Why the Expressive Power of Programming Languages such as Ada is needed for future Cyber Physical Systems

Alan Burns

Department of Computer Science, University of York, UK.

Abstract. If Cyber Physical Systems (CPS) are to be built with efficient resource utilisation it is imperative that they exploit the wealth of scheduling theory available. Many forms of real-time scheduling, and its associated analysis, are applicable to CPS, but it is not clear how the system developer/programmer can gain access to this theory when real CPS are being constructed. This short paper gives the background to the associated presentation where the facilities available in the Ada programming language are highlighted and reviewed. The aim of the presentation is to show that Ada provides most of the programming abstractions needed to deliver future CPS.

1 Introduction

The design and implementation of reliable and cost-effective Cyber-Physical Systems (CPS) incorporates many issues in both the hardware and software aspects of the system. For reliability (and for many CPS, safety), timing constraints such as deadlines must be satisfied, and in many applications evidence for such compliance must be provided (perhaps in a safety-case to be used during certification). For cost-effectiveness, resources must be used effectively and sparingly. Resources in this context being platform hardware and run-time energy consumption.

The branch of Computer Science that deals with resource usage in this context is real-time scheduling. Scheduling protocols promote efficient (and at times optimal) resource utilisation. And scheduling analysis provides the means of verifying that, even in the worst-case, deadlines will be met.

Over the last 50 years a considerable body of knowledge and volume of results in the field of real-time scheduling have been produced. These results can, potentially, have a significant impact on how further CPS are developed. But to realise this potential it must be possible for software developers to exploit this theory; they must be able to apply the scheduling protocols and to subject their software to the relevant forms of scheduling analysis.

The means of exploiting scheduling theory is via the programming languages (PLs) and operating systems (OSs) available to the developer. PLs provide abstractions, such as tasks with priorities and priority based processor scheduling, whilst OSs provide interfaces that can support, for example, threads, priorities and priority-based dispatching.

In this paper and talk I want to give a somewhat high level, and necessarily brief, review of the many scheduling results that may be of utility to CPS development. I then want to see to what extent these results are available to developers via OSs and programming languages. I shall argue that for all but the the most basic level of support, OSs (and certainly OS Standards) lack the expressive power required. However, programming language abstractions do exist that give access to many (but of course not all) forms of scheduling protocols and analysis. And the programming language that implements/supports most of these abstractions is Ada. Therefore the expressive power of the Ada programming language will form the main focus of this keynote.

2 Scheduling Results

Here some of the major results from scheduling theory, which could be employed in CPS development are outlined. Requirements are linked to applicable scheduling protocols. Of course not all requirements are needed for all CPS, but each one has the potential to be useful. We looks at this work under four headings, uniprocessor systems, multiprocessor systems, mixed criticality systems and then a *catch all* section.

2.1 Uniprocessor Theories

We start with some very basic requirements

- *Interactions with the parallel world* – requires concurrency (tasks, threads, processes etc).
- *Safe Sharing between distinct software components* – synchronisation controls (semaphores, mutexes, monitors etc).
- *Synchronisation with external real-time* – clock abstractions and delay primitives.
- *Synchronisation with external events* – interrupt handling.

Concurrency allows tasks to execute non-deterministically. Basic scheduling allows this freedom to be constrained so that the possibility of meeting timing requirements is optimised (and verified if the appropriate analysis is applied):

- *Predicable task ordering* – (static) priority attributes for tasks, priority ceilings for monitors.
- *Deadline aware task execution* – deadline attributes for tasks, protocols for effective sharing.
- *Deterministic execution order* – Non-preemptive scheduling (with static priorities).

Building upon these basic schemes there are a wealth of protocols that aim to improve resource utilisation, for example:

- Deferred preemption [18].
- Dual priorities [13].
- Dynamic priorities (which can be used to program a wide variety of protocols).

or support more general computational models:

- Logical Execution Time (LET) [21, 20, 27].
- Anytime or imprecise algorithms (where following the production of an adequate result, further processing will improve the result up to the point when the result must be output) citeimprecise:computation,drteeb,zilberstein1996optimal.

- Dynamic periods and deadlines (elastic task model) [16].
- N in M constraints (only N out of every M jobs need meet their deadlines) [6].
- Multiframe Tasks (tasks execute through a series of frames with different resource requirements) [28].
- Generalised Task Model (where tasks are related by DAG models) [5].
- Open Systems with admission control (dynamic task creation) [12, 24].

Many CPS have to demonstrate resilience as well as functional correctness; for this, fault tolerance behaviour needs to be supported. This takes the form of error recognition, firewall protection and various forms of adaptive resource management. For example,

- Deadline miss detection
- Budget monitoring
- Budget overrun detection
- Budget enforcement
- Watchdog timers
- Aborting rogue computation
- Budget management per task
- Budget management per group of tasks
- Early task termination identification

Aborting rogue computation is needed at the task/thread level and at the functional code level. It must be possible to undertake this abandonment without leaving shared data in an undefined state.

2.2 Multiprocessor Theories

Once the execution platform changes from uniprocessor to one with multiple processors (or cores) then other requirements have to be addressed by the scheduling theory. The basic features that must be supported are:

- Partitioned scheduling – managing the static assignment of tasks/threads to processors/cores.
- Global scheduling – managing the run-time migration of tasks/threads to follow the rules of the scheduling protocol.
- Semi-partitioned scheduling – managing the controlled migration of individual tasks/threads at run-time [22, 9, 10].
- Sharing – controlling the sharing of resources between potentially parallel executing tasks/threads (this is a major open problem, in that effective general purpose protocols are not yet available).

On top of these basic protocols there are more advanced schemes that again deliver more efficient execution or support more general models of computation:

- TkC, and DkC (global schemes with priority-based scheduling then non-preemptive) [2, 19].
- tasklets to model parallelism within a task/thread [25].
- barriers to efficiently synchronise tasks/threads on multiprocessor platforms.

Finally there is the need to support heterogeneous as well as homogeneous hardware platforms.

2.3 Mixed Criticality Theories

Mixed criticality system introduce new scheduling protocols that aim to increase the resilience of such systems but keep their resource usage as low as possible. Most of the proposed protocols involve forms of change management. For example:

- task/thread parameter modification (extend period and deadlines),
- suspending tasks/threads,
- modifying scheduling attributes: priorities and deadlines,
- resume tasks/threads.

Also important is the means of supporting partitioning; in particular the allocation of processor time – the overrun of one subsystem must not impact on the resources available to another subsystem. This is especially true if the subsystems are of different levels of criticality. However, the complete separation of subsystems can lead to the over provisioning of resources as the certification of safety-critical software requires very conservative resource-usage estimates. Mixed criticality research has focused on scheduling protocols that provide managed sharing as well as adequate separation [8].

2.4 Others Requirements and Scheduling Features

Here we note some other issues:

- Control of when tasks/threads that preform I/O execute (e.g. minimising input and output jitter).
- Control of memory used by tasks/threads.
- Control of power used by tasks/threads.
- Control over the speed of variable rate processors.
- Control over placement on FPGA type hardware

3 Required Abstractions and/or Interfaces

To satisfy the above extensive list of requirements and protocols, one can either provide a (large) set of high-level abstractions/models, or aim to support primitives from which these abstractions can be programmed. For example, the notion of a periodic task could be provided in a model-based development scheme. Such a concept would have a defined period and deadline; but to give an elastic task it will be necessary to allow period and deadline to change – will the abstraction allow this? Also when executed by a priority based dispatcher how are the priority changes managed? Alternatively, the more general notion of a task or event handler, supported by clocks, a delay statement and a dynamic priority routine will allow any form of time-triggered computation to be programmed. In general a high-level abstraction is easier to use if it is exactly what is required; but lacks the expressive power to allow variants to be derived. Lower-level abstractions, however, present the programmer with more challenges.

Another problem with high level abstractions come from composition. (e.g. how to obtain a periodic tasks with deadline overrun protection, budget enforcement and an N in M execution requirement). Will the three or four high level models work together? Yes programming the required behaviour from lower level abstractions is more effort (and hence is potentially more error prone). But it does allow the actual necessary semantics to be delivered as long as the low-level abstractions are adequate and themselves provide the necessary expressive power.

The interfaces provided by a modern Real-Time operating system, such as one based on POSIX or Linux, give a good level of support for the protocols defined above. Threads have priorities, there are mutexes and priority ceiling protocols, priorities can by modified, budgets can be monitored, signals sent from one thread to another, and mappings to processors can be managed at run-time via affinities. But not all the requirements can be satisfied by OS standards. And composition via library APIs is error-prone and lacks the usability one would hope to have in a programming environment. Programming languages can however embed the protocols within the syntax as well as provide standard library routines, and this allows programmers to directly address the needs of CPS. But of course the programming language must be up to this challenge. In the following section I will look at what Ada provides and argue that it does indeed have (mostly) the required expressive power. Ada is chosen as it provides more low-level abstractions than any other programming language.

4 Ada's Provisions

For those very familiar with Ada they will find nothing new in this brief overview, but for others I hope to at least remind you of what is now supported in the full Ada language. The Ravenscar profile is an important technology for simple real-time scheduling. But for the schemes likely to be needed in future CPS the expressive power of the full language is required. Again I will do this in the form of lists. So first basic concurrency, Ada supports

- Calendar and real-time clocks.
- Static and dynamic creation of tasks.
- Delay mechanisms.
- Priority assignment.
- Protected objects with requeue to give controlled sharing.
- Dynamic task priorities and dynamic priority ceilings.

The requeue facility allows the full expressive power of a monitor to be provided without the need for very low level condition variables.

By supporting dynamic priorities and flexible delay mechanisms many different forms of behaviour can be programmed (such as an elastic task that changes its period).

To support scheduling protocols directly Ada supports:

- Priority based dispatching with priority ceiling protocol.
- EDF scheduling with the Stack Resource Protcol (SRP) [4, 3] (and possibly in the future the Deadline Floor Protocol, DFP [11, 1]).
- Round Robin and non-preemptive dispatching.
- Hierarchical scheduling (for example, combined priority-based and EDF).

– Primitives to allow tasks to suspend themselves and other tasks.
– Timing events – code that executes at a specified time (can be used to control input and output jitter).
– Group budget monitoring and control that allows standard execution time servers such as the Periodic Server, Sporadic Server and Deferrable Server to be constructed [26, 17, 7, 14, 15].

To support more resilient software Ada supports:

– Budget clocks that monitor task execution time, and can signal when specified levels of usage have been reached.
– Task aborting, and the ability to abandon computation at the sub-task level (ATC – select then abort))
– Timing events – that are only execute in error conditions, i.e. programmed watchdog timers.
– Signaling when a task terminates (useful when the task should not!).

The ATC (Asynchronous Task Control) facility is not only of use in error handling for it also allows anytime algorithms to be easily programmed – set the ATC at a deadline, loop through some code to improve quality of computation, storing result in a protected object, abandon when deadline is reacted.

Timing events are another language feature with multiple usages. They can be used positively to control when I/O operations occur, but they can be 'not used' for watchdog timers. Here the 'alarm' is programmed to occur at some future event. The 'I'm alive' signal simple pushes the alarm time further into the future.

To support multiprocessor execution, Ada provides:

– Affinities that can control where a task executes; a task can be restricted to just one CPU, a groups of CPUs or be allowed to execute on any CPU.
– Dynamic affinities to allow semi-partitioned schemes to be programmed.

Other features that are potential important in CPS:

– Use of memory pools to control this important resource.

Having introduced a wide range of important features currently supported by Ada it is only fair to consider some that are missing:

– Support for parallel execution within a task – a plan for including the notion of tasklet into the language is currently under consideration [23].
– Support for energy aware programming – API to whatever is supported by the underlying hardware/run-time is the only current approach available – I would like to execute a loop within a bound determined by energy available.
– Support for an effective synchronisation scheme for multiprocessor execution – many schemes have been proposed in the literature but there is not yet consensus on which Ada can build.

To illustrate the expressive power of Ada a couple of illustrative examples will be included in the presentation.

5 Conclusions

A brief introduction to the scheduling requirements for future Cyber Physical Systems has been given. The list of requirements has been mapped against the provisions of the Ada programming language. In general, Ada provides a rich set of facilities from which higher level abstractions can be built.

The alternative to the use of a facilitating programming language is to rely upon the provisions of the operating systems upon which the software executes. Unfortunately the APIs provided by real-time operating systems (RTOSs) are not flexible enough to deal with the emerging approaches to scheduling resources that are being considered for cost-effective future systems.

Clearly Ada is not a static finished language, it has proved itself to be adaptable and to be able to embrace new ideas and programming styles. New challenges will continue to emerge, such as support for fine grain parallelism, and Ada must be as adaptable going forward as it has been in the past.

References

1. M. Aldea, A. Burns, M. Gutirrez, and M. González Harbour. Incorporating the deadline floor protocol in Ada. *ACM SIGAda Ada Letters – Proc. of IRTAW 16*, XXXIII(2):49–58, 2013.
2. B. Andersson and J. Jonsson. Fixed-priority preemptive multiprocessor scheduling: to partition or not to partition. In *Proc. of the International Conference on Real-Time Computing Systems and Applications*, 2000.
3. T.P. Baker. A stack-based resource allocation policy for realtime processes. In *Proc. IEEE Real-Time Systems Symposium (RTSS)*, pages 191–200, 1990.
4. T.P. Baker. Stack-based scheduling of realtime processes. *Journal of Real-Time Systems*, 3(1), March 1991.
5. S.K. Baruah. A general model for recurring real-time tasks. In *rtss*, page 114. IEEE, 1998.
6. G. Bernat and A. Burns. Combining (n m)-hard deadlines with dual priority scheduling. In *Proc. 18th IEEE Real-Time Systems Symposium*, pages 46–57, 1997.
7. G. Bernat and A. Burns. New results on fixed priority aperiodic servers. In *Proc. 20th IEEE Real-Time Systems Symposium*, pages 68–78, 1999.
8. A. Burns and R.I. Davis. Mixed criticality systems: A review. Technical Report MCC-1(e), available at http://www-users.cs.york.ac.uk/burns/review.pdf, Department of Computer Science, University of York, 2015.
9. A. Burns, R.I. Davis, P. Wang, and F. Zhang. Partitioned EDF scheduling for multiprocessors using a C=D scheme. In *Proc. of 18th International Conference on Real-Time and Network Systems (RTNS)*, pages 169–178, 2010.
10. A. Burns, R.I. Davis, P. Wang, and F. Zhang. Partitioned EDF scheduling for multiprocessors using a C=D task splitting scheme. *Real-Time Systems Journal*, 48(1):3–33, 2012.
11. A. Burns, M. Gutierrez, M. Aldea, and M. González Harbour. A deadline-floor inheritance protocol for EDF scheduled embedded real-time systems with resource sharing. *IEEE Transactions on Computers*, 64(5):1241–1253, 2015.
12. A. Burns, D. Prasad, A. Bondavalli, F. Di Giandomenico, K. Ramamritham, J. Stankovic, and L. Stringini. The meaning and role of value in scheduling flexible real-time systems. *Journal of Systems Architecture*, 46:305–325, 2000.
13. A. Burns and A.J. Wellings. Dual priority scheduling in Ada 95 and real-time POSIX. In *Proc. of the 21st IFAC/IFIP Workshop on Real-Time Programming, WRTP'96*, pages 45–50, 1996.

14. A. Burns and A.J. Wellings. Programming execution-time servers in Ada 2005. In *Proc. of IEEE Real-Time Systems Symposium (RTSS)*, pages 47–56, 2006.
15. A. Burns and A.J. Wellings. Programming execution-time servers in ada 2005. In *Proc. of the 27th IEEE Real-Time Systems Symposium*, pages 47–56, 2006.
16. G. Buttazzo, G. Lipari, and L. Abeni. Elastic task model for adaptive rate control. In *IEEE Real-Time Systems Symposium*, pages 286–295, 1998.
17. M. Caccamo and L. Sha. Aperiodic servers with resource constraints. In *Proc. of the IEEE Real-Time Systems Symposium*, December 2001.
18. R.I. Davis and M. Bertogna. Optimal fixed priority scheduling with deferred pre-emption. In *Proc. IEEE Real-Time Systems Symposium*, pages 39–50, 2012.
19. R.I. Davis and A. Burns. Priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems. In *Proc. of IEEE Real-Time Systems Symposium (RTSS)*, pages 398–409, 2009.
20. H. Hagenauer, N. Martinek, and W. Pohlmann. Ada meets giotto. In *Reliable Software Technologies-Ada-Europe 2004*, pages 237–248. Springer, 2004.
21. T.A. Henzinger, B. Horowitz, Kirsch, and C. Meyer. Giotto: A time-triggered language for embedded programming. In *Embedded Software*, pages 166–184. Springer, 2001.
22. S. Kato and N. Yamasaki. Semi-partitioned fixed-priority scheduling on multiprocessors. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 23–32, 2009.
23. S. Michell, B. Moore, and L.M. Pinho. Tasklettes ? a fine grained parallelism for ada on multicores. In *Proc. Reliable Software Technologies - Ada Europe*, 2013.
24. D. Prasad, A. Burns, and M. Atkin. The measurement and usage of utility in adaptive real-time systems. *Journal of Real-Time Systems*, 25(2/3):277–296, 2003.
25. A. Saifullah, J. Li, K. Agrawal, C. Lu, and C. Gill. Multi-core real-time scheduling for generalized parallel task models. *Real-Time Systems*, 49(4):404–435, 2013.
26. B. Sprunt, L. Sha, and J. P. Lehoczky. Aperiodic task scheduling for hard real-time systems. *Journal of Real-Time Systems*, 1:27–69, 1989.
27. A. Wellings and A. Burns. *Proc. 15th Ada-Europe International Conference on Reliable Software Technologies*, chapter The Evolution of Real-Time Programming Revisited: Programming the Giotto Model in Ada 2005, pages 196–207. Springer Berlin Heidelberg, 2010.
28. A. Zuhily and A. Burns. Exact scheduling analysis of non-accumulatively monotonic multi-frame tasks. *Real-Time Systems Journal*, 43:119–146, 2009.