DEBUGGING TASKED ADA PROGRAMS

by

R.G. Fainter
Virginia Tech,
and
T.E. Lindquist
Arizona State University

Abstract

The applications for which Ada was developed require distributed implementations of the language and extensive use of tasking facilities. Debugging and testing technology as it applies to parallel features of languages currently falls short of needs. Thus, the development of embedded systems using Ada poses special challenges to the software engineer. Techniques for distributing Ada programs, support for simulating distributed target machines, testing facilities for tasked programs, and debugging support applicable to simulated and to real targets all need to be addressed. This paper presents a technique for debugging Ada programs that use tasking and it describes a debugger, called AdaTAD, to support the technique. The debugging technique is presented together with the user interface to AdaTAD. The component of AdaTAD that monitors and controls communication among tasks has been designed in Ada and is presented through an example with a simple tasked program.

1. INTRODUCTION

Because of the distributed nature of the Space Station and its unmanned platforms, software that the Space Station uses must be highly distributed. This implies, therefore, that the task will be used extensively in Space Station software. Because of the difficulties associated with locating errors in tasked programs and because of the cost of programming errors in Space Station software, tools to aid in the production of correct programs must be developed. Such a tool is currently under development and is described in this paper.

One view of program testing [1] indicates that a program has been tested when every statement in the program has been executed at least once and every possible outcome of each program predicate has occurred at least once. Considerable literature addressing techniques for testing software reflects a view of testing that is consistent with this definition. Although this definition does not naturally extent to tasked programs, it is indicative of the view that testing occurs late in software development and is oriented toward validation.

In contrast, debuggers have traditionally had utility in earlier software development activities. Accordingly, debuggers are used as automated support for locating errors and determining what is needed to correct errors. Ideally, testing is used to identify the presence of errors and debuggers to support location and correction. When tasking facilities are included in a language, however, the software designer is left without good testing techniques, and debugging must enter into the process of identifying the existence of errors.

Helmbold [2] suggests that "Debuggers for parallel programs have to be more than passive information gatherers--they should automatically detect errors". When tasking errors directly depend on the semantics of the language, a debugger is able to actively aid in detecting errors. More commonly, errors are also dependent on the specific logic of task interaction and the use of the language.To take an active role in identifying this more complex type of errors, the debugger must include facilities to analyze the logic of the program. Helmbold distinguishes types of tasking errors as "Task Sequencing Errors" and "Deadness". AdaTAD provides task information that may be used to detect either type of tasking errors, although it does not actively detect errors.

AdaTAD is a debugger whose capabilities are specific to the problems of concurrent programs. The name AdaTAD is an acronym for Ada Task Debugger. Most debuggers allow the user to trace the execution of a program, but the program remains under control of the operating system. AdaTAD differs from other debuggers by exercising direct control over the execution of a

program's tasks. The user is able to specify which tasks run when, at what rate and for how long. Of course, to emulate more closely the environment in which a program is to execute, the user may defer these decisions to the runtime system, and simply monitor task synchronization and communication. AdaTAD combines typical debugging facilities with others specific to supporting the Ada constructs for rendezvous.

Space Station software may be configured in many different ways. One possible scenario might involve an Ada program with tasks running on an Earth-based computer, on one or more computers aboard the main station and on computers on one or more unmanned platforms. AdaTAD has the capability to allow the software engineer to debug such a program in at least two different ways. Firstly, the software engineer may construct, solely on ground based computers, an environment similar to that which exists on the Space Station for debugging purposes. Secondly, because AdaTAD itself may be distributed, the program may be run under AdaTAD in the actual Space Station environment. This allows the software engineer a great deal of flexibility in exercising the program under a variety of conditions.

A method for debugging tasked Ada programs and AdaTAD are presented jointly in this paper. Our approach to task debugging centers on removing task errors from three successive levels of consideration. Errors within tasks, which are principally independent of other tasks, are first addressed. Next, the communication and synchronization structure among tasks is addressed, and finally, any application specific concerns are addressed. AdaTAD, as it relates to these levels, is discussed in the following three sections together with a discussion of our approach to debugging. A subsequent section addresses the design of AdaTAD. Ada is used in the design to allow increased effectiveness on multiprocessor applications, and to show how the rendezvous constructs can be used to control the execution of tasked Ada programs. An Ada implementation of AdaTAD would require emitting special code from the compiler for synchronization with AdaTAD.

2. LOGIC ERRORS WITHIN A TASK

The first level of usage for the debugger is to address logic errors within each of a program's tasks. These errors are exclusive of intertask communication and synchronization. Removing them is synonomous to removing errors detected during unit testing of software. At this level, we assume that interactions with other tasks are correct and examine the activities of the task itself. Testing and debugging at this stage considers a piece of software in absence of all elements of its environment except any procedures or functions it calls. For example, a task may use information

obtained from other tasks to retrieve and update information in a database. Task logic to perform operations on the database is considered, at this level, exclusive of synchronization with other tasks.

AdaTAD facilities are used in conjunction with a testing strategy in which some form of code analysis may be performed. AdaTAD is designed to aid in executing test cases and in removing any errors subsequently found.

2.1 User's View of AdaTAD

AdaTAD provides many facilities which are common to source level debuggers in addition to those specific to tasks. After introducing the manner in which AdaTAD includes common functions, facilities specific to removing logic errors from tasks are presented.

Command Entry

In accordance with the findings of Wixon [3], AdaTAD is designed to use command driven user input instead of either a menu or iconic input. Commands exist to control the initiation, configuration, and completion of an AdaTAD session as well as to control the execution of the task being debugged. Arguments to commands are entered as parameters to the command line itself. Each task has a keyboard assigned to it for interactive input. When a task is the current task its keyboard is the physical terminal to which the task has been assigned.

Information Display

Since so much information is made available to the user of AdaTAD, a well engineered display is critical. We have designed an interface that combines textual and graphical status information in a windowing framework. The concept of windowing has recently received much attention. Windows allow a process to assume that it has a dedicated output device, independent of whether the window is being viewed. Assignment of screen geography can vary dynamically under user control to allow variable presentation of information.

The AdaTAD display consists of a set of task windows and a task interaction status display. The user may configure windows on the screen by using the **WINDOW DEFINITION** command. Figure 1 shows a task window and the panes that are included (the task interaction status display is presented in the next section). The panes display information about the current

execution state of the task, information on designated variables, the source code context and task output.

AdaTAD control commands manage the appearance of the debugger to the user and perform basic initiation and termination of users programs. The commands include:

EXECUTE	--initiate program and enable execution
DEFINE_WINDOW	--specify size and location of a window
ZOOM	--alter the size of a window
CURRENT_TASK	--task to which taskless commands apply
ASSIGN	--associate i/o device with a task
TERMINATE	--complete the interactive session

Although these commands are not specific to a particular task, they are needed in tailoring a specific debugging session for logic errors.

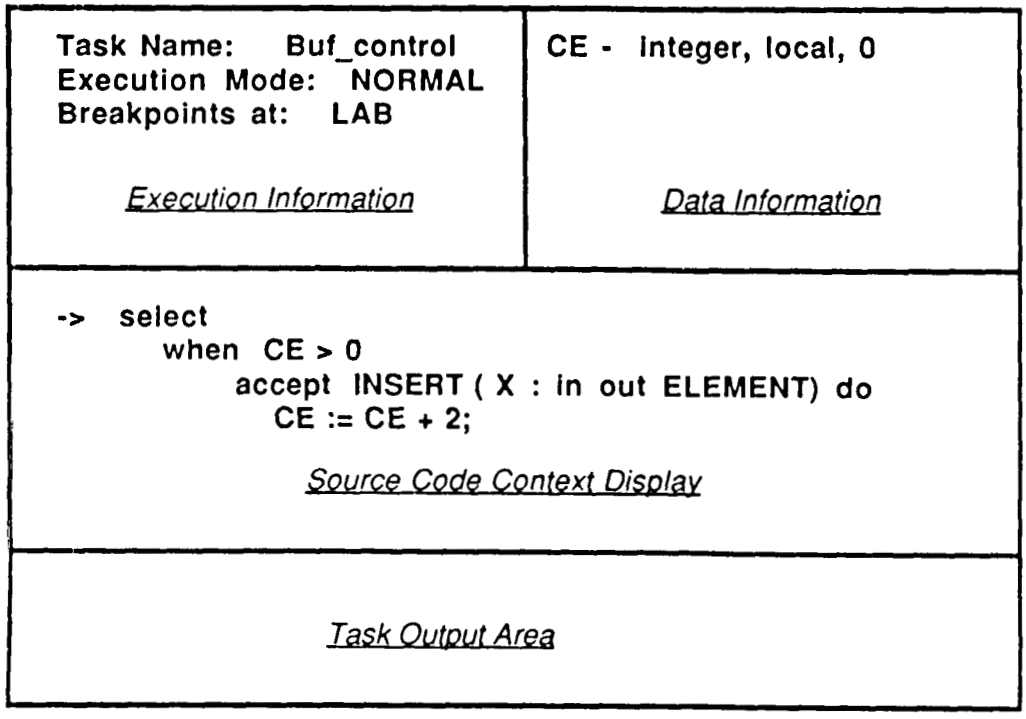


Figure 1. Task Window Format.

Task Execution Information and Control

Two breakpoint facilites are provided for controlling the execution of statements within a task. Assertion breakpoints may be placed within the a task by adding an **ASSERT** statement to the program, and unconditional breakpoints may be associated with any statement of a task. Since several allocated tasks may have the same task body, breakpoints cause breaks to occur in all tasks having the body.

Four modes of task execution are provided to accommodate various debugging techniques.

NORMAL	--execute until encountering break
SINGLE STEP	--user initiated statement execution
TIMED	--execute statements at a given rate
WAIT	--suspend task execution

When a task halts execution at a true or unconditional breakpoint, the task is placed in a **WAIT** mode of execution. Execution is resumed by explicitly placing the task in another execution mode (**NORMAL, SINGLE_STEP,** or **TIMED**).

Examination of Data.

AdaTAD provides facilities for viewing or altering the values of program objects by the object's source code name. If tasks communicate via shared variables, then AdaTAD aids in detecting any attempt to violate the assumptions described in section 9.11 of the Ada Language Reference Manual [4].

2.2 Using AdaTAD to Remove Logic Errors

Testing and debugging the logic errors within tasks can best be done by removing the influence exerted by the task's environment. The environment must be specified by the test case and controlled by the debugger. All interactions with other tasks, such as entry calls to the tested task, accepts of calls made by the tested task, or the use of shared variables are controlled during testing and debugging by AdaTAD stub facilities.

The test cases for this phase can be characterized as including input, environment and expected results. When the task is initiated in a state satisfying the input condition and executed in the environment specified then it should exhibit the expected results. The input condition describes

the values of inputs to the task. These may include the initial state of a database used by the task or of objects obtained through input.

The environment specification must describe the necessary interactions with other tasks to carry out a test case. When selective waits or conditional or timed entry calls are contained in the task, the specification indicates specific paths through the constructs relevant to the test. For example, a test case that is to examine a specific delay alternative must specify in its environment section conditions causing that delay to be executed. Further, to obtain the information needed for a test case it may be necessary to specify which task is to call a specific entry to the tested task.

The anticipated results of executing a test case may not be as simply expressed as an output condition to be true when execution completes. Tasks may execute indefinitely, may terminate in synchronization with others, or may transmit their results to other tasks through entry parameters. Accordingly, the anticipated result may be a condition to be true at a specific point during execution of the task (possibly within an iteration).

AdaTAD Support

AdaTAD facilities are used to execute test cases, and debugging can be done in conjunction with testing if needed. Facilities supporting the execution of test cases can be compared to those of other debuggers for handling procedure stubs. In AdaTAD, these facilites include commands to:

1. Cause a terminate condition to evaluate true,
2. Provide a dummy entry call to a task with specific arguments,
3. Cause an entry call to another task to be accepted and out parameters from that call to be set,
4. Deterministically select an alternative in a nondeterministic selective wait,
5. Selectively satisfy durations on delay statements.

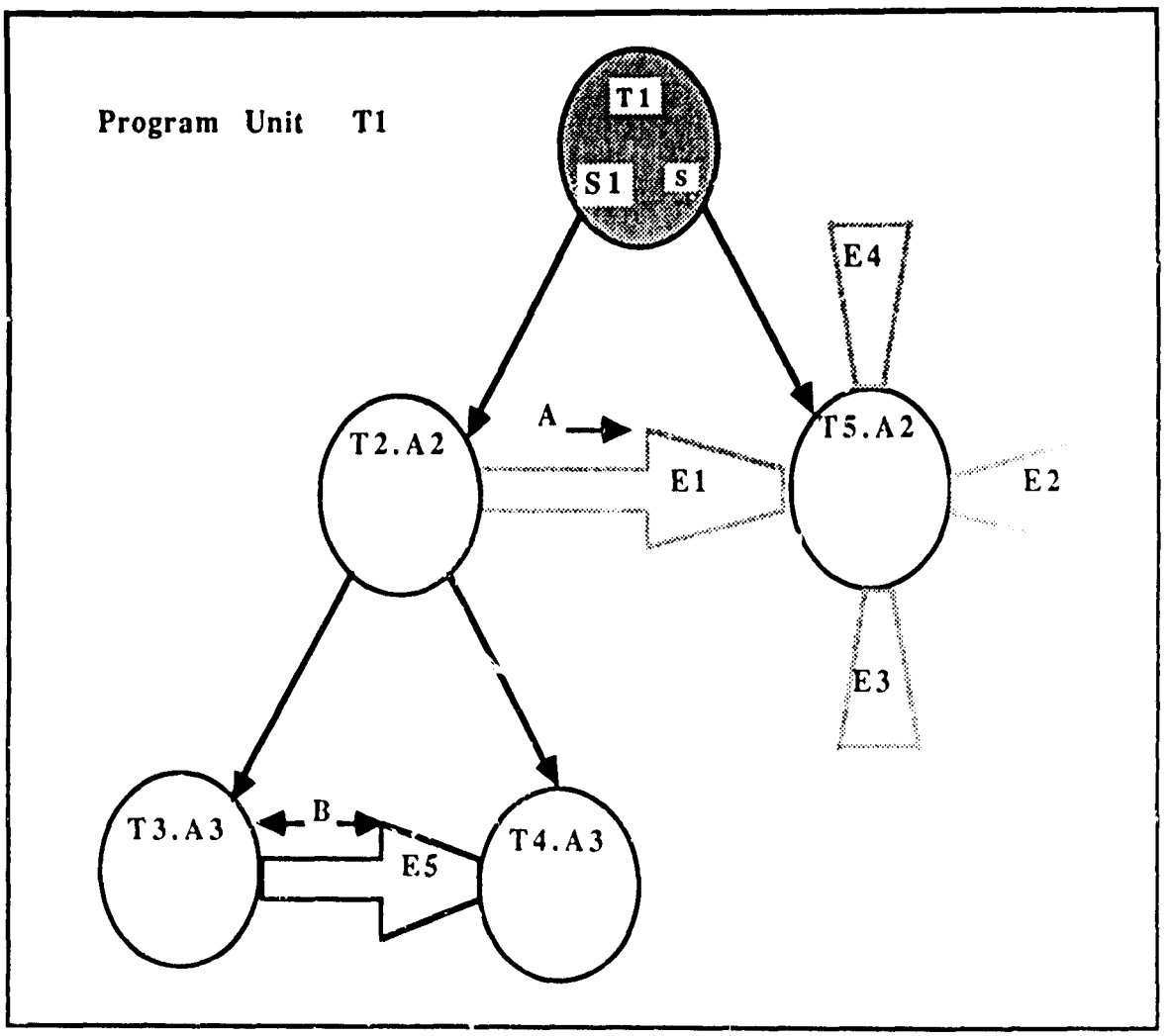
3. SYNCHRONIZATION AMONG TASKS

After checking the logic within a task, the communication and synchronization among tasks is considered. This step is analogous to integration testing in that the cooperation among possibly several tasks is addressed. Data flow and control flow through tasks of the program are observed at this level of testing and debugging. From the perspective of a single task, this level checks, in a rudimentary manner, the task's tasking environment. Subtle timing interactions and interactions with the operating environment are left to the final level of checking.

The scenario for testing and debugging follows the same approach as with task logic. Test cases are identified using source code analysis. Test cases are run using AdaTAD support, and errors are located and removed using AdaTAD debugging facilities. Test cases focus on task interaction. Input conditions and expected results are included, but no specific information describing task execution constraints is included.

3.1 Task Interaction Status

AdaTAD's Task Interaction Status window depicts the state of rendezvous and consequently is particularly useful for synchronization testing. Within the window a graph is used to represent tasks of the program and relationships between tasks. Each task has a corresponding node in the graph, and relationships such as "**depends on**" and "**is in rendezvous with**" are depicted by directed edges from one task to another. Figure 2 shows a hypothetical program unit, called **T1**, at some point of execution, and Figure 3 is the legend for the task status area. **T1** has four subordinate tasks, **T2, T3, T4** and **T5**. Each of these subordinates has an underlying task type: **A2** for **T2, T5** and **A3** for **T3, T4**. Arcs with solid arrow heads indicate the dependent relation among tasks. Thus in this example, **T1** has caused initiation of **T2** and **T5**. Rendezvous and communication status is conveyed through double-line arcs. The arc from **T2** to **T5**, with shaded lines, indicates that **T2** is waiting at an unaccepted entry call to **T5**'s entry **E1**. **E1** has a single input (IN) parameter, and for this call **A** is the argument.



Task Interaction Status Window

The large shaded arrowheads (without bodies) pointing at **T5** indicate the task will not be immediately accepting the call to **E1. T5** is waiting at a selective-wait with three open accepts (**E2, E3, E4**). The large solid arc from **T3** to **T4** indicates that these two tasks are currently in rendezvous. **T3** is the calling task and **T4** is the accepting task, as indicated by the arrow-head. For entry **E5** the argument is B, which is an IN OUT parameter.

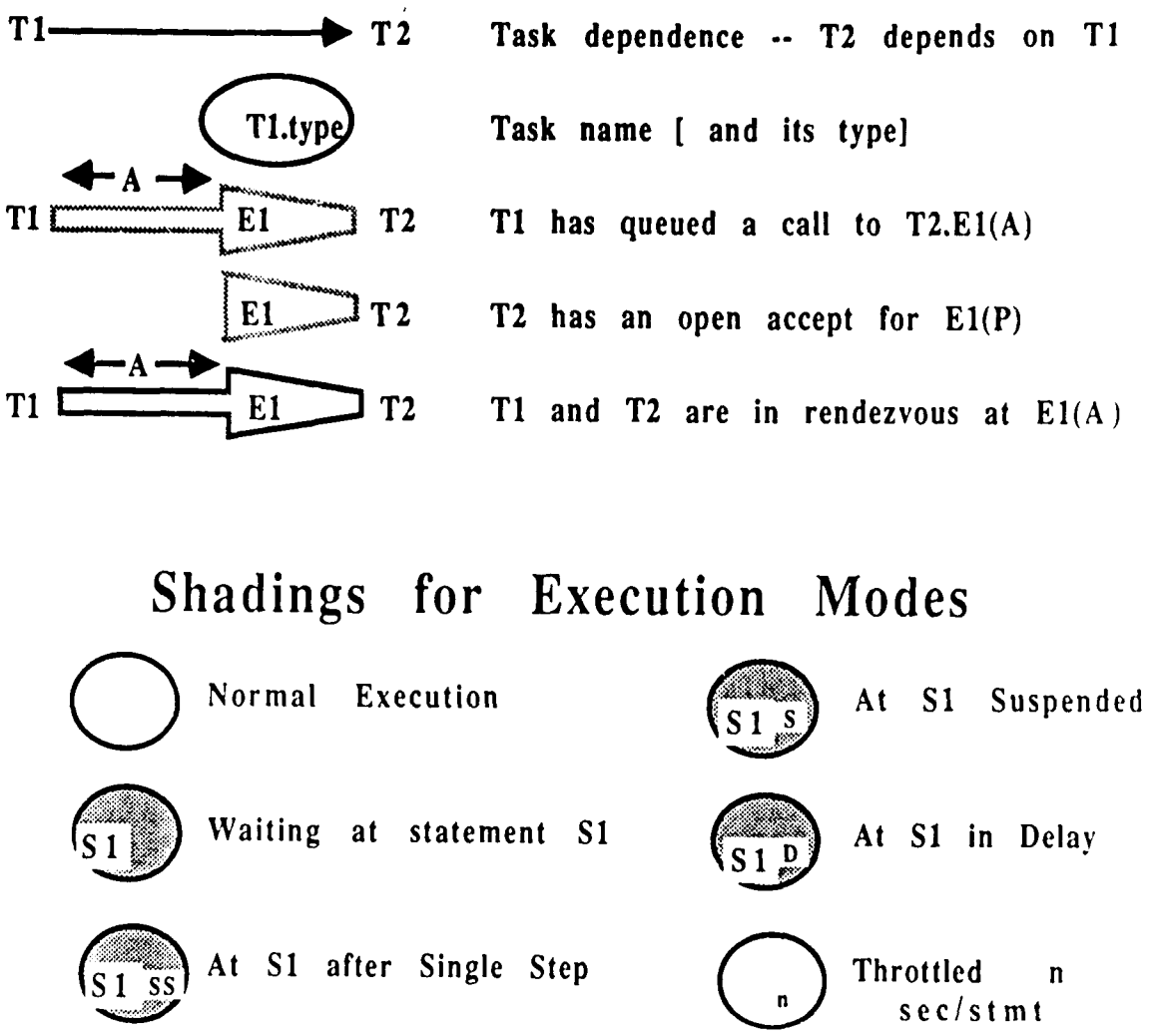


Figure 3. Legend for the task interaction status display

The main program unit, **T1**, is currently in a **WAIT** state of execution, as indicated by the shaded task node for **T1**. The small s in the lower right of the task node indicates the task is

waiting because it was suspended.

3.2 Execution Control For Checking Interactions

The displays generated by AdaTAD for checking task interactions are the same as those for logic checking within a task, but the capabilities available to the user differ. When checking task interactions, AdaTAD does not allow the user to:

1. Provide a dummy entry call to another task, or
2. Provide a dummy accept of an outstanding entry call.

Additional facilites are provided to specifically aid in debugging task interactions. These include:

1. Break at rendezvous beginning/completion,
2. Examine the calling queue for an entry,
3. Reorder the calling queue for an entry
4. Examine/alter arguments to an entry call.

Rendezvous breakpoints provide a means for control to return to the user at the boundaries of a rendezvous. When both tasks reach the synchronization point, the user may need to examine assertions, arguments, or results to determine correct communication between tasks. Rendezvous breakpoints may be associated with either pairs of tasks or with entries within a task. In one situation, the user may be interested in examining communication between tasks **T1** and **T2** each time they rendezvous, indepenf of the entry at which rendezvous occurs. In another situation, a user may need to know parameter information each time that a specific entry within a task in called, independent of what task is calling.

4. APPLICATION SPECIFIC USES OF ADATAD

The final stage of debugging considers the operating environment in which the tasks must execute. For an embedded system, this may include operating within a set of heterogeneous processors, each with different resources and capabilities. Testing and debugging at this level is

often accomplished with a simulation of the operating environment. While specific tools are necessary to support this activity, AdaTAD provides facilities that are useful in a general manner to the problem of addressing the operating environment.

The problems that may arise in this phase of testing include timing inconsistencies among tasks, space requirements of a task, or resource contention caused by task interaction. Device interactions for special purpose input or output may be one cause. Another cause may be constraints imposed on the program by task distribution or the interaction between the task scheduling strategy and the operating environment.

AdaTAD provides facilities that allow the user to monitor program elements that will reveal these environment related problems. Ultimately, we recognize that the program under observation may to some extent be perturbed by the debugger. Nonetheless, a certain amount of debugging can be useful in this phase. To a large degree, testing technology is not appropriate for revealing application specific errors. This is an area in which ad hoc stress testing has been most successfully applied.

The capabilities that support this aspect of testing include:

1. Call Queue Display,
2. Entry Call Frequency,
3. Accept Entry Frequency,
5. Statement Execution Frequency,
6. Object Update Frequency.

The user can request that certain entry call queues be displayed automatically when modified. This provides a monitoring ability for a service rendezvous that is used by several tasks. The frequency displays allow the user to selectively obtain information that will show the contention points in a program. Entry call frequency may be obtained in two forms, entry call by any task and entry call by a named task. Statement and Object frequency information is useful in determining the dynamic space requirements of a task. One can observe executions of allocator statements or updates to objects detailing the size of dynamic structures. Although these facilities do not directly support monitoring interactions with the external environment, often internal objects or statements reflect their status.

5. THE DESIGN OF ADATAD

As with any debugger, AdaTAD requires specific modifications to the compiler and linker. To allow the debugger itself to be designed and implemented in Ada, source code changes are made to provide synchronization through AdaTAD entries. AdaTAD is, itself, a set of Ada tasks. There

are four major cooperating tasks including:

1. AdaTAD Coordinator,
2. Data Base Monitor,
3. Command Processor, and
4. Terminal Communicator.

There are also two arrays of tasks, including:

1. Logical Processor Tasks and
2. Terminal Drivers.

Additionally, there is a task to handle input/output between the user's program and non-terminal input/output devices. Figure 4 is a diagram of the overall structure of AdaTAD.

AdaTAD tasks communicate via the rendezvous and a shared variable. The data base stores execution information about the user's tasks. AdaTAD effectively makes each user task part of a logical processor task, which controls its exeuction. The terminal communicator is responsible for receiving user commands and updating task displays. The data base monitor provides operations that both synchronize access to the data and perform data storage and retrieval functions. The coordinator mediates communication among logical processors whose user tasks are synchronizing. The coordinator is also responsible for directing parsed user commands to the appropriate logical processor.

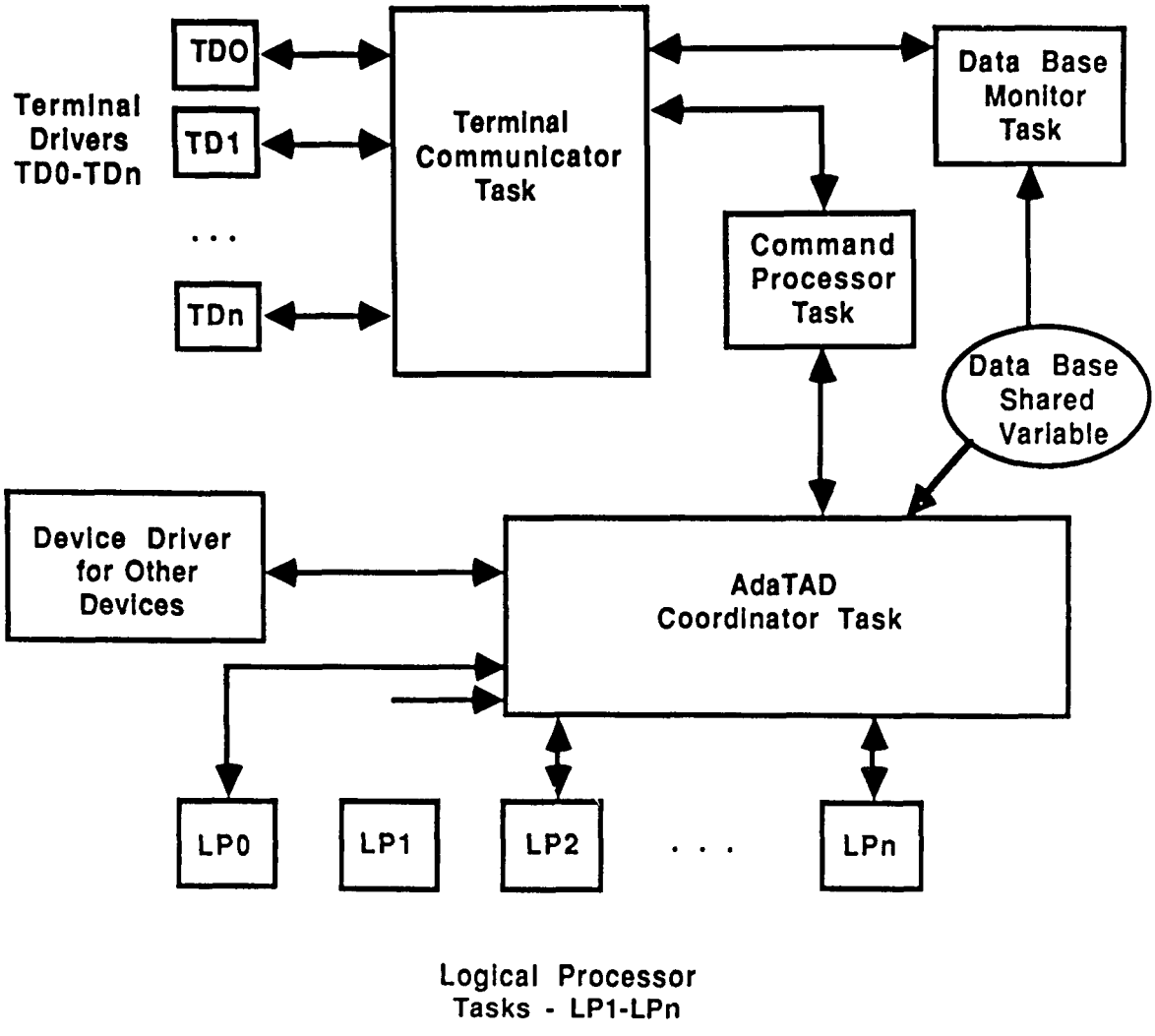


Figure 4. Task structure of AdaTAD.

5.1 Design of the Logical Processors

Logical processors are the most complex tasks in AdaTAD, because they monitor and control the synchronization among user tasks. Synchronization with other AdaTAD tasks is used to communicate the current state of execution to the data base maintained by the Coordinator. Logical processors have four entries for receiving input from the command interpreter, for servicing rendezvous requests from user tasks, for notifying rendezvous completion from servicing tasks, and for notifying task termination from other logical processors. Three tasks are defined within

each Logical Processor. The EXECUTOR task directly controls environment for the user task, the TRANSMITTER task serves as a funnel for messages to the coordinator, and the EXECUTION_AREA_MONITOR maintains the variables which reflect the current execution state of the user task. Although the presence of three tasks complicates the Logical Processor, it allows for maximal parallelism in the execution of the Logical Processor, and it minimizes the time spent by the user task in synchronization with AdaTAD.

Receive_User_Command

Through this entry, the logical processor is called by the coordinator when a user command is to be executed by the logical processor. A case statement within 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 of a task, the command is channeled to the appropriate logical processor and the execution state variable is changed.

Receive_Rendezvous_Completion.

When a rendezvous between two user tasks completes, the calling task must be released for further execution. To do this, the AdaTAD coordinator calls Receive_Rendezvous_Request. The call indicates that a rendezvous requested by the task running on the logical processor has been completed. The entry updates the local data base so that the user task can continue execution. Any arguments which were changed by the rendezvous exist in the argument list and are copied to the appropriate area.

The Executor Task

This task directly controls execution of the user's task. The compilation system modifies the user's task to physically nest it within the Executor. The Executor has one entry which is called when another user task has issurd an entry call to this task. The call is forwarded to the Executor by the logical processor's Receive_Rendezvous_Request entry when the coordinator sends an entry call. The compilation system converts rendezvous code into procedures that may be called to perform the rendezvous code. Thus, when the user task is ready to accept the call, the appropriate procedure is called.

Transmitter Task

The Transmitter sends messages to the AdaTAD coordinator. It is called by the user's task to request an input/output service or to inform the coordinator that a rendezvous has begun or completed. Transmitter is called by the execution area monitor to send the current state of the data base to the coordinator.

Execution Area Monitor Task

Since the execution data base is a shared variable that must be accessed by the Executor, the Transmitter and the Logical Processor itself, synchronization to the information is provided by the Execution_Area_Monitor. Tasks requiring information from the data base get the information by making an entry call to the monitor. The task services the following entries.

Sing_step_rel: 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 enables execution of the next statement.

Set_bk_state: Called by the Logical Processor to enable or disable breakpoint checking.

Set_ex_md: Called by the Logical Processor whenever the execution mode is to be changed.

Set_ex_rt: Called by the Logical Processor to set the rate for timed execution.

Set_timed: Called by the Logical Processor to enter timed execution mode.

Examine_exe: Called by the statement prologue to see whether statement execution is enabled.

When there are no outstanding entry calls to the monitor, the current execution mode is determined and the appropriate action is taken. If the execution mode is TIMED, the monitor determines whether it is time to execute the next statement.

5.2 The Coordinator Task

The AdaTAD coordinator mediates communication among AdaTAD tasks. When user tasks rendezvous, the coordinator handles communication among their Logical Processors. This mediation occurs when a rendezvous is requested, when a rendezvous completes and when a rendezvous begins. The Coordinator also mediates input/output requests for user tasks. To allow all appropriate information regarding the execution status of tasks, all communication with the underlying operating system must be recorded. This is done when a user's task requests service and when control returns from the operating system facility. Two further functions of the

Coordinator are to dispatch AdaTAD user commands to the appropriate Logical Processor and to collect status information for data base modifications. The coordinator interactively accepts entry calls to its entries in the order in which they arrive. We now describe the Coordinator in terms of its entries.

Rendezvous_request.

When one user task requests a rendezvous with another, the requesting task's Logical Processor makes a call to this entry of the Coordinator to initiate the rendezvous. The coordinator, in executing the call, looks up the Logical Processor for the called task. The name of the called task is taken from a descriptor list which also includes parameters for the call. Before making an entry call to the Logical Processor of the called task, Coordinator sets an indicator to show that the calling task is awaiting synchronization.

Rendezvous_begin.

When a rendezvous begins, the called task calls this entry with the names of the two synchronized tasks. The entry updates the synchronization information for the two tasks. It clears the **waiting** indicator, sets the **is_synchronized** indicator and records the names of the called and calling tasks in the synchronization data base.

Rendezvous_completion.

When the called task completes its rendezvous code, its logical procesor calls this entry. This occurs when the servicing task either terminates or encounters the end of the synchronized code of an accept statement. The entry updates the synchronization data base to reflect the rendezvous has completed. Further, an entry call is made to the logical processor running the served task so it may continue execution. The single parameter for this entry is the name of the task which has been served.

Data_base_update.

Each logical processor has local data that controls the execution of the user's task. When that data changes, the central data base is periodically informed through calls to this entry by Logical

Processors. Parameters convey the task name and its execution state. A local procedure, which the entry uses to perform the update, blocks the data base monitor task from looking at the data base while doing the update.

5.3 Data Base Monitor Task

The data base monitor is used to implement exclusive modification of the data base and to drive terminal updates of task status. An AdaTAD task acquires exclusive access to the data base through the Monitor's Hold and Release entries (P and V). For example, Hold is called by the Coordinator prior to making data base modifications required by a user command. After completing the modifications, Release is called.

The current state of the data base is transmitted to the Terminal Communicator task for display when no other task is modifying the data base. This is accomplished with an else clause on the selective wait for the Monitor's Hold entry. If no AdaTAD task has queued a call to Hold when the selective wait is encountered, then the else clause is executed and information is sent to the Terminal Communicator.

5.4 The Command Processor Task

The Command Processor analyzes 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, and an error message is sent back to the Terminal Communicator. The internal procedure Analyze_Command does the lexical and syntactic analysis of the command.

Parse is the only entry into the Command Processor. Parse is called by the Terminal Communicator when unsolicited input occurs on a terminal.

5.5 The Terminal Communicator Task

A task's terminal input and output is controlled by the logical processor, through the mediation of the Terminal Communicator. The Terminal Communicator also provides the 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 Processor.

From_Terminal and From_Coordinator

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. 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 From_Coordinator entry is called by the Coordinator when a user task requires input or output. We call this solicited input or output. The second accept for the From_Terminal entry is used for input of solicited information. From_Terminal is accepted after accepting the From_Coordinator entry. These entries are called when a user task has requested terminal input.

From_Command_Processor

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 the command was entered.

5.6 The Terminal Drivers

The Terminal Drivers are an array of tasks that handle the transmission of data between the physical terminals and the Terminal Communicator. The Terminal Driver has four entries and one internal task which has no entries.

The Output entry is called by the Terminal Communicator to write a string on a terminal. It then calls the Output entry in the Terminal Driver. Output accepts the string and writes it on the device through the appropriate Terminal Driver. The Input entry passes the string and the Terminal Driver number to the Terminal Communicator.

Terminal_watcher

Internal to the Terminal Driver is a task whose sole job is to wait for an input string from the terminal. When a string is received, as indicated by a terminal character, the task makes an entry

call to the terminal driver's input entry, passing the string. The identify entry is called by the Terminal Communicator as soon as the driver begins execution, to assign the driver a number, which is used in all communication.

5.7 An Example of Synchronization Among User Tasks

Controlling the synchronization of user tasks is the most complex of actions that AdaTAD performs. AdaTAD must intervene when a rendezvous request is made, when the rendezvous begins, and again when the rendezvous ends. To keep track of these interactions, the compiler converts user entry calls to calls of AdaTAD task entries. The compiler also generates code to inform AdaTAD when a rendezvous actually starts and when it completes. In this manner, AdaTAD can record the status of all user task synchronization. These actions occur whenever a rendezvous request is made, but they are normally transparent to the user. The following paragraphs describe what occurs in each case of AdaTAD intervention.

As an example of how AdaTAD controls execution, assume that two tasks (A and B) are running. Assume that task A wants to make an entry call to task B's entry named E1. Since the example is concerned with synchronization only, we assume that no data are passed during the rendezvous. Assume further, task A is running on logical processor one and task B is running on logical processor two.

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 names the calling task, the called task and the called entry. The compiler converts the statement B.E1 into:

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

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 argument list, to get the name of the called task, 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, A, will be updated to reflect that it is waiting for a rendezvous, and 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 Logical Processor accepts the rendezvous request, it passes the argument list to the executor running task B. The Executor receives the request at its Rendezvous entry and extracts the name of the called task and entry from the argument list. The name of the calling task (A) 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. If appropriate, 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. Assuming that the called task, B, is waiting at the entry being called, the Executor's entry call is answered immediately and the user's task begins execution.

Accepting 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 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 in the display area. After the user's task indicates that the rendezvous has been accepted, AdaTAD does not intervene. 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. 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, as indicated by calling Receive_Rendezvous_Completion 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 it can proceed.

Wait for Synchronization

If the called task in the above scenario is not waiting at the entry, it would not immediately inform the coordinator that the rendezvous had begun. Thus, the coordinator would reflect the wait in its data base. The user would be able to see the called task executing elsewhere and the calling task waiting.

6. SUMMARY

The problem of testing and debugging Ada programs that make extensive use of tasking facilities has been addressed in this paper. We have considered an approach to debugging tasks that is similar to the scenario in which software units are first considered. Following units, interactions among units are addressed. Our approach recommends a three tier approach to debugging tasked programs. The first tier considers the logic of tasks independent of their interactions. The second tier addresses interactions among tasks that take place through rendezvous and synchronized access to shared data. The final tier deals with application specific concerns. Here, the subtlities of the interactions between a tasked program and its operating environment are considered.

We have presented the design of a debugger suitable for applying this methodology. AdaTAD , which stands for Ada TAsk Debugger, includes facilities specific to each of the tiers. When used in conjunction with a testing methodology, AdaTAD supports the execution of test cases and the process of locating and fixing errors uncovered through testing. We have presented the user interface to AdaTAD in conjunction with an explanation of the three tiered approach to debugging tasked programs.

The applications for which Ada is intended require a level of technology that currently doesn't exist in today's Ada compilation systems. For embedded real-time systems, a compiler must support the distribution of an Ada program across a set of possibly hetergeneous processors. When such compilation systems appear, we will immediately be faced with the challenge of demonstrating the reliability of Ada software. In addition to modifying existing testing and debugging methodologies, special purpose tools such as AdaTAD will be required. To ease the implementability of a system such as AdaTAD, we have designed the bulk of the system in Ada. While an Ada design certainly compromises execution efficiency, it also eases implementations. The final section of this paper has presented the Ada design of AdaTAD together with an example of how synchronization can be controlled and monitored using Ada primitives.

7. REFERENCES

1. Miller, E.; et.al. Program Testing, IEEE Computer, Vol. 11, No. 4, April 1978 pp. 10-12.

2. Helmbold and Luckham, "Debugging Ada Tasking Programs," IEEE Software, March 1985.

3. 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.

4. Ada Language Reference Manual.