Formal Analysis of the Priority Ceiling Protocol*

Presented at RTSS'00, November 2000.

Bruno Dutertre
System Design Laboratory, SRI International
bruno@sdl.sri.com

Abstract

We present a case study in formal specification and tool-assisted verification of real-time schedulers, based on the priority ceiling protocol. Starting from operational specifications of the protocol, we obtain rigorous proofs of both synchronization and timing properties, and we derive a schedulability result for sporadic tasks.

1. Introduction

Scheduling and synchronization services are critical components of real-time operating systems that are being used in safety-critical applications. In such contexts, one must obtain strong guarantees of correctness, and rigorous development and verification methods are required. Using the priority ceiling protocol [10] as a case study, we illustrate how mechanical theorem proving can give high assurance that a scheduler ensures critical synchronization and timing properties, and that associated schedulability results are valid.

Previous applications of formal methods to the priority ceiling protocol typically consider only parts of the issues. For example, Pilling et al. [9] give a formal specification of the protocol and prove manually that the protocol ensures mutual exclusion and prevents deadlocks, but they do not consider timing aspects. Other work focuses on high-level scheduling properties – for example, developing formal proofs of Liu and Layland theorems [13, 14] – but ignores synchronization issues. Our objective is to obtain a tool-assisted specification and verification methodology for addressing both aspects. We want to prove low-level synchronization properties and high-level schedulability results.

In real-time systems, high-level properties, such as whether or not tasks meet their deadlines, depend on the task characteristics as much as on the scheduler. On the other hand, synchronization properties are typically satisfied for arbitrary tasks. We need a system model that allows us to prove task-independent properties in a generic way, but can be easily customized to reason about particular classes of tasks. Traditional models of real-time schedulers (e.g., [4, 9]) essentially represent tasks as nondeterministic automata cycling through states such as idle, ready, running, or blocked. Such models are adequate for proving generic synchronization properties but make it difficult to specify the timing characteristics of tasks.

The modeling approach we propose is also state based but relies on representing tasks as sequences of jobs, each characterized by attributes such as dispatch time and duration. We construct a state-machine model parameterized by a fixed but arbitrary set of independent jobs. The job attributes are assumed constant and known in advance, and this greatly simplifies the modeling and verification. At this level, tasks and their temporal characteristics are not relevant. Generic properties can be established that do not depend on the timing relationship between jobs and are valid for any class of tasks. In the case of the priority ceiling protocol, such generic properties include mutual exclusion and properties related to blocking.

The analysis proceeds by examining the execution traces of the state-machine model. General timing properties are obtained, such as bounds on blocking delays and on the processing time allocated to jobs. As before, no assumption is made on job attributes, and the timing properties are valid for any set of jobs. In a final step, high-level schedulability results for specific types of tasks are obtained by considering the execution traces of specific instances of the parameterized model.

The remainder of this paper describes the application of this analysis approach to the priority ceiling protocol.[1] The formalization and verification was supported by the PVS theorem prover [7] and some familiarity with the PVS notation is assumed. Section 2 introduces semaphores, programs, and critical sections. Section 3 presents the param-

*This work was supported by the Defense Advanced Research Projects Agency and by the Air Force Research Laboratory under contract F30602-97-C-0040.

[1] A more detailed presentation is given in [3] and the PVS developments are available at `http://www.csl.sri.com/~bruno/pvs/prio-ceiling.txt`.

eterized state-machine model and Section 4 describes the verification of synchronization properties. Section 5 defines execution traces and schedules, and examines timing properties such as bounds on blocking. Section 6 derives a well-known feasibility criterion for sporadic tasks from the preceding results.

2. Basic Notions

Priorities are represented by integers in the range $0, \ldots, \mathtt{maxprio} - 1$, where `maxprio` is a positive constant. An uninterpreted type represents semaphores, and a fixed function `ceil` assigns a ceiling between 0 and $\mathtt{maxprio} - 1$ to every semaphore. All these notions are fixed throughout the whole PVS development, but constraints relating job priorities and semaphore ceilings are introduced as PVS assumptions when necessary.

Programs are represented as finite sequences of commands. Commands are of three different forms: $\mathtt{P}(\mathtt{s})$, $\mathtt{V}(\mathtt{s})$, or `Step`. $\mathtt{P}(\mathtt{s})$ and $\mathtt{V}(\mathtt{s})$ are the commands for requesting and releasing semaphore `s`, respectively, and the constant `Step` represents any other command whose precise effect is not relevant for the protocol. We assume that all the commands take one time unit to execute.

The critical sections of a program are defined as in Figure 1. For a program `p` of length `l`, the type $\mathtt{pc}(\mathtt{p})$ denotes integers in the range $0, \ldots, \mathtt{l}$ that play the role of program counters. Given such an index `i`, the set $\mathtt{needs}(\mathtt{p}, \mathtt{i})$ contains all the semaphores `p` acquires after executing the commands $0, \ldots, \mathtt{i} - 1$. Critical sections are defined as follows:

- A critical section of `p` is an interval $[\mathtt{i}, \mathtt{j})$ where $\mathtt{i} < \mathtt{j} \leqslant \mathtt{l}$ and $\mathtt{needs}(\mathtt{p}, \mathtt{k}) \neq \emptyset$ for all `k` such that $\mathtt{i} \leqslant \mathtt{k} < \mathtt{j}$.
- A critical section is of level `n` if, for all `k` such that $\mathtt{i} \leqslant \mathtt{k} < \mathtt{j}$, $\mathtt{needs}(\mathtt{p}, \mathtt{k})$ contains a semaphore of ceiling at least `n`.

Proper nesting of critical sections is not required: semaphores can be acquired and released in an arbitrary order.

3. State-Machine Model

3.1. Jobs

The protocol is specified in a theory `priority_ceiling` whose parameters represent a set of jobs and their attributes. Each job is characterized by its dispatch time, its priority, and its program. The corresponding declarations and assumptions are shown in Figure 2. Since we use discrete time, dispatch times are natural numbers. Assumption `good_ceiling` requires that all the semaphores accessed by a job `j` have a ceiling at least as high as the priority of `j`. Assumption `good_programs` requires that all the jobs release all the semaphores they have acquired on termination.

```
programs: THEORY

  BEGIN
  ...
  prog: TYPE =
     [# length: posnat,
        clist: [below(length) -> command] #]

  p: VAR prog
  s: VAR semaphore

  pc(p): NONEMPTY_TYPE = upto(p`length)

  ...

  needs(p, (i: pc(p))): RECURSIVE rsrc_set =
     IF i=0 THEN emptyset ELSE
       CASES cmd(p, i-1) OF
          P(s): add(s, needs(p, i-1)),
          V(s): remove(s, needs(p, i-1)),
          Step: needs(p, i-1)
       ENDCASES
     ENDIF
   MEASURE i

  ...

  cs(p, (i: pc(p))): bool = not empty?(needs(p, i))

  cs(p, (i: pc(p)), n): bool =
    EXISTS s: member(s, needs(p, i))
               AND ceil(s) >= n

  ...

  critical_section(p,  (i, j: pc(p))): bool =
     i < j AND
       FORALL (k: pc(p)):
         i <= k AND k < j IMPLIES cs(p, k)

  critical_section(p,  (i, j: pc(p)), n): bool =
     i < j AND
       FORALL (k: pc(p)):
         i <= k AND k < j IMPLIES cs(p, k, n)
```

Figure 1. Programs and Related Notions

A precedence relation between jobs is the basis of the processor allocation policy. In the absence of blocking, jobs of high priority are executed first, and jobs of equal priority are executed in dispatch order:

```
precedes(j, k): bool =
   prio(j) > prio(k) OR
     prio(j) = prio(k) AND
       dispatch(j) <= dispatch(k).
```

3.2. Resource Management

The protocol is specified as a state machine with two main components: a resource-management subsystem con-

```
priority_ceiling [ (IMPORTING programs)
        job: TYPE,
        prio: [job -> priority],
        dispatch: [job -> nat],
        prog: [job -> prog] ] : THEORY

  BEGIN

  ASSUMING

  j: VAR job
  s: VAR semaphore

  good_ceiling: ASSUMPTION
      member(s, resources(prog(j)))
                IMPLIES prio(j) <= ceil(s)

  good_programs: ASSUMPTION  well_behaved(prog(j))

  ENDASSUMING
```

Figure 2. State Machine Parameters

trols the allocation of semaphores to jobs, and a scheduling subsystem selects the job to activate.

A state component stores the set of semaphores allocated to or requested by each job. This component is a record of type `rsrc_state` defined in Figure 3. Given a job `j` and a resource allocation state `r`, `r‘alloc(j)` is the set of semaphores owned by `j` in `r`, and `r‘request(j)` is the set of semaphores that `j` has requested but has not obtained. This set is empty unless `j` has attempted and failed to enter a critical section.

The set `blk(r, j)` is the set of jobs other than `j` that own a semaphore of ceiling as high as `j`'s priority. These jobs will block `j` if `j` ever tries to enter a critical section. As defined by predicate `blocked`, a job `j` is then blocked in `r` if both `blk(r, j)` and `r‘request(j)` are nonempty. This happens if `j` has attempted to enter a critical section in a state `r0` where `blk(r0, j)` was not empty.

Three operations control semaphore allocation and deallocation:

- `alloc_step(r, j, s)` executes `P(s)` on behalf of `j`. The semaphore `s` is allocated to `j` if `blk(r, j)` is empty; otherwise, `s` is stored in `r‘request(j)` and `j` becomes blocked.

- `release_step(r, j, s)` executes `V(s)` on behalf of `j`. It simply removes `s` from the set of semaphores allocated to `j`. After this operation, jobs that were blocked by `j` may be no longer blocked. The operation has no effect if `j` does not own `s`.

- `wakeup(r, j)` allocates to `j` all the semaphores `j` requested. This operation is used to reactivate a job `j` when it is no longer blocked.

```
rsrc_state: TYPE =
   [# alloc, request: [job -> rsrc_set] #]

r: VAR rsrc_state
s: VAR semaphore
j, k: VAR job

blk(r, j): set[job] =
   { k | k /= j AND
          EXISTS s: member(s, r`alloc(k)
                       AND ceil(s) >= prio(j) }

blocked(r, j): bool =
   not empty?(blk(r, j)) AND
         not empty?(r`request(j))

alloc_step(r, j, s): rsrc_state =
   IF empty?(blk(r, j)) THEN
      r WITH [ `alloc(j) := add(s, r`alloc(j)) ]
   ELSE
      r WITH [ `request(j) := add(s, r`request(j)) ]
   ENDIF

release_step(r, j, s): rsrc_state =
   r WITH [ `alloc(j) := remove(s, r`alloc(j)) ]

wakeup(r, j): rsrc_state =
   r WITH [ `alloc(j) := union(r`alloc(j),
                               r`request(j)),
            `request(j) := emptyset ]
```

Figure 3. Semaphore Allocation and Blocking

3.3. Scheduler

The scheduler specifications are shown in Figure 4. The scheduler state includes a resource-management component, a global time counter, and a program counter for every job. For every job `j` and every state `q`, the program counter `q‘pc(j)` is an integer in the range `0, ..., l`, where `l` is the length of `prog(j)`. If the program counter is less than `l`, it points to the next command of `prog(j)` to be executed. If it is equal to `l` then the job is completed.

Predicate `ready(q, j)` indicates whether `j` is ready to run in `q`. Predicate `topjob(q, j)` is true if `j` is ready in `q` and has precedence over all the other jobs ready in `q`. Such a job does exist unless there are no jobs ready in `q`. The key part of the specifications is predicate `eligible`. A job `j` can be activated in state `q` if the condition `eligible(q, j)` is satisfied. This holds if `j` is not blocked and has top priority among the jobs that are ready to run in `q`, or if `j` is blocking a job `k` of top priority. There can be more than one eligible job if two jobs of the same priority are dispatched at the same time.

3.4. Complete Protocol

The resource manager and the scheduler provide basic functions to specify the protocol. It remains to define the initial state and transition relation. Initially, program coun-

```
sch_state: TYPE = [#
      rsrc: rsrc_state,
      pc: [j:job -> pc(prog(j))],
      time: nat #].

q: VAR sch_state

finished(q, j): bool = q`pc(j) = prog(j)`length

ready(q, j): bool =
  dispatch(j) <= q`time AND not finished(q, j)

topjob(q, j): bool =
  ready(q, j) AND
    (FORALL k: ready(q, k) IMPLIES precedes(j, k))

eligible(q, j): bool =
     topjob(q, j) AND not blocked(q`rsrc, j)
  OR (EXISTS k: topjob(q, k) AND blocked(q`rsrc, k)
                  AND member(j, blk(q`rsrc, k)))
```

Figure 4. Scheduler and Eligible Jobs

ters point to the first command of each program, and the allocation and request sets are all empty:

```
init_rsrc: rsrc_state =
     (# alloc   := lambda j: emptyset,
        request := lambda j: emptyset #)

init_sch: sch_state =
     (# rsrc := init_rsrc,
        pc := lambda j: 0,
        time := 0 #)
```

The transition relation distinguishes two cases. If no job is eligible in state q then the next state, idle_step(q), is obtained by simply incrementing the time counter. Otherwise, one eligible job j is activated, and the next state, step(q, j), is obtained by incrementing the time counter, executing the command of prog(j) to which q‘pc(j) points, and incrementing j's program counter. The transition relation T is then as follows:

```
T(q1, q2): bool =
    idle(q1) AND q2 = idle_step(q1)
 OR EXISTS j:
     eligible(q1, j) AND q2 = step(q1, j)
```

For this definition to be sound, we must show that any job eligible in q1 is not already finished.[2] Since this property is not satisfied in general, we restrict our attention to states q that satisfy the following predicate.

```
P(q): bool =
      (FORALL j:
         union(q`rsrc`alloc(j),
               q`rsrc`request(j)) =
                  needs(prog(j), q`pc(j)))
   AND (FORALL j:
          q`time <= dispatch(j)
               IMPLIES q`pc(j) = 0)
```

Considering only states that satisfy P does not cause loss of generality: the following lemmas, together with the fact that the initial state satisfies P, show that all reachable states satisfy P.

```
step_P: LEMMA
   P(q) AND ready(q, j)
          IMPLIES P(step(q, j))

idle_P: LEMMA
   P(q) IMPLIES P(idle_step(q))

eligible_ready: LEMMA
   P(q) AND eligible(q, j)
           IMPLIES ready(q, j)
```

3.5. Modeling Choices

Our model differs from traditional descriptions of the protocol (e.g., [9, 10]) by avoiding dynamic priorities and priority inheritance altogether. The essential requirements are captured by the blocking rules (Figure 3) and by the definition of eligible jobs (Figure 4). The key property is that if a job of highest precedence is blocked by job j then j must be eligible irrespective of its priority. Priority inheritance is a mechanism for ensuring this property but is not a fundamental requirement.

The organization of jobs in tasks is irrelevant to the basic protocol mechanisms, and our model ignores this aspect. Modeling tasks more explicitly, as in [5] or [9], would lead to a more complex state machine. The current status of each process (e.g., suspended, ready, or running) would need to be included and the transitions between process states to be specified.

In implementations, the number of semaphores and tasks must be finite, but we do not make such an assumption. The only finiteness property we use is that there are finitely many priorities. This ensures that there is always a job of top precedence among the ready jobs. Other finiteness assumptions would only complicate the formalization. Our model does not exclude the possibility that an infinite number of jobs may be ready to execute at the same time, but this does not create any difficulty (from the verification point of view).

4. Synchronization Properties

Fundamental properties of the protocol include mutual exclusion, absence of deadlocks, and the fact that a job can have at most one blocker. All these properties are consequences of the invariance of two more basic state properties,

[2]The PVS typechecker generates a proof obligation, known as a TCC, to ensure that this is the case.

```
P2(r): bool =
  FORALL j, k, s:
    member(s, r`alloc(j)) AND prio(j) <= prio(k)
      AND prio(k) <= ceil(s) AND j /= k
        IMPLIES empty?(r`alloc(k))

Q(g): bool =
  FORALL j, k:
    ready(g, j) AND prio(k) <= prio(j)
      AND dispatch(j) < dispatch(k)
        IMPLIES empty?(g`rsrc`alloc(k))
```

Figure 5. Inductive Invariants

shown in Figure 5. These predicates are *inductive invariants*: they are satisfied in the initial state and are preserved by any state transition. It follows that any reachable state of the system satisfies these two predicates.

The most important of the two invariants is P2. Given two distinct jobs j and k of priority p_j and p_k, respectively, P2 states that if j owns a semaphore of ceiling c_s and $p_j \leqslant p_k \leqslant c_s$ then k does not own any semaphore. It is easy to see that P2 implies mutual exclusion: the same semaphore cannot be allocated to two distinct jobs. Absence of deadlocks also follows from P2. More precisely, P2 implies that blocking is intransitive: if j blocks k then j must own a semaphore of ceiling as high as k's priority. By P2, k does not own any semaphore, and then k cannot block any job at all. Absence of deadlocks is then obvious since a job cannot block itself.

Mutual exclusion and absence of deadlocks are important properties, but do not play a major role in schedulability analysis. The most relevant properties are those from which blocking bounds can be derived: a job j can have at most one blocker and this blocker is within a critical section of a level equal to j's priority. The definition of blockers and the associated properties are shown in Figure 6. The set $\mathtt{blockers}(\mathtt{r}, \mathtt{j})$ is the set of jobs that can block j in a state whose resource allocation component is equal to r. As an easy consequence of P2, either this set is empty or it contains a unique element k. This k owns a semaphore of ceiling at least as high as $\mathtt{prio}(\mathtt{j})$ so it must be within a critical section of level $\mathtt{n} = \mathtt{prio}(\mathtt{j})$. Finally, the invariance of Q ensures that, as long as j is ready, any eligible job either has precedence[3] over j or is the blocker of j.

The properties presented above are all fairly easy to prove with PVS. The only difficulty is determining the inductive invariants P2 and Q. The streamlined protocol specification is crucial for simplifying the proofs. In particular, eliminating priority inheritance and avoiding priority adjustment rules is an essential simplification. Similarly,

[3] The relation precedes is reflexive, so j itself can be eligible.

```
blockers(r, j): set[job] =
  { k | member(k, blk(r, j)) AND prio(k) < prio(j) }

unique_blocker: LEMMA
   P2(r) AND member(j1, blockers(r, k)) AND
     member(j2, blockers(r, k))
       IMPLIES j1 = j2

blockers_in_cs: LEMMA
   member(k, blockers(g`rsrc, j))
     IMPLIES cs(prog(k), g`pc(k), prio(j))

eligible_prio: LEMMA
   Q(g) AND ready(g, j) AND eligible(g, k) IMPLIES
      precedes(k, j) OR
        member(k, blockers(g`rsrc, j))

blockers_step: LEMMA
   Q(g1) AND ready(g1, j) AND T(g1, g2) IMPLIES
      subset?(blockers(g2`rsrc, j),
              blockers(g1`rsrc, j))
```

Figure 6. Blockers

using jobs as parameters makes the notion of readiness trivial and simplifies the analysis.

As a first result, one can see that the protocol works under weaker assumptions than made by its authors in [10]. Sha et al. [10] assume that the critical sections of a job are properly nested, but this requirement is actually unnecessary. All the important properties of the protocol are satisfied even if critical sections overlap.

5. Support for Scheduling Analysis

The previous developments consider synchronization properties expressed as state invariants. To obtain schedulability results, we define the execution traces of the state machine and introduce a notion of schedule that records the successive active jobs (if any) in each state of a trace.

5.1. Traces and Schedules

A trace of the protocol is a sequence w of the following type:

```
trace: NONEMPTY_TYPE =
   { w | w(0) = init_sch AND
             FORALL t: T(w(t), w(t+1)) }
```

The first state is equal to init_sch, and successive states of w are related by the transition relation T. Every trace u satisfies the following properties:

```
invariance_P2: PROPOSITION
   FORALL t: P2(u(t)`rsrc)

invariance_Q: PROPOSITION
   FORALL t: Q(u(t))
```

```
active_prio: LEMMA
   active(u, k, t) AND ready(u, j, t)
     IMPLIES precedes(k, j) OR
             member(k, blockers(u, j, t))

single_blocker: LEMMA
   member(j1, blockers(u, k, t)) AND
     member(j2, blockers(u, k, t))
       IMPLIES j1 = j2

blocker_in_cs: LEMMA
   member(j, blockers(u, k, t)) IMPLIES
     cs(prog(j), pc(u, j, t), prio(k))

blocker_step: LEMMA
   ready(u, j, t) IMPLIES
      subset?(blockers(u, j, t+1),
              blockers(u, j, t)).
```

Figure 7. Trace-level Formulation of Blocker Properties

```
time_invariant: PROPOSITION
   FORALL t: u(t)`time = t
```

The first two propositions say that P2 and Q are invariant, as discussed previously, and the third is proven by a trivial induction.

The following predicate indicates whether job j is active at time t in trace u. It is easy to show that no more than one job is active at a time:

```
active(u, j, t): bool =
   eligible(u(t), j) AND
      u(t+1) = step(u(t), j)
```

Various properties follow immediately from this definition and the synchronization properties established previously. For example, Figure 7 summarizes the important properties of blockers.[4] The lemmas state at the trace level the same properties as in Figure 6. These new formulations make the further PVS developments cleaner and less dependent on the internals of the state-machine model. This increases flexibility and limits the impact of any change in state representation on the remainder of the analysis.

A schedule sch is a sequence that records the active jobs and idle steps corresponding to a particular trace. At time t, $\mathtt{sch(t)}$ is either of the form $\mathtt{some(j)}$ to indicate that job j is active, or equal to none to indicate that no job is active at that time. With any trace u of the protocol, one can immediately associate a schedule $\mathtt{sch(u)}$.

5.2. Support Theories

Using schedules rather than traces allows us to obtain timing results that are independent of the state representation used. This also enables us to develop generic results that can be applied to other types of schedulers. All the relevant notions and properties are grouped in a set of support theories that are independent of the protocol specifications.

The bases of the timing analysis are functions that measure the processing time allocated to a job or set of jobs. Given a schedule sch and a set of jobs E, $\mathtt{process\_time(sch, t_1, t_2, E)}$ denotes the processing time allocated to jobs of E between $\mathtt{t_1}$ and $\mathtt{t_2 - 1}$. A similar function, idle_time, measures the amount of idle time in an interval.

A large part of the support theories provides various properties of process_time and idle_time. A typical example is the following

```
partition(E)(A, F) IMPLIES
  process_time(sch, t1, t2, E) =
    sum(A, lambda i:
      process_time(sch, t1, t2, F(i))).
```

In more standard notations, this lemma states that if $E = \bigcup_{i \in A} F_i$ and the sets of jobs F_i are pairwise disjoint, then

$$\mathtt{process\_time(sch, t_1, t_2, E)} \;=\; \sum_{\mathtt{i} \in \mathtt{A}} \mathtt{process\_time(sch, t_1, t_2, F_i)}.$$

Although it is not difficult to prove this lemma, a fair amount of work is required in developing the underlying theories. The function sum for finite sets is already defined in a PVS library. The finite set library provides elementary results, but was not sufficient and had to be extended.[5] Other notions such as partitions and sums over sequences were defined from scratch. In total, the support theories contain 134 lemmas and theorems.

5.3. Blocking Bounds

Using the support theories and the synchronization properties, we can obtain lower and upper bounds on the processing time allocated to a job or a set of jobs in a schedule $\mathtt{sch(u)}$. As previously, these bounds are established while assuming a fixed but arbitrary collection of jobs and are valid for any task set.

A first easy step is to relate the processing time allocated to a job j and the program counter of j:

```
process_time2: LEMMA
    t1 <= t2 IMPLIES
       process_time(sch(u), t1, t2, j) =
           pc(u, j, t2) - pc(u, j, t1).
```

[4]Functions of state are extended to traces in a straightforward way, for example, $\mathtt{pc(u, j, t) = u(t)`pc(j)}$.

[5]Some of these extensions were due to the new judgment facilities of PVS2.3. The finite set library was designed with earlier versions of PVS that did not include the same judgment facilities.

```
blockers_in_critical_section: LEMMA
  ready(u, j, t) AND t1=dispatch(j) AND
    member(k, blocker(u, j))
       IMPLIES pc(u, k, t) = pc(u, k, t1) OR
         critical_section(prog(k),
            pc(u, k, t1), pc(u, k, t), prio(j))

blocking(u, j): nat =
  IF empty?(blocker(u, j)) THEN 0
    ELSE max_cs(prog(the_blocker(u, j)), prio(j))
      ENDIF

blocking_time: LEMMA
   ready(u, j, t2) AND t1=dispatch(j) IMPLIES
     process_time(sch(u), t1, t2, blocker(u, j))
       <= blocking(u, j).
```

Figure 8. Upper Bound on Blocking Time

Figure 8 shows an essential property derived from this lemma and from previous results: if job $\mathtt{j}$ has priority $\mathtt{n}$, then the blocking time of $\mathtt{j}$ is no longer than the longest critical section of level $\mathtt{n}$ of $\mathtt{j}$'s blocker (if it exists).

Similar bounds are obtained for the minimal amount of processing time allocated to $\mathtt{j}$ in an interval where $\mathtt{j}$ is ready to execute. A generalized notion of blockers is also useful: the set $\mathtt{blockers}(\mathtt{u},\mathtt{p},\mathtt{t})$ contains the jobs of priority less than $\mathtt{p}$ that own a semaphore of ceiling at least $\mathtt{p}$ in state $\mathtt{u}(\mathtt{t})$. As before, this set contains at most one job $\mathtt{k}$ and we denote by $\mathtt{blocking}(\mathtt{u},\mathtt{p},\mathtt{t})$ the length of the longest critical section of level $\mathtt{p}$ in $\mathtt{k}$. The following property is essential

```
busy_time2: LEMMA busy(u, p, t1, t2)
  AND t1 <= t2 IMPLIES
    process_time(sch(u), t1, t2, K(p)) >=
      t2 - t1 - blocking(u, p, t1).
```

$\mathtt{K}(\mathtt{p})$ is the set of jobs of priority at least $\mathtt{p}$. The lemma gives a lower bound on the processing time allocated to such jobs, in an interval $[\mathtt{t}_1,\mathtt{t}_2]$ where there is always a job of priority at least $\mathtt{p}$ ready to run.

6. Feasibility of Sporadic Tasks

We now examine how a feasibility criterion for sporadic tasks is derived from the previous results.

6.1. Task Model

We consider a finite set of tasks represented by integers in the interval $[0,\mathtt{nbtasks}-1]$. A task $\mathtt{i}$ in this interval is characterized by the following attributes:

- $\mathtt{prio}(\mathtt{i})$: task priority
- $\mathtt{T}(\mathtt{i})$: minimal delay between dispatch of successive jobs of task $\mathtt{i}$
- $\mathtt{D}(\mathtt{i})$: task deadline
- $\mathtt{C}(\mathtt{i})$: maximal length of jobs of task $\mathtt{i}$

```
prio(i, n): priority = prio(i)

good_dispatch: ASSUMPTION
  dispatch(i, n) + T(i) <= dispatch(i, n + 1)

bound_length: ASSUMPTION
  length(prog(i, n)) <= C(i)

blocking: ASSUMPTION prio(i, n) < p
  IMPLIES max_cs(prog(i, n), p) <= B(p)
```

Figure 9. Model of Sporadic Tasks

We also assume a constant $\mathtt{B}(\mathtt{p})$ that bounds the blocking delay for jobs of priority $\mathtt{p}$.

The protocol model and all the results derived so far are parameterized by a $\mathtt{job}$ type and by functions defining job attributes. To apply these results to sporadic tasks, we instantiate these parameters by appropriate types and constants. Each job is represented by a pair $(\mathtt{i},\mathtt{n})$, where $\mathtt{i}$ is a task and $\mathtt{n}$ is a natural number: this pair corresponds to the $\mathtt{n}$-th instance of task $\mathtt{i}$. As required by the protocol model, each job has the following attributes:

- $\mathtt{prio}(\mathtt{i},\mathtt{n})$: job priority
- $\mathtt{dispatch}(\mathtt{i},\mathtt{n})$: dispatch time
- $\mathtt{prog}(\mathtt{i},\mathtt{n})$: program

These parameters are assumed to satisfy the constraints listed in Figure 9. We also assume that the priority of jobs is consistent with the ceiling of semaphores, that every job releases the semaphores it uses on termination, and that the total cpu utilization is less than 1.

6.2. Feasibility Criterion

From the above assumptions and the blocking bounds, we obtain a PVS proof of a well-known feasibility test based on computing worst-case response times (e.g., [2, 11]).

The proof is built in two stages. Let $\mathtt{A}(\mathtt{p})$ denote the set of tasks of priority at least $\mathtt{p}$. First, we showed that the sequence $\mathtt{u}$ defined as

$$\begin{aligned} \mathtt{u}_0 &= \mathtt{B}(\mathtt{p}) \\ \mathtt{u}_{\mathtt{n}+1} &= \mathtt{B}(\mathtt{p}) + \sum_{\mathtt{i}\in\mathtt{A}(\mathtt{p})} \mathtt{C}(\mathtt{i}) \times \left\lceil \frac{\mathtt{u}_\mathtt{n}}{\mathtt{T}(\mathtt{i})} \right\rceil \end{aligned}$$

reaches a fixed point $\mathtt{M}(\mathtt{p})$. This fixed point is the smallest solution of the equation

$$\mathtt{M}(\mathtt{p}) = \mathtt{B}(\mathtt{p}) + \sum_{\mathtt{i}\in\mathtt{A}(\mathtt{p})} \mathtt{C}(\mathtt{i}) \times \left\lceil \frac{\mathtt{M}(\mathtt{p})}{\mathtt{T}(\mathtt{i})} \right\rceil .$$

The proof relies on a standard argument and uses part of the support theories discussed previously.

Given a set of sporadic tasks, let u be an arbitrary trace of the protocol. Let $[\mathtt{t}_1, \mathtt{t}_2]$ be an interval such that for any t between $\mathtt{t}_1$ and $\mathtt{t}_2$ there is a job of priority at least p ready to execute at time t in u. From Lemma busy_time2 and the assumption on $\mathtt{B}(\mathtt{p})$, we obtain

```
process_time(sch(u), t1, t2, K(p))
   >= t2 - t1 - B(p).
```

For a fixed priority p, we say that a time t is quiet if the jobs of priority at least p that started before t are all finished at time t:

```
quiet(u, p, t): bool =
  FORALL j:
    dispatch(j) < t AND prio(j) >= p
      IMPLIES finished(u, j, t).
```

The following lemma is important:

```
busy_interval2: LEMMA
   quiet(u, p, t1) AND t1 <= t2 IMPLIES
      process_time(sch(u), t1, t2, K(p))
          <= sum(A(p), lambda i: C(i) *
                 ceiling((t2 - t1)/T(i)))
```

This says that, if $\mathtt{t}_1$ is a quiet time, the processing time allocated in $[\mathtt{t}_1, \mathtt{t}_2]$ to jobs of priority at least p is no more than

$$\sum_{\mathtt{i} \in \mathtt{A(p)}} \mathtt{C(i)} \times \left\lceil \frac{\mathtt{t}_2 - \mathtt{t}_1}{\mathtt{T(i)}} \right\rceil.$$

Now, let $\mathtt{t}_1$ be a quiet time and let $\mathtt{t}_2 = \mathtt{t}_1 + \mathtt{M(p)}$. If, at some point t in the interval $[\mathtt{t}_1, \mathtt{t}_2]$, no job of priority p or higher is ready to run, then t is a quiet point. Otherwise, the definition of $\mathtt{M(p)}$ and the two bounds above give

$$\mathtt{process\_time(sch(u), t_1, t_2, K(p))} = \sum_{\mathtt{i} \in \mathtt{A(p)}} \mathtt{C(i)} \times \left\lceil \frac{\mathtt{M(p)}}{\mathtt{T(i)}} \right\rceil,$$

which implies that all jobs of $\mathtt{K(p)}$ started between $\mathtt{t}_1$ and $\mathtt{t}_2$ are finished at $\mathtt{t}_2$: $\mathtt{t}_2$ is then a quiet point. This shows that the delay between two successive quiet points is no more than $\mathtt{M(p)}$ and this is sufficient to prove the propositions of Figure 10.

Proposition schedulability_criterion is the classic result: if distinct tasks have distinct priorities, the deadline $\mathtt{D(i)}$ of task i is not larger than $\mathtt{T(i)}$, and for all task i there exists $\mathtt{M} \leqslant \mathtt{D(i)}$ such that

$$\mathtt{B(i)} + \mathtt{C(i)} + \sum_{\mathtt{l} \in \mathtt{H(i)}} \mathtt{C(l)} \left\lceil \frac{\mathtt{M}}{\mathtt{T(l)}} \right\rceil \quad = \quad \mathtt{M},$$

```
schedulability1: PROPOSITION
   (FORALL i: M(prio(i)) <= D(i)) IMPLIES
      (FORALL u, j: finished(u, j, deadline(j)))

schedulability_criterion: PROPOSITION
      (FORALL l1, l2: prio(l1) = prio(l2)
           IMPLIES l1 = l2)
  AND (FORALL i: D(i) <= T(i))
  AND (FORALL i: EXISTS M:
         M = sum(H(i),
            lambda l: C(l) * ceiling(M/T(l)))
                    + B(i) + C(i))
         AND M <= D(i))
 IMPLIES
    (FORALL u, j: finished(u, j, deadline(j)))
```

Figure 10. Feasibility Criteria

where $\mathtt{H(i)}$ is the set of tasks of higher priority than i, then the task set is feasible. All jobs meet their deadlines in any trace u of the protocol.

The proof sketched above relies on a variant of the notion of busy periods.[6] A notable simplification is that we do not use worst-case scenarios or critical instants. Many feasibility results refer to Liu and Layland's theorem 1, which states that the worst-case response time for a task τ is obtained when all the tasks of priority higher than τ are dispatched at the same time as τ [6]. The worst case is when all the tasks are released at the same time. Proofs are often based on finding the response time for jobs released at such a critical time. Looking for such worst-case scenarios is an unnecessary detour since the essential bound given by lemma busy_interval2 is satisfied whether or not $\mathtt{t}_1$ is a critical instant.

The feasibility tests of Figure 10 apply if deadlines are less than or equal to job inter-arrival delays. Still, many results necessary for handling the general case, where $\mathtt{D(i)}$ can be larger than $\mathtt{T(i)}$, have already been established. This general case could then be formally analyzed in PVS with some extra work.

7. Discussion and Related Work

The PVS formalization includes theories and results that are specific of the priority ceiling protocol, and more general theories that are reusable in other contexts. The general modeling and verification methodology presented is applicable to other types of scheduling algorithms. How much effort is required and how much of the existing developments can be reused depend on the focus of the analysis and on the complexity of the scheduling algorithm under investigation.

[6] A busy period is an interval $[\mathtt{t}_1, \mathtt{t}_2]$, where $\mathtt{t}_1$ and $\mathtt{t}_2$ are successive quiet times and a job of priority p or more is ready at $\mathtt{t}_1$.

A state-machine model is convenient for understanding the low-level properties of an algorithm. Such models are mostly useful for verifying synchronization properties and for algorithms whose mechanisms are complex and subtle. Clearly, such state-machine models are algorithm specific and their verification may require substantial effort. In some cases, more abstract descriptions are sufficient. For example, earliest-deadline-first or priority-based scheduling can be defined by simple rules and do not require complex machine models.

The PVS developments that we specially intend to be reusable focus on higher-level issues. The support theories presented in Section 5.2 define a notion of schedules, introduce functions for measuring processing and idle time, and provide a large set of lemmas about these basic concepts. Our goal is for this PVS library to conveniently support feasibility analysis for various scheduling approaches and task models. Although the library is still being extended, we have had reasonable success in recent experiments. We have developed PVS proofs of the optimality of earliest-deadline-first and minimal-laxity-first scheduling and of a simple schedulability criterion valid in these two models. Each proof took less than a day to complete.

Other parts of the protocol formalization can also be reused but in more narrow contexts. For example, the proof of the feasibility criterion presented in Section 6 actually relies on only a few properties of the protocol, namely, that jobs are executed in priority order and that blocking is bounded. The corresponding PVS theorems are then directly applicable to any other scheduling algorithm that satisfies the same properties.

Apart from the PVS specifications and theories, an important outcome of the case study is the general methodology for formalizing and verifying real-time schedulers. The PVS language is a variant of higher-order logic that offers great freedom for writing specifications. Many modeling styles can be envisaged for real-time schedulers but not all induce the same verification effort. Proof difficulty varies greatly with the form and organization of the specifications. Designing and structuring the specifications to take advantage of the prover's capabilities is one of the most challenging aspects of using tools such as PVS. Formalizing the priority ceiling protocol helped us identify an effective modeling and verification methodology that addressed this issue, and is applicable to many real-time scheduling problems.

Except at the lowest levels, our analysis of the priority ceiling protocol relies mostly on traces and schedules. Timing and feasibility properties are established without referring to automata. Alternative approaches, that rely on variants of timed automata, enable the verification of timing and schedulability properties using model-checking techniques.

Vestal [12] models the MetaH executive using hybrid linear automata and uses reachability analysis to verify timing properties (i.e., deadlines are met) and other invariants (e.g., all variables are initialized before they are used). The analysis discovered several defects in the executive and in MetaH tools. The verification examines several scenarios, each consisting of one or two tasks with different timing characteristics and synchronization constraints. Each of these simple scenarios is analyzed using state-exploration algorithms.

Using the same type of automata, Altisen et al. [1] propose a general methodology for examining scheduling issues. Tasks are modeled by timed automata with controllable and uncontrollable transitions. Controllable transitions represent scheduler actions, such as activation or preemption of tasks, and resource allocations. Uncontrollable transitions represent events such as job arrival or termination. In this setting, schedulability is equivalent to the following controllability problem: Can a controller ensure that all tasks eventually progress? Such questions can be answered using fixed point computations and incremental controller constructions. Constraints on acceptable controllers correspond to specific types of scheduling algorithm. The approach is very general; it provides a rich priority model and can handle task sets with complex interactions and timing characteristics.

Penix et al. [8] use more standard automata and model-checking tools for modeling and verifying the kernel of an existing real-time operating system. They build a state-machine model directly from kernel code and analyze the model to verify time partitioning properties. By a combination of abstraction and model-checking, they discovered a serious flaw in the kernel code, that prevents a thread from receiving its alloted CPU time.

As these examples illustrate, fully automatic verification of real-time schedulers is out of the reach of current model-checking technology. The models used in [12] and [1] are timed automata with integrators, for which reachability and controller computation are only semi-decidable (i.e., computations may fail to converge). Even when computations terminate, the practical complexity of the algorithms is very high and only small automata can be analyzed. More standard automata do not suffer from the same theoretical limitations, but, as shown by [8], model checking complex real-time systems still requires substantial effort from the users. Expertise is required for constructing the right system abstraction and guiding the model-checking process.

In the domain of tool-assisted analysis of real-time systems, deductive methods supported by interactive theorem proving are viable alternatives to these more algorithmic methods. Although they may require more effort, theorem-proving techniques achieve a level of generality that is not attainable with model checking. Timed automata and state-exploration techniques are invaluable for rapidly analyzing a specific system, modeled by a specific automaton. When one is interested in more general issues – such as feasibility criteria for a generic task model – which amounts to reasoning about a class of systems (possibly represented as a class of automata), theorem proving has no alternative.

8. Conclusion

We present a complete specification and analysis of the priority ceiling protocol, from low-level resource management and job selection aspects, to high-level schedulability conditions for sporadic tasks. The formalization shows that it is possible to perform a very thorough and detailed verification of a fairly subtle scheduling mechanism such as the priority ceiling protocol.

The case study illustrates how new insights can be gained from such an analysis: our formalization is more precise, and we believe simpler, than traditional descriptions of the protocol. Priority adjustment rules are avoided by using a slightly modified rule for selecting active jobs. Other results emerged from the analysis such as the fact that proper nesting of critical sections is not necessary. Our proof of the feasibility test for sporadic tasks is also more direct than other approaches based on worst-case scenarios and critical instants.

All the developments and proofs were performed with tool support, using the PVS specification and verification system. This provides high assurance of correctness and ensures a complete and rigorous analysis. The formalization and verification effort represented between two and three personmonths, and the PVS developments contain 417 lemmas and theorems for around 2500 lines of specifications. Many theories and results are actually independent of the protocol and can be reused for analyzing other scheduling policies and proving other scheduling results. As often in tool-assisted verification, the key to success is to model the relevant properties of the system under investigation but stay abstract enough to make mechanical proofs manageable. We achieved this objective by relying on jobs rather than tasks, modeling the protocol behavior for a fixed collection of jobs, and avoiding dynamic priorities. These ideas are applicable to other types of schedulers and other theorem provers than PVS. In future work, we intend to apply the same approach to more complex schedulers that provide temporal partitioning guarantees as required in applications such as Integrated Modular Avionics.

References

[1] K. Altisen, G. Gößler, and J. Sifakis. A Methodology for the Construction of Scheduled Systems. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, Pune, India, September 2000. To appear.

[2] N. Audsley, A. Burns, M. Richardson, and A. Wellings. Hard Real-Time Scheduling: The Deadline Monotonic Approach. In *Proceedings of the 8th IEEE Workshop on Real-Time Operating Systems and Software*, pages 127–132, Atlanta, GA, May 1991.

[3] B. Dutertre. The Priority Ceiling Protocol: Formalization and Analysis Using PVS. Technical report, System Design Laboratory, SRI International, Menlo Park, CA, October 1999. Available at `http://www.sdl.sri.com/dsa/publis/prio-ceiling.html`.

[4] S. Fowler. *A Development Method for Trusted Real-Time Kernels*. PhD thesis, University of York, Department of Computer Science, August 1998.

[5] S. Fowler and A. Wellings. Formal Analysis of a Real-Time Kernel Specification. In *Proceedings of the Fourth Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 1135 of *Lecture Notes in Computer Science*, pages 440–458, Uppsala, Sweden, September 1996.

[6] C. Liu and J. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, 20(1):46–61, January 1973.

[7] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal Verification for Fault-Tolerant Architectures: Prolegomena to the Design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[8] J. Penix et al. Verification of Time Partitioning in the DEOS Scheduler Kernel. In *International Conference on Software Engineering (ICSE 2000)*, pages 488–497, Limerick, Ireland, June 2000.

[9] M. Pilling, A. Burns, and K. Raymond. Formal Specifications and Proofs of Inheritance Protocols for Real-Time Scheduling. *Software Engineering Journal*, 5(5):263–279, September 1990.

[10] L. Sha, R. Rajkumar, and J. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, September 1990.

[11] L. Sha, R. Rajkumar, and S. Sathaye. Generalized Rate-Monotonic Scheduling Theory: A Framework for Developing Real-Time Systems. *Proceedings of the IEEE*, 82(1):68–82, January 1994.

[12] S. Vestal. Formal Verification of the MetaH Executive Using Linear Hybrid Automata. In *Procedings of the 6th IEEE Real-Time Technology and Application Symposium (RTAS'2000)*, pages 134–144, Washington, DC, May-June 2000.

[13] M. Wilding. A Machine-Checked Proof of the Optimality of a Real-Time Scheduling Policy. In *Computer-Aided Verification – CAV'98*, volume 1427 of *Lecture Notes in Computer Science*, pages 369–378, Vancouver, Canada, June-July 1998. Springer-Verlag.

[14] Z. Yuhua and Z. Chaochen. A Formal Proof of the Deadline Driven Scheduler. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 863 of *Lecture Notes in Computer Science*, pages 756–775, Lubeck, Germany, September 1994. Springer Verlag.