

A collection of historical artifacts is arranged on a light-colored surface. In the top left, a portion of a wooden chessboard with a checkered pattern and several chess pieces is visible. Below it, a blue ribbon with a circular emblem is attached to a silver star-shaped medal. To the right, a red ribbon with a similar emblem is attached to another silver star-shaped medal. In the bottom left, a circular compass with a white face and black markings is visible. A pair of gold-rimmed glasses with thin temples lies across the center of the image. The text 'Programming Real-Time Systems' is overlaid on the right side of the image in a black, serif font.

Programming Real-Time Systems

Alan Burns

University of York, UK



Contrast

- ◆ Lots of exciting notions/ideas/concepts/techniques etc etc in research domain
- ◆ Neither real-time programming languages (RTLs) or RTOSs support these features

How do programmers gain access to the right level of expressive power?

How do researchers do experiments rather than just simulations



Contents of Talk

- ◆ Assessment of programming languages
- ◆ Mechanisms or abstraction
- ◆ What do we need to support
- ◆ Mechanisms and meta-mechanisms
- ◆ Actual language developments



How to evaluate RTL and RTOS

- ◆ Expressive power
 - What can you do with the features
- ◆ Ease of use
 - How easy is it to use the features together

Level of support

- ◆ Abstractions
- ◆ Policies
- ◆ Mechanisms
- ◆ No support – ad hoc





Level of support

- ◆ Abstractions – usually too high
- ◆ Policies – some mature enough to support
- ◆ Mechanisms – lots more needed
- ◆ No support



Level of Abstraction

- ◆ Are there high level abstractions/policies that need to be identified and supported

No – at this time

I believe we are still in the position of needing mechanism, and the means of checking orthogonality (combined use)



Example

- ◆ A periodic thread
 - Good abstraction for many applications
 - What happens is abstraction not quite right
 - Change period for example
- ◆ Provide a mechanism for programming a periodic thread
 - delay statement
- ◆ Static analysis harder without the abstraction
 - Templates are perhaps a useful compromise



Languages or APIs

- ◆ Languages allow richer semantic models to be developed
 - Better understanding of feature interaction
 - Better static checking available



Current Expressive Power

- ◆ Concurrency
 - dynamic creation etc
 - delay a thread
- ◆ Shared objects (monitors)
- ◆ Fixed Priority Scheduling for single CPU
 - dynamic priorities
- ◆ Priority Ceiling Protocol (immediate variety) on single CPU
- ◆ Limited exception handling, not temporal



What do we need?

- ◆ Control over time
- ◆ Control over resource usage
- ◆ Control over (temporal) faults
- ◆ Ignore non-temporal issues

What do we need?

- ◆ Control over time
 - Clocks (with knowledge of precision)
 - Deadlines
 - System goes fast enough
 - Delays
 - System does not go too fast
 - synchronous and asynchronous interactions
- ◆ Model checking a program against its requirements is possible with these features





What do we need?

- ◆ Control over resource usage
 - Flexibly use of single CPU
 - Multiple CPUs
 - Novel architectures - SoC
 - Memory and Busses
 - Other application-specific resources
 - Power
 - Networks
- ◆ Protection from misuse
- ◆ Recognition of timing faults



CPU Scheduling

- ◆ Events and threads
- ◆ EDF (urgency) and preemption level control over shared objects
(policy/mechanism)
- ◆ User defined scheduling
- ◆ Hierarchical scheduling
- ◆ Integrated scheduling
 - eg. CPU and memory bus



Budgeting

- ◆ Monitoring the resource use of a thread
 - On CPU, or other resource
 - Event generated when thread has used “enough”
- ◆ Allocating budget to a group of threads/events
 - Manage budget exhaustion, replenishment
 - Manage budget sharing



To increase expressive power

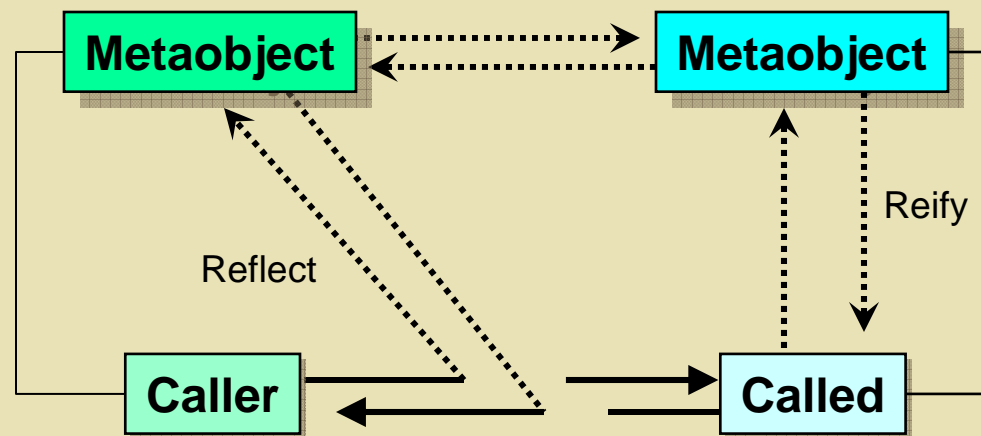
- ◆ Add new mechanisms to RT languages
- ◆ Add new mechanisms to RTOS and Middleware APIs
- ◆ Allow adaptability (meta-mechanisms)
 - Reflective programming
 - Aspect-oriented programming



Reflection

- ◆ Structural
 - Break into the computation model
 - Reflect on a method call for example
 - Can be delivered by compile time transformations of code

Structural Reflection





Behavioural Reflection

- ◆ Gain access to key run-time events
 - context switch, beginning and/or end
 - job arrival
 - first execution of a job
 - resource use (other than CPU)
- ◆ Behavioural reflection is not new, but still little experience in real-time languages
 - Meta-mechanism could be event firing
 - Run-time overheads an issue



Mainstream RT Languages

- ◆ The Real-Time Specification for Java
 - Events and threads are schedulable objects
 - Mechanisms for control use of memory
 - Admissions control (unproved)
 - Thread groups
- ◆ Ada 95 and Ada 05
 - A number of new mechanisms



Ada (05)

- ◆ Asynchronous transfer of control (Ada 95)
 - fault recognition
 - anytime algorithms
- ◆ Timing Events
 - Code that is executed at a fixed point in time
 - Can be reset for repeated events
 - Can be reset before firing for a watchdog



Ada (05)

- ◆ Execution time clocks for each thread (task)
 - Event fired when $T.\text{clock} = X$
- ◆ Budgets for groups of threads
 - Event fired when budget is exhausted
 - Replenishment under programmer control
 - Can program Servers or various forms

Ada (05)

- ◆ Dynamic priorities for threads (Ada 95)
- ◆ Dynamic ceilings for shared objects (protected objects in Ada)





Ada (05)

- ◆ Support for EDF dispatching
- ◆ Support for preemption level locking
 - Can be used with other measures of urgency
- ◆ EDF and Fixed Priority (and Round Robin) can be used together
- ◆ Non-preemptive dispatching

Ada (05)

- ◆ Worth having a look at!!





Conclusions

- ◆ To give control over time;
 - Time and CPU resource must be first class notions with support mechanisms for
 - Deadlines
 - Budgeting
 - Events
 - Asynchronous interactions



Conclusions

- ◆ To give control over other resources
 - Some form of reflective power
 - On context switching
 - On resource usage
 - On power consumption
 - On ...



Conclusions

- ◆ The number of features and the application domain requirements are too extensive to be left to APIs
- ◆ The development of Real-Time Programming Languages should again be the subject of active research