Testing Conformity to the Real-Time Annex

A. Burns and A.J. Wellings
Department of Computer Science,
University of York, York, UK.
email: {alan.burns, andy.wellings}@york.ac.uk

Abstract

In this paper we consider the many features of the Real-Time annex that are not as yet covered in ACATS. We identify a number of tests, but note that for the real-time programmer many of the crucial features of the annex cannot be validated by the type of tests found in ACATS.

1 Introduction

One of the reasons that Ada can justifiable claim to be a technology that can be applied to the highest levels of safety requirements is that the language has an internationally agreed definition and that Ada compilers can be assessed to demonstrate their conformity to this standard. This conformity is checked by a process known as ACATS [1] (Ada Conformity Assessment Test Suite). Release 4.0 of this suite has some 3,910 tests (Ada programs) that cover the whole of the language. In the main these tests check the semantics of specific language features. Unfortunately only 192 of these test are aimed at the annexes, with Annex D (the Real-Time annex) being addressed by only 53[1]. All of these focus on Ada 95 features. Although much has been added to Annex D in Ada 2005 and Ada 2012 (largely as a result of the deliberation of the IRTAW community), no further tests have been added. Arguably IRTAW should address this deficiency.

The aim of the features contained in the Real-Time Annex is to support necessary abstractions and functionality, and to improve the predictability and efficiency of the Ada language. Conforming to the Real-Time Annex is therefore more than just adherence to language rules, it requires the behaviour of the language features to meet the expectations of the real-time programmer. To help in this regard the Real-Time Annex, in the 95 version of the language, introduced metrics to assist

[1] A listing of the main ones is contained in an Appendix.

users to understand the performance to be expected from the run-time. It is not clear, however, that these metrics were adequate or complete.

To illustrate the difficulty with defining what it means to comply with the spirit as well as the wording of the language's definition consider two features: the abort statement and timing events.

It is clear what is expected of the abort statement. It is an extreme measure that will allow a wayward task to be stopped. Such a task may be in an infinite loop that must be interrupted. To give some implementation freedom (especially for multiprocessor platforms), an earlier version of the language rules for abort allowed some flexibility. But as a result, tasks could execute for an unbounded time before being aborted – thereby invalidating the whole purpose of the language feature. This problem was put right in later versions of the language (at least for single processor platforms) and is checked with ACATS test CXD6001 (see Appendix).

Timing events are a useful abstraction that were introduced at a previous IRTAW. They allow short code segments to be executed directly by the clock interrupt handler. As a result they have minimum temporal jitter and have much less overhead than would be the case if a full task was used that 'delayed until' the allotted time. Unfortunately the easiest way to implement timing events (especially as they were an additional new feature) is to map them to a task. The programmer declares a timing event, the compiler delivers a task. Again the whole point of using a timing event is lost.

A final difficulty to note with the real-time features of the language is that the required behaviour of a test program may be dependent on the hardware platform. Single processor, multiprocessor and multi-core chips may require different behaviour from a dispatching policy (for example), whilst sequential parts of the language will be invariant to such matters.

In this paper we give an overview of the features found in the Real-Time Annex and attempt to define the nature of the conformance tests that should be developed within the ACATS. We do not go as far as defining detailed tests, although the workshop could attempt to define them.

There are six different classes of ACATS tests, reflecting different testing requirements of language conformance. Class A checks legality; tests should compile without error. Class B tests checks illegal constructs; tests should not compile. Class C is in many ways the most important; here the tests must compile and execute correctly. Class D is concerned with exact arithmetic. Tests in Class E may require inspection to verify compliance as they can give rise to the outcome, 'tentatively passed'. Finally Class L is concerned with library unit dependencies. For the annexes the tests fall into the B, C and L classes.

Tests in the A, C, D and E classes generally use the following format:

```
with Report;
procedure Testname is
   <declarations>
begin
   Report.Test ("Testname", "Description␣...");
   ...
   <test situation yielding result>
   if Post_Condition /= Correct_Value then
      Report.Failed ("Reason");
   end if;
   ...
   Report.Result;
end Testname;
```

2 Real-Time Annex Features

The post Ada 95 features that require ACATS tests to be developed are the new dispatching policies, the Ravenscar profile, dynamic PO (Protected Object) priorities, execution timers, timing events and the support for multiprocessor execution. We shall look at each of these in turn. Potential tests are highlighted as bullet points. Tests should ideally be automatically verified (i.e. fall into Class C), but for some this may not be possible and inspections will be necessary (i.e. fall into Class E). Reporting must, of course, retain the order of items sent for output, as correct order maybe required to pass a test.

2.1 Dispatching policies

Fixed priority scheduling was in Ada 95 and is covered to some degree in the current ACATS. What is needed are tests to cover the new dispatching policies. Initially these should only be executed on a single processor to ensure that the correct ordering is observed.

- Non-preemptive dispatching – 3 tasks (say) with different priorities; their releases are staggered to ensure high priority tasks are released while lower priority ones are executing. These test could follow the format of existing tests for FIFO within priority dispatching (i.e. CXD2*** tests – see the Appendix).
- Yield – with non-preemptive dispatching, allow some tasks to have the same priority, tests for Yield_To_Higher and Yield_To_Same_Or_Higher (Yield).

Round Robin scheduling is difficult to test accurately, even identical tasks may not progress at the same rate when given the same execution time (due to cache effects etc). Hence we recommend the use of execution time clocks (if available).

- Round-Robin dispatching – 3 identical busy tasks of the same priority, run the program for an extended duration, and then compare their execution-times.
- Round-Robin dispatching with inherited priority – one task to run for an extended period in a PO (i.e. with an inherited priority), this task should be seen to have more than its fair share of execution time.
- Round-Robin set-quantum – again hard to check correctness here, perhaps compute (elapsed time - total execution time of all tasks) and see that this value goes up for smaller values of the quantum (i.e. observe that the over-heads increase for smaller quantum).
- EDF dispatching (1) – 3 busy tasks with different deadlines but released at the same epoch, show that they finish in the right order.
- EDF dispatching (2) – 3 busy tasks with different deadlines, but staggered releases, show that they finish in the right order (i.e. that preemptions work correctly).
- Set and Get deadline – a simple test to check that changes are effected (i.e. Set followed by Get yields the same value).
- Set deadline – one task (A) reduces the deadline of another task (B); initially B has the longer deadline, after the call of Set, B then has shortest deadline; test should show that B finishes before A.
- SRP (Stack Resource Protocol) – if SRP is to be retained, then some quite complicated tests will be needed to check ordering.
- DFP (Deadline Floor Protocol) – if DFP is adopted then a test is needed to show that the task's deadline is reduced within a PO.
- Hierarchical scheduling – run a fixed priority, EDF and a Round-Robin scheme together as a three policy scheme (other combinations could also be tested).

2.2 Dynamic Priorities and Ceilings

Dynamic priorities were part of Ada 95. There is a metric to provide the execution time of a call to Set_Priority. There is also a metric to give a measure of the time it takes to change the ceiling priority of a PO.

- Dynamic ceilings – a test in which a task calls a PO three times, first a correct call, second a call to lower the ceiling, third to call again (now with too high a priority); exception should be raised on the final call (only). These test could be adapted from the existing test (CXD3001) that checks that PO ceiling priorities are correctly assigned.

2.3 Monotonic time

There are a rich set of features to support clocks and time abstractions. It is, of course, impossible to check the quality of a clock without the use of another clock. It would be possible to compare Calendar.Clock and Real_Time.Clock, but ideally they should be sourced from the same time base (this is Implementation Advice). Tests exist for these clocks.

2.4 Synchronous Barriers

- Wait on synchronous barrier – 5 tasks call a barrier with a threshold of 3, as the third task arrives all three tasks should be released (with only one having the notification flag set to true); the other two tasks should remain blocked.

2.5 Tasking restrictions and the Ravenscar Profile

A collection of relatively simple programs can check that restrictions are enforced (Class L tests). A key aspect of the Ravenscar profile is that it allows a much simpler, smaller and more efficient run-time to be used. It is not clear how a metric could address this issue.

2.6 Execution Time

Execution time is another time abstraction, tests are needed for its methods and its run-time behaviour.

- Execution time operations – a single task could undertake a wide range of operations that check the implementation of the time operations, these could mimic those for Calendar.Clock and Real_Time.Clock.

- Execution time accuracy – a long running program (on a target single processor) with 1, 4 and 16 tasks (i.e. three programs) but no interrupts other than for the system clock; compare total execution time and elapsed time; execution time (including that of the system clock) should always be less, but within 1% of elapsed time.

- On a processor board with a source of interrupts, the above test could be adapted to include the measurement of time spend in the interrupt handler.

2.7 Timers

This feature will probably need a number of tests to cover all its operations. Also metrics are needed to cover some of the timing issues (such as time to cancel a handler).

- Timer operations – a single task could run through a series of set, read and cancel handlers to check this aspect of the implementation; test should include an immediate read of time remaining to confirm it is close to the time set.

- Timer behaviour – a program of three tasks all with timers set, each uses an execution time clock to check that the timer has fired at the right time (approximately).

2.8 Group Budgets

Again this feature will probably need a number of tests to cover all its operations.

- Membership operations – a small number of tasks could run through a series of Add, Remove, Is Member and Members operations.

- Budget setting – a single task could run through a series of Add, Replenish, Has_Expired, Budget_Remaining etc. to check this aspect of the implementation; the test should include an immediate read of the budget remaining to confirm that it is close to the budget set.

- Group budget behaviour – a program of 12 tasks in three groups, they join and are replenished twice (by the handler); each task uses its execution time clock to check that the budgets have expired at the right time (approximately); test should use delay statements or yields to ensure that all the tasks execute.

2.9 Timing events

- Timing event operations – a single task could run through a series of set, read and cancel handlers to check this aspect of the implementation; the test should include a read of the time of the event to check that it is the same as the time set.

- Timing event behaviour(1) – 2 tasks set 5 timing events (each); some are for the same time; handlers should be executed in the correct order at the correct time.

- Timing event behaviour(2) – a single task sets up a Timing Event in the future and then terminates (before the event has fired). The program should not terminate until the event handler has executed.

2.10 Multiprocessor issues

Once the platform moves beyond a single processor then there are many issues to consider. Firstly many of the above tests need to be rerun (and reinterpreted) on say a 4 core platform. Then the specific features defined in D16 need to be addressed. It is hard to check that a task is running on the right CPU unless the underlying RTOS supports some form of monitoring. And if tasks can migrate then this is even harder. There are however some test that can be defined. Assume all test are run on a 4-core platform.

- Dispatching domain operations – a single task could allocate 9 tasks to three domains and check the Set, Get etc routines.

- Partitioned execution – with a single dispatching domain of 4 CPUs, allocate two busy tasks to the same CPU (only). Assign different priorities and show that the lower priority one is starved. Execution time budgets could be used.

- Parallel execution - with a single dispatching domain of 4 CPUs, allocate four busy tasks to all CPUs and show that they are all executing in parallel. Again execution time budgets could be used.

- Migrating execution - with a single dispatching domain of 4 CPUs, allocate four busy tasks, with priorities 8, 7, 6 and 5 to all CPUs; then release a higher priority busy task that moves around the 4 CPUs by calling Set CPU; show that the lower priority task never runs.

- Dispatching Domain execution - with two dispatching domains, repeat the above tests, this time in parallel so that one domain is partitioned and the other involves task migration.

3 Conclusion

In this paper we hope to initiate a discussion as to the best way to test conformance with the many features defined in the Real-Time Annex. Issues relating to the operators defined with the abstractions (e.g. Set and Get a Timing Event) are relatively easy to address. They would naturally fit into ACATS and lead to automatic validation (i.e Class C tests). Those that test the run-time behaviour of the feature are harder to define. There may be more than one valid order of execution, and for some the 'passing criteria' may be hard to define precisely. Many of these tests may be better placed in Class E and allow for the judgement 'tentatively passed'.

For real-time systems the temporal data, as identified in metrics, is crucial. Should a set of tests be defined that validate the information provided by the implementor for these metrics?

References

[1] ACATS – http://www.ada-auth.org/acats-files/4.0/docs/UG-1.HTM

Appendix - ACATS Tests for Annex D

Within the ACATS there are two Class B test (concerned with priority pragma), forty-seven Class C tests and a number of Class L tests that are concerned with pragma restrictions). Here we outline the scope of the Class C tests.

- CXD1001 – CXD1008: Concern priorities and the correct behaviour of tasks with different priorities; CXD1003 considers priority during a rendezvous; CXD1004 deals with priority during activation; CXD1008 addresses floating point operations and exception raising (to check that the exception is raised in the correct task).
- CXD2001 – CXD2008: Concern FIFO priority based dispatching, making sure tasks are added to the correct end of the ready queue.
- CXD3001 – CXD3003: Concern PO locking policies, including exception being raised if calling task has priority greater than ceiling.
- CXD4001 – CXD4010: Concern priority queues, multiple barriers (in POs) and accept alternatives in select statements.
- CXD5001: Concerns Get_Priority from a terminated tasks (i.e Program_Error raised).
- CXD6001 – CXD6003: Concern immediacy of abort (and ATC in CXD6002).
- CXD8001 – CXD8003: Concern Ada.Real_Time and delay statements.
- CXD9001: Concerns delay of zero in a timed entry call.
- CXDA001 – CXDA004: Concern synchronous task control.
- CXDB001 – CXDB004: Concern asynchronous task control.