGIC

The desirability of an implementation of temporal logic is clear. It is the natural next step in logic implementation for languages. Though a design for temporal logic is not completely specified in AdaTAD, a syntax for temporal expressions is described, "Description of AdaTAD commands" on page 59, and ideas for the approximation of eternity are discussed, "Temporal Logic" on page 11. These ideas form a basis for incorporating temporal logic in a future version of AdaTAD.

6.0 DISCUSSION AND CONCLUSION

This chapter discusses the significance of the research conducted and presents some ideas for future work.

6.1 SIGNIFICANCE

The significance is that a concrete design for a task debugger now exists where none did before. The "Stoneman" [Ston81] report specifically mentions inclusion of automated debuggers in the APSE. AdaTAD is such a debugger. Since it incorporates features of sequential debuggers in addition to its task capabilities, it may be the only debugger required for the APSE. The debugger satisfies the requirements specified in the literature [Baia83, Webe83] and presents a start toward implementation of temporal logic [Mann81].

This design is an important advance in debugging technology because it permits distributed debugging and because it is machine independent, allowing for transportability of the debugger among all systems running Ada.

Though not innovative, AdaTAD's modular design allows for easy modification of the user interface. This means that an implementation of AdaTAD might embody command, menu-driven and iconic user interfaces and the user may choose whichever is best. Moreover, as the technology of human-computer

interfaces is advanced, new interface types may be included with a minimum of modification to AdaTAD.

6.2 FUTURE WORK

Clearly, the first step for work beyond that reported here is to implement the design of AdaTAD. This will entail no small amount of work because not only must the debugger be implemented, but the Ada compiler's generation of code must be modified and the linker must also be modified. This will be a major undertaking.

Additional future work includes implementing temporal logic [Mann81]. The work reported in this document can form a basis for that implementation.

The ideas of task debugging are intriguing and application of them to other languages is an interesting idea. Versions of PL/I, Pascal, Lisp and FORTRAN exist which support concurrency. The ideas developed in designing AdaTAD may well apply to these languages as well. A concurrent debugger may be particularly attractive to users of multiprocessor or multicomputer machines since such machines often achieve much of their performance with parallel processing.

The concept of task debugging is but one example of a larger, more general area which might be called "smart debugging." Historically, automated debugging has been a machine code oriented activity in that the user needed to be

familiar with the execution time specifics of the program. Smart debugging is a concept which would lessen the requirement the requirement that the user be familiar with the machine code implementation of the program. A smart debugger would allow the user to ignore the low-level implementation of the program and concentrate on the high-level abstraction represented by the source language program. Such a debugger would go beyond modern technology source level debuggers by providing facilities for handling higher level concepts such as tasks.

7.0 BIBLIOGRAPHY AND REFERENCES

ACM72 Proceedings of an ACM Conference on Proving Assertions about Programs. New Mexico State University, Las Cruces, New Mexico, 1972. Association for Computing Machinery, New York, 1972.

Ada83 Ada Language Reference Manual.

Aho74 Aho, A. V., Hopcroft, J. E. and Ullman, J. D. The Design and Analysis of Computer Algorithms. Addison-Wesley, Boston, Mass., 1974.

Aiel83 Aiello, L. and Nardi, D. "Intelligent man-machine interfaces." in Theory and practice of software techonology. Proc. international seminars on software engineering (Capri, Italy, 1980 & 1982), D. Ferrari et al. (Eds.), Elsevier North-Holland, Inc., New York, 1983, 1-20.

Alle68 Allen, J. and Luckham, D. C. "An Interactive Theorem Proving Program". Machine Intelligence, (vol 5), New York, American Elsevier, 1968.

Alle76 Allen, F. E. and Cocke, J. "A Program Data Flow Analysis Procedure". CACM, 19(3), Mar. 1976, pp 137-147.

Alle82 Allen, R. "Cognitive factors in human interaction with computers." in Directions in human/computer interaction. A. Badre and B. Shneiderman (Eds.), Ablex Publ. Corp., Norwood, NJ, 1982, 1-26.

Ambl77 Ambler, A. L., Good, D. I., Browne, J. C., Burger, W. F., Cohen, R. M., Hock, G. C. and Wells, R. E. "GYPSY: A Language for Specification and Implementation of Verifiable Programs". Proceedings ACM Conference on Language Design for Reliable Software SIGPLAN Notices, 12(3), Mar. 1977.

Andr83 Andrews, G. and Schneider, F. "Concepts and notations for concurrent programming." Comput. Surv. 15, 1 (Mar. 1983), 3-43.

Baia83 Baiardi, F., De Francesco, N., Matteoli, E., Stefanini, S. and Vaglini, G. "Development of a Debugger for a Concurrent Language." Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering

Symposium on High-Level Debugging Mark Scott Johnson, ed. ACM, March, 1983.

Balz79 Balzer, R. and Goldman, N. "Principles of Good Software Specification and Their Implications for Specification Languages". Proc IEEE Conference on Specification of Reliable Software, Cambridge, Mass., Apr. 1979.

Bate83 Bates, Peter and Wileden, Jack C. "An Approach to High-Level Debugging of Distributed Systems (Preliminary Draft)." Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging Mark Scott Johnson, ed. ACM, March, 1983.

Bell83 Bell, D., Kerridge, J., Simpson, D. and Willis, N. Parallel programming - a bibliography. John Wiley & Sons, Inc., New York, 1983.

Ben82 Ben-Ari, M. Principles of concurrent programming. Prentice-Hall, Inc., Englewood Cliffs, N.J., 1982.

Ben84 Ben-Ari, M. "Algorithms for on-the-fly garbage collection." ACM Trans. Prog. Lang. Syst. 6, 3 (Jul. 1984), 333-344.

Berg82 Berg, H. K., Boebert, W. E., Franta, W. R., Moher, T. G. Formal Methods of Program Verification and Specification. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1982.

Blum82 Blum, E. "Programming parallel numerical algorithms in ADA." in The relationship between numerical computation and programming languages. Proc. IFIP TC 2 working conference (Boulder, CO, Aug. 3-7, 1982), J. K. Reid (Ed.), Elsevier North-Holland, Inc., New York, 1982, 297-304.

Boye81 Boyer, R. and Moore, J. (Eds.) The correctness problem in computer science. Academic Press, Inc., New York, 1981.

Bran80 Branstad, M. A., Cherniavsky, J. C. and Adrion, W. R. "Validation, Verification and Testing for the Individual Programmer". Computer, 13(12), Dec. 1980, pp 24-30.

Bran80A Branstad, M. A. and Adrion, W. R. (eds). "High Level Language Programming Environments". ACM SIGSOFT, 6(4), Aug. 1981.

Bran81A Branstad, M. A. and Adrion, W. R. (eds). "Contemporary Software Development Environments". ACM SIGSOFT, 6(4), Aug. 1981.

Brow72 Brown, J. R., DeSalvio, A. J., Heine, D. E. and Purdy, J. G. "Automated Software Quality Assurance". in Program Test Methods, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1972, pp 76-92.

Broy82 Broy, M. "A fixed point approach to applicative multiprogramming." in Theoretical foundations of programming methodology. Lecture notes of an international summer school directed by F. L. Bauer, E. W. Dijkstra and C. A. R. Hoare (Munich, West Germany), M. Broy and G. Schmidt (Eds.), D. Reidel Publ, Co., Hingham, MA., 1982, 565-623.

Budd78 Budd, T. A., DeMillo, R. A., Lipton, R. J. and Sayward, F. G. "The Design of a Prototype Mutation System for Program Testing". Proceedings 1978 NCC.

Burs68 Burstall, R. M. "Semantics of Assignment". Machine Intelligence, (vol 2), New York, American Elsevier, 1968.

Burs69A Burstall, R. M. "Proving Properties of Programs by Structural Induction". The Computer Journal, 12(1), 1969, pp 41-48.

Burs69B Burstall, R. M. "Programs and Their Proofs - An Algebraic Approach". Machine Intelligence, (vol 4), New York, American Elsevier, 1969.

Char83 Charlton, C. and Leng, P. "Aids for pragmatic error detection." Softw. Pract. Exper. 13 1 (Jan. 1983), 59-66.

Chea79A Cheatham, T. E., Jr., Townley, J. A. and Holloway, G. H. "A System for Program Refinement". Proceedings 4th International Conference on Software Engineering, Munich, 1979.

Chea79B Cheatham, T. E., Jr., Holloway, G. H. and Twonley, J. A. "Symbolic Evaluation and the Analysis of Programs". EE Trans. Soft. Eng., SE-5, 4, July 1979.

Cher84 Cherry, G. Parallel programming in ANSI Standard ADA. Reston Publishing Co., Reston, VA, 1984.

Clar76 Clarke, L. A. "A System to Generate Test Data and Symbolically Execute Programs". IEEE Trans. Soft. Eng., SE-2, Sept. 1976, pp 215-222.

Clar83 Clarke, Lori A. and Richardson, Debra J. "The Application of Error Sensitive Testing Strategies to Debugging." Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging Mark Scott Johnson, ed. ACM, March, 1983.

Cole82 Coleman, D and Gallimore, R. "Partial correctness of distributed programs." in Program Specification. Proc. workshop (Aarhus, Denmark, Aug. 4-7, 1981), J. Staunstrup (Ed.), Springer-Verlag, New York, 1982, 138-180.

Coop67 Cooper, D. C. "Mathematical Proofs About Computer Programs". Machine Intelligence, (vol 1), New York, American Elsevier, 1967.

Coop69 Cooper, D. C. "Program Scheme Equivalences and Second Order Logic". Machine Intelligence, (vol 4), New York, American Elsevier, 1969.

Dahl72 Dahl, O.-J., DIjkstra, E. W. and Hoare, C. A. R Structured Programming. Academic Press, London and New York, 1972.

Darr78 Darringer, J. A. and King, J. C. "Applications of Symbolic Execution to Program Testing". Computer, 11(4), Apr. 1978, pp 51-60.

Davi83 Davis, M. and Weyiker, E. "A formal notation of program-based test data adequacy." Inf. Control, 56, 1/2 (Jan./Feb. 1983), 52-71.

Dean82 Dean, M. "How a computer should talk to people." IBM Syst. J. 21, 4 (1982), 424-453.

DeBa82 Bakker, J. and Zucker, J. "Processes and the denotational semantics of concurrency." Inf. Control 54, 1/2 (Jul./Aug. 1982), 70-120.

DeFr85 Francesco, N., Latelle, D. and Vaglini, G. "Integrated Environments for Concurrent Languages: An Interactive Debugger." Unpublished.

Demi78 DeMillo, R. A., Lipton, R. J. and Sayward, F. G. "Hints on Test Data Selection: Help for the Practicing Programmer". Computer, 11(4), Apr. 1978, pp 34-43.

DeMi79 Millo, R. A., Lipton, R. J. and Perlis, A. J. "Social Processes and Proofs of Theorems and Programs". CACM, 22(5), 1979, 271-280.

Dijk76 Dijkstra, E. W. A Discipline of Programming. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1976.

Dijk82 Dijkstra, E. "A tutorial on the split binary semaphore." in Theoretical foundations of programming methodology. Lecture notes of an international summer school directed by F. L. Bauer, E. W. Dijkstra and C. A. R. Hoare (Munich, West Germany), M. Broy and G. Schmidt (Eds.), D. Reidel Publ, Co., Hingham, MA., 1982, 555-564.

Dix83 Dix, T. "Exceptions and interrupts in CSP." Sci. Comput. Program. 3, 2 (Aug. 1983), 189-204.

Dunn84 Dunn, R. Software defect removal. McGraw-Hill Inc., New York, NY, 1984.

Fain83 Fainter, R. G., Guy, S. R., Maynard, J. F. and Lindquist, T. E. "Generic ENvironment for Interactive Experiments". Technical Report CS830009, Department of Computer Science, Virginia Polytechnic Institute and State University, Blacksburg, Va., 1982.

Fair71A Fairley, R. E. "Modern Software Design Techniques". Computer Software Engineering, Polytechnic Press, New York, pp 11-30.

Fair75 Fairley, R. E. "An Experimental Program-Testing Facility". IEEE Trans. Soft. Eng., SE-1, Dec. 1975, pp 350-357.

Fair76 Fairley, R. E. "Dynamic Testing of Simulation Software". Proceedings Summer Computer Simulation Conference, Washington, D. C., Jul. 1976.

Fair78 Fairley, R. E. "Tutorial: State Analysis and Dynamic Testing of Computer Software". Computer, 11(4), Apr. 1978, pp 14-23.

Flor68 Florentine, J. J. "Language Definition and Compiler Validation". Machine Intelligence, (vol 3), New York, American Elsevier, 1968.

Floy67A Floyd, R. W. "Assigning Meanings to Programs". Proceedings American Mathematics Society Symposia in Applied Mathematics, Vol 19, Providence, Rhode Island. American Mathematics Society, 1967, pp 19-31.

Floy67B Floyd, R. W. "The Verifying Compiler". Computer Science Research Review, Carnegie-Mellon University, Pittsburgh, Pa., 1967, pp 18-19.

Floy72 Floyd, R. W. "Toward Interactive Design of Correct Programs". Proceedings IFIP Congress71, Amsterdam: North Holland, 1972, pp 7-10.

Fosd76 Fosdick, L. D. and Osterweil, L. J. "Data Flow Analysis in Software Reliability". ACM Computing Surveys, (8)3, 1976, pp 305-330.

Fost80 Foster, K. A. "Error Sensitive Test Case Analysis". IEEE Trans. Soft. Eng., SE-6, May 1980, pp 258-264.

Gabo76 Gabow, H. N., Maheshwari, S. N. and Osterweil. L. J. "On Two Problems in the Generation of Program Test Path." IEEE Trans Soft Eng. SE-2, Sept 1976, 227-231.

Gall82 Gallimore, R. and Coleman, D. "Specification of distributed programs." in Program Specification, Proc. workshop (Aarhus, Denmark, Aug. 4-7, 1981), J. Staunstrup (Ed.), Springer-Verlag, New York, 1982, 181-214.

Gann79 Gannon, C. "Error Detection Using Path Testing and Static Analysis." Computer, Vol 12, Aug 1979, 26-31.

Geha84 Gehani, N. ADA: concurrent programming. Prentice-Hall, Inc., Englewood Cliffs, N.J., 1984.

Gerg81 Gergely, T. and Ury, L. "Time models for programming logics." in Mathematical logic in computer science, Proc. colloquium (Salgorarjan, Hungary, Sept. 10-15, 1978), B. Domolki and T. Gergely (Eds.), Elsevier North-Holland Inc., New York, 1981, 359-427.

Gert82 Gerth, R. "A sound and complete Hoare axiomatization of the ADA-rendezvous." in Automata, languages, and programming, Ninth Colloquium (Aarhus, Denmark, July 12-16, 1982), M.

Nielsen and E. M. Schmidt (Eds.), Springer-Verlag, New York, 1982, 252-264.

Gert84 Gerth, B. and De Roever, W. "A proof system for concurrent ADA programs." Sci. Comput. Program. 4, 2(Aug. 1984), 159-204.

Gols83 Golson, W. and Rounds, W. "Connections between two theories of concurrency: metric spaces and synchronization trees." Inf. Control 57, 2/3 (May/Jun. 1983), 102-124.

Good75 Goodenough, J. B. and Gerhart, S. L. "Toward a Theory of Test Data Selection". IEEE Trans. Soft. Eng. SE-1, Jun. 1975, pp 156-173.

Good79 Goodenough, J. B. "A Survey of Program Testing Issues." in Research Directions in Software Technology, P. Wegner, ed., Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1979, 316-340.

Gram83 Gramlich, Wayne C. "Debugging Methodology (Session summary)." Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging Mark Scott Johnson, ed. ACM, March, 1983.

Grie75 Gries, D and Levin, G. "Assignment and Procedure Call Proof Rules". ACM Transactions on Programming Languages and Systems, Vol. 2, No. 4, October, 1980, pp 564-579.

Hals77 Halstead, Maurice. Elements of Software Science. Elsevier North-Holland, Inc., New York, 1977.

Haml77 Hamlet, R. G. "Testing Programs with the Aid of a Compiler." IEEE Trans. Soft. Eng., SE-3, 4, Jul 1977, 279-290.

Haml81 Hamlet, R. "Reliability Theory of Program Testing." Acta Informatica, April, 1981.

Hans73 Hansen, Per Brinch. Operating System Principles. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1973.

Hant76 Hantler, S. L. and King, J. C. "An Introduction to Proving the Correctness of Programs." ACM Computing Surveys, Vol. 8, #3, Sept 1976, 331-353.

Hetz73 Hetzel, W. C., (ed.). Program Test Methods. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1973.

Hibb83 Hibbard, P., Hisgen, A., Rosenberg, J., Shaw, M. and Sherman, M. Studies in ADA style (2nd ed.). Springer-Verlag, New York, 1983.

Hill83 Hill, Charles R. "A Real-Time Microprocessor Debugging Technique." Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging Mark Scott Johnson, ed. ACM, March, 1983.

Hoar69 Hoare, C. A. R. "An Axiomatic Basis for Computer Programming". CACM, 12(10), 1969, pp576-583.

Hoar81 Hoare, C. A. R. "The Emperor's Old Clothes". CACM, 24(2), Feb., 1981.

Hoar82 Hoare, C. A. R. and McKeag, R. "Structure of an operating system." in Theoretical foundations of programming methodology. Lecture notes of an international summer school directed by F. L. Bauer, E. W. Dijkstra and C. A. R. Hoare (Munich, West Germany), M. Broy and G. Schmidt (Eds.), D. Reidel Publ, Co., Hingham, MA., 1982, 643-658.

Hold83 Holdsworth, D. "A system for analyzing ADA programs at run-time." Softw. Pract. Exper. 13, 5(May 1983), 407-421.

Howd75 Howden, W. E. "Methodology for the Generation of Program Test Data". IEEE Trans. Comput., C-24, May 1975, pp 554-560.

Howd76 Howden, W. E. "Reliability of the Path Analysis Testing Strategy". IEEE Trans. Soft. Eng., SE-2, Sept. 1976, pp 208-215.

Howd77 Howden, W. E. "Reliability of Symbolic Evaluation". Proceedings Computer Software and Applications Conference, Chicago, Ill., Nov. 1977, pp 442-447.

Howd77A Howden, W. E. "Symbolic Testing and the DISSECT Symbolic Evaluation System". IEEE Trans. Soft. Eng., SE-3, 1977, pp 266-278.

Howd78 Howden, W. E. "Algebraic Program Testing". Acta Informatica, Vol 10, 1978, pp 53-56.

Howd78A Howden, W. E. "Theoretical and Empirical Studies of Program Testing". IEEE Trans. Soft. Eng., SE-4, Jul. 1978, pp 293-297.

Howd78B Howden, W. E. and Eichorst, H. P. "Proving Properties of Programs from Program Traces". in Software Testing and Validation Techniques, E. Miller and W. E. Howden, eds., IEEE Computer Society, 1978, pp 46-56.

Howd80 Howden, W. E. "Functional Program Testing". IEEE Trans. Soft. Eng., SE-6, Mar. 1980, pp 162-169.

Howd81 Howden, W., (ed.). "Contemporary Software Development Environments," Software Engineering Notes 6, 4(Aug. 1981), 6-14.

Huan75 Huang, J. C. "An Approach to Program Testing," Computing Surveys, 7, 3(Sept. 1975), 113-127.

Huan78 Huang, J. C. "Program Instrumentation and Software Testing." Computer, Vol 11, Apr 1978, 25-33.

Hull84 Hull, M. and McKeag, R. "Communicating sequential processes for centralized and distributed operating system design." ACM Trans. Program. Lang. Syst. 6, 2 (Apr. 1984), 175-191.

Jone83 Jones, C. "Tentative steps toward a development method for interfering programs." ACM Trans. Program. Lang. Syst. 5, 4 (Oct. 1983), 596-619.

Kaha82 Kahan, W. and Coonen, J. "The near orthogonality of syntax, semantics, and diagnostics in numerical programming environments." in The relationship between numerical computation and programming languages. Proc. IFIP TC 2 working conference (Boulder, CO, Aug. 3-7, 1982), J. K. Reid (Ed.), Elsevier North-Holland, Inc., New York, 1982, 297-304.

Karp83 Karp, R. Proving operating systems correct. UMI Research Press, Ann Arbor, MI., 1983.

Karp84 Karp, R. A. "Proving failure-free properties of concurrent systems using temporal logic." ACM Trans. Program. Lang. Syst. 6, 2(April 1984), pp 239-253.

Kell76 Keller, R. M. "Formal Verification of Parallel Programs." CACM, Vol 19, 371-384, 1976.

Kess82 Kessels, J. "Arbitration without common modifiable variables." Acta Inf. 17, 2 (Jun. 1982), 135-131.

King76 King, J. C. "Symbolic Execution and Program Testing." CACM, Vol 19, 7, Jul 1976, 385-394.

Koza82 Kozma, L. and Laborczi, Z. "On implementation problems of shared abstract data types." in Specification and design of software systems, Proc conference on operating systems (Visegrad, Hungary, Jan. 1982), E. Knuth and E. J. Neuhold (Eds.), Springer-Verlag, New York, 1982, 146-152.

Lamp77 Lamport, L. "Proving the Correctness of Multiprocess Programs." IEEE Trans. Soft. Eng., Vol SE-3, 125-143, 1977.

Lamp83 Lamport, L. "Specifying concurrent program modules." ACM Trans Program. Lang. Syst. 5, 2 (Apr. 1983), 190-222.

Lee85 Lee, John A. N. Personal communication, 1985.

Leve83 Leveson, N. and Harvey, P. "Analyzing software safety." IEEE Trans. Softw. Eng. SE-9, 5 (Sep. 1983), 569-579.

Lewi82 Lewis, T. Software engineering: analysis and verification. Reston Publ. Co., Inc., Reston, VA, 1982.

Lisk82 Liskov, B. "On linguistic support for distributed programs." IEEE Trans. Softw. Eng. SE-8, 3 (May 1982), 203-210.

Luca78 Lucas, P. "On the Formalization of Programming Languages: Early History and Main Approaches," in Lecture Notes in Computer Science, D. Bjorner and C. B. Jones, eds., Springer-Verlag, Berlin, Heidelberg, New York, 1978.

Mann68 Manna, Z. "The Correctness Problem of Computer Programs". Computer Science Research Review, Carnegie-Mellon University, Pittsburgh, Pa., 1968, pp 34-36.

Mann69 Manna, Z. "The Correctness of Programs". J. Computer and System Science, 3(2), 1969, pp 119-127.

Mann81 "Verification of concurrent programs: the temporal framework." in The Correctness Problem in Computer Science, R. S. Boyer and J. S. Moore, eds., Academic Press, London, New York, Toronto, Sydney, San Francisco, 1981.

Mann81 Manna, Z. and Waldinger, R. "Problematic features of programming languages: a propositional calculus approach." Acta Informatica, 16, 4(Dec. 1981), 371-426.

Mann84 Manna, Z. and Wolper, P. "Synthesis of communicating processes from temporal logic specifications." ACM Trans. Program. Lang. Syst., 6, 1 (Jan. 1984), pp 68-93.

McCa62 McCarthy, J. "Toward a Mathematical Science of Computation". Proceedings, IFIP62 Congress, C. M. Popplewell (ed), Amsterdam, North Holland, 1962, pp21-23.

McCa76 McCabe, T. "A Complexity Measure." IEEE Trans. Soft. Enf., Vol SE-2, No. 4, Dec. 1976, pp 308-320.

McGe82 McGetterick, A. Program verification using ADA. Cambridge Univ. Press, New York, 1982.

Mil177 Miller, E. Program Testing Techniques. IEEE Computer Society, Long Beach, CA, 1977.

Mil178 Miller, E. and Howden, W. Tutorial: Software Testing and Validation Techniques. IEEE Computer Society Publications Office, 5855 Naples Plaza, Suite 301, Long Beach, CA., 90803, 1978.

Mil178A "Program Testing." Computer, Vol 11, No. 4, April 1978, pp 10-12.

Mino82 Minoura, T. "Deadlock avoidance revisited." JACM, 29, 4 (Oct. 1982), 1023-1048.

Morl83 Morland, D. "Human factors guidelines for terminal interface design." CACM, 26, 7 (Jul. 1983), 484-494.

Myer79 Myers, G. The Art of Software Testing Prentice-Hall, Englewood, NJ., 1979.

Naur66 Naur, P. "Proof of Algorithms by General Snapshots". BIT, 6(4), Kronprinsengade 14, K, 1114 Copenhagen K, Denmark, 1966, pp 310-316.

Ntaf79 Ntafos, S. and Hakimi, S. "On Path Cover Problems in Digraphs and Application to Program Testing." IEEE Trans. Soft. Eng., SE-5, Sep 1979, pp 520-529.

Osse83 Ossefort, M. "Correctness proofs of communicating processes: three illustrative examples from the literature." ACM Trans. Program. Lang. Syst. 5, 4 (Oct. 1983), 620-640.

Oste74 Osterweil, L. and Fosdick, L. "Data Flow Analysis As an Aid in Documentation, Assertion Generation, Validation and Error Detection." Department of COmputer Science, University of Colorado, Boulder, Co., Technical Report 55, Sep., 1974.

Owic76 Owicki, S. and Gries, D. "Verifying Properties of Parallel Programs: an Axiomatic Approach." CACM, 19, (5), May, 1976, 279-285.

Paga81 Pagan, F. G. Formal Specification of Programming Languages: A Panoramic Primer. Prentice-Hall, Inc., Englewood Cliffs, N. J., 1981.

Paig73 Paige, M. and Balkovich, E. "On Program Testing." Proc. 1973 IEEE Symposium of Computer Software Reliability. New York, NY, May 2, 1973.

Paig74 Paige, M. and Benson, J. "The Use of SOftware Probes in Testing Fortran Programs." Computer, Vol, 7, No. 7, July 1974, pp 40-47.

Pain67 Painter, J. A. "Semantic Correctness of a Compiler for an Algol-like Language (AIM-44)". Stanford University, Stanford, Ca., Mar., 1967.

Panz76 Panzl, D. "Test Procedures: A New Approach to Software Verification." Proc. 2nd International Conference on Software Engineering. Oct., 1976, pp 477-485.

Panz78 Panzl, D. "Automatic Software Test Drivers." Computer, Vol. 11, 1978.

Panz78A Panzl, D. "Automatic Revision of Formal Test Procedures." Proc. 3rd International Conference on Software Engineering. May, 1978.

Pete83 Peterson, G. "A new solution to Lamport's concurrent programming problem using small shared variables." ACM Trans. Program. Lang. Syst. 5, 1 (Jan. 1983), 56-65.

Ploe79 Ploedereder, E. "Pragmatic Techniques for Program Analysis and Verification." Proc. 4th International Conference on Software Engineering. Munich, FRG., 1979.

Pool73 Poole, P. "Debugging and Testing." in Advanced Course on Software Engineering, F. L. Bauer, (ed), Springer-Verlag, New York, 1973, pp 278-318.

Prat82 Pratt, T. "Formal analysis of computer programs." In Studies in computer science. S. V. Pollack (Ed.), The Mathematical Association of America, Washington, DC, 1982, 169-195.

Prat83 Pratt, T. Programming Languages: design and implementation (2nd ed.). Prentice-Hall, Inc., Englewood Cliffs, N.J., 1983.

Prie83 Priese, L. "Automata and concurrency." Theor. Comput. Sci. 25, 3 (Jul. 1983), 221-265.

Prob82 Probert, R. "Optimal insertion of software probes in well-delimited programs." IEEE Trans. Softw. Eng. SE-8, 1 (Jan. 1982), 34-42.

Pyle81 The ADA Programming Language. Prentice-Hall International, Inc., London, England, 1981.

Rama73 Ramamoorthy, C. V., Meeker, R. J. and Turner, J. "Design and Construction of an Automated Software Evaluation System". Proceedings 1973 Symposium on Computer Software Reliability, 1973, pp 28-37.

Rama75 Ramamoorthy, C. V. and Ho, S. B. F. "Testing Large Software with Automated Software Evaluation Systems". IEEE Trans. Soft. Eng., SE-1, #1, Mar. 1974, pp 46-58.

Rama75A Ramamoorthy, C. V., Kim, K. H. and Chen, W. T. "Optimal Placement of Software Monitors Aiding Systematic Testing". IEEE Trans. Soft. Eng., SE-1, #4, Dec. 1975, pp 403-411.

Rama76 Ramamoorthy, C. V., Ho, S. B. F. and Chen, W. T. "On the Automated Generation of Program Test Data".

IEEE Trans. Soft. Eng., SE-2, Dec. 1976, pp 293-300.

Reid82 Reid, L. Control and communication in programs, UMI Research Press, Ann Arbor, MI., 1982.

Samm82 Sammet, J., Waugh, D. and Reiter, R. "PDL/ADA - a design language based on ADA." in ACM '81, Conference proceedings (Los Angeles, CA., Nov. 9-11, 1981), New York, 1982, 217-229.

Schl84 Schlichting, R. and Schneider, F. "Using message passing for distributed programming: proof rules and disciplines." ACM Trans. Prog. Lang. Syst. 6, 3 (Jul. 1984), 402-431.

Shne82 Shneiderman, B. "The future of interactive systems and the emergence of direct manipulation." Behav. Inf. Technol. 1, 3 (Jul.-Sep. 1982), 237-256.

Shne82 Shneiderman, B. "System message design: guidelines and experimental results." in Directions in human/computer interaction. A. Badre and B. Shneiderman (Eds.), Ablex Publ. Corp., Norwood, NJ, 1982, 55-78.

Shne83 Shneiderman, B. "Human factors of interactive software." in Enduser systems and their human factors. Proc. scientific symposium conducted on the occasion of the 15th anniversary of the science center (Hekdelberg, West Germany, Mar. 18, 1983), A. Blaser and M. Zoeppritz (Eds.), Springer-Verlag, New York, 1983, 9-29.

Site74 Sites, S. "Proving that Computer Programs Terminate Cleanly." Report STAN-CS-74-418, Department of Computer SCience, Stanford University, Stanford, CA., May, 1974.

Stef84 Steffen, J. "Experience with a portable debugging tool." Softw. Pract. Exper. 14, 4 (Apr. 1984), 323-334.

Stev83 Stevens, G. "User-friendly computer systems? A critical examination of the concept." Behav. Inf. Technol. 2, 1 (Jan.-Mar. 1983), 3-16.

Ston80 "Stoneman": Requirements for Ada Programming Support Environments, Department of Defense, February, 1980.

Stoy77 Stoy, J. E. Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory. The MIT Press, Cambridge, Massachusetts, and London, England, 1977.

Stuc72 Stucki, L. "A Prototype Automatic Program Testing Tool." "pub "AFIPS Conf. Proc." Vol. 41, 1972/FJCC, pp 829-836.

Tai80 Tai, Kuo-Chung. "Program Testing Complexity and Test Criteria". IEEE Trans. Soft. Eng., SE-6, #6, Nov. 1980, pp 531-538.

Tayl83 Taylor, R. "An integrated verification and testing environment." Softw. Pract. Exper. 13, 8(Aug. 1983), 697-713.

Tayl83 Taylor, R. "A general-purpose algorithm for analyzing concurrent programs." CACM, 26, 5(May 1983), 361-376.

Teit79 Teitelbaum, R. T. The Cornell Program Synthesizer: A Microcomputer Implementation of PL/CS. Technical Report, Department of Computer Science, Cornell University, Ithaca, New York, 1979.

Turi50 Turing, A. "Checking a Large Routine." Report of a Conference of High-Speed Automatic Calculating Machines. Cambridge University Mathematical Laboratory, January, 1950, pp 67-69.

Webe83 Weber, Janice C. "Interactive Debugging of Concurrent Programs (Extended Abstract)." Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging Mark Scott Johnson, ed. ACM, March, 1983.

Weyu80 Weyuker, E. and Ostrand, T. "Theories of Program Testing and the Application of Revealing Subdomains." IEEE Trans. Soft. Eng., SE-6, May, 1980, pp 236-246.

Weyu82 Weyuker, E. "On testing nontestable programs." Comput. J. 25, 4 (Nov, 1982), 465-470.

Whit80 White, L. and Cohen, E. "A Domain Strategy for Computer Program Testing." IEEE Trans. Soft. Eng., SE-6, May, 1980, pp 247-257.

Whit85 Whiteside, J., Jones, S., Levy, P. and Wixon, D. "User Performance with Command, Menu, and Iconic

Interfaces." in Proc. CHI '85 Human Factors in Computer Systems. (San Francisco, April 14-18, 1985), ACM, New York, pp 185-191.

Wits83 Witschorik, C. "The real-time debugging monitor for the Bell System 1A processor." Softw. Pract. Exper. 13, 8 (Aug. 1983), 727-743.

Wood80 Woodward, M., Hennell, M. and Hedley, D. "Experience with Path Analysis and Testing of Programs." IEEE Trans. Soft. Eng., SE-6, May, 1980, pp 278-286.

Yeh77 Yeh, R., ed. Current Trends in Programming Methodology: Vol II, Program Validation. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1977.

Youn82 Young, S. Real time languages: design and development. Halsted Press, New York, 1982.

Zelk78 Zelkowitz, M. "Perspectives on Software Engineering." ACM Computing Surveys, Vol. 10, June 1978, pp 197-216.

APPENDIX A. ADATAD SOURCE CODE

```
Task AdaTAD Is

-- ------------------------------------------------------
--
--  ADATAD
--  Robert G. Fainter
--  Dept. of Computer Science
--  Virginia Polytechnic Institute and
--    State University
--  Blacksburg, Va  24061
--  (C) COPYRIGHT, AUGUST 1984, JUNE 1985.
--  ALL RIGHTS RESERVED
--
--  This is the main task of the ADATAD task debugger.
--
--  AdaTAD is a debugger specific to the task environment
--  of Ada.  Unique features include
--     1) Real time monitoring of task execution.
--     2) User-specification of temporal interaction of
--        task.
--     3) Monitor and control of task communication.
--     4) Maintenance of lists of synchronized tasks.
--     5) Maintenance of lists of waiting tasks.
--     6) Implementation of temporal logic.
--     7) Setting of temporal breakpoints.
--        Note to self: set a breakpoint in task A that
--        depends upon some state in Task B.
--     8) User specification of execution modes, including
--        a "throttle" to control the execution speed.
--  The source code listing constitutes a definition
--  of AdaTAD.  AdaTAD comprises several tasks.
--
--  TASK                        Purpose
--
--  LOGICAL_PROCESSOR_TASKS     An array of tasks each of
--                              which implements a logical
--                              processor on which a user's
--                              task can run.
--
--  AdaTAD_COORDINATOR          A task which monitors and su-
--                              pervises execution of a user's
--                              program.
--
--  MONITOR                     A task which scans the user
--                              task data base and transmits
```

```
--                                  all pertinent information to
--                                  the display terminal.
--
--   COMMAND_INTERPRETER            A task which translates user
--                                  commands into executable form.
--
--   TERMINAL_COMM                  A task which manages input/
--                                  output to and from the user's
--                                  video terminals.
--
--   VIDEO_TERMINAL_DRIVER          An array of tasks each of
--                                  which handles I/O for a single
--                                  terminal.
--
--   IODRIVERS                      A series of tasks which will
--                                  serve as drivers for whatever
--                                  IO devices are used with the
--                                  program.
-- ------------------------------------------------------------
--
  Entry ABORTER;

  Entry GRACEFUL_HALT;
--
End AdaTAD;
```

```
Task Body AdaTAD Is
--
-- Begin with declarations
--
-- ------------------------------------------------------
-- ------------------------------------------------------
--
-- User defined types.
--
-- The following are miscellaneous type declarations;
--
Type BYTES Is
  range 0 .. 255;

Type SUCCESS_CODE_TYPE Is
  (non_dbms_msg, dbms_msg, failure);

Type UNSIGNED Is Range
  0 .. Integer'Last

Type INTER_TASK_MESSAGE Is
  String (1..256);

Type TASK_NAME_TYPE Is
  String (1..31);

Type WINDOW_NAME_TYPE Is
  String (1..31);

Type ENTRY_NAME_TYPE Is
  String (1..31);

Type ADDRESS_OFFSET_TYPE Is
  Unsigned;

Type IO_DEV_NAME Is
  String (1..10);

-- The following two types are used to define a list of I/O
-- devices owned by a task.
-- The data structure is a singly linked list of device names
-- and numbers.
--
Type IO_DEV_LIST_POINTER Is Access io_device_list;

Type IO_DEVICE_LIST Is
  Record
    device_name: io_dev_name;
    device_number: integer;
    next_device: io_dev_list_pointer;
  End record;
```

```
-- The following several types define the information that
-- will be kept on each task.  In general, three different
-- kinds of information will be kept on each task: HEADER in-
-- formation is task-specific information that will not
-- change during  execution of the program; EXE_INFO_RECORD
-- information is dynamic information that changes from in-
-- stant to instant as the task executes; SYNCH_INFO_RECORD
-- information is synchronization information that changes
-- as the task requests or answers rendezvous.
--
Type EXE_INFO_RECORD Is
  Record
    done: boolean;
    breakpoint_state: (enabled, disabled);
    breakpoint_set: boolean;
    breakpoint_type: (unconditional, assertion);
    assertion_code: string;
    execution_mode: (wait, normal, timed, singlestep);
    release_character: character;
    execution_rate: real;
    release_time: real;
    execution_unit: (st_per_sec, sec_per_st);
        -- st_per_sec => statements per second;
        --    The value in execution_rate is the
        --    number of SOURCE statements that must
        --    be executed per seconds.
        -- sec_per_st => seconds per statement
        --    The value in execution_rate is the
        --    number of seconds that must elapse
        --    between the start of consecutive
        --    SOURCE statements.
    release_character: character;
    execution_enable: boolean;
  End record;

Type SYNCH_RECORD Is
  Record
    awaiting_os_service: boolean;
    awaiting_synchronization: boolean;
    is_synchronized: boolean;
    task_called: task_name_type;
    called_by: task_name_type;
    trace_of_called_tasks: task_name_pointer;
    trace_of_calling_tasks: task_name_pointer;
  End record;

Type HEADER Is
  Record
    task_name: task_name_type;
    default_io_number: integer;
```

```
    default_io_name: IO_dev_name;
    other_io_devices: io_dev_list;
  End record;

Type WINDOW_RECORD Is
  Record
    visible: boolean;
    name: window_name_type;
    x_anchor: integer;
    y_anchor: integer;
    x_extent: integer;
    y_extent: integer;
    zoomed: boolean;
  End record;

Type EXECUTION_STATUS_AREA_TYPE Is
  Record
    header: header_record;
    synchronization_information: synch_record;
    execution_information: exe_info_record;
    window_information: window_record;
  End record;

-- Then next two types are used to define the symbol table.
-- The compiler creates the symbol table as the program is
-- compiled.
--
-- The data structure of the symbol table is a general tree.
-- The name of the program is in the root node while its sub-
-- tree consists of all the identifiers in the main program.
-- Procedures and tasks may have their own sub-trees defining
-- their identifiers.  Any procedure or task knows all the
-- identifiers of its own and of any procedure or task above
-- it (not its siblings).
--
Type TREE_LIST_POINTER Is Access symbol_table_node;

Type SYMBOL_TABLE_INFO Is
  Record
    SYMBOL: String (1..31);
    TYPE: Unsigned;
    TASK: Task_name_type;
    OFFSET: Address_offset_type;
  End record;

Type SYMBOL_TABLE_NODE Is
  Record
    info: symbol_table_info;
    subtree: tree_list_pointer;
    sibling: tree_list_pointer;
  End record;
```

```
-- The following is a descriptor of arugments to be passed
-- among tasks when they rendezvous.  When one task makes an
-- entry call to another, the arguments are passed as a
-- doubly linked list of descriptors.
--
-- location: the offset of the variable in the defining
--           module.
-- total_length: the aggregate length of the argument.
-- element_length: the length of a single element of the
--                 argument.
-- argument_type: an integer representing the type of the
--                argument.  This allows run-time type check-
--                ing.  When the user defines a type, it will
--                be assigned a unique number.  This will be
--                stored in the appropriate symbol table by
--                the compiler.  Then, when an entry call is
--                made, the compiler will place the type of
--                the argument in the argument descriptor.
--                At runtime the type number in the descriptor
--                will be checked against the type number
--                also in the appropriate symbol table) of
--                the formal parameter.  A mis-match is
--                easily spotted.
-- mode: carries a run-time description telling whether or
--       not the argument can be read and/or modified.  Since
--       AdaTAD implements task parameter passing by refer-
--       ence, the called task must be explicitly told how to
--       handle the received argument.  If the mode is IN,
--       the called taskmay not modify the argument.  If the
--       mode is OUT, the called task may not use the value
--       in the argument.  If the mode is IN OUT, the called
--       task may use and modify the argument.  AdaTAD will
--       implement ment IN mode arguments by requiring the
--       called task to create local storage for the argument
--       and copying the argument into that storage.
-- value: this is a byte-array used to communicate values be-
--        tween rendezvoused tasks.  The compiler will gen-
--        erate thnecessary code to place the value of the
--        arguments into this field at the time of the entry
--        call.  This code will vary according to the type of
--        the data.  This field of the record cannot be fur-
--        ther defined until compile time.  That is why the
--        compiler must generate it.
--
Type ARGUMENT_DESCRIPTOR_TYPE Is
  Record
    location: unsigned;
    total_length: unsigned;
    element_length: unsigned;
    argument_type: unsigned;
    mode: (in, out, in_out);
```

```
    value: array (0..1023) of bytes;
  End record;

-- The following types define the doubly linked list
-- containing the parameter descriptors.
--
Type DESCRIPTOR_LIST_LINK Is Access DESCRIPTOR_LIST_NODE;

Type DESCRIPTOR_LIST_NODE Is
  Record
    forward,
    backward: descriptor_list_link;
    argument_descriptor: argument_descriptor_type;
  End record;

-- The following types define the head of the linked list of
-- descriptors.  Each descriptor list will have exactly one
-- node of this type as its first node.
--
Type DESCRIPTOR_LIST_HEAD_POINTER Is Access
     DESCRIPTOR_LIST_HEAD;

Type DESCRIPTOR_LIST_HEAD Is
  Record
    descriptor_list_pointer: descriptor_list_link;
    task_name: task_name_type;
    entry_name: entry_name_type;
    served_task: task_name_type;
  End record;
```

```
-- The following diagram is an example of a complete argument
-- descriptor list with two argument nodes.
--
--   HEAD        (points to an allocation of
--    ||          descriptor_list_head)
--    ||
--    \/
--   DESCRIPTOR_LIST_HEAD
--     descriptor_list_pointer--->--++
--     task name                    ||
--     entry name                   ||
--     served task name             ||
--                                  ||
--                                  \/
--    ++-----------<----------------++
--    ||
--    ||
--    \/
--   DESCRIPTOR_LIST_NODE
--                  forward--------->---------------------++
--           ++-----backward                               ||
--           ||     location(a1)                           ||
--           ##     length(a1)                             ||
--                  length(ele(a1))                        ||
--         /\       type(a1)                               ||
--         ||       mode(a1)                               ||
--         ||       value(a1)                              ||
--         ||                                              ||
--         ||                                              \/
--         ||      ++--------------------<----------------++
--         ||      ||
--         ||      ||
--         ||      \/
--         || DESCRIPTOR_LIST_NODE
--         ||       forward----------------->-------------++
--        ++-<-----backward                               ||
--                  location(a2)                          ##
--                  length(a2)
--                  length(ele(a2)
--                  type(a2)
--                  mode(a2)
--                  value(a2)
--
```

```
-- ------------------------------------------------------
-- ------------------------------------------------------
-- Global Variables
--
-- Controls execution.  Initialli false, can be made true by
-- 1) normal termination of user program.
-- 2) fatal error in user program.
-- 3) execution of AdaTAD STOP command.
  done:                  boolean := false;

-- Symbol table generated by the compiler for runtime use.
  symbol_table:          Array ( 1..sym_tab_limit )
                           Of symbol_table_type;

-- This array holds all pertinent information about tasks.
-- It is a shared variable which is used by the coordinator
-- and the data base monitor.
  task_information_table:
     Array ( 0 .. 255 ) Of Task_Information_node;
```

```
-- ------------------------------------------------------
-- ------------------------------------------------------
-- Procedure declarations
--
Procedure MESSAGE_HEAD
          ( inmessage: in out
                       inter_task_message;
            outmessage: out
                       inter_task_message )

Is Separate;

  -- Format of parameters:
  --  <string>0<string>0<string>0 ... 0<string>00
  --
  -- where <string> is any string of ASCII characters except
  -- character 0. 0 is ASCII character zero and is used as a
  -- delimiter.  The string is terminated with two adjacent
  -- ASCII zeroes.
  --
  -- This procedure removes the leftmost string of characters
  -- from the parameter "inmessage" and places them in the
  -- parameter "outmessage".  The value of both parameters
  -- is changed.  The length of "inmessage" is decreased
  -- by the length of its leftmose string + 1; the 0
  -- delimiter is also removed from it.
  --
  -- This procedure is used whenever an inter_task message
  -- has to be decoded.

Procedure BUILD_MESSAGE
          ( message_string: in out
                            inter_task_message;
            message:        in
                            inter_task_message )
Is Separate;
--
-- This procedure is used to append a message to the end of a
-- string of messages.
-- Format of message_string:
--   <string>0<string>0 ... 0<string>00
--
-- Format of message:
--   <string>00
--
-- Then characters of "string", prefaced by an ASCII 0, are
-- inserted in "message_string" immediately before the
-- closing delimiter (00).  "message" remains unchanged; the
-- length of "message_string" is increased by the length of
-- "message" + 1.
```

```
-- ------------------------------------------------------
-- ------------------------------------------------------
-- Task declarations.
--
-- Logical processor type
Task Type LOGICAL_PROCESSOR Is

  Entry RECEIVE_USER_COMMAND (command: in
                                       integer;
                              message: in out
                                       inter_task_message);

  -- The entry RECEIVE_USER_COMMAND is called by the AdaTAD
  -- coordinator whenever the user has entered a command from
  -- the keyboard that affects the task running on the
  -- logical processor.  The parameter "command" is an
  -- integer containing the code for the command; the
  -- parameter "message" contains other information if the
  -- command needs it.

  Entry RECEIVE_RENDEZVOUS_REQUEST
          (argument_list: in out
                          descriptor_list_head_pointer);

  -- The entry RECEIVE_RENDEZVOUS_REQUEST is called by the
  -- AdaTAD coordinator whenever another user task has
  -- requested a rendezvous with this task.  The parameter
  -- contains the address of the list of argument
  -- descriptors.

  Entry RECEIVE_RENDEZVOUS_COMPLETION;

  -- This entry is called by the AdaTAD coordinator whenever
  -- a servicing task has signalled the coordinator that a
  -- rendezvous that had been requested by the task on this
  -- processor has been completed.

  Entry LP_IDENTIFY
        ( task_name: in task_name_type );

  -- Activation of AdaTAD will cause the coordinator to call
  -- this entry for all tasks that are not dynamically
  -- allocated.  For tasks that are dynamically allocated,
  -- this entry is called by the coordinator when the
  -- allocated task is activated.

End LOGICAL_PROCESSOR;
```

```
--
-- ------------------------------------------------------
-- AdaTAD Coordinator
--
Task AdaTAD_COORDINATOR Is

  Entry RENDEZVOUS_REQUEST  (messag0: in out
                             descriptor_list_head_pointer);

  -- The above entry is called by the logical processor when
  -- the user's task desires rendezvous with another user
  -- task.

  Entry RENDEZVOUS_BEGIN (caller0: in
                                   task_name_type;
                          callee0: in
                                   task_name_type);

  -- The above entry is called by the logical processor when
  -- the user task accepts a rendezvous request and begins
  -- service.  It is the CALLED task that calls this entry.

  Entry RENDEZVOUS_COMPLETION (caller1: in
                                        task_name_type;
                               callee1: in
                                        task_name_type);

  -- The above entry is called by the logical processor when
  -- the user's task has completed the requested rendezvous.
  -- It is called by the servicing user task.

  Entry OS_REQUEST     (alist: in
                               descriptor_list_head_pointer);

  -- The above entry is called by the logical processor when
  -- the user's program needs an OS service.  Currently that
  -- is limited to input/output.

  Entry IO_DRIVER_INTERRUPT   (messag1: in
                                        inter_task_message);

  -- The above entry is called by the I/O driver task when an
  -- external device has interrupted.

  Entry COMMAND_IN            (command: in
                                        integer;
                               messag2: in out
                                        inter_task_message );

  -- The above entry is called by the command processor when
  -- it has successfully parsed a user command entered from
```

```
   -- the terminal.

   Entry DB_UPDATE                (lp_execution_state: in
                                   lp_execution_state_type;
                                   task_name: in
                                   task_name_type);

   -- The above entry is called by the logical processor when
   -- a change has occurred in the execution information data
   -- base.

End AdaTAD_COORDINATOR;
```

```
--
-- ------------------------------------------------------
-- User task monitor
--
Task MONITOR Is

  Entry HOLD;

  Entry RELEASE;

  Entry DONE;

End MONITOR;
```

```
--
-- ------------------------------------------------------
-- Command processor
--
Task COMMAND_INTERPRETER Is

  Entry PARSE (c_string: in
                         inter_task_message);

End COMMAND_INTERPRETER;
```

```
--
-- ------------------------------------------------------
-- Terminal Communicator
--
Task TERMINAL_COMM Is

  Entry FROM_TERMINAL (id: in
                           integer;
                       strl: in
                             inter_task_message);

  Entry FROM_MONITOR  (task_node: in
                       task_information_node);

  Entry FROM_COORD    (term_id: in
                                integer;
                       io_action: in
                                  integer;
                       arg_list: in out
                              descriptor_list_head_pointer);

  Entry FROM_CMD_INT  (message2: in
                                 inter_task_message;
                       drvidx2:  in
                                 integer);

  Entry DONE;

End TERMINAL_COMM;
```

```
--
-- ------------------------------------------------------
-- Terminal Drivers
--
Task Type VIDEO_TERMINAL_DRIVER Is

  Entry IDENTIFY ( id: in integer );

  Entry OUTPUT ( out_string: in
                             inter_task_message );

  Entry INPUT ( in_string: in
                           inter_task_message );

  Entry DONE;

End VIDEO_TERMINAL_DRIVER;
```

```
--
-- ------------------------------------------------------
-- Generic Device Drivers
--
Task IODRIVERS

     Entry DONE;

Is Separate;
```

```
-- ------------------------------------------------------
-- ------------------------------------------------------
-- Now, the task bodies, in the declaration order.
--
-- Logical processors
--
Task Body LOGICAL_PROCESSOR Is
-- A task object of this type executes the user's tasks.
-- There is one user task per logical processor.  The logical
-- processor is responsible for mediating rendezvous requests
-- and controlling the execution behavior of the user's task.
--
-- The following is a description of what the compiler must
-- do in order for user tasks to rendezvous under AdaTAD.
-- In this discussion, the calling task will be known as
-- TASK1 and the called task will be TASK2.  Case I describes
-- what is necsssary for a task to make a rendezvous request;
-- Case II describes what is necessary for a task to answer a
-- rendezvous request.
--
-- Case I.  The task running in this logical processor makes
-- an entry call to another task running on another logical
-- processor.  That is, TASK1 contains the statement
-- TASK2. e1 (x, y) where e1 is an entry in TASK2 and x and y
-- are arguments passed to TASK2. e1.
--
-- At compile time, this is what will happen.  When the
-- compiler encounters the entry call, it will build a doubly
-- linked list The first node of the list contains the name
-- of the called task and the name of the entry being called.
-- The remaining nodes in the list contain the information on
-- each argument.
-- TASK1 will be executing on a logical processor and will
-- have to make the entry call through that processor's
-- TRANSMITTER task to the AdaTAD coordinator.  (The AdaTAD
-- coordinator will use the task name in the first node of
-- the argument list to determine which logical processor is
-- running TASK2.) Therefore, the compiler will replace the
-- statement TASK2. e1 (x,y) with the statement
-- TRANSMITTER. SEND (2, head).
--
-- In summary, the compiler performs two transformations to
-- allow a user task to make an entry call: 1) the called
-- task's name and the called entry's name are placed in the
-- first node of a doubly linked list; the arguments are
-- placed in following nodes of the list, one argument per
-- node and 2) the entry call itself is converted into an
-- entry call to the logical processor's transmitter task.
--
-- Case II.  The compiler will perform the following actions
-- for the task which will accept the entry call.  For each
```

```
-- entry in the user's task, the compiler will create a CASE
-- statement aternative in the EXECUTOR task.  The code
-- generated in each alternative will "decompose" the
-- linked list of arguments and make the entry call to the
-- user's task with these arguments.  This code will, at run
-- time, check the type of the arguments against
-- the types of the formal parameters.
--
-- At execution time, the entry call to TRANSMITTER results
-- in an entry call to the AdaTAD coordinator with the
-- linked list of arguments being passed.
-- The AdaTAD coordinator gets the called task's name from
-- the first node of the list and looks it up in the log-
-- ical processor assignment table to determine which log-
-- ical processor is running the called task.  The AdaTAD
-- coordinator then examines the database to 1) determine if
-- the rendezvous can be made (if it can't, the request is
-- queued to the called task) and 2) to resolve any external
-- references to shared variables.
-- Next, the AdaTAD coordinator updates the database to re-
-- flect the rendezvous request.  Assuming that the
-- rendezvous is possible, the AdaTAD coordinator makes an
-- entry call to the logical processor coordinator on the
-- appropriate logical processor and passes the argument
-- list along.  The logical processor's coordinator will then
-- look up the name of the called entry and execute the code
-- of the CASE statement alternative which corresponds to the
-- called entry.  This code decomposes the list of arguments,
-- checking the types and preserving the mode.
-- Finally, the user's entry itself is called with the passed
-- arguments (if any).  When the rendezvous between the
-- EXECUTOR and the user task is complete, the EXECUTOR calls
-- the TRANSMITTER which in turn informs the AdaTAD coord-
-- inator.  The AdaTAD coordinator then updates the data base
-- and tells the calling logical processor that the rendezvous
-- is complete.  The calling task is then released from the
-- rendezvous and execution proceeds asynchronously.
--
-- The transmitter task sends messages to the AdaTAD coordi-
-- nator. This task is called whenever the users task
-- requests a rendezvous, needs a supervisor service (e.g.,
-- I/O) or encounters an exceptional condition.
--
-- This is the high level structure of the logical processor.
--
-- Task Body LOGICAL_PROCESSOR Is
--
--   Task Body EXECUTOR Is
--
--     Task Body USER Is
--       Begin
```

```
--          accept STARTER;
--            User code
--          EXECUTOR. EXECUTOR_DONE;
--          TRANSMITTER. XMTR_DONE;
--          EXECUTION_AREA_MONITOR. EXE_MONIT_DONE;
--        End USER;
--
--      Begin
--      End EXECUTOR;
--
--    Task Body TRANSMITTER Is
--      Begin
--      End TRANSMITTER;
--
--    Task Body EXECUTION_AREA_MONITOR Is
--      Begin
--      End EXECUTION_AREA_MONITOR;
--
--    Begin
--    End LOGICAL_PROCESSOR;
--
  Task EXECUTION_AREA_MONITOR Is
    Entry SET_BK_ST (in bkval: bk_states );
    Entry SET_EX_MD (in exval: ex_states );
    Entry SET_EX_RT (in rtval: real );
    Entry SET_EX_UN (in unval: un_states );
    Entry EXE_MONIT_DONE;
  End EXECUTION_AREA_MONITOR;

  Task TRANSMITTER Is
    Entry SEND_RENDEZVOUS_REQUEST (alist: in out
                          descriptor_list_head_pointer);
    Entry SEND_RENDEZVOUS_BEGINNING (caller: in
                                            task_name_type;
                                     callee: in
                                            task_name_type);
    Entry SEND_RENDEZVOUS_COMPLETION (served_task: out
                                       task_name_type );
    Entry SEND_DB_UPDATE;
    Entry SEND_SUPERVISOR_REQUEST (alist: in out
                             descriptor_list_head_pointer);
    Entry XMTR_DONE;
  End TRANSMITTER;

  Task EXECUTOR Is

    Entry STARTER;
    Entry RENDEZVOUS (alist: in out
                             descriptor_list_head_pointer);
    Entry EXECUTOR_DONE;
  End EXECUTOR;
```

```
Task Body EXECUTOR Is
--
-- Declaration part
--
  done: boolean;
--
--
-- This task executes the user's task.  It therefore
-- exercisesupervisory control over the user's task.
-- When another desires rendezvous with this task (if
-- appropriate), the AdaTAD coordinator makes rendezvous
-- with the coordinator of this logical processor.
--
-- The compiler makes the following changes to the user's
-- task:
--
--  1).  The entry STARTER is added as the first executable
--       statement of the user's task.  This is so that
--       execution of the user's task will, in effect, not
--       begin until this entry is called.  This entry will
--       be called based upon the value of the execution
--       state variable.
--  2).  For each entry call made by this task, a linked
--       list of arguments will be constructed with the
--       task name entry name and served task name in the
--       first node.
--  3).  Each entry call made by this task will be convert-
--       ed into an entry call to the transmitter: to wit,
--       TRANSMITTER. SEND_RENDEZVOUS_REQUEST (a_list)
--  4).  For each entry into this task, a case statement
--       alternative will be inserted in EXECUTOR.  The
--       code in the alternative will decompose an incoming
--       argument list and use the addresses, lengths and
--       modes found therein to call the entry in the
--       user's taskThe code can also perform run-time
--       type checking of arguments and parameters.
--  5).  The machine code which implements each source
--       statement in the user's task will have a short
--       prologue included.  That prolog will consist of
--       the following things:
--         a) Code to store the number of the source
--            statement in the execution data area.
--         b) Code to determine if breakpoint checking is
--            currently enabled and, if so, whether this
--            statement has a breakpoint set for it.  If it
--            does, the code must determine what kind of
--            breakpointit is (hard or assertion) and, if
--            assertion, call the assertion executor with
--            the appropriate assertion code.
--         c) A small data area holding information on
```

```
--              whether this statement is a breakpoint and
--              what kind. If this statement is an assertion
--              breakpoint, a pointer to the assertion code
--              is also here.
--           d) Code to examine the execution enable variable.
--              This section of code will continually examine
--              the execution enable variable until execution
--              is enabled.
--
--   6).  The user task calls the executor entry
--        EXECUTOR_DONE when the user task terminates
--        normally.
--
  Task USER Is
    Entry STARTER;
    -- Remainder of user's declaration follows.
    .
    .
    .
    Entry ...
  End User;
  Task Body USER Is
    -- X and Y are assumed to be user declarations.
    x: integer;
    y: real;
    -- The following two declarations are added by the
    -- compiler
    alist: descriptor_list_head_pointer;
    arg_desc: descriptor_list_link;
    --
    Begin
      Accept STARTER;
      --
      -- The following is an example of a possible entry
      -- call to another user task TASKN.  Assume that
      -- TASKN haan entry called e1.  Assume that x is
      -- an integer, y is a double precision real and that
      -- TASKN. e1 expects an integer and a real with
      -- IN and IN OUT mode, respectively.
      TASKN. e1 (x, y);
      -- ************************************************
      -- ************************************************
      -- The above entry call will be REPLACED by the
      -- following code.
      --
      -- First, the new argument list will be built.
      --
      -- This sets up the argument list head.
      new alist (null,
                 "TASKN",
                 "e1",
```

```
            "USER");

-- This sets up the first node of the argument list.
new arg_desc (null,
              null,
              loc(x),    -- offset of argument "x"
              4,         -- 4-byte integer
              4,
              1,         -- possible code for integer
              IN,        -- depends on declaration of
                         -- TASKN. e1; this is example.
              x          -- value
               );

-- This sets up the second node of the argument list.
new arg_desc. forward (null,
                       null,
                       loc(y),
                       8,       -- double prec real
                       8,
                       2,       -- code for real
                       IN OUT,  -- see above comment
                       y        -- value
                        );

-- This sets the backward link of the second node to
-- point to the first node.
arg_desc. forward. backward := arg_desc;
--
-- This sets the pointer in the head node to point to
-- the first argument node.
alist. descriptor_list_pointer := arg_desc;
--
-- This makes the entry call to the transmitter.
TRANSMITTER. SEND_RENDEZVOUS_REQUEST (alist);
-- ************************************************
-- ************************************************
--
--
-- The following is an example of an entry into task
-- USER
-- More code applies to this in after the EXECUTOR's
-- entry RENDEZVOUS.
--
-- The following is done before each accept to set
-- the data base correctly.
EXECUTION_AREA_MONITOR. SET_EX_MD (wait);
Accept e2 ( p: in integer ) Do;
  -- "caller" is declared in EXECUTOR and is there-
  -- fore visible here.  Since it is set just after
  -- the EXECUTOR's "RENDEZVOUS" entry, there could
```

```
  -- be no conflict in using it here.
  --
  TRANSMITTER. SEND_RENDEZVOUS_BEGINNING
    (caller);
  --
  -- Here is the code for the rendezvous.
  --
End e2;
TRANSMITTER. SEND_RENDEZVOUS_COMPLETION
  (caller);
--
-- The following is an example of how a terminal
-- output request is handled.
put (x);
--
-- The compiler will replace the above with the
-- following:
-- First, the new argument list will be built.
--
-- This sets up the argument list head.
new alist (null,
           "PUT", -- This will be recognized by the
                  -- AdaTAD coordinator as an output
                  -- request.  If this were an input
                  -- request, this would be GET.
           "",
           "USER");

-- This sets up the first node of the argument list.
new arg_desc (null,
              null,
              loc(x),    -- offset of argument "x"
              4,         -- 4-byte integer
              4,
              1,         -- possible code for integer
              OUT,
              x          -- value
               );
-- This sets the backward link.
arg_desc. forward. backward := arg_desc;
--
-- This sets the pointer in the head node to point to
-- the first argument node.
alist. descriptor_list_pointer := arg_desc;
TRANSMITTER. SEND_OS_REQUEST (alist);
--
-- The remainder of the code here is added by the
-- compiler to effect normal termination.  When the
-- user task terminates, the logical processor which
-- is running it must also terminate.  The code here
-- makes entry calls which cause that termination.
```

```
      --
      EXECUTOR. EXECUTOR_DONE;
      TRANSMITTER. XMTR_DONE;
      EXECUTION_AREA_MONITOR. EXE_MONIT_DONE;
    End;
  Begin
    Accept STARTER;
    USER. STARTER;
    done := false;
    Loop
      Select;
        Accept RENDEZVOUS (alist: in out
                           descriptor_list_head_pointer);
        --
        --
        -- started.
        caller := alist. served_task;
        callee := alist. server;
        e_name := alist. entry_name;
        Case e_name Is
          -- The compiler will insert here a
          -- case_statement_alternative for each entry in
          -- the user's task.  The choice for that
          -- alternative will be the corresponding entry
          -- name.  The code in each alternative will pass
          -- the address, length and mode of each argument
          -- to the called entry.
          When e1 =>
            -- The code in here will decompose the argument
            -- list and produce the dummy arguments p1
            -- through pn. These will be in local storage.
            -- The compiler will generate the necessary
            -- machine-dependent code for storage and
            -- offsets into the area.  The contents of the
            -- locations of arguments having IN mode will be
            -- copied to the storage here created.
            --
            USER. E1 (p1, p2, ... , pn);
            --
            -- After the rendezvous is complete, the value
            -- of those arguments with OUT mode will be
            -- copied into the addresses specified in the
            -- argument list.  Then, a call will be made to
            -- the transmitter signalling rendezvous completion.
            --
          When e2 => ...
            .
            .
            .
        End case;
      Or
```

```
        -- When the user task calls this entry, the execu-
        -- tor task will terminate.  The user task will
        -- call this only when it is about done.
        accept EXECUTOR_DONE;
        done := true;
      End Select;
    Exit When done;
    End loop;
  End EXECUTOR;

Task Body TRANSMITTER Is
  -- The following variable will have the name of the task
  -- running on the logical processor.  This will be set by
  -- the linker.
  --
  callee: task_name_type;
  done: boolean;
  Begin
    done := false;
    Loop
      Select
        -- This entry is called by the user's task when it
        -- needs to make an entry call to another task.
        -- The compiler converts all user task entry calls
        -- to an entry call to this entry.
        --
        -- The needed action is to update the execution
        -- mode to "wait" during the rendezvous and then to
        -- notify the coordinator of the rendezvous request.
        --
        Accept SEND_RENDEZVOUS_REQUEST (alist: in out
                          descriptor_list_head_pointer) Do;
          EXECUTION_AREA_MONITOR. EXECUTION_MODE (wait);
        End SEND_RENDEZVOUS_REQUEST;
        AdaTAD_COORDINATOR. RENDEZVOUS_REQUEST (alist);
      Or
        Accept SEND_RENDEZVOUS_BEGINNING (served_task: out
                                          task_name_type);
        AdaTAD_COORDINATOR. RENDEZVOUS_BEGIN (served_task,
                                              callee);
        EXECUTION_AREA_MONITOR. SET_EX_MD (running);
      Or
        -- The user's task calls this entry when it has
        -- completed a rendezvous.
        Accept SEND_RENDEZVOUS_COMPLETION (served_task: out
                                          task_name_type );
        AdaTAD_COORDINATOR. RENDEZVOUS_COMPLETION
          (served_task);
      Or
        Accept SEND_OS_REQUEST (alist: in out
                              descriptor_list_head_pointer)
```

```
                                    Do;
            AdaTAD_COORDINATOR. OS_REQUEST (alist);
        Or
          -- The EXECUTION_AREA_MONITOR calls this entry when
          -- it needs to send newly updated information to
          -- the AdaTAD coordinator.
          Accept SEND_DB_UPDATE;
          AdaTAD_COORDINATOR. DB_UPDATE (execution_area);
        Or
          -- The user's task calls this entry when it needs a
          -- service from the supervisor (such as I/O).
          Accept SEND_SUPERVISOR_REQUEST (alist: in out
                                descriptor_list_head_pointer);
          AdaTAD_COORDINATOR. OS_REQUEST (alist);
        Or
          Accept SEND_RENDEZVOUS_BEGINNING
            (caller: in
                     task_name_type;
             callee: in
                     task_name_type);
          AdaTAD_COORDINATOR. RENDEZVOUS_BEGIN
            (caller, callee);
        Or
          Accept XMTR_DONE;
          done := true;
        End select;
      Exit When done;
      End loop;
    End TRANSMITTER;

  Task Body EXECUTION_AREA_MONITOR Is
    execution_status_area:
      Record
        breakpoint_state: (enabled, disabled);
        execution_mode: (wait, normal, timed, singlestep);
        execution_rate: real;
        release_time: real;
        execution_unit: (st_per_sec, sec_per_st);
           -- st_per_sec => statements per second;
           --     The value in execution_rate is the
           --     number of SOURCE statements that must
           --     be executed per seconds.
           -- sec_per_st => seconds per statement
           --     The value in execution_rate is the
           --     number of seconds that must elapse
           --     between the start of consecutive
           --     SOURCE statements.
        statement_number: integer;
        execution_enable: boolean;
      End record;
      done: boolean;
```

```
Begin
  -- The following five statements set the initial
  -- conditions.
  execution_area. breakpoint_state := disabled;
  execution_area. execution_mode := wait;
  execution_area. execution_rate := 0.0;
  execution_area. execution_unit := st_per_sec;
  execution_area. execution_enable := true;
  done := false;
  Loop
    Select
      -- This entry is called by AdaTAD coordinator when-
      -- ever the release character is entered.
      Accept SING_STEP_REL Do;
        execution_area. execution_enable := true;
      End SING_STEP_REL;
    Or
      -- This entry is called when a set command changes
      -- the breakpoint checking state.
      Accept SET_BK_ST (in bkval: bk_states ) Do;
        execution_area, breakpoint_state := bkval;
      End SET_BK_ST;
    Or
      -- This entry is called when the execution mode
      -- must be changed (by a set command or a
      -- rendezvous request.)
      Accept SET_EX_MD (in exval: ex_states ) Do;
        execution_area. execution_mode := exval;
      End SET_EX_MD;
    Or
      -- This entry is called when a set command changes
      -- the rate of statement execution.
      Accept SET_EX_RT (in rtval: real ) Do;
        execution_area. execution_rate := rtval;
      End SET_EX_RT;
    Or
      -- This entry is called when the execution rate
      -- units change from statements per second to
      -- seconds per statement or vice versa.
      Accept SET_EX_UN (in unval: un_states ) Do;
        execution_area. execution_unit := unval;
      End SET_EX_UN;
    Or
      -- This entry is called by code that the compiler
      -- generates for each user statement to see if that
      -- statement is to be executed.
      Accept EXAMINE_EXE (OUT result: boolean);
        result := execution_area. execution_enable;
      End EXAMINE_EXE;
    Or
      -- When this entry is called by the user task, it
```

```
      -- marks the task as having completed by setting
      -- the "done" variable to true.  Also, the logical
      -- processor control variable, "done", is set to
      -- true.
      Accept EXE_MONIT_DONE;
      execution_area. done := true;
      done := true;
    Else
      -- This code is executed continually unless inter-
      -- rupted by one of the entry calls above.  It
      -- first checks for timed execution.  If that is
      -- the case, it checks for time to execute a
      -- statement.  If that's the case, the
      -- execution_enable is set to true and the time
      -- for the following statement to execute is com-
      -- puted.  If it is not time for the next statement
      -- to execute, this code insures against its execu-
      -- tion by setting execution_enable to false.  If
      -- the exeution cmode is single step, this code
      -- sets execution_enable to false (it takes a
      -- call to SING_STEP_REL, above, to set
      -- execution_enable to true).  If the execution
      -- mode is NORMAL, then execution_enable is set to
      -- true; otherwise, it is set to false (WAIT mode).
      If execution_area. execution_mode = TIMED Then
        If now = release_time Then
          execution_area. execution_enable := true;
          If execution_area. execution_unit = st_per_sec
          Then
            time_increment := 1 /
                        execution_area. execution_rate;
          Else
            time_increment :=
                        execution_area. execution_rate;
          End If;
          release_time := now + time_increment;
        Else
          execution_area. execution_enable := false;
        End If;
      Elseif execution_area. execution_mode = SINGLESTEP
      Then
        If execution_area. execution_enable Then
          execution_area. execution_enable := false;
        End If;
      Elseif execution_area. execution_mode = NORMAL
      Then
        execution_area. execution_enable := true;
      Else
        execution_area. execution_enable := false;
      End if;
    End select;
```

```
    Exit When DONE;
    End Loop;
  End EXECUTION_AREA_MONITOR;

Begin
  --
  -- This entry is called by the coordinator when AdaTAD
  -- begins running (for non-allocated tasks) and when an
  -- allocated task is activated (for allocated tasks).  In
  -- this way, the logical processor knows the name of the
  -- task running on it.
  --
  Accept LP_IDENTIFY (task_name: task_name_type);

  Loop
    Select

      -- This accept will receive commands and messages
      -- from the AdaTAD coordinator.  Anything except a
      -- rendezvous request will enter the logical
      -- processor through this entry.
      Accept RECEIVE_USER_COMMAND
        (command: in
                  integer;
         message: in out
                  inter_task_message);

      Case command Is

        When  1 =>  -- The user has issued a "start"
                    -- message for this processor.
                    EXECUTOR. STARTER;

        When 11 =>  -- The user has issued the "wait" com-
                    -- mand for this processor.  The exe-
                    -- cute enable variable will be so set.
                    EXECUTION_AREA_MONITOR.
                      SET_EX_MD (WAIT);

        When 12 =>  -- Resume normal execution.  The exe-
                    -- cute enable variable will be so set.
                    EXECUTION_AREA_MONITOR.
                         SET_EX_MD (NORMAL);

        When 13 =>  -- Set "breakpoint enable" variable.
                    EXECUTION_AREA_MONITOR.
                         SET_BK_ST (ENABLE);

        When 14 =>  -- Set single step mode.
                    EXECUTION_AREA_MONITOR.
                      SET_EX_MD (SINGLESTEP);
```

```
            When 15 =>  -- Set timed execution mode.  The mes-
                        -- sage contains the rate of execution.
                        message_head (message, rate_c);
                        rate_real := cvt_ch_real (rate_c);
                        EXECUTION_AREA_MONITOR.
                          SET_EX_RT (rate_real);

            When 16 =>  -- The message contains a statement
                        -- label which will be a breakpoint.
                        -- Parse the label, look it up in the
                        -- local symbol table and enter the ad-
                        -- dress in the list of active break-
                        -- points.

            When 17 =>  -- Message contains a label and an
                        -- expression.  The label will be
                        -- parsed entered in the list of active
                        -- breakpoints and marked as an asser-
                        -- tion break. The expression will be
                        -- compiled (lexical, syntactic and se-
                        -- mantic analysis, code generation)
                        -- and its location entered into the
                        -- list of active breakpoints.  When-
                        -- ever this breakpoint is encountered,
                        -- this code will be executed.

            When 18 =>  -- Definition of eternity.

            When 19 =>  -- The logical processor has just re-
                        -- eived the release character (in sin-
                        -- gle step ode) so an entry call is
                        -- now to be made to the execution area
                        -- monitor to release the task for
                        -- execution of the next statement.
                        EXECUTION_AREA_MONITOR. SING_STEP_REL;

          End case;

        Or
          -- This entry is where rendezvous requests enter.
          Accept RECEIVE_RENDEZVOUS_REQUEST
            (argument_list: in out
                            descriptor_list_head_pointer);
          EXECUTOR. RENDEZVOUS (argument_list);

        Or
          -- This entry will be called by the AdaTAD coordi-
          -- nator whenever a task servicing this one has com-
          -- pleted the rendezvous.
          Accept RECEIVE_RENDEZVOUS_COMPLETION;
```

```
      EXECUTION_AREA_MONITOR. SET_EX_MODE (running);

    Or
      terminate;

    End select;
    Exit When done;
  End loop;
End LOGICAL_PROCESSOR;
```

```
-- ------------------------------------------------------
-- ------------------------------------------------------
--
-- AdaTAD Coordinator
--
Task Body ADATAD_COORDINATOR Is
--
-- The following are local objects used in data base update
-- operations.
--
  temp_synch_info: synch_record;
  temp_exe_info: exe_info_record;
  temp_window: window_record;

--
-- This task is the heart of AdaTAD.  It mediates
-- communication among the user tasks running on the logical
-- processors.  Also, it maintains the database which is
-- continally examined by the "monitor" task. Notice that the
-- database is a shared variable.  Therefore, the coord-
-- inator must synchronize with the monitor in order to
-- update that database.  When the monitor and coordinator
-- are not synchronized, then the monitor continually exam-
-- ines the database and transmits its contents to the term-
-- inal communicator for display.  When the monitor and
-- coordinator are synchronized, then the monitor cannot
-- access the database but the coordinator can update it.
-- Update will occur when one task requests a rendezvous with
-- another user task, when a user task requests a supervisor
-- service or when a user task has changed its execution
-- state.
--
-- The following two procedures allow update of the database.
-- Since there are two distinct parts of the task database,
-- two procedures are used.  The first, MODIFY_EXECUTION_INFO,
-- updates the execution information that is supplied by the
-- logical processors. The second, MODIFY_SYNCH_INFO, updates
-- the information on synchronization. The coordinator itself
-- mediates this information.  The caller must supplu the
-- number of the affected processor and the new information.
-- Then, the appropriate procedure signals a hold on the
-- monitor process and, when that's been answered, makes the
-- database change.  Then the monitor is released to continue
-- sending information to the terminal.
--
Procedure Body MODIFY_EXECUTION_INFO

  (      lp_number: in integer;
   execution_state: in EXE_INFO_RECORD) Is

  Begin
```

```
    MONITOR. HOLD;

    task_information_table (lp_number).
      execution_information := execution_state;

    MONITOR. RELEASE;
  End MODIFY_EXECUTION_INFO;

Procedure Body MODIFY_SYNCH_INFO

  (        lp_number: in integer;
   synch_information: in synch_record) Is

  Begin
    MONITOR. HOLD;

    task_information_table (lp_number).
      synchronization_information := synch_information;

    MONITOR. RELEASE;
  End MODIFY_SYNCH_INFO;

Procedure Body MODIFY_WINDOW

  (        lp_number: in integer;
  window_information: in window_record) Is

  Begin
    MONITOR. HOLD;

    task_information_table (lp_number).
      window_information := window_information;

    MONITOR. RELEASE;
  End MODIFY_WINDOW;

Procedure Body MODIFY_HEAD

  (        lp_number: in integer;
    head_information: in header) Is

  Begin
    MONITOR. HOLD;

    task_information_table (lp_number).
      header := head_information;

    MONITOR. RELEASE;
  End MODIFY_HEAD;

Function LOOK_UP (TASK_NAME: TASK_NAME_TYPE)
```

```
                    return INTEGER is
  begin
    FOUND := false;
    INDEX := 0;
    while (not FOUND) and (INDEX <= 255) loop
      if TASK_INFORMATION_TABLE. HEADER. TASK_NAME =
         TASK_NAME then
         FOUND := true;
      else
        INDEX := INDEX + 1;
      endif;
    end loop;
    return INDEX;
  end LOOK_UP;

Function GET_LP_WITH_RELCHAR (relchar: character)
                               return INTEGER is
  begin
    FOUND := false;
    INDEX := 0;
    while (not FOUND) and (INDEX <= 255) loop
      if TASK_INFORMATION_TABLE. EXE_INFO_RECORD.
         RELEASE_CHAR = RELCHAR Then
         FOUND := true;
      else
        INDEX := INDEX + 1;
      endif;
    end loop;
    return INDEX;
  end GET_LP_WITH_RELCHAR;

  Begin
    Loop;
      Select;
        Accept RENDEZVOUS_REQUEST
               (messag0: in out
                         descriptor_list_head_pointer) Do;
          -- A request for service has been made to the task
          -- named in called_task.  The calling task has
          -- already been placed in a wait state by its
          -- logical processor and here the call is made to
          -- the logical processor running the appropriate
          -- task.
          --
          -- Get the name and index of the called task.
          called_task := messag0. task_name;
          called_task_index := look_up (called_task);
          --
          -- Get the name and index of the calling task.
          calling_task := messag0. served_task;
          calling_task_index := look_up (calling_task);
```

```
      --
      -- Update the data base to show that the calling
      -- task is awaiting synchronization.
      temp_synch_info := (false,true,false,called_task,
                          '');
      MODIFY_SYNCH_INFO (calling_task_index,
                         temp_synch_info);
      --
      -- Make the entry call to the called task.
      LOGICAL_PROCESSOR (called_task_index).
        RECEIVE_RENDEZVOUS_REQUEST (messag0);
    End RENDEZVOUS_REQUEST;
  Or
    Accept RENDEZVOUS_BEGIN (caller0: in
                                      task_name_type;
                             callee0: in
                                      task_name_type);
    -- The rendezvous between caller and callee has now
    -- begun.  The database will be updated to reflect
    -- this.
    callee_task_index := look_up (callee0);
    caller_task_index := look_up (caller0);
    temp_synch_info   := (false, false, true, callee0,
                          '');
    MODIFY_SYNCH_INFO (callee_task_index,
                       temp_synch_info);
    temp_synch_info   := (false, false, true, '',
                          caller0);
    MODIFY_SYNCH_INFO (caller_task_index,
                       temp_sync_info);
  Or
    Accept RENDEZVOUS_COMPLETION (caller1: in
                                        task_name_type;
                                  callee1: in
                                        task_name_type);
    -- The task which has been serving the task named
    -- "served_task" has completed the service.  Here,
    -- look up "served_task" in the logical processor
    -- allocation table and send that processor a message
    -- that the service has been performed.  Also, the
    -- database will be updated to reflect the
    -- completion.
    callee_task_index := look_up (callee1);
    caller_task_index := look_up (caller1);
    temp_synch_info   := (false, false, false, '', '');
    MODIFY_SYNCH_INFO (caller_task_index,
                       temp_synch_info);
    MODIFY_SYNCH_INFO (callee_task_index,
                       temp_synch_info);
    LOGICAL_PROCESSOR (caller1).
      RECEIVE_RENDEZVOUS_COMPLETION;
```

```
Or
  Accept OS_REQUEST
         (alist: in
                 descriptor_list_head_pointer);
    -- A user's task has requested an operating system
    -- I/O service.
    -- "service": 1 => input; 2 => output
    If alist. task_name = "PUT" Then
      service := 2;
    Else;
      service := 1;
    End If;
    calling_task := alist. served_task;
    temp_synch_info := (true,false,false,'','');
    lp_number := look_up (calling_task);
    MODIFY_SYNCH_INFO (lp_number,
                       temp_synch_info);
    TERMINAL_COMM. FROM_COORD
      (lp_number,
       service,
       alist);

    temp_synch_info := (false, false, false, '', '');
    MODIFY_SYNCH_INFO (lp_number,
                       temp_synch_info);
Or
  Accept IO_DRIVER_INTERRUPT
         (messag2: in
                   inter_task_message);
    -- Some peripheral device wants attention.
    -- Provide it.
Or
  Accept COMMAND_IN
         (command: in
                   integer;
          messag3: in out
                   inter_task_message);
  -- The command interpreter has determined that a
  -- command has to be executed. Here it comes.  Do it.
  Case command Is

    When =>  1  -- EXECUTE
      For lp_index In (0..number_user_tasks-1) Loop
        LOGICAL_PROCESSOR (lp_index). LP_IDENTIFY
          (task_information_table (lp_index).
           task_name);
        LOGICAL_PROCESSOR (lp_index).
          RECEIVE_USER_COMMAND
            (1, messag3);
      End loop;
```

```
When =>  2  -- EXIT
    AdaTAD. ABORTER;

When =>  3  -- SET DEFAULT TASK
  message_head (messag3, lp_index_c);
  lp_index := cvt_to_int (lp_index_c);
  controlled_task := lp_index;

When =>  4  -- Define a window
  message_head (messag3, window_name);
  message_head (messag3, visibility);
  message_head (messag3, ax_c);
  message_head (messag3, ay_c);
  message_head (messag3, ex_c);
  message_head (messag3, ey_c);
  If ax_c /= '' Then
    ax := cvt_to_int (ax_c);
  End if;
  If ay_c /= '' Then
    ay := cvt_to_int (ay_c);
  End if;
  If ex_c /= '' Then
    ex := cvt_to_int (ex_c);
  End if;
  If ey_c /= '' Then
    ey := cvt_to_int (ey_c);
  End if;
  temp_window := (window_name, visibility, ax, ay,
                  ex, ey, false);
  modify_window (controlled_task, temp_window);

When =>  5  -- ZOOM IN
  temp_window :=
    task_information_table (controlled_task). all;
  temp_window. zoomed := true;
  modify_window (controlled_task, temp_window);

When =>  6  -- ZOOM OUT
  temp_window :=
    task_information_table (controlled_task). all;
  temp_window. zoomed := false;
  modify_window (controlled_task, temp_window);

When =>  7 -- Give a logical name to an I/O device.
  message_head (messag3, iodevname);
  -- The host operating system will perform the
  -- assignment of the iodevname to the physical
  -- device.

When =>  8 -- Assign a default I/O device to a task
  message_head (messag3, defdevice);
```

```
      temp_head :=
        task_information_table (controlled_task).
        header. all;
      temp_head. default_io_name := defdevice;
      MODIFY_HEADER (controlled_task, temp_head);

    When =>  9  -- SHOW <variable>
      message_head (messag3, vbl);
      LOGICAL_PROCESSOR (controlled_task).
        RECEIVE_USER_COMMAND (9, vbl);

    When => 10  -- SHOW ALIASES
      message_head (messag3, vbl);
      LOGICAL_PROCESSOR (controlled_task).
        RECEIVE_USER_COMMAND (10, vbl);

    When => 11  -- SET WAIT
      LOGICAL_PROCESSOR (controlled_task).
        RECEIVE_USER_COMMAND (11, messag3);

    When => 12  -- SET NORMAL
      LOGICAL_PROCESSOR (controlled_task).
        RECEIVE_USER_COMMAND (12, messag3);

    When => 13  -- SET BREAKPOINT
      LOGICAL_PROCESSOR (controlled_task).
        RECEIVE_USER_COMMAND(13, messag3);

    When => 14  -- SET SINGLE STEP MODE
      LOGICAL_PROCESSOR (controlled_task).
        RECEIVE_USER_COMMAND (14, messag3);

    When => 15  -- SET TIMED MODE
      LOGICAL_PROCESSOR (controlled_task).
        RECEIVE_USER_COMMAND (15, messag3);

    When => 16  -- SET BREAKPOINT AT
      LOGICAL_PROCESSOR (controlled_task).
        RECEIVE_USER_COMMAND (16, messag3);

    When => 17  -- ASSERT
      LOGICAL_PROCESSOR (controlled_task).
        RECEIVE_USER_COMMAND (17, messag3);

    When => 18  -- SET ETERNITY
      LOGICAL_PROCESSOR (controlled_task).
        RECEIVE_USER_COMMAND (18, messag3);

    When => 19  -- Code for a release character.
      message_head (messag3, relchar);
      lp := get_lp_with_relchar (relchar);
```

```
            LOGICAL_PROCESSOR (lp). RECEIVE_USER_COMMAND
              (19, relchar);

        End case;

      Or
        -- One of the logical processors has just updated its
        -- task execution database.  Information must now be
        -- changed in the AdaTAD database.  (A parameter must
        -- be accepted here telling which logical processor.
        -- The logical processor will pass its name to the
        -- coordinator tor and then the coordinator will look
        -- up the name of the task in the logical processor
        -- allocation table.
        Accept DB_UPDATE              (lp_execution_state: in
                                       lp_execution_state_type;
                                       task_name: in
                                       task_name_type);
        lp_number := look_up (task_name);
        modify_exe_info (lp_number, lp_execution_state);

      End Select;
    Exit When done;
    End loop;
  End ADATAD_COORDINATOR;
```

```
-- ------------------------------------------------------
-- ------------------------------------------------------
-- The User Task Monitor
--
Task Body MONITOR Is
  done: boolean;

  Begin
    done := false;
    Loop
      Select;
        --
        -- If the Coordinator has an entry call pending, then
        -- wait until it releases.
        Accept HOLD;
        Accept RELEASE;
      Or
        --
        -- Termination.
        Accept DONE;
        done := true;
      Else
        --
        -- If there is no pending entry call, send the cur-
        -- rent data base to the terminal communicator for
        -- display.
        TERMINAL_COMM. FROM_MONITOR
          ( Task_Information_Array );
      End Select;
    Exit When done;
    End Loop;
  End MONITOR;
```

```
-- ------------------------------------------------------
-- ------------------------------------------------------
-- The Command Processor
--
Task Body COMMAND_INTREPRETER Is

  success_code: success_code_type;
  command_code: integer;
  err_msg,
  param_string: inter_task_message;
  done: boolean;

  -- This procedure analyzes the user's commands.
  -- It performs lexical and syntactic analysis of user commands.
  -- Parameters:
  --   a:  The command string entered by the user.
  --   b:  success code - (success, failure).  This is set
  --       according to the successful analysis of the command.
  --   c:  An integer code representing the command.  If an error
  --       was encountered, this is a code for the error.
  --   d:  The parameters of the command, if any.  These are sent
  --       to the logical processor for further analysis.
  --
  Procedure ANALYZE_COMMAND ( a: in
                                 inter_task_message;
                              b: in out
                                 success_code_type;
                              c: in out
                                 integer;
                              d: in out
                                 inter_task_message )
  Is Separate;

  -- This procedure takes an error code from the command analyzer
  -- and finds its text.
  --
  -- Parameters:
  --   a: The code of the error message to be found.
  --   b: The message itself.
  --
  Procedure LOOK_UP_ERROR_MESSAGE ( a: in
                                       integer;
                                    b: in out
                                       inter_task_message )
  Is Separate;

  Begin
    done := false;

    Loop;
      Accept PARSE (c_string: in inter_task_message);
```

```
--
-- For synchronization.  Once a rendezvous is made,
-- the command be parsed while the next command is
-- being read.
--
--
-- A language analyzer will be used to parse the user's
-- command, producing a "success code", a "command_code"
-- and a "parm_string".
--
analyze_command (c_string, success_code, command_code,
                 parm_string);

--   Table of command strings and meanings
--
--  COMMAND        CODE  Meaning
--
-- EXECUTE          1    AdaTAD coordinator sends a signal
--                       to each logical processor which
--                       causes the logical processor coord-
--                       inator to make an entry call to the
--                       EXECUTOR's STARTER entry.
--                       Synonyms for this command are GO,
--                       RUN, EXECUTE and START.
-- EXIT             2    AdaTAD coordinator calls the
--                       "ABORTER" entry in the AdaTAD main
--                       program which causes sudden termi-
--                       nation of the entire debugger.
-- <lp ref>         3    Marks the entry in the LP table so
--                       that all following commands apply
--                       to that task.  Typically is used
--                       just before one or more SET com-
--                       mands.
--                       Also used just before making a win-
--                       dow definition.
-- WINDOW           4    Causes definition of window para-
--                       meters in the task data base.
-- ZOOM IN          5    Causes the window of the default
--                       task to be zoomed in.
-- ZOOM OUT         6    Causes the window of the default
--                       task to be zoomed out.
-- NAME             7    Causes an entry in the IO device
--                       table associating an o/s device
--                       with a logical name.  Similar to
--                       DEC VAX/VMS logical name assignment
-- ASSIGN           8    Causes an entry in the task infor-
--                       mation table associating a task
--                       with an I/O device.  That device is
--                       then that tasks default I/O device.
-- SHOW             9    Causes the value of the named vari-
--                       able to be displayed in the default
```

```
--                                  task's window.
-- SHOW ALIASES    10               Causes all aliases of the named
--                                  variable to be displayed.
-- SET WAIT        11               Causes the most recently referenced
--                                  logical processor (see 9 above) to
--                                  enter an indefinite wait state.
-- SET NORMAL      12               Causes the most recently referenced
--                                  logical processor to execute nor-
--                                  mally.
-- SET BREAKPOINT 13                Causes the most recently referenced
--                                  logical processor to enter a wait
--                                  state whenever it encounters a
--                                  breakpoint.
-- SET SINGLE
--  STEP RELEASE
--        <char>    14               Causes the most recently referenced
--                                  logical processor to execute source
--                                  statements one at a time; the des-
--                                  ignated character is a signal for
--                                  the next statement to be executed.
-- SET TIMED       15               Causes the most recently referenced
--                                  logical processor to execute at the
--                                  specified rate.
-- SET BREAK-
-- POINT AT
--  <label>        16               Designates <label>, from the symbol
--                                  tabel, as a breakpoint.
-- ASSERT          17               Establishes a condition to be
--                                  tested whenever a label is reached.
-- SET ETERNITY    18               Defines how the temporal logic pro-
--                                  cessor is to interpret the meaning
--                                  of temporal operators refering to
--                                  eternity.
-- release char    19               The command analyzer has recognized
--                                  a release character.  When the user
--                                  enters a single character, the
--                                  analyzer assumes that it may be a
--                                  single step mode release character
--                                  for a logical processor.  It will,
--                                  therefore, look up this character
--                                  in the task information table and,
--                                  if the analyzer finds this charac-
--                                  ter in the table, it reports a
--                                  successful analysis and a command
--                                  code of 19 to the coordinator.  The
--                                  coordinator will determine which
--                                  processor has been released and
--                                  then instructs that processor to
--                                  execute the next statement.
--
Case success_code Is
```

```
When => success
  Case command_code Is

    When =>  1  -- EXECUTE
      AdaTAD_coordinator. command_in
        (1, parm_string);

    When =>  2  -- EXIT
      AdaTAD_coordinator. command_in
        (2, parm_string);

    When =>  3  -- <lp reference>
      AdaTAD_coordinator. command_in
        (3, parm_string);

    When =>  4  -- WINDOW definition
      AdaTAD_coordinator. command_in
        (4, parm_string);

    When =>  5  -- ZOOM IN
      AdaTAD_coordinator. command_in
        (5, parm_string);

    When =>  6  -- ZOOM OUT
      AdaTAD_coordinator. command_in
        (6, parm_string);

    When =>  7  -- NAME
      message_head (parm_string, iodev);
      message_head (parm_string, l_name);
      n_logical_names := n_logical_names + 1;
      logical_name_table(n_logical_names).
        io_dev := iodev;
      logical_name_table(n_logical_names).
        logical_name := lname;

    When =>  8  -- ASSIGN
      AdaTAD_coordinator. command_in
        (8, parm_string);

    When =>  9  -- SHOW <variable>
      AdaTAD_coordinator. command_in
        (9, parm_string);

    When => 10  -- SHOW ALIASES
      AdaTAD_coordinator. command_in
        (10, parm_string);

    When => 11  -- SET WAIT
      AdaTAD_coordinator. command_in
```

```
            (11, parm_string);

          When => 12  -- SET NORMAL
            AdaTAD_coordinator. command_in
              (12, parm_string);

          When => 13  -- SET BREAKPOINT (checking)
            AdaTAD_coordinator. command_in
              (13, parm_string);

          When => 14  -- SET SINGLE STEP
            AdaTAD_coordinator. command_in
              (14, parm_string);

          When => 15  -- SET TIMED
            AdaTAD_coordinator. command_in
              (15, parm_string);

          When => 16  -- SET BREAKPOINT AT <label>
            AdaTAD_coordinator. command_in
              (16, parm_string);

          When => 17  -- ASSERT
            AdaTAD_coordinator. command_in
              (17, parm_string);

          When => 18  -- SET ETERNITY
            AdaTAD_coordinator. command_in
              (18, parm_string);

        End case;

      When => failure
        look_up_error_msg (command_code, err_msg);
        TERMINAL_COMM. from_cmd_int (err_msg);

    End case;
  Exit When done;
  End loop;
End COMMAND_INTERPRETER;
```

```
-- ------------------------------------------------------
-- ------------------------------------------------------
-- The Terminal Communicator
--
Task Body TERMINAL_COMM Is
-- This task serves as a clearinghouse for terminal I/O
-- requests to and from AdaTAD.
--
-- "done" controls normal termination.
  done: boolean;
--
  most_recent_command_terminal: integer;

  Begin
    done := false;

    Loop
      Select
        Accept FROM_TERMINAL (id: in
                                  integer;
                              strl: in
                                  inter_task_message);
        -- When an unsolicited line is typed on a terminal,
        -- the terminal driver calls this entry.  It is as-
        -- sumed that unsolicited input is a command, so the
        -- string is passed to the command interpreter.
        --
        most_recent_command_terminal := id;
        COMMAND_INTERPRETER. parse
          (strl);

      Or
        Accept FROM_COORD   (term_id: in
                                  integer;
                            io_action: in
                                  integer;
                             arg_list: in out
                             descriptor_list_head_pointer) Do;
        -- When the user task requests terminal I/O, it calls
        -- the coordinator to provide that service.  The co-
        -- ordinator in turn looks up the terminal associated
        -- with the task's window.  THe coordinator also de-
        -- termines whether the I/O request is for input or
        -- output and passes the appropriate value.  Finally,
        -- the task will have done any necessary data conver-
        -- sion from the base type to ASCII for display.
          If io_action = 1 Then
            -- This is for soliciting input from a terminal.
            -- "t_id" starts out not equal to any terminal
            -- index so the loop always starts.  Then, wait
            -- at the accept from_terminal.  If the input
```

```
        -- comes from the proper terminal, then the loop
        -- terminates and the value of the argument in
        -- the descriptor is set to the string that was
        -- typed on the terminal; this value will event-
        -- ually wend its way back to the task that re-
        -- quested I/O.
        t_id := -1;
        While t_id /= term_id Loop
          Accept FROM_TERMINAL (t_id: in integer;
                                str2: in
                                inter_task_message );
        End loop;
        -- For the sake of clarity, a step has been left
        -- out here.  At this point, the character string
        -- in "str2" will be converted to what ever data
        -- type the input value is to be.  This informa-
        -- tion is in the "type" field of the descriptor.
        arg_list. descriptor_list_pointer.
          argument_descriptor. value := str2;
      Else
        -- This branch is for output.  The value in the
        -- arg ument list is written on the designated
        -- terminal.  For the sake of clarity, a step has
        -- been left out.
        -- In actuality, this is where data conversion
        -- takes place.  That means that "value" will be
        -- converted to a character string here for dis-
        -- play on the terminal.  The conversion will
        -- take place using information provided by the
        -- "type" field of the descriptor.
        VIDEO_TERMINAL_DRIVER (term_id). OUTPUT
          (arg_list. descriptor_list_pointer.
           argument_descriptor. value);
      EndIf;
    End FROM_COORD;

  Or

    Accept FROM_MONITOR ( task_node: in
                          Array (0..255) of
                          task_information_node);
    --
    -- This entry accepts the status information from the
    -- database monitor, converts it to displayable form,
    -- then sends the appropriate information to the app-
    -- ropriate terminal.

    For i In (0..255) Loop
      If task_node(i). on_screen Then
      -- Do data conversion
      drvidx0 := task_node(i). default_id_number;
```

```
        VIDEO_TERMINAL_DRIVER (drvidx0). OUTPUT (message0);
      End Loop;
      End FROM_MONITOR;

    Or

      Accept FROM_CMD_INT ( message2: in
                                  inter_task_message ) Do;
      --
      -- This entry is called when the command interpreter
      -- has detected an error in the previously entered
      -- command.  This entry prints the error message;
      --
        VIDEO_TERMINAL_DRIVER
        (most_recent_command_terminal).
          OUTPUT (message2);
      End FROM_CMD_INT;

    Or
      Accept DONE Do;
      -- This entry is called by the coordinator when
      -- AdaTAD terminates.
        done := true;
      End DONE;

    End select;
  Exit When done;
  End loop
End TERMINAL_COMM;
```

```
-- ------------------------------------------------------
-- ------------------------------------------------------
-- The Terminal Drivers
--
Task Body VIDEO_TERMINAL_DRIVER Is
-- There is one of these tasks assigned to each video termi-
-- nal.  The task TERMINAL_WATCHER waits at a GET statement
-- for input from the terminal.  When it comes, it makes an
-- entry call to the main task to pass the string
-- along.  The main task then calls the terminal communicator
-- and sends the string along.  When AdaTAD wants to write to
-- a terminal, it makes an entry call to the entry OUTPUT,
-- which then writes the string on the terminal.

  out_string,
  in_string: inter_task_message;

  Task TERMINAL_WATCHER;

  Task Body TERMINAL_WATCHER Is
  -- This tasks sole purpose is to wait for input on a
  -- terminal. When it detects input, it calls the terminal
  -- driver entry with that string.

    c_string: inter_task_message;

    Begin
      Loop
        get (terminal, c_string);
        VIDEO_TERMINAL_DRIVER. INPUT (c_string);
      Exit When done;
      End loop;
    End TERMINAL_WATCHER;

  Begin

    -- The following statement is the first executed when the
    -- driver begins execution.  It accepts an entry call
    -- from the terminal communicator telling the driver who
    -- the terminal is.
    Accept identify (id: in integer);
    done := false;

    Loop

      Select;
        Accept OUTPUT (out_string: in
                                   inter_task_message ) Do;
        -- This accept is called by the terminal communicator
        -- when there is output to be done.
          put (terminal, out_string);
```

```
      End output;
    Or
      Accept INPUT (in_string: in
                                inter_task_message ) Do;
      -- This entry is called from the local task
      -- "terminal watcher" whenever a terminator character
      -- (usually the <return> key) has been pressed.  The
      -- entry adds the identification and calls the termi-
      -- nal communicator.
        TERMINAL_COMM. FROM_TERM (id, in_string);
      End input;
    Or
      Accept stopper;
      -- This entry is called by the terminal communicator
      -- when AdaTAD is terminating.
      done := true;
    End Select;

  Exit When done;
  End loop;

End VIDEO_TERMINAL_DRIVER;
```

```
-- ------------------------------------------------------
-- ------------------------------------------------------
Begin
  -- ------------------------------------------------------
  -- The main program will eventually initialize the AdaTAD
  -- environment.
  --
  -- The selective wait handles termination.  If the user
  -- enters the HALT command, the coordinator makes an en-
  -- try call to ABORTER, which causes all of AdaTAD to stop.
  -- If the user tasks terminate normally, the termination
  -- is propogated through AdaTAD gracefully.  After all
  -- user tasks have been stopped, the coordinator calls the
  -- GRACEFUL_HALT entry, ending AdaTAD.
  --
  Select
    Accept ABORTER;
      abort ADATAD;
  Or
    Accept GRACEFUL_HALT;
  End select;
End ADATAD;
```

APPENDIX B. PRODUCER-CONSUMER SOURCE CODE

```
With text_io;
User text_io, integer_io;
-- ------------------------------------------------------
--
-- Consumer/producer
--
-- ------------------------------------------------------
procedure CONSUMER_PRODUCER is

type TYPE_0 is integer;
type TYPE_1 is integer;
type ELEMENT is integer;
type POINTER is access BUF_NODE_TYPE;
type BUF_NODE_TYPE is
  record
    LINK: POINTER;
    DATUM: ELEMENT;
  end record;

N: constant := 3;

task BUF_CONTROL is

  entry INSERT ( X: in ELEMENT );
  entry EXTRACT ( Y: in out ELEMENT );

end BUF_CONTROL;

task PRODUCER;

task CONSUMER;
```

```
task body PRODUCER is
  Y: ELEMENT;
  X: TYPE_0;

function F (X: in TYPE_0) return ELEMENT is separate;

begin
  loop
    Y := F (X);
    BUF_CONTROL. INSERT (Y);
  end loop;
end PRODUCER;
```

```
task body CONSUMER is
  Y: ELEMENT;
  T: TYPE_1;

function G (X: in ELEMENT) return TYPE_1 is separate;

begin
  loop
    delay 5.0;
    BUF_CONTROL. EXTRACT (Y);
    T := G(Y);
  end loop;
end CONSUMER;
```

```
task body BUF_CONTROL is

CE, CF: integer;
BUFFER_FRONT, BUFFER_BACK: POINTER;

-- The procedures ADD_ELEMENT and DELETE_ELEMENT
-- add and delete elements from a FIFO queue.
-- They are not detailed here but could be
-- easily implemented.
--
procedure ADD_ELEMENT (BUFFER_FRONT,
                       BUFFER_BACK: in out POINTER;
                       X: ELEMENT) is separate;

procedure DELETE_ELEMENT (BUFFER_FRONT,
                          BUFFER_BACK: in out POINTER;
                          X: in out ELEMENT) is separate;
begin

  BUFFER_FRONT := null;
  BUFFER_BACK := null;

  CE := N;
  CF := 0;

  loop
    select
      when CE > 0
        accept INSERT (X: in out ELEMENT) Do
          CE := CE - 2;
          CF := CF + 1;
          ADD_ELEMENT (BUFFER_FRONT,
                       BUFFER_BACK,
                       X);
        end INSERT;

    or

      when CF > 0
        accept EXTRACT ( Y: in out ELEMENT ) Do
          CF := CF - 1;
          CE := CE + 1;
          DELETE_ELEMENT (BUFFER_FRONT,
                          BUFFER_BACK,
                          Y);
        end EXTRACT;

    else
      null;
    end select;
```

```
  end loop;
end BUF_CONTROL;

begin
  null;
end CONSUMER_PRODUCER;
```

The two page vita has been removed from the scanned document. Page 1 of 2

The two page vita has been removed from the scanned document. Page 2 of 2