ADA LANGUAGE CONSIDERATIONS FOR CONCURRENCY ANALYSIS

Douglas L. Long⋆
Lori A. Clarke†
Joseph Fialli†

COINS Technical Report 89-42
May 1989

⋆Department of Computer Science
Wellesley College
Wellesley, Massachusetts 02181

†*Software Development Laboratory*
Department of Computer & Information Science
University of Massachusetts
Amherst, Massachusetts 01003

This paper to appear in
The Proceedings of the
6th Washington Ada Symposium

This work was supported in part by Office of Naval Research grant N00014-88-K-0025 and by National Science Foundation grant CCR-87-04478 in cooperation with the Defense Advanced Research Projects Agency (ARPA Order No.6104).

This work was supported in part by Office of Naval Research grant N00014-88-K-0025 and by National Science Foundation grant CCR-87-04478 in cooperation with the Defense Advanced Research Projects Agency (ARPA Order No.6104).

Abstract

The concurrency features of Ada can result in very complex program behavior. This can inhibit both program understandability and the development of analysis tools to help programmers write correct concurrent programs. This paper discusses some of the causes of this complex behavior and suggests ways of avoiding the resulting problems by limiting the use of certain concurrency features. When adhered to, these limitations result in programs that are easier to comprehend and analyze.

1 Introduction

Concurrent programming is an important tool for solving complex problems, but the increased complexity of concurrent programs makes them more difficult to write and test. In fact, careless usage of concurrency and poor design may lead to concurrent programs that behave in ways that are too complex to understand or to analyze. This problem is alleviated somewhat in Ada because it provides extensive concurrent programming facilities that are simpler to understand and easier to use than those provided by many other languages. Nevertheless, Ada contains features that may make concurrent programs harder to understand than necessary. This paper discusses several potential problems and suggests ways to avoid them.

The difficulty of writing concurrent programs suggests that automated analysis tools are necessary to help in the development and testing of such programs. We are in the process of developing such tools. Our approach, derived from the static concurrency analysis of Taylor [Tayl83b], is based on *Task Interaction Graphs*(TIG), a graphical representation of a task that emphasizes information about task interactions over information about control flow. The TIG approach, described more fully in [Long89], is designed to reduce the number of states that must be considered in static concurrency analysis. At the same time, the TIG approach is able to model a number of concurrency constructs, such as hierarchically related tasks and shared variable accesses, that other analysis techniques have chosen to ignore [Dill88,Kemm88].

2 The Rendezvous Model

In Ada, a concurrent program consists of a number of *tasks* that may be executed concurrently. Ada requires that tasks synchronize with each other at certain points. These synchronization points allow the coordination of and the exchange of data between tasks and are central to the understanding of the behavior of a concurrent Ada program. Synchronization points occur at the start and end of a rendezvous between two tasks, between master and dependent tasks, and when a pragma SHARED variable is read or updated.

The primary way tasks interact with each other is through rendezvous. The Ada rendezvous provides synchronous communication between two tasks and occurs when a client task makes an *entry call* to an entry of a server task and the server task *accepts* the entry call. The execution of a task making an entry call or an accept is suspended until another task makes a corresponding accept or call. The rendezvousing tasks synchronize at the start and the end of the rendezvous. Calls and accepts may have parameters that are used to exchange data at this time.

Tasks may be nested in other tasks. One or more tasks that are nested within the scope of a master task are said to be *dependent* on the master. Dependent tasks become active just prior to the start of execution of the first statement in the body of their master and must terminate before the program can leave the scope of the their declaration. A master task synchronizes with its dependents at the point at which the dependents are activated, i.e., the dependent tasks begin the elaboration of their declarative regions, when all its dependent tasks have finished the elaboration of their declarative regions and again when all its dependent tasks have completed execution.

A read or an update of a pragma SHARED variable is also a synchronization point. Particular attention must be paid to the use of shared variables and to the difference between those shared variables that are declared as pragma SHARED variables and those that are not. A shared variable is a variable that is accessed by two or more tasks. Since Ada tasks are defined as independent entities that can be distributed on computers connected by a network, one cannot assume that access to a shared variable is the same as an access to a global variable. Since tasks may execute in completely separate address spaces, a shared variable access can

translate into a network communication. Since the access to shared variables can be costly and time-consuming, the specifications of Ada allow a task to maintain a local copy of a shared variable between synchronization points. Shared variables are updated at synchronization points, e.g., at the start and end of rendezvous and when master and dependent tasks synchronize. Because of this delay, a read or an update of a shared variable is not a synchronization point and it is considered erroneous to have one task modifying a shared variable at the same time another task could be referencing it. On the other hand, declaring a variable to be a pragma SHARED variable forces a read or an update of that variable to be a synchronization point. I.e, an access to a pragma SHARED variable causes data to be communicated immediately with other tasks. Thus we do not consider the concurrent read and update of a pragma SHARED variable to be erroneous; the correctness or incorrectness of this must be determined from the context.

In the TIG model these synchronization points are known as *task interactions*. These task interactions are the points at which one task can influence the behavior of another; between synchronization points a task executes sequentially and independently of the other tasks in the program. The areas between task interactions are known as *task regions*. In a TIG, task regions are represented by nodes of the graph and task interactions are represented by edges of the graph. The execution state of a concurrent program can be described by indicating the region that each task composing the program is currently executing. The state of the program changes when a task interaction occurs, causing one or more tasks to execute in different regions. Certain types of analysis require the examination of different states of the program. For example, erroneous usage of shared variables can be detected in this way. It is simply a matter of looking for a state that contains a task that is in a region that contains a write to a shared variable and another task that is in a region that contains a read or write of the same shared variable. If all states are examined, then all potential erroneous usage of shared variables can be detected. One way that these states can be generated is by constructing a concurrency graph [Tayl83b], which, when based on TIG's, is referred to as a Task Interaction Concurrency Graph (TICG) [Long89]. The TICG for a program provides a means for determining the correct usage of pragma SHARED variables, deadlock detection, and other anomalies [Youn86].

3 Task Activation and Concurrency Analysis

The state based analysis technique described in the previous section may require the generation of many states; if too many states are generated then the analysis may become intractable [Tayl83a,DeMi79]. Many of these states are necessary to correctly represent the behavior of the program, but others only represent possible orderings of sequences of independent events. For such analysis to work it is necessary to minimize the latter without losing information about the former. The number of possible states also has an impact on the understandability of the program; a change in a program that reduces the number of states in its TICG should make the program easier to understand.

One place where a large number of states may be generated is during the activation of dependent tasks and the elaboration of their declarations. Consider the three tasks shown in Figure 1. In this example, the MASTER task declares and activates two dependent tasks, T1 and T2. Ada requires that certain parts of the program complete before other parts of the program may begin. The precedence diagram in Figure 2 shows which parts must finish before other parts may begin. In this diagram, an arrow from one part of the program to another indicates that the first part must complete before the second may start. On the other hand, if there is no path from one part of a program to another in this diagram then those parts may execute concurrently. For example, the dependent tasks are activated when MASTER reaches the end of its declarative region. At this point MASTER suspends execution and T1 and T2 may begin the elaboration of their declarations. MASTER cannot resume execution until both T1 and T2 have finished elaboration of their declarations. On the other hand, once either T1 or T2 has finished the elaboration of its declarations it may immediately begin execution of its body. Thus the execution of T1 and T2 proceed independently of each other as shown in Figure 3. As can be seen in this picture T1 and T2 may elaborate their declarative regions concurrently, T1 may elaborate its declarative region while T2 executes its body, T2 may elaborate its declarative region while T1 executes its body, or both may execute their bodies concurrently. These four cases must be differentiated from one another because the MASTER's body may not exe-

```
procedure MASTER is
   --BEGIN MASTER decl_region
   task T1,T2;

   task body T1 is separate;
   task body T2 is separate;
   --END MASTER decl_region
begin -- MASTER
   MASTER_body;
end MASTER;

separate(MASTER)
task body T1 is
   T1_decl_region;
begin
   T1_body;
end T1;

separate(MASTER)
task body T2 is
   T2_decl_region;
begin
   T2_body;
end T2;
```

Figure 1: An Ada program containing nested tasks

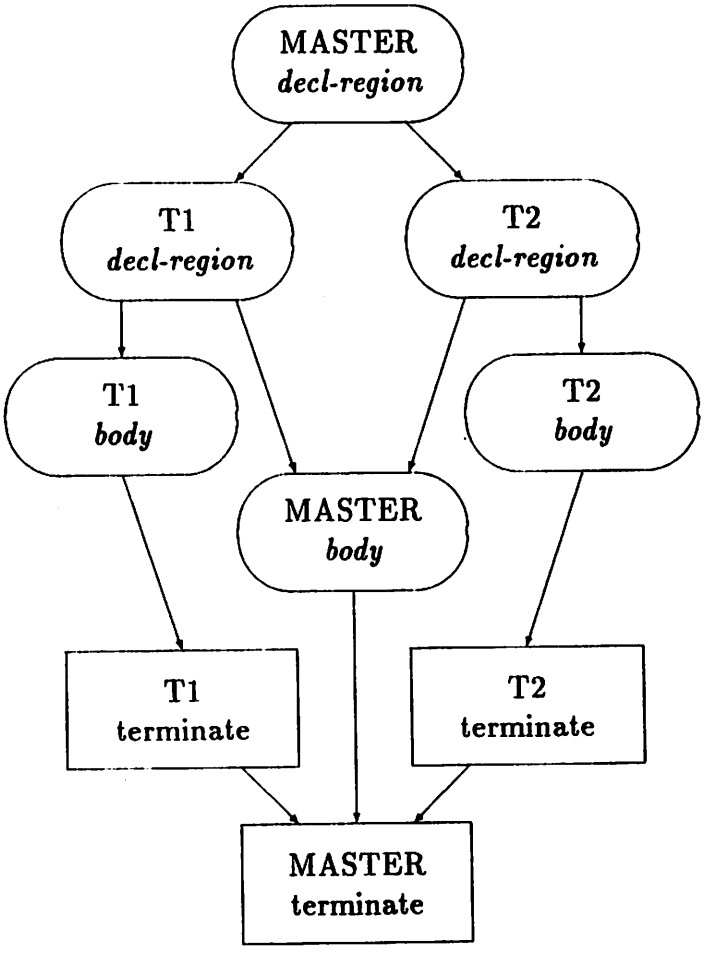

Figure 2: Precedence graph of a nested Ada program

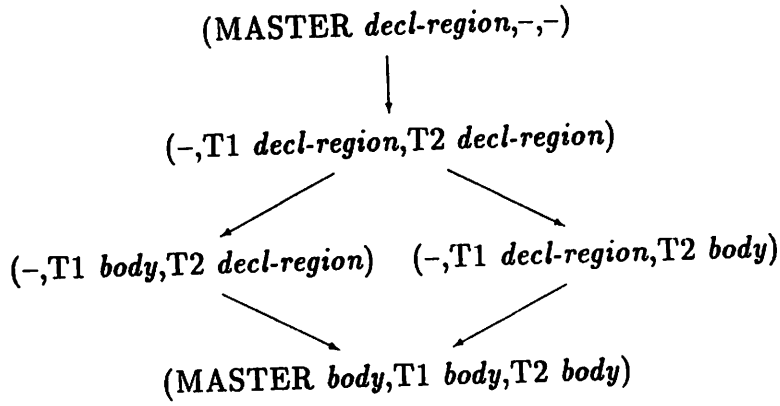

Figure 3: Concurrent execution

cute until after T1 and T2 have finished elaborating their declarations. At least four states are required to represent these four cases. The situation is much worse if there are more dependent tasks. If there are n dependent tasks then at least 2^n states are necessary.

The presence of task interactions in declarative regions complicates the behavior of a concurrent program. These may occur as references to pragma SHARED variables, calls to functions that have task interactions as side effects, or in the elaboration code of nested packages. Regardless of their form, these task interactions force the consideration of all the cases outlined above for the activation and execution of dependent tasks because the task interactions that occur in the declarative regions of the subtasks must occur before any of the task interactions in the body of the MASTER task. Consider, for example, the program shown in Figure 4. At the end of execution of procedure MASTER the variable M may have a value of 10, 20, 30, or 40, depending on which of the 16 possible orderings of the elaboration of the declarative regions and the execution of the task bodies actually occurs. The shared variable SHARED may have a value of 20 or 40, depending on which of the 6 possible orderings of the execution of the task bodies actually occurs. In addition to the proper consideration of the task interactions in declarative regions, the possibility that a declarative region might contain a shared variable (not pragma SHARED) also requires

```
procedure MASTER is
    M,SHARED : INTEGER := 0;
    pragma SHARED(SHARED);
    function SIDE_EFFECT(X : in INTEGER)
        return INTEGER is
    begin
        SHARED := X;
        return X + 1;
    end
    task T1,T2;

    task body T1 is separate;
    task body T2 is separate;
begin -- MASTER
    M := SHARED;
end MASTER;

separate(MASTER)
task body T1 is
    N : INTEGER := SIDE_EFFECT(10);
begin
    SHARED := 20;
end T1;

separate(MASTER)
task body T2 is
    N : INTEGER := SIDE_EFFECT(30);
begin
    SHARED := 40;
end T2;
```

Figure 4: An Ada program containing nested tasks

the consideration of all these cases.

The complete analysis of a program may require the consideration of a very large number of program states. Part of the reason is that task interactions that occur in the declarative region of the master task must occur before the task interactions that occur in the declarative region of any of the dependents, and the task interactions that occur in the declarative region of the dependent tasks must occur before any task interactions that occur in the body of the master. The other part of the reason is that the dependent tasks are free to move from their declarative regions to their bodies independently of the other tasks. It is this combination of events that requires the consideration of so many possible orderings of execution.

4 Restrictions to Ada

The source of much of the trouble described in the last section is the synchronization point that forces the master task to wait for all its children to finish elaboration of their declarations. This synchronization point was added, not for concurrency related reasons, but for reasons related to exception handling [Barn84]. While this synchronization point may be useful from the exception handling perspective, it makes the analysis of concurrent Ada programs with nested tasks more difficult. The question that arises is, are the benefits of this feature worth the cost of this added complexity? One could accept this added complexity as the cost of doing business and proceed accordingly. On the other hand, it is not clear that what is gained is worth the cost. The use of language features that cause task interactions in declarative regions should be approached with some caution. These interactions occur as side effects of the declarations and in many cases it is not readily apparent to the programmer that these side effects are occurring. Our approach to this problem is to look for restrictions on these features that will mitigate the impact of the troublesome synchronization point on the analysis. There is more to gain from analysis tools that aid the development of concurrent programs than there is to lose from the restrictions necessary to make these tools possible.

We have considered several possible restrictions and evaluated each based on three criteria. First, the restriction must have the desired effect of reducing the cost of the analysis. Second, it must not be so restrictive

that it will eliminate useful programming constructs. Third, an analysis tool must be able to detect and warn the programmer of an occurrence of a violation of the restriction that may invalidate the concurrency analysis. This latter problem is of particular interest. Certain queries, such as does a function call have any task interactions as side effects, may require a difficult global analysis of the entire program. Further work is needed in this area. For the initial implementation of our analysis tools we have adopted restrictions that are easy to enforce. Later implementations will relax these restrictions as much as possible while maintaining the ability to detect violations.

The restrictions that we consider focus on the declarative regions of tasks. If there are no interactions occurring in the declarative regions of the dependent tasks (or in the body of the master task) then the synchronization point that forces the delay in execution of the body of the master can be safely ignored without introducing errors into the analysis and without requiring the analysis of a large number of states. Each potential source of task interactions in declarative regions, shared variables, function calls and nested tasks, must be considered.

The first potential source of problems is shared variables, both plain and pragma SHARED. Shared variables may be referenced in a declarative region any place that an expression may be used and the value of a shared variable may be changed as a side effect of a function call or the execution of the elaboration code of a nested package. One could eliminate the problem by disallowing references to shared variables in declarative regions of tasks. This is not a very desirable choice for two reasons. First of all, it is very restrictive. It would eliminate useful programming techniques such as the one shown in Figure 5, where a master task uses a shared variable to broadcast information to all its dependent tasks. Second, the identification of shared variables requires global analysis.

There is a less restrictive alternative that would allow limited forms of broadcasting from a master task to all its dependents. This alternative would allow writes to a shared variable only in the declarative region of the master tasks. The declarative regions of the dependent tasks, the bodies of the dependent task, and the body of the master task would be allowed to read the value of the shared variable but not to write to it. This restriction allows limited use of shared variables, such as that shown in Figure 5, and will allow the synchronization point that forces the delay in the execution

```
procedure MASTER is
    S : INTEGER := 100;
    task T1,T2;

task body T1 is
    N : INTEGER := S;
begin
    ⋮
end T1;
task body T2 is
    N : INTEGER := S;
begin
    ⋮
end T2;
begin -- MASTER
    ⋮
end MASTER;
```

Figure 5: Broadcasting information to dependent tasks

of the body of the master to be ignored. Thus this restriction will help keep the analysis tractable by helping to minimize the number of states that must be considered. Unfortunately, it is more difficult to enforce such a restriction because it requires being able to determine whether or not a shared variable can be changed as a side effect of a function call or package body elaboration.

An alternative restriction is to allow only static simple expressions in declarative regions. This is only slightly less restrictive than the first restriction that was considered, but has the decided advantage that it is easy to enforce. The initial implementation of the analysis tools we are building uses this restriction.

The two other ways that task interactions can occur in declarative regions are as side effects of function calls and in nested packages. The best way to deal with function calls is to check each function call for side effects. Once again, this is a more difficult analysis that might be included in a later implementation. An alternative is to disallow function calls in declarative regions altogether. This is easy to enforce and is not so restrictive that it is impossible to find ways around, but may lead to some awkward code. This restriction is subsumed by the previous restriction that only allows static simple expressions in declarative regions.

The easiest way to deal with nested packages is to disallow their use. This is easy to enforce but may be considered by some to be too restrictive. An alternative is to allow nested packages, but not to allow package body elaboration code. This is easy to detect and is not very restrictive. The package body elaboration code is easily replaced with an explicit initialization subprogram. Even less restrictive would be to allow package body elaboration code as long as it didn't contain task interactions as side effects. Unfortunately, this restriction is more difficult to enforce.

The initial implementation of the analysis tools allows only static simple expressions and disallows function calls and nested packages in declarative regions or tasks. We will continue to look for ways that will allow us to ease these restrictions in later implementations while at the same time being able to guarantee the detection of violations of the restrictions.

While the restrictions that we have considered have been evaluated in the context of the analysis tools we are building, they can also be used as guidelines to help programmers avoid unnecessarily complex programs. The avoidance of all task interactions in the declarative regions and the careful

use of shared variables, such as is suggested in this section, will result in programs that are are easier to understand and less likely to behave in unanticipated ways. Unfortunately, even the most diligent programmer can make mistakes or may not not have enough information available to be sure that there are no task interactions in the declarative regions. Automated tools should be used to verify that these restrictions have not been violated. The tools we are constructing will help the programmer detect features of concurrent programs that might cause unanticipated or incorrect program behavior.

5 Conclusion

We are in the process of developing tools that will aid in the development of concurrent programs. In the process of designing these tools we have identified a number of features of Ada that make the analysis less tractable. In particular, the synchronizations required between master and dependent tasks cause a combinatorial explosion in the number of states that must be considered during static concurrency analysis. We have identified several restrictions to Ada that will make the analysis more tractable without being overly restrictive. Violations of these restrictions are easy to recognize. In addition, these restrictions provide guidelines that programmers can use to avoid overly complex and unexpected program behavior. In future work, we will investigate relaxations to these restrictions and the global analysis techniques necessary to recognize violations of these relaxed restrictions.

References

[Ada83] **Reference Manual for the Ada Programming Language (ANSI/MIL–STD–1815A)**, United States Department of Defense, Washington, D.C., January 1983.

[Barn84] J. G. P. Barnes *Programming in Ada.* Addison-Wesley, 1984.

[Brin78] Per Brinch Hansen. Distributed Processes: A Concurrent Programming Concept. *Communications of the ACM*, 21(11):934–941, November 1978.

[Bris79] G. Bristow, C. Drey, B. Edwards and W. Riddle. Anomaly Detection in Concurrent Programs. *Proceedings of the 4th International Conference on Software Engineering*, 265–273, 1979.

[DeMi79] Richard DeMillo and Raymond Miller. Implicit Computation of Synchronization Primitives. *Information Processing Letters*, 9(1):35-38, July 1979.

[Dill88] Laura K. Dillon. Symbolic Execution-Based Verification of Ada Tasking Programs. *Proceedings of the Third International IEEE Conference on Ada Applications and Environments*, 3-13, May 1988.

[Geha83] Narain Gehani. *Ada, An Advanced Introduction.* Prentice-Hall, Englewood Cliffs, New Jersey, 1983.

[Helm85] David Helmbold and David Luckham. Debugging Ada Tasking Programs. *IEEE Software*, 2(2):47–57, March 1985.

[Hoar78] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[Kemm88] L.J. Harrison and R.A. Kemmerer. An Interleaving Symbolic Execution Approach for the Formal Verification of Ada Programs with Tasking. *Proceedings of the Third International IEEE Conference on Ada Applications and Environments*, 3-13, May 1988.

[Long89] Douglas L. Long and Lori A. Clarke. Task Interaction Graphs for Concurrency Analysis. *Proceedings of the Eleventh International Conference on Software Engineering*, May 1989, to appear.

[Tai85] K.C. Tai. On Testing Concurrent Programs. *Proceedings of COMPSAC 85*, 310–317, October 1985.

[Tayl80] Richard N. Taylor and Leon J. Osterweil. Anomaly Detection In Concurrent Software By Static Data Flow Analysis. *IEEE Transactions on Software Engineering*, SE-6(3):265–278, May 1980.

[Tayl83a] Richard N. Taylor. Complexity of Analyzing the Synchronization Structure of Concurrent Programs. *Acta Informatica*, 19:57–84, 1983.

[Tayl83b] Richard N. Taylor. A General-Purpose Algorithm For Analyzing Concurrent Programs. *Communications of the ACM*, 26(5):362–376, May 1983.

[Youn86] Michal Young and Richard N. Taylor. Combining Static Concurrency Analysis With Symbolic Execution. In *Proceedings of the Workshop on Software Testing*:170–178, IEEE Computer Society Press, July 1986.