Annotating Ada for Real-Time Program Synthesis

Aloysius K. Mok†
Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712

Key words: Ada, real-time programming, annotation, specification, verification, real time logic

ABSTRACT

Although Ada is intended to be a programming language for embedded real-time systems, the tasking features of Ada are in general not sufficient for expressing the stringent timing constraints such as those found in time-critical applications. Without detailed knowledge of the run-time system, the timing behavior of real-time Ada programs is hard to analyze and is likely to be unpredictable. In this paper, we shall introduce a logic-based system of annotating Ada programs so as to make it easier to control the timing behavior of real-time systems written in Ada.

Introduction

As the Ada‡ language is being applied to build time-critical embedded systems, it is becoming apparent that the tasking features of Ada do not provide sufficient support for meeting stringent timing constraints. A number of researchers have proposed enhancements to Ada for dealing with various timing and scheduling issues. A fundamental question, however, is whether Ada is adequate for specifying the wide spectrum of timing behavior found in real-time systems. The purpose of this paper is to examine this question by considering the difficulties involved in deriving the timing behavior of real-time systems from their Ada source code. Our investigation leads us to propose an approach for specifying the timing behavior of Ada programs by adopting a system of annotation based on RTL (Real Time Logic).

The Basic Issues

In our research (e.g., see [Mok 84], [Mok 85a]), we have been investigating the fundamental problems in automating the production of efficient schedulers to meet the stringent timing constraints in real-time programs. We have observed that there are two crucial properties that a real-time programming language should have:

1. It should support direct expression of the desired real-time behavior. (A single

† Supported by a research grant from the Office of Naval Research under ONR contract number N00014-85-K-0117 and also URI grant ONR contract number N00014-86-K-0763
‡ Ada is a registered trademark of the United States Department of Defense.
execution of a program generates a sequence of events. The behavior of a program is commonly described by the sequences of events that can result from an execution of the program. The timing behavior of a real-time program is described by the restrictions on the time of occurrence of events generated by an execution of the program.

(2) It should permit the exploitation of nondeterministic constructs to achieve predictable real-time behavior, i.e., the scheduling of nondeterministic constructs should be accessible to the programmer so that the functional and timing behavior of the program can be made as deterministic as is necessary to satisfy system requirements.

Since Ada is a general-purpose programming language, one may use a Turing universality argument to show that Ada can be used to simulate any desired real-time behavior. However, in addition to the efficiency penalty incurred by an extra level of interpretation that a simulation usually requires, such arguments miss the important issue: To be effective for real-time applications, the timing behavior of a real-time program should be readily deducible from the program text, preferably by a syntactic analysis. For otherwise, there is insufficient information to allocate resources to meet the specified timing constraints. More importantly, a real-time program whose timing behavior is difficult to analyze is especially hard to maintain, since it is all too easy to introduce subtle time-related errors when modifications are made. We shall discuss why and how Ada should be augmented to facilitate timing behavior analysis. Our approach exploits a system of annotations based on RTL (Real Time Logic, see [Jahanian & Mok 86]).

**Examining the Timing Behavior of an Ada Task**

Consider the following skeleton of an Ada task:

```ada
  task body T1 is
    declare
      use CALENDAR;
      NEXT_TIME : TIME := CLOCK+INTERVAL;
    begin
      loop
        delay NEXT_TIME-CLOCK;
        T2.SYNCHRONIZE; --synchronize with task T2
        CRITICAL_SECTION(); --execute a critical section
        NEXT_TIME := NEXT_TIME+INTERVAL;
      end loop;
    end T1
```

The above task, T1 is intended to be scheduled exactly once every INTERVAL time units. Every time T1 is run after the delay, it will synchronize with another task T2 before executing a critical section, and then compute the time at which it should be next
scheduled. In a real-time application, there may be a large number of periodic tasks like T1, and they may interact with one another through synchronization and resource sharing, i.e., there are precedence and mutual exclusion constraints that the run-time scheduler must enforce, in addition to meeting the periodic timing requirements. In order that resources can be allocated efficiently to meet these different types of constraints, the scheduler must be able to deduce the intended timing behavior of the tasks from the program text. This may be quite difficult with Ada tasks.

Suppose we want to write an analysis tool to mechanically determine the intended timing behavior of the task T1. Since the `delay` statement in Ada only puts a lower bound on when a task can be next scheduled, the semantics of Ada only permits an analysis tool to suggest that the task T1 should be scheduled at most once every INTERVAL time units. A correct implementation can legally execute T1 far less frequently. Furthermore, if the actual parameter to the delay statement evaluates to a negative number, the `delay` statement will have no effect, and the task T1 might be executed more than once within a time interval shorter than INTERVAL time units. Thus the semantics of `delay` does not guarantee that the task T1 will be executed no more than or no less than once every INTERVAL time units.

One might attempt to prescribe the intended periodic timing constraint by raising an exception whenever the actual parameter of `delay` is negative as follows.

```ada
task body T1 is
  declare
    use CALENDAR;
    NEXT_TIME : TIME := CLOCK+INTERVAL;
  begin
    loop
      TIME_LEFT := NEXT_TIME-CLOCK;
      if TIME_LEFT<0 then
        raise ERROR;
      else
        delay TIME_LEFT;
      end if;
      T2.SYNCHRONIZE; --synchronize with task T2
      CRITICAL_SECTION(); --execute a critical section
      NEXT_TIME := NEXT_TIME+INTERVAL;
    end loop;
  end T1
```

While the use of an exception does force the run-time system to signal an error if a period is missed, it does not in itself lead to the mechanical derivation of the intended timing constraint which requires task T1 to be executed exactly once every INTERVAL
time units. Specifically, a mechanical tool may not be able to ascribe the condition that causes the exception (TIME_LEFT<0) to the failure of this particular timing constraint. It may be the case that the programmer intends the variable TIME_LEFT to be always non-negative for some reason other than to enforce a timing constraint. For example, the TIME_LEFT variable may be used subsequently in an arithmetic calculation whose result is meaningless unless TIME_LEFT is non-negative. It is also possible that some other timing constraint which also makes use of the TIME_LEFT variable will fail if TIME_LEFT is negative. In other words, an analysis tool cannot simply assume that the exception is meant to signal the violation of the intended periodic timing constraint.

In general, the delay statement gives the programmer only limited control over the timing behavior of the program. In fact, unless the programmer can take into account the time it takes to evaluate the actual parameter of a delay statement, the actual duration of the delay is bound to be bigger than that specified by the computed value. To be fair to the designers of the Ada language, however, one should understand their reluctance to adopt a powerful construct whose semantics effectively dictates an upper bound on how soon certain computation must be completed. To do so would require the compiler to guarantee that the generated code will indeed meet the specified timing constraints, a technical challenge beyond the technology of the time.

We do not intend to give a formal proof here to show that it is impossible to write an analysis tool which can mechanically deduce the intended timing behavior of an Ada program from its text alone. It should not be hard to concoct an example to show that such a tool would have to be able to solve the halting problem which is of course undecidable. We hope to have convinced the reader, however, that it is non-trivial even to deduce simple timing constraints when they are expressed in a straightforward fashion.

It has been suggested by some authors that a pragma can be used to inform an Ada compiler of a program’s intended timing behavior. For example, a simple pragma may be used to indicate that a task is cyclic and must be executed at a specified rate. This is a good approach, but there are two difficulties with it:

1) A pragma is only a suggestion to the compiler. The actual timing behavior of the program is still determined by the delay statements in the program text. It is possible that the timing behavior as suggested by a pragma may conflict with the delay statements. Thus the compiler may end up having to check that no conflict exists as it takes advantage of the pragma information to meet the intended timing constraints. This consistency checking problem is just as hard as the previous one.

2) The timing behavior of a complex real-time program may be quite involved, and the official repertoire of pragmas may not be sufficient to express the wide spectrum of timing constraints. For example, in every period, the task T1 must also synchronize with task T2 and to gain access to a critical section before the end of the period runs out. These
requirements involve interactions among tasks and complicate the timing behavior of the program. In Ada, enforcement of both the synchronization and critical section constraints are usually carried out by using the rendezvous construct. It turns out that in order to meet all the timing constraints, it is very important to be able to distinguish the places in the program text where the rendezvous is used for synchronization from where the rendezvous is used for implementing critical sections. (See [MOK 84] for a discussion of the related scheduling theory.) The need to convey this type of information to the compiler calls for more pragmas. However, this tends to encourage proliferation of implementation-defined pragmas, thus hurting program portability. Portability is, of course, a major design objective of Ada.

A Uniform System for Timing Behavior Annotation

We now propose a system for annotating Ada programs that will help us to deal with the two difficulties discussed above. For ease of understanding, the reader may regard our approach as introducing a uniform system (a language really) to define timing-related pragmas. Thus instead of establishing an official repertoire of timing-related pragmas which every validated Ada compiler must support, we advocate a facility for interpreting timing-related annotations (formalized pragmas). We shall explain our system by annotating the tasking program which is our running example.

Program Text::

```ada
task body T1 is
  declare
    use CALENDAR;
    --NEXT_TIME : TIME := CLOCK+INTERVAL;
    --v- RUN_T1
  begin
    --delay NEXT_TIME-CLOCK;
    T2.SYNCHRONIZE; --synchronize with task T2
    --^- SYNC_WITH_T2
    CRITICAL_SECTION(); --execute a critical section
    --NEXT_TIME := NEXT_TIME+INTERVAL;
  end;
  --^- SUSPEND_T1
end T1

task body T2 is
  begin
    --statements of T2 before synchronization
    accept SYNCHRONIZE; --synchronize with task T1
    --^- SYNC_WITH_T1
    --statements of T2 after synchronization
end T2;
```
procedure CRITICAL_SECTION() is
begin
--v- ENTER_CS
--body of critical section
--ˆ- EXIT_CS
end CRITICAL_SECTION;

Timing Behavior Specification::
∀ i ( (i-1)*INTERVAL ≤ @(RUN_T1, i) ^
 @(SUSPEND_T1, i) ≤ i*INTERVAL
∀ i @ (SYNC_WITH_T2, i) = @(SYNC_WITH_T1, i)
∀ i @ (ENTER_CS, i+1) ≥ @(EXIT_CS, i)

(Note: For simplicity of explanation, we have left out the axioms for system initialization.)

The reader may notice that in task T1, we have commented out the delay statement and related time calculations. This is because the semantics of the delay construct in Ada does not lend itself to specifying stringent timing constraints which involve upper bounds on task suspension, as we have discussed earlier; the intended periodic timing constraint is being taken care of by the annotations. We have also included the source codes for task T2 and the critical section which is a procedure.

Our annotation system has two parts: event marker definitions and timing behavior specification. Event markers are stylized comments that are placed strategically in the program text. There are two syntactic forms:

--v- <event name>
--ˆ- <event name>

A event marker can be thought of as a time-keeper of computational activity. Every time a CPU executes a statement on one side of and right next to an event marker, the event marker records the time instant at which it happens. More precisely, if a CPU initiates the execution of a statement which is right below a event marker, say "--v- E" at time t, then we say that an instance of the event E occurs at time t. If a CPU completes the execution of a statement which is right above a event marker, say "--ˆ- E'" at time t, then we say that an instance of the event E’ occurs at time t. For example, if the first time the scheduler starts running the task T1 is at 8:10 a.m., then the first occurrence of the event RUN_T1 is at 8:10 a.m.

A timing behavior specification is a set of assertions that relate the time of occurrences of different events to one another. The notation @(<event name>,<index>) denotes an application of the function "@" (the occurrence function) to an event and an integer argument. The "@" function can be thought of as the master time-keeper who can
interrogate an event marker for the time at which some instance (specified by the <index> argument) of the event occurs. The three assertions in our running example should be interpreted as follows.

The first assertion states that the $i^{th}$ time at which the task $T1$ is run must be after the $i^{th}$ period has started, and $T1$ must be suspended, having completed its execution before the end of the $i^{th}$ period. (For ease of explanation, we assume that the system starts executing at time=0.)

The second assertion states that the $i^{th}$ time the task $T1$ completes the issue of an entry call to $T2$ must be at the same time at which $T2$ completes the acceptance of the same entry call.

The third assertion states that the $i+1^{th}$ time the critical section is entered by some task must not occur before the $i^{th}$ time the critical section is exited by some task.

The timing behavior specifications should be regarded as obligations that any correct implementation should honor. However, if a compiler (or an analysis tool working in collaboration with the compiler) determines that an implementation might miss a timing constraint, then some of the timing behavior assertions cannot be regarded as axioms. If this is acceptable, an exception can be raised to indicate which assertion has been violated. Assertions in the form of implications can be added to the program to catch the exceptions and force recovery actions to happen.

Our system of annotation helps to avoid the two difficulties with pragmas because unlike the latter, our annotation is based on formal logic (actually extensions to Presburger Arithmetic) and so does not permit ambiguity in expressing timing behavior. Our uniform syntax for describing timing properties does away with implementation-dependent pragmas which may take many special forms.

**Relation with RTL (Real Time Logic)**

The approach of annotating Ada programs described here in fact makes use of RTL (Real Time Logic) which is a formal system for reasoning about timing behavior. Details of RTL can be found in [Jahanian & Mok 86]. Briefly, RTL is invented to describe systems for which the absolute timing of events and not only their relative ordering is important. RTL reasons about occurrences of events. We distinguish between four classes of events: (1) External event, e.g., device interrupts, (2) Start event which marks the initiation of an action (an action in this case is the execution of an Ada statement), (3) Stop event which marks the completion of an action (executing an Ada statement), and (4) Transition event which marks a change in the system state. (Transition events have not been discussed in this paper. They can be used, however, to keep track of the results of expression evaluations, but we shall not develop this idea here.)
In RTL, time is captured by the occurrence function, denoted by the character "@", which assigns time values to event occurrences. The occurrence function is a mapping from the space (E,W) to W where E, W are respectively the set of events and non-negative integers.

**Definition:**

\[ @ (e,i) \equiv \text{Time of the } i^{\text{th}} \text{ occurrence of event } e; \text{ where } e \text{ is a start, stop, external or transition event, and } i \text{ is an integer constant/variable.} \]

The notion of the occurrence function is central to RTL. In particular, timing requirements imposed by the system specifications are restrictions on the "@" function. A system satisfies a timing property P if there is no mapping of event occurrences to time values which is consistent with the negation of the property P in conjunction with the system specification. RTL formulas are constructed using the equality/inequality predicates, universal and existential quantifiers, and the first order logic connectives. We are currently investigating practically efficient procedures for deciding interesting subsets of RTL.

**Conclusion**

In this paper, we have described some serious difficulties in deriving the timing behavior of real-time Ada programs. We advocate a formal system of annotation that should help real-time system programmers to prescribe and control the timing behavior of Ada programs. Because of space limitations, our description of this annotation system is necessarily brief. We are in fact building a set of tools under the SARTOR project which makes use of annotations (for the C language) to synthesize efficient code and customized schedulers from system specifications. A review of SARTOR (Software Automation for Real-Time OpeRations), a design environment for hard-real-time systems currently under development, can be found in [Mok 85b].
Bibliography

[Mok 84]

[Mok 85a]

[Mok 85b]

[JAH & MOK 86]