Protected Ceiling Changes

Jorge Real*, Alfons Crespo
Dept. of Computer Engineering (DISCA)
Technical University of Valencia
Camino de Vera, s/n. E-46022. Valencia (SPAIN)
Tel: +34 96387 9702 Fax: +34 96387 7579
Corresponding author: jorge@disca.upv.es

Andy Wellings, Alan Burns
Dept. of Computer Science
University of York
Heslington
York YO10 5DD
Great Britain

Abstract

The inclusion of dynamic ceiling priorities in Ada is a topic of discussion in the Real-Time Ada community. Several approaches have been discussed in previous editions of the International Real-Time Ada Workshop, which have allowed to identify the problems associated with this issue, mainly the interaction of such feature with tasking, consistency and efficiency of implementation. In this paper, a new approach for dynamic ceilings is presented. It is based on implementing the ceiling change as a predefined protected operation itself, which allows to eliminate the potential for data corruption that other approaches introduce, without the need for additional locks.

The paper also discusses the use of the proposed operation in the context of mode changes.

1 Introduction

The inclusion of dynamic ceilings in Ada has been a topic of discussion in the previous two editions of the International Real Time Ada Workshop [5, 4]. The main motivation is the ability to support mode changes as well as dynamically adapting existing program libraries containing protected objects. The different approaches considered have allowed to discuss in depth the issues involved with such a feature. Nevertheless, an agreement on a particular implementation and semantics has not been reached yet, due to the actual complexity of the problem.

Previous work has focused on implementing dynamic ceilings by means of a *special* operation in the sense that, as the ceiling change is an operation that affects the protected object's state, it was considered apart from the ceiling protocol itself. This approach has been shown to introduce new problems, specially the need for a dedicated lock in the implementation of the ceiling change and other sources of overhead, as discussed in the next sections.

In this paper, a different approach is presented to implement dynamic ceilings. We propose to implement the ceiling change as a predefined protected operation associated with each protected object. This allows to predict, by means of the ceiling locking protocol, that no task will be executing a protected procedure at the time of the ceiling change.

This solution opens the new problem of designing suitable protocols for changing the ceilings, specially in the case of mode changes, where a number of tasks may be active at the time the ceiling change must be performed.

The paper is structured as follows. Section 2 presents the main approaches to implement dynamic ceilings, including past proposals and also the proposal for protected ceiling changes. The section 3 discusses some implementation aspects of the proposed operation. In the section 4, different ways of using the new semantics are proposed and compared in the context of mode changes. Finally, section 5 presents our conclusions.

2 Approaches to dynamic ceilings

Two alternatives will be presented in this section:

1. Implement `Set_Ceiling` as a special operation.
2. Implement `Set_Ceiling` as a predefined protected operation.

2.1 `Set_Ceiling` as a special operation

The implementation of `Set_Ceiling` as a special operation, not necessarily adherent to the ceiling locking protocol, was motivated by the aim to be able to change a protected object's ceiling from a task of any priority. Particu-

*Partially funded by the Spanish Government CICYT project TIC99-1043-C03-02.

larly, from a very high priority task performing the necessary actions for a *fast* mode change.

In [5] three cases are identified with respect to the relative priorities of tasks executing a protected action, the task calling `Set_Ceiling` (referred to as *the caller*) and the old and new ceilings of the protected object:

Case 1: The caller has a lower or equal priority to the current ceiling and is changing the ceiling to a higher priority. In this situation the ceiling change can take place without problems. The priority change takes place and tasks on entry queues are not affected. They will just inherit the new ceiling when they execute the protected operation, which will be higher or equal to their base priority.

Case 2: The caller has a lower or equal priority to the current ceiling, as in Case 1, but now it is changing the ceiling to a lower priority. Here, it is possible that tasks queued on entries might have active priorities higher than the new ceiling. In this situation it is necessary to raise an appropriate exception to the queued tasks. Currently, in Ada, `Program_Error` is raised when a task tries to access a lower priority protected object.

Case 3: The caller has a higher priority than the current ceiling. Hence, *the ceiling change cannot adhere to the ceiling protocol to change the ceiling priority*. It should be noted that the POSIX 1003.1b standard says exactly the same about the operation provided for changing the ceiling of a mutex [1]. Cases 1 or 2 still apply with respect to the relationship between the old and new ceilings.

The worst case is represented by the third situation, where data corruption may occur. It could be the case that a task with a priority 10 wants to change the ceiling priority of a protected object with a ceiling lower than 10, say 8. Figure 1 shows graphically this scenario, where τ_i represents a task with priority i. A task with a lower priority, say 5, could be executing a protected action when it was preempted by the high priority task. The main problem here is that a task with a medium priority, say 8, that uses the same protected object, can be released whilst τ_5 is preempted by τ_{10}. When τ_{10} completes, the medium priority task τ_8 starts to run when τ_5 is still in the middle of the protected operation. The risk of data corruption in the protected object is clear.

A way to avoid this problem is providing a lock, indicating that the protected object is in use when τ_8 tries to access it. This would effectively eliminate the potential for data corruption, but may unfortunately produce a large priority inversion as depicted in figure 2. Moreover, the ceiling locking protocol in Ada has the characteristic that it does not need a lock to implement mutual exclusion: using the

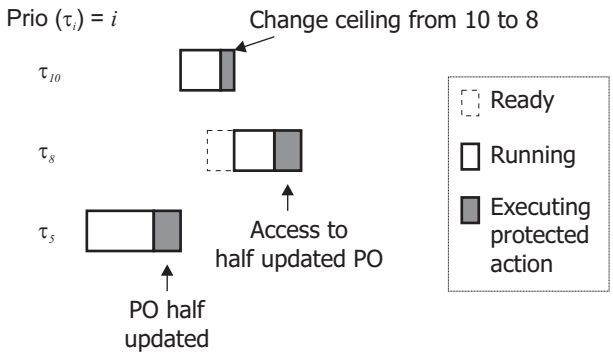

Figure 1. Potential access to corrupted data

appropriate priorities is sufficient to provide consistency. Therefore, adding a lock is not an attractive solution to dynamic ceilings, due to its impact on performance.

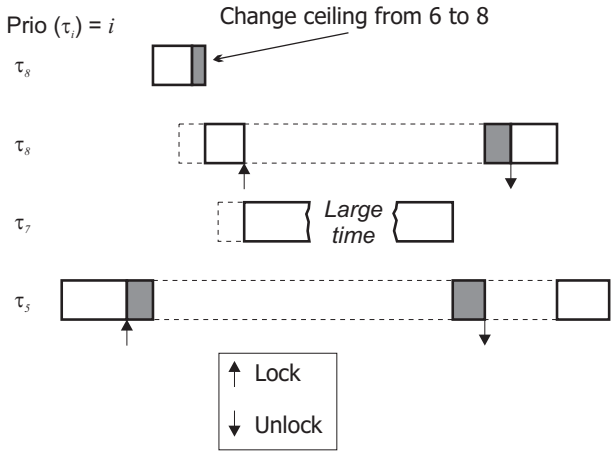

Figure 2. Large priority inversion using an additional lock

A second way to avoid the potential for data corruption is that the task executing the protected action when a ceiling change happens to occur, immediately inherits the new ceiling. In such case, no other task accessing the protected object can preempt it, therefore the protected action will be normally completed even in the presence of a ceiling change. A possible implementation of such semantics would be the following:

```
1.  Change variable "ceiling" to New_Ceiling
2.  if PO_In_Use and Prio(Caller) < New_Ceiling then
3.     Caller inherits New_Ceiling
4.  end if
```

At least two problems can be identified in this approach. First, the implementation of the priority inheritance adds the overhead of changing a task's priority, which can be relatively costly (line 3 above). Second, the runtime needs to know the identity of the task running the protected action

(the *caller*), something that is not needed in Ada but for this particular case.

2.2 `Set_Ceiling` as a predefined protected operation

A second approach to dynamic ceilings —the one proposed in this paper— is to implement it as a protected operation. As such, it is attached to the protected object as a predefined protected procedure. For a given protected object PO, we shall assume the existence of a procedure `Set_Ceiling`, implemented as an attribute to avoid name clash `PO'Set_Ceiling` where an *in* parameter of the type `System.Any_Priority` expresses the new ceiling priority for PO.

As `Set_Ceiling` is a protected procedure, it is subject to the ceiling locking policy. This means that a task calling it must have a priority lower than or equal to the protected object's ceiling. More precisely, the task's priority should not be above the old or the new ceiling.

According to this premise, a task calling `Set_Ceiling` will be executed in mutual exclusion with other potential users of the protected object, therefore ensuring that no task is executing a protected action when the ceiling is changed. This approach avoids the situation described in the previous section, where data consistency was at risk due to the *asynchronous* nature of the ceiling change when it is implemented as an special operation. If `Set_Ceiling` is implemented as a protected operation, the priority assignment is sufficient to guarantee its execution in mutual exclusion with other users of the protected object.

With respect to tasks queued in protected entries, it could be the case that the ceiling of the protected object is lowered below the queued task's base priority, which represents a bounded error in Ada. According to the Ada Reference Manual [6], section D.5-11: *If a task is blocked on a protected entry call, and the call is queued, it is a bounded error to raise its base priority above the ceiling priority of the corresponding protected object.* In other words, if dynamic ceilings are considered, this rule would also apply to the case where a protected object's ceiling is set below the base priority of task queued on one of the protected object's entries. The situation is already considered by the language, therefore no new problems are introduced in this sense.

We see that the implementation of `Set_Ceiling` as a protected operation avoids the problems of inconsistency and large priority inversion introduced by the approach of implementing dynamic ceilings as a special operation, separated from the ceiling locking protocol. The main motivation to investigate the case of changing the ceiling from *any* priority, even higher than the protected object's ceiling (as shown in section 2.1), was to be able to implement prompt mode changes: a high-priority task acts as the mode changer, changing periods, deadlines and priorities of tasks and protected objects before releasing new-mode tasks. A sensible question to study is the impact of limiting the priority of the task changing the ceiling on the mode change protocol. In section 4, different options are studied and compared.

3 Implementation aspects

After having presented the two main approaches in the previous section, we deal now with some aspects concerning the implementation of the protected dynamic ceiling.

An aspect in favor of the protected ceiling change is related to ease of use and implementation. In particular, the explicit mention of the protected object, as in the call `PO'Set_Ceiling(New_Ceiling)`, avoids the need for protected object identifiers in the language –as opposed to the need for task identifiers to implement dynamic task priorities.

The disadvantage of the syntax for the predefined protected ceiling change, in principle, is that the mode changer (in the mode change example) needs to know the names of the protected objects. This could be a problem for implementing a *generic* mode changer object. Nevertheless, this problem can be worked around by means of an *access-to-protected-subprogram*: the mode changer can access the different protected objects by means of a dereference to the adequate protected subprogram.

On the other side, the predefined protected operation approach can be very efficiently implemented by just changing the ceiling value associated with the protected object, as its modification is guaranteed to be performed in mutual exclusion by the use of the ceiling protocol.

The concerns on the implementation of dynamic ceilings outlined at the last edition of the International Real-Time Ada Workshop include [8]:

1. The proposed solution should not require an additional lock.

2. It should minimise the potential for priority inversion.

3. The behaviour of nested protected objects should be studied and worked out.

4. The rules should be consistent with dynamic task priorities and should result in minimum impact on the existing semantics for requeue, tasking, etc.

As mentioned above, an additional lock is not needed if the ceiling change is a protected operation. We note that other approaches require a lock to avoid access to corrupted data (see section 2.1).

Regarding the priority inversion produced by the ceiling change, two aspects are of importance: (i) the ceiling change is a very fast operation: just change the ceiling value; (ii) as it follows the ceiling locking protocol, the blocking may affect only to tasks with a priority below or equal to the ceiling.

The problem of nested protected objects becomes more an application problem than a language problem. The protected ceiling change does not introduce new problems with respect to dynamic tasks priorities. A task can have its base priority changed whilst executing a chain of nested protected actions. Indeed, if its base priority is raised above the ceiling of the protected object the task is using at a particular instant, a bounded error is said to occur and `Program_Error` should be raised.

Finally, the protected implementation avoids the problem of a ceiling changing whilst being used by a task, therefore limiting the interactions between dynamic ceilings and dynamic priorities for tasks.

4 Protocols for changing ceilings

In this section we shall assume the language has been extended with a new attribute `'Set_Ceiling` for protected objects, implemented as a protected operation, automatically generated by the compiler. The problem we illustrate here is how to use this feature in the context of a mode change. Another motivation for dynamic ceilings is to be able to dynamically adapting program libraries containing protected objects. This situation can be viewed as a simplified mode change problem, where an initial mode is before the initialisation of the library and we want to switch to a *fully-initialised* mode.

As the ceiling change is a protected operation, it must follow the ceiling locking protocol. This implies that the mode changer must have its priority adequately lowered to change the ceilings of protected objects with lower priorities. The lowering of the mode changer priority can be performed in two ways:

- Set the mode changer priority to the minimum in the system and change all the ceilings when it gets the processor.
- The mode changer lowers its priority in steps, dealing with different priority levels at different stages.

The first approach is valid for certain mode change protocols, such as the *idle-time* protocol [7]. In this protocol, upon a mode change request, the change in the schedule takes place as soon as the processor becomes idle. The mode changer can thus execute at the lowest priority, stop the release of all tasks, change the ceilings and tasks profiles when it gets the processor and then release the new-mode tasks. The protocol is simple to implement, but may not be suitable for prompt mode changes, such as a change to an emergency mode, because the time for the system to become idle is, in the worst case, not lower than the response time of the lowest priority task. Moreover, theoretically the idle time can only occur if the workload is below 100%.

Certain applications may impose a *promptness* requirement to the mode change transition. In these cases, it is more adequate to execute the urgent new-mode tasks as soon as possible after the mode change request. The mode changer can then deal with the different ceiling priority levels in steps, from the highest to the lowest ceilings, allowing new-mode tasks to gradually start executing, as far as the schedulability of the transition is preserved [3].

By changing the ceilings in decreasing priority order, we want to prompt the higher priority new-mode tasks. But, as some ceilings may need to be raised and others be lowered, we see two possible orders of adjusting ceilings:

1. Decreasing old-mode ceilings, that is, from higher to lower old-mode ceilings, and
2. Decreasing new-mode ceiling, which proceeds from higher to lower new-mode ceilings.

The figure 3 shows an example scenario of a multi-moded system. There exist two operating modes, denoted as *Old mode* and *New mode*, each consisting of four or five tasks and two protected objects. We shall study the transition from the old mode to the new one. Inside each task we find the associated priority: the greater the value, the higher the priority. For simplicity, we shall denote τ_i as *the task with a priority equal to* i. If necessary, we shall also say whether it is an old- or a new-mode task. The arrows denote the use relationship between tasks and objects. In the old mode, the protected object PO1 has a ceiling priority equal to $C_1^{old} = 10$, whereas $C_2^{old} = 8$. In the new mode, two tasks are introduced with priorities 12 and 5 (shadowed in the figure), which gives new ceiling values for the protected objects. The new-mode tasks τ_8 and τ_6 remain unchanged (with respect to their priorities in the old mode). The priority assignments in this example provide a case where one protected object has its ceiling lowered and the other has it raised. The example also considers the situation where a task uses more than one protected object.

Regarding the mode change protocol, we assume that old-mode tasks are allowed to complete their execution if they were running when a mode change request arrives in the system, but they are not released again as old-mode tasks after the mode change request. The worst case scenario, according to promptness in the mode change transition, occurs when the mode change request arrives coinciding with the simultaneous release of all the old-mode tasks [2]. We also assume tasks do not suspend themselves in any delay or entry queue.

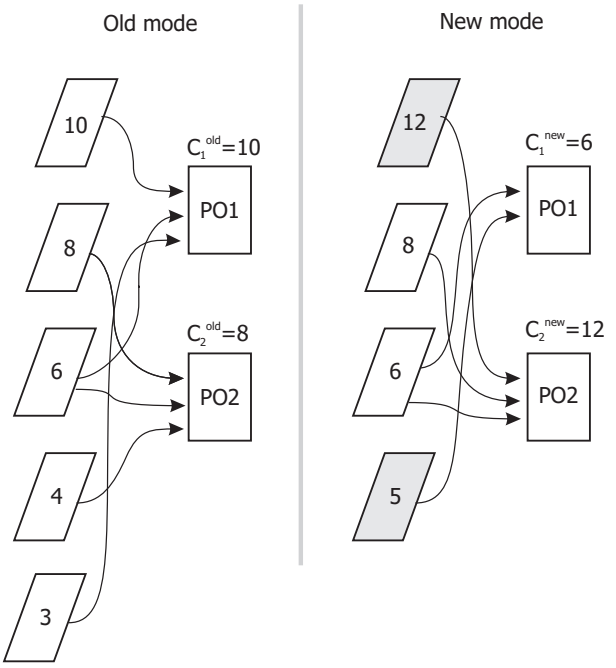

Figure 3. An example multi-moded system

Decreasing Old-Mode Ceilings Algorithm The first approach, the *Decreasing Old-Mode Ceilings* algorithm (DOC for short), is shown in figure 4. Upon a mode change request, the mode changer becomes ready. Assume it has (initially) the highest priority in the system. First, the ceiling priority of PO1 will be changed from 10 to 6 and then we should proceed with PO2, raising its ceiling from 8 to 12. The mode changer should lower its priority to consistently change the ceilings. The question is: what priority should the mode changer be set to in the different stages?

If we lowered the mode changer priority to 10, the old-mode ceiling, then it could be the case that the task τ_4 is ready to execute. The mode changer could change PO1's ceiling and afterwards the old-mode task τ_6 may call a protected operation in PO1. This could lead to an erroneous access, in the sense that τ_6 would inherit the new-mode ceiling, 6, instead of the old-mode ceiling, invalidating the blocking analysis. To avoid such situation, the mode changer should lower its priority to the lowest priority among the tasks using PO1. We call this priority level the *floor* priority of PO1.

Definition 1 FLOOR PRIORITY

The floor priority of a protected object PO is the lowest base priority among the tasks that use PO.

We shall use the floor priorities to consistently change the ceilings from the mode changer. If a task is executing a protected action on PO1, then its active priority is equal to 10, due to the ceiling locking protocol. On the other hand, if an old-mode task is ready to execute when the mode change request arrives and it is a potential user of a protected object PO, then its priority is not below the floor priority of PO. Setting the mode changer's priority to 3, the floor of PO1, places it at the tail of the queue for the priority level 3 (ARM D.5-15). This means that the rest of active tasks at that priority level will be allowed to complete before the mode changer finally changes the ceiling. In particular, the old-mode task τ_3 will use PO1 with the old-mode ceiling. After changing the ceiling of PO1 to the new value (6), the new-mode task τ_5 can be safely released. Regarding τ_6, the mode changer should also check that all the ceilings of the protected objects used by the task have been updated before it can be released. Therefore, the rules for releasing a new-mode task τ become:

1. The priority of τ is lower or equal to the new-mode ceiling, AND
2. all the protected objects used by τ have been updated.

This double check avoids τ_6 being released before the ceiling of PO2 has been changed to its new-mode value. To implement the algorithm, the application only needs to know the floor priorities of the protected objects and the use relationship between tasks and objects. Both can be obtained statically, before run time.

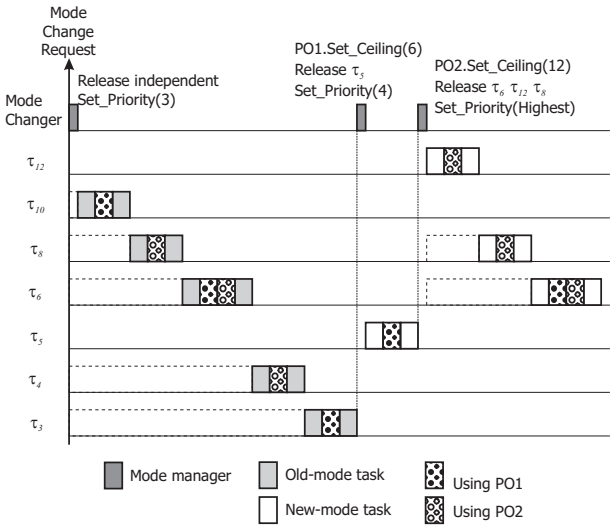

Figure 4. DOC: The Decreasing Old-mode Ceiling algorithm

Once PO1 has been updated, the mode changer sets its priority to 4, the floor priority of PO2, in order to change its ceiling. Again, it will be placed at the tail of the queue for that priority level, therefore allowing the protected actions in course (and the ones yet to come from old-mode tasks)

to complete. Once τ_4 has completed, the ceiling of PO2 is changed to 12 and the new-mode tasks τ_{12} and τ_8 can be safely released. Also τ_6 is released because all the protected objects it uses have been updated.

The last action of the mode changer is to set its priority back to the highest level, leaving it prepared to process a new mode change request.

As we can see, the DOC algorithm behaves correctly, in the sense that tasks never use the protected object with the wrong priority or ceiling. Nevertheless, the algorithm seems not appropriate to speed up the release of high-priority new-mode tasks, as it considers the ceilings in the old mode and not the new-mode ceilings. We see that τ_{12} has a large latency, as it has to wait for all the old-mode tasks to complete, including τ_{10}, τ_8, τ_6, τ_4, τ_3 and the new-mode task τ_5. Also, the new-mode version of τ_6 is delayed by the completion of τ_4, τ_3 and the new-mode version of τ_5, apart from the higher priority new-mode tasks τ_{12} and τ_8.

Decreasing New-Mode Ceiling Algorithm The figure 5 shows the behaviour of the Decreasing New-mode Ceiling algorithm (DNC). Accordingly, the ceiling priority of PO2 is changed first and then we proceed with PO1. The rules applied for releasing new-mode tasks are those introduced in the previous algorithm.

The release of higher-priority new-mode tasks is faster than in the case of DOC. This is a natural consequence of considering the new-mode ceilings to decide in what order to proceed. In particular, τ_{12} only has to wait for the tasks τ_{10}, τ_8, τ_6 and τ_4 to complete. The new-mode version of τ_6 is also released earlier than with the previous algorithm.

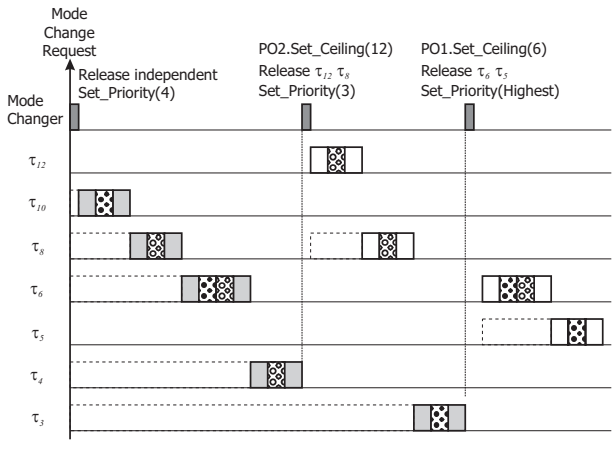

Figure 5. DNC: The Decreasing New-mode Ceiling algorithm

5 Conclusions

This paper has presented a new approach to dynamic ceilings in Ada. Previous work [5, 4] discussed its implementation by means of a special operation, apart from the ceiling locking protocol, whereas a protected operation automatically generated by the compiler has been introduced here. This approach eliminates the problems of potential inconsistency and the need of a lock to safely change the ceiling priority. It also results in a very efficient implementation of the ceiling change.

As changing the ceiling is here proposed as a protected operation, the priority of the task changing the ceiling of a protected object needs to be subject to the ceiling locking protocol. Two ways of using this feature have been presented and discussed, showing how a multi-moded application can benefit from protected ceiling changes, allowing the implementation of different mode change protocols.

References

[1] IEEE. Portable Operating System Interface: Amendment 1: Realtime Extensions [C Language]. IEEE 1003.1b, IEEE, 1993.

[2] J. Real. *Protocolos de Cambio de Modo para Sistemas de Tiempo Real* (Mode Change Protocols for Real Time Systems). Ph.D. thesis, Universidad Politécnica de Valencia, 2000. In spanish.

[3] J. Real and A. Crespo. Offsets for scheduling mode changes. In *Proceedings of the 13th Euromicro Conference on Real-Time Systems. Delft, The Netherlands*, pages 3–10. IEEE Computer Society Press, 2001.

[4] J. Real, A. Llamosí, and A. Crespo. A semantics for dynamic ceiling priorities in Ada. *Ada Letters*, XXI(1):91–95, March, 2001.

[5] J. Real and A. Wellings. Dynamic ceiling priorities and Ada 95. *Ada Letters*, XIX(2):41–48, July, 1999.

[6] S. Tucker Taft and Robert A. Duff (eds.). Ada 95 Reference Manual. Language and Standard Libraries. Springer, Lecture Notes on Computer Science, vol 1246, International Standard ISO/IEC-8652:1995(E), 1995.

[7] K. Tindell and A. Alonso. A very simple protocol for mode changes in priority preemptive systems. Technical report, Universidad Politécnica de Madrid, 1996.

[8] J. Tokar. Tasking and object orientation: session report. In *Proceedings of IRTAW10, Ada Letters*, volume XXI, nr 1, pages 9–10, 2001.