The Priority Ceiling Protocol: Formalization and Analysis Using PVS*

Bruno Dutertre
System Design Laboratory
SRI International
Menlo Park, CA 94025

October, 1999

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

Abstract

Common real-time operating systems rely on priority-based, preemptive scheduling. Resource sharing in such systems potentially leads to *priority inversion*: processes of high priority can be prevented from entering a critical section and be delayed by processes of lower priority. Since uncontrolled priority inversion can cause high-priority processes to miss their deadlines, a real-time operating system must use resource-sharing mechanisms that limit the effects of priority inversion. The priority ceiling protocol is one such mechanism. It ensures mutual exclusion and absence of deadlocks, and minimizes the length of priority inversion periods. This paper presents a formal specification and analysis of the protocol using PVS and the rigorous proof of associated schedulability results.

Chapter 1

Introduction

Fixed-priority, preemptive scheduling is being increasingly used in real-time safety-critical applications. This approach to real-time scheduling is more flexible than static scheduling, and the runtime behavior of a system is still sufficiently predictable to obtain guarantees that timing constraints are satisfied. Many theoretical results provide such guarantees for different classes of systems, with different assumptions about tasks and deadlines [19, 20, 32, 37].

Subtle issues arise in this context when processes of different priorities are allowed to share resources. Mutual exclusion can cause high-priority tasks to be blocked and delayed by tasks of lower priority. To still obtain guarantees that timing constraints are satisfied, one must ensure that the delays caused by blocking are bounded and that the bounds can be determined statically. Since high-priority tasks are usually the most urgent, it is also important to minimize blocking delays. The priority ceiling protocol is a resource allocation mechanism introduced by Sha et al. [33,34] that achieves these two objectives. The protocol ensures mutual exclusion, minimizes blocking, prevents deadlocks, and allows a bound on blocking to be computed statically.

For systems based on the priority ceiling protocol to support safety-critical applications, one needs high assurance that the protocol is correct and properly implemented, and that the associated schedulability results are valid. Unfortunately, the description given by Sha et al. [34] is quite informal and possibly open to misinterpretation. The key properties are also proved in a fairly informal manner, and the reasoning relies more on intuition than on rigorous arguments. The main objective of this paper is to provide precise specifications of the priority ceiling protocol and develop rigorous proofs that the associated results are valid. The developments are supported by the PVS theorem prover [24].

Chapter 2 gives an overview of the priority ceiling protocol. Chapter 3 presents a state-machine specification of the protocol in PVS and the proof of key properties, such as mutual exclusion and absence of deadlocks. Chapter 4 describes the PVS model used to establish schedulability properties. Bounds on blocking and on the amount of processing time allocated to jobs according to their priority are obtained. Chapter 5 shows how this model can be used to derive a schedulability test for sets of sporadic tasks. [1] In Chapter 6, we com-

[1]This result extends immediately to periodic tasks since the latter can be considered as a subclass of sporadic tasks

pare our model and verification approach with related work in formal analysis of real-time operating systems and real-time scheduling.

Chapter 2

Overview of the Priority Ceiling Protocol

2.1 Assumptions

Only uniprocessor systems are considered in this document. A real-time application is modeled as a finite set of tasks that run concurrently on a single processor. A task consists of a succession of jobs, each requiring a finite amount of computation. In other words, the computation associated with each job always terminates. Each job is characterized by its release or dispatch time and by the amount of processing it requires. For example, a process sampling an input signal at a frequency of 100 Hz can be modeled as a periodic task of period 10ms. The successive jobs of this task are released at time t_0, t_0+10ms, t_0+20ms, and so forth, and the length of each job is the amount of time required for processing a single sample.

Other resources than the processor are shared by the different jobs. For example, the jobs may share common I/O channels or communicate with each other via shared variables. Access to these shared resources is controlled by binary semaphores that ensure mutual exclusion. The operations for requesting and releasing the lock on a semaphore S are denoted by $P(S)$ and $V(S)$, respectively.

Each task and job has a fixed priority; the priority of a job is the priority of the task to which it belongs. Jobs are executed in priority order: if two jobs of different priorities are ready to run at the same time, then the one with the highest priority is allocated the processor. If two competing jobs have the same priority and different dispatch times, the one that was dispatched first is given the processor. If two competing jobs have both the same priority and the same dispatch time, the processor is allocated to one of them, arbitrarily. For full generality, we do not exclude the latter case but in practice, priority assignment techniques such as the rate monotonic or the deadline monotonic approaches avoid this situation by giving different priorities to the different tasks.

2.2 Priority Inversion

At time t, all jobs that have been dispatched before t and are not completed yet are ready to execute. In systems where all the tasks are independent, there is no resource sharing and the active job at time t is the job of highest priority among those that are ready. Because of the need for mutual exclusion, this cannot be guaranteed anymore if resources are shared. A job j can be prevented from entering a critical section guarded by a semaphore S if a job of lower priority has locked S before j was dispatched. As a result, a job j of top priority can be unable to execute and a job of lower priority than j can become active. This phenomenon is called priority inversion: the top-priority job j is blocked and delayed by jobs of lower priority.

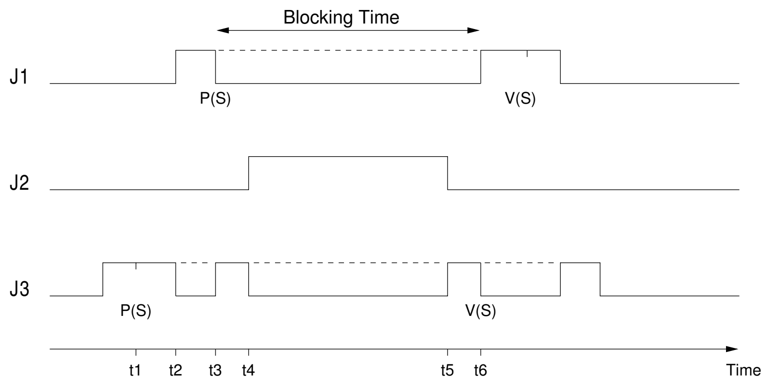

Figure 2.1: Blocking Time Caused by Priority Inversion

To ensure mutual exclusion, a job j is blocked when executing $P(S)$ if the semaphore S is locked by another job k. In such a case, j remains blocked until k releases the lock via a call to $V(S)$. As illustrated by Figure 2.1, this can lead to long blocking periods. The figure shows a system execution involving three jobs J_1, J_2, and J_3 in decreasing priority order. J_1 and J_3 synchronize by using a semaphore S. J_3 starts first and enters in critical section at time t_1. At time t_2, job J_1 is released and preempts J_3. At time t_3, J_1 attempts to enter a critical section guarded by S and is blocked by J_3. J_3 is then restarted and runs until it is preempted by the job J_2 of intermediate priority. At time t_5, J_2 terminates and J_3 restarts. J_3 stays active until it releases S at time t_6. Overall, J_1 is delayed for the interval $[t_3, t_6]$: J_1 waits not only for J_3 to release S but also for J_2 to execute. The high-priority job J_1 can then be delayed by the low-priority job J_3 that locks S but also by any job of intermediate priority that might preempt J_3. Since high-priority jobs are usually the most urgent and may have tight deadlines, such unrestricted priority inversion can be disastrous.

2.3 Priority Inheritance

To limit the effects of priority inversion, one must ensure that a low-priority job releases a semaphore S as soon as possible once a high-priority job has requested S. In the example above, this means that J_2 must not be allowed to preempt J_3 as long as J_1 is blocked by J_3.

A simple way to prevent such preemption is to momentarily increase J_3's priority to that of J_1. More generally, when a high-priority job j is blocked by a low-priority job k that owns a semaphore S, then k inherits the priority of j until it releases S. During the interval, only jobs of higher priority than j can preempt k.

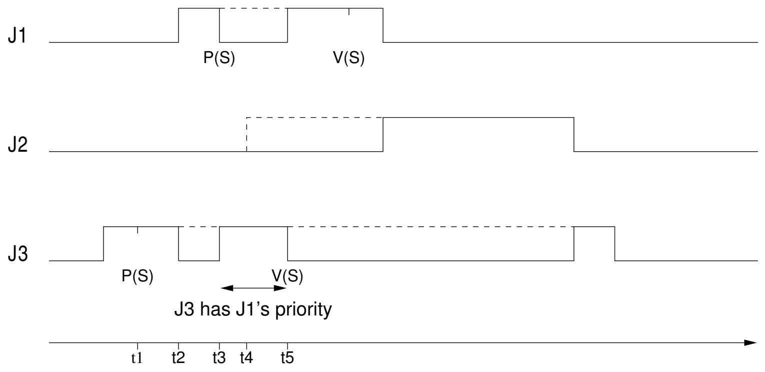

Figure 2.2: Priority Inheritance

Figure 2.2 illustrates the benefits of priority inheritance. As previously, J_3 locks semaphore S at time t_1 and is preempted at time t_2 when J_1 starts. J_1 runs until time t_3 where it is blocked by J_3. According to the priority-inheritance principle, J_3 runs with J_1's priority from t_3 to t_5. When J_2 is dispatched at time t_4, it cannot preempt J_3 anymore and then cannot indirectly delay J_1. J_1 is blocked only for the interval $[t_3, t_5]$, that is, for the delay needed by J_3 to exit its critical section.

The basic priority inheritance protocol described by Sha et al. [34] relies on this simple priority adjustment scheme. This approach was also discussed earlier, for example by Lampson and Redell [18]. Under the assumption that deadlocks do not occur, this protocol ensures that blocking delays are bounded. For each semaphore S used by a job j, at most one job k of lower priority can block j by locking S. The corresponding blocking delay cannot be longer than the longest critical section of k guarded by S. The worst-case blocking of j can then be bounded a priori by finding the semaphores used by j and examining the jobs of lower priority than j that share a semaphore with j.

Although this basic protocol is an improvement on the preceding scheme, it still suffers from two limitations. First, the protocol does not prevent deadlocks by itself, and some other deadlock-prevention technique must be used. Second, blocking chains can occur, that is, a job j can be blocked by several jobs of lower priority in succession. For example, this happen if j needs to access two semaphores S_1 and S_2 successively, and S_1 and S_2 are locked by two different jobs of lower priority than j. Job j will be delayed by the time required for both jobs to exit their respective critical sections. To address these issues, Sha et al. [34] designed the priority ceiling protocol that prevents both deadlocks and blocking chains.

2.4 The Priority Ceiling Protocol

The problem with the basic protocol is that job priorities are not taken into account when access to critical sections is granted. If a semaphore S is free, a job j executing $P(S)$ obtains access to S, irrespective of any other jobs already in a critical section. Two jobs j_1 and j_2 can then lock two distinct semaphores S_1 and S_2. If the two semaphores are requested later on by a job k of higher priority than j_1 and j_2, two successive blocking periods will occur. To prevent such a situation, the priority ceiling protocol enforces stronger rules for accessing a critical section. If two jobs j_1 and j_2 could potentially block a common job k via two semaphores S_1 and S_2, the protocol does not grant S_1 to j_1 and S_2 to j_2 at the same time. For example, if j_1 has lower priority than j_2 and enters a critical section first, then the request $P(S_2)$ by j_2 will not be granted even though S_2 may be free.

To decide whether it is safe to allocate a semaphore S to a job j, the protocol must have some information about the other jobs that might request S in the future. For this purpose, each semaphore S is assigned a fixed *ceiling* that is equal to the highest priority among the jobs that need access to S. If S is allocated to j then jobs of priority higher than j and lower than or equal to the ceiling of S might become blocked by j. The rule for entering critical sections is based on the priority of the requesting job and the ceiling of the semaphores already locked:

> A job j executing $P(S)$ is granted access to S if the priority of j is strictly higher than the ceiling of any semaphore locked by a job other than j. Otherwise, j becomes blocked and S is not allocated to j.

In the previous example, since S_1 and S_2 are both accessed by k, the ceiling of these two semaphores is at least equal to the priority of k and is then higher than the priority of j_1 or j_2. The above rule ensures that j_1 cannot obtain access to S_1 if S_2 is locked by j_2, and, conversely, that j_2 cannot obtain access to S_2 if S_1 is locked by j_1.

Apart from the new rule for accessing semaphores, the priority ceiling protocol works like the basic priority inheritance protocol. A job k is said to block j if k has lower priority than j and owns a semaphore of ceiling at least equal to the priority of j. Such a job k prevents j from entering a critical section. If j requests access to a semaphore, then j becomes blocked and k inherits j's priority. An essential property of the protocol is that j cannot have more than one blocker k.

Figure 2.3 illustrates how the priority ceiling protocol operates. The ceiling of S_1 and S_2 is at least as high as the priority of J_1. At time t_1, the job J_4 of lowest priority obtains access to S_1. At time t_2, job J_3 attempts to lock semaphore S_2 but is denied access to S_2 even though S_2 is free. Instead, J_3 is blocked because the semaphore S_1 locked by J_4 has higher ceiling that J_3's priority. Since J_4 blocks J_3, J_4 inherits J_3's priority and runs until it is preempted by J_1. J_1 is blocked by J_4 when executing $P(S_2)$ and, at this point, J_4 inherit J_1's priority. J_4 keeps J_1's priority until it releases semaphore S_1. In the interval $[t_3, t_4]$, priority inheritance prevents J_2 from preempting J_4 and indirectly delaying J_1. The figure also shows that the priority ceiling protocol avoids a deadlock that would occur if J_1 were allowed to lock S_2 at time t_3.

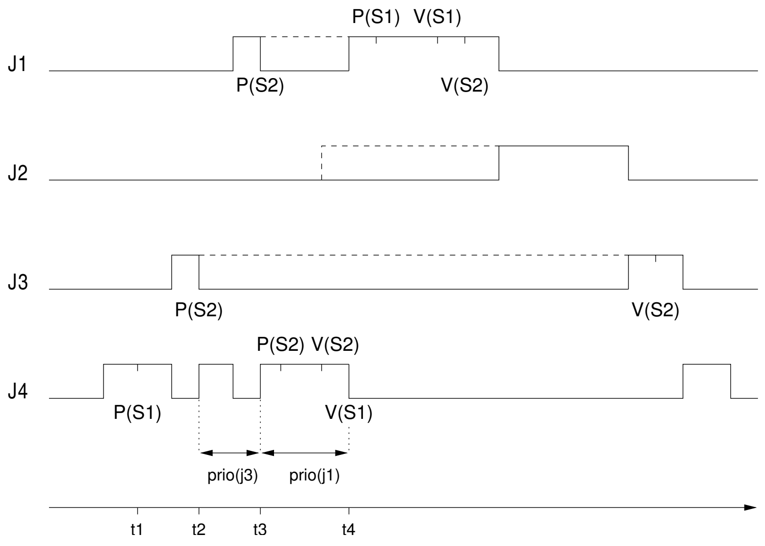

Figure 2.3: Priority Ceiling Protocol

2.5 Formalizing the Protocol

The priority ceiling protocol is fairly subtle, and it is difficult to give an informal description that is precise and complete enough to enable rigorous analysis. In particular, it is difficult to specify precisely how job priorities should be adjusted on entering or exiting critical sections. In this respect, the short description given above is far from complete. Sha et al. [34] give a better definition, including rules for managing resources and priorities when $V(S)$ is executed, but the specifications are still very informal and may be erroneous in places.[1]

The objective of this paper is to develop a PVS model of the protocol suitable not only for proving absence of deadlock or mutual exclusion, but also for developing various schedulability results. We start by building a state-machine model of the protocol behavior for a fixed but arbitrary set of jobs. We then show that the model satisfies the following key properties: distinct jobs cannot be allocated the same semaphore at the same time (mutual exclusion), every job can have at most one blocker, and blocking is intransitive (which implies that deadlocks cannot occur).

Schedulability analysis proceeds by examining the execution traces of the state-machine model. A bound on the worst-case blocking time that a job can encounter is obtained from the single-blocker property. General results can then be established, concerning the allocation of processing time to jobs according to priority and blocking. These general properties do not refer to tasks or deadlines but are the basis for further schedulability analysis.

In a last step, we apply the general results to the specific case of sporadic tasks. From assumptions on the inter-arrival time between successive jobs of a task and the maximal

[1] Sha et al. [34] erroneously state that when a job exits a critical section, it should resume the priority it had on entering the critical section.

length of each job, we obtain standard schedulability conditions that ensure that all the jobs meet their deadline. The proof essentially reproduces a fairly standard argument based on calculating the worst-case response time for jobs at a given priority level.

The PVS formalization is presented in the following chapters. Some familiarity with the PVS notation is assumed. More information about the PVS language and system, including manuals [25–27] and tutorials [8, 9, 31], can be obtained via the PVS web site: `http://pvs.csl.sri.com`.

Chapter 3

The Protocol in PVS

A set of PVS theories defines the basic concepts used to build the protocol model. These theories introduce types to represent priorities, semaphores, and programs, and define related notions such as the critical sections of a program. The protocol is modeled as a state machine and we show that the machine satisfies various invariant properties, including mutual exclusion and the uniqueness of blockers.

3.1 Basic Notions

3.1.1 Priorities and Semaphores

Task and job priorities are represented by integers in the range i$0, \dots,$ maxprio $- 1$ where maxprio is a positive constant; 0 is the lowest priority and maxprio $- 1$ the highest. An uninterpreted type is used to represent semaphores, and a fixed function ceil assigns a ceiling to every semaphore. The corresponding PVS definitions are shown in Figure 3.1. The type below(maxprio) corresponds to the natural numbers smaller than maxprio[1] and rsrc_set is an abbreviation for set[semaphore], the type of sets of semaphores. For simplicity, all these basic notions are fixed throughout the whole PVS development. When necessary, constraints relating job priorities and semaphore ceilings are introduced as assumptions on programs and jobs.

3.1.2 Programs and Critical Sections

To obtain timeliness guarantees in real-time systems, one must assume that jobs have finite length. The runtime activity of a job can then be modeled as a finite sequence of commands or instructions performed by the processor. As far as the priority ceiling protocol is concerned only two classes of commands are relevant, namely, the instructions of the form $P(S)$ or $V(S)$ for requesting and releasing semaphores. We also use a third class of commands representing any other instruction performed in one step. These three classes of commands are represented as an abstract datatype in PVS:

[1]For any natural number n, below(n) is defined as {x:nat | x<n}.

```
basic_types : THEORY

  BEGIN

  %--------------------
  %  Priority of jobs
  %--------------------

  maxprio: posnat

  priority: NONEMPTY_TYPE = below(maxprio)


  %--------------------------
  %  Semaphores and ceiling
  %--------------------------

  semaphore: NONEMPTY_TYPE

  ceil: [semaphore -> priority]

  rsrc_set: TYPE = set[semaphore]

  END basic_types
```

Figure 3.1: Priorities, Semaphores, and Ceiling

```
command : DATATYPE
  BEGIN
    IMPORTING basic_types
    P(request: semaphore): P?
    V(sem: semaphore): V?
    Step: step?
  END command.
```

A program is a nonempty finite sequence of commands. A program p of length l can then be represented as an array of commands numbered from 0 to $l-1$. Given an integer i in the range $0, \dots, l$, we denote by $\mathit{needs}(p, i)$ the set of semaphores p owns after successfully executing the commands $0, \dots, i-1$. More precisely, a semaphore S belongs to $\mathit{needs}(p, i)$ if the command $P(S)$ occurs at some point j of p such that $j < i$ and the command $V(S)$ does not occur in the interval $[j+1, i-1]$. The set $\mathit{needs}(p, i)$ allows us to define critical sections as follows:

- A critical section of p is an interval $[i, j)$ where $i < j \leqslant l$ and $\mathit{needs}(p, k) \neq \emptyset$ for all k such that $i \leqslant k < j$.

- A critical section $[i, j)$ is said to be of level n if, for all k such that $i \leqslant k < j$, $\mathit{needs}(p, k)$ contains a semaphore of ceiling at least n.

According to these definitions, any nonempty subinterval of a critical section is also a critical section. A critical section is maximal if it is not a proper subinterval of another critical section. Proper nesting of critical sections is not required. A maximal critical section is enclosed by two commands $P(S)$ and $V(S')$ where S and S' may be distinct. For example, a program of the form

$$\dots P(S_1) \dots P(S_2) \dots V(S_1) \dots V(S_2) \dots$$

contains a maximal critical section starting after the request for S_1 and ending after the release of S_2. This critical section is of level n for any n that is smaller than the ceiling of S_1 and of S_2.

The concrete PVS definition of programs, critical sections, and related notions is given in theory `programs`. Figure 3.2 shows an excerpt from this theory. A program `p` is a record of type `prog` with two components: a length `length(p)` and a list of commands `clist(p)`.[2] The definition uses PVS's dependent types: `clist(p)` is a mapping whose domain, `below(length(p))`, depends on the first component of `p`. The important functions and predicates defined in theory `programs` are:

- `cmd(p, i)`: Command of index `i`.
- `complete(p, i)`: `p` is finished after `i` steps.
- `needs(p, i)`: Set of semaphores owned by `p` after `i` steps.
- `resources(p)`: Set of semaphores used by `p`.
- `well_behaved(p)`: `p` releases all its semaphores on termination.
- `cs(p, i)`: `p` is in a critical section after `i` steps.
- `cs(p, i, n)`: `p` is in a level-n critical section after `i` steps.
- `critical_section(p, i, j)`: $[i, j)$ is a critical section of `p`.
- `critical_section(p, i, j, n)`: $[i, j)$ is a level-n critical section of `p`.
- `max_cs(p)`: Length of the longest critical section of `p`.
- `max_cs(p, n)`: Length of the longest level-n critical section of `p`.

A semaphore `s` belongs to `resources(p)` if the command `P(s)` occurs in `clist(p)`. In case `p` does not use any semaphore, or any semaphore of ceiling at least `n`, then `max_cs(p)` or, respectively, `max_cs(p, n)`, is equal to 0.

In addition to the above functions, the type `pc(p)` denotes integers in the range $0, \dots,$ `length(p)`. Such indices play the role of program counters in the protocol model.

An expression of the form `cmd(p, i)` is type-correct if the index `i` is of type `below(length(p))`, that is, in the range $0, \dots,$ `length(p)` $- 1$. Similarly, in all the

[2]PVS 2.3 offers an alternative syntax for record components: `length(p)` and `clist(p)` could be written `p`length` and `p`clist`.

```
programs: THEORY

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

  p: VAR prog
  s: VAR semaphore
  n: VAR priority

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

  complete(p, (i: pc(p))): bool = i = length(p)

  cmd(p, (i: below(length(p)))): command = clist(p)(i)

  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

  ...

  well_behaved(p): bool = empty?(needs(p, length(p)))

  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 3.2: Programs and Associated Notions

```
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 3.3: Parameters to the `priority_ceiling` Theory

other expressions above, `i` and `j` must be of type `pc(p)`, that is, integers between 0 and `length(p)`. The PVS type system enforces these constraints by generating proof obligations known as type-correctness conditions (TCCs). The TCC mechanism is described in [25].

3.1.3 Jobs

The priority ceiling protocol is specified in a parameterized theory `priority_ceiling` with the parameters representing a fixed set of jobs and their attributes. Each job is characterized by its dispatch time, its priority, and its program, that is, a finite list of commands as described previously. The corresponding declarations and assumptions are shown in Figure 3.3. We use discrete time: the dispatch time of a job is a natural number. 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.

Using these parameters, a precedence relation on jobs is defined as follows:

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

This 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. A first useful property is that every nonempty set of jobs has a maximal element with respect to this precedence relation:

```
A: VAR (nonempty?[job])

topjob_exists: LEMMA
   EXISTS (j: (A)): FORALL (k: (A)): precedes(j, k).
```

There can be more than one maximal element since two jobs can have the same priority and be dispatched at the same time.

3.2 State-Machine Model

The state-machine specifications are organized in two parts: a resource management subsystem controls the allocation of semaphores to jobs, and a scheduling subsystem selects the job to activate.

3.2.1 Resource Management

The global state of the system contains a resource management component that stores the set of semaphores allocated to or requested by each job. This component is a record of type `rsrc_state` defined in Figure 3.4. 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.[3] 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 `request(r, j)` are nonempty. Informally, this happens if `j` has attempted to enter a critical section in a state `r0` where `blk(r0, j)` was not empty. Job `j` remains blocked until a state `r1` is reached where `blk(r1, j)` is empty.

Three operations control semaphore allocation and deallocation:

- `alloc_step(r, j, s)` executes command `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 command `V(s)` on behalf of `j`. It simply removes `s` from the set of semaphores allocated to `j`. After this operation, some 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 intended for a job `j` that is reactivated after having been blocked.

[3]The expressions `` r`alloc(j) `` and `` r`request(j) `` use the new PVS2.3 syntax for record components. In previous versions of PVS, these expressions would be written `alloc(r)(j)` and `request(r)(j)`, respectively.

```
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 3.4: Semaphore Allocation and Blocking

3.2.2 Scheduler

The state-machine that models the priority ceiling protocol operates on a global state:

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

Every state includes a resource management component, a global time counter, and a program counter for every job. The definition uses dependent types: for every job `j` and every `q` of type `sch_state`, the program counter `q`pc(j)` is of type `pc(prog(j))`. In other words, if the program associated with job `j` is of length $n$ then `q`pc(j)` is in the range $0, \ldots, n$. A program counter between 0 and $n-1$ points to the next instruction to be executed by `j`, and a counter equal to n indicates that `j` is complete.

The time counter is incremented with every transition of the machine. Each transition corresponds either to an idle step when there is no job ready or to a single instruction

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

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 3.5: Eligible Jobs

```
run_step(r, j, cmd): rsrc_state =
    CASES cmd OF
       P(s): alloc_step(wakeup(r, j), j, s),
       V(s): release_step(wakeup(r, j), j, s),
       Step: wakeup(r, j)
    ENDCASES

step(q, (j | ready(q, j))): sch_state =
      (# rsrc := run_step(q`rsrc, j, cmd(prog(j), q`pc(j))),
         pc   := q`pc WITH [(j) := q`pc(j) + 1],
         time := q`time + 1 #)
```

Figure 3.6: Step Function

performed by the active job. For simplicity, we have not modeled delays caused by context switching, that is, we assume that changing active jobs is instantaneous.

The remainder of the scheduler specification is shown in Figure 3.5 and defines which jobs are ready to execute and which jobs can be activated in a state `q`. A job `j` is ready to execute in `q` if the time in `q` is larger than the dispatch time of `j` and `j` is not completed in `q`. A job `j` can be activated in state `q` either if it is a ready job of highest precedence and is not blocked or if it is currently blocking a job of highest precedence.

3.2.3 Next State

If job `j` is activated in state `q`, then the resulting successor state is defined by the function `step` shown in Figure 3.6. First, function `wakeup` is applied to the resource management component of `q`. If `j` was blocked, `j`'s pending requests are granted; otherwise, `wakeup` has no effect. The command indicated by `` q`pc(j) `` is then executed, and both the time counter and `j`'s program counter are incremented.

The function `step` requires two arguments `q` and `j` such that `ready(q, j)` holds. This constraint on the domain of `step` ensures that the function is well defined. When typechecking the definition, PVS generates the two following TCCs:

```
step_TCC1: OBLIGATION
  FORALL (q, j: job | ready(q, j)): q`pc(j) + 1 <= prog(j)`length;

step_TCC2: OBLIGATION
  FORALL (q, j: job | ready(q, j)): q`pc(j) < prog(j)`length;
```

The first one ensures that the result of `step(q, j)` is effectively a record of type `sch_state`. This amounts to checking that the expression

```
q`pc WITH [(j) := q`pc(j) + 1]
```

is of the expected dependent type, which reduces to showing that ``q`pc(j) + 1`` is smaller than or equal to `length(prog(j))`. The second TCC is caused by the occurrence of ``cmd(prog(j), q`pc(j))`` in the definition of `step`. This expression requires ``q`pc(j)`` to be of type `below(length(prog(j)))`. The assumption that `ready(q, j)` holds is sufficient to prove the two TCCs.

3.2.4 Complete Protocol

The resource manager and the scheduler define the basic functions necessary to specify the protocol. It remains to define the initial state and transition relation. The initial resource management and scheduler states are defined as follows:

```
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 #)
```

Initially, program counters point to the first command of each program, and the allocation and request sets are all empty.

The transition relation needs to distinguish two cases. If no job is eligible in state `q`, then no job is active in `q` and the next state, `idle_step(q)`, is obtained by simply incrementing the time counter:

```
idle(q): bool = not EXISTS j: eligible(q, j)

idle_step(q): sch_state = q WITH [time := q`time + 1]
```

If there are jobs eligible in `q`, then one of them is activated and the next state is `step(q, j)` where `j` is the job selected. A tentative definition of the transition relation could then be as follows:

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

Unfortunately, this definition does not typecheck. Since the function `step` requires `j` to be ready in `q1`, we must show that any job that is eligible in `q1` is also ready in `q1`. This is true if `j` satisfies `topjob(q1, j)`, but `j` can also be eligible in `q1` by blocking a job `k` of top priority. In this case, we know that `j` belongs to `blk(q1, k)` and that `topjob(q1, k)` holds. This is not sufficient to show that `j` is ready in `q1`.

The solution is to restrict our attention to the states that are reachable from `init_sch`. We want to show that, in such states, every eligible job is ready. For this purpose, we are looking for a predicate `P` on `sch_state` that satisfies the following properties for all `q` and `j`:

```
P1: P(q) AND eligible(q, j) IMPLIES ready(q, j)

P2: P(init_sch)

P3: P(q) AND idle(q) IMPLIES P(idle_step(q))

P4: P(q) AND eligible(q, j) IMPLIES P(step(q, j)).
```

This technique is the usual approach for proving invariants of a transition system. Condition `P1` ensures that, in all the states satisfying `P`, eligibility implies readiness. Conditions `P2` to `P4` show that `P` is an inductive invariant: `P` is true initially and is preserved by legal transitions. If conditions `P1` to `P4` are satisfied, then all the reachable states satisfy `P` and every job eligible in a reachable state is ready.

An adequate `P` is shown in Figure 3.7. The first clause in the definition of `P` states that the set of semaphores owned or requested by a job `j` in state `q` is equal to `needs(prog(j), q`pc(j))`. The second clause states that the program counter of a job that has not started is zero. In conjunction with assumption `good_programs`, predicate `P` implies that a job that has not started yet or has already finished cannot own a semaphore. As a consequence, a job that is not ready cannot block any other job and then cannot be eligible.

The lemmas in Figure 3.7 are used to prove that `P` is invariant. Conditions `P3` and `P4` above are replaced by stronger lemmas `idle_P` and `step_P` that are easier to prove. It is also trivial to show that `init_sch` satisfies `P`. Once the invariance of `P` has been established, we define the type `good_states` to denote the states that satisfy `P`, and it is straightforward to prove the following properties:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  Invariant for type correctness         %
%   ensures that eligible jobs are ready  %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

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)


%-------------------
%  Invariance of P
%-------------------

alloc_P: LEMMA
   union(alloc_step(r, j, s)`alloc(k),
         alloc_step(r, j, s)`request(k)) =
      IF j=k THEN add(s, union(r`alloc(k), r`request(k)))
             ELSE union(r`alloc(k), r`request(k)) ENDIF

wakeup_P: LEMMA
   union(wakeup(r, j)`alloc(k), wakeup(r, j)`request(k)) =
      union(r`alloc(k), r`request(k))

release_P: LEMMA
   union(release_step(r, j, s)`alloc(k),
         release_step(r, j, s)`request(k)) =
      IF j=k THEN union(remove(s, r`alloc(k)), r`request(k))
             ELSE union(r`alloc(k), r`request(k)) ENDIF


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

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


%-------------------------
%  States that satisfy P
%-------------------------

good_state: NONEMPTY_TYPE = (P)
```

Figure 3.7: Good States

```
g, g1, g2: VAR good_state

alloc_not_ready: LEMMA
   not ready(g, j) IMPLIES empty?(g`rsrc`alloc(j))

eligible_ready: LEMMA eligible(g, j) IMPLIES ready(g, j)

ceiling_prop1: LEMMA
   member(s, g`rsrc`alloc(j)) IMPLIES prio(j) <= ceil(s)

ceiling_prop2: LEMMA
   member(s, g`rsrc`request(j)) IMPLIES prio(j) <= ceil(s)
```

Lemma `eligible_ready` is the essential result: in any state that satisfies `P`, eligible jobs are also ready. The proof relies on lemma `alloc_not_ready`, which says that a job that has not started yet or is already finished cannot hold any resource. From assumption `good_ceiling` we also immediately obtain that in all good states the semaphores allocated to `j` or requested by `j` have ceiling at least as high as `j`'s priority.

The protocol specifications can now be completed as shown in Figure 3.8. The transition relation `T` is the same as shown above but restricted to `good_states`. This ensures that the definition of `T` typechecks. PVS generates a TCC that is easily discharged using lemma `eligible_ready`.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%
%   Transition system    %
%%%%%%%%%%%%%%%%%%%%%%%%%%

%------------------
%  initial state
%------------------

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

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

%------------------------
%  transition relation
%------------------------

T(g1, g2): bool = (idle(g1) AND g2 = idle_step(g1))
      OR (EXISTS j: eligible(g1, j) AND g2 = step(g1, j))
```

Figure 3.8: Transition System

3.2.5 Modeling Choices

Our model differs from traditional descriptions of the priority ceiling protocol by avoiding dynamic priorities and priority inheritance. The essential requirements of the protocol are captured by the blocking rules and by the definition of eligible jobs. 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 and the associated timing constraints are irrelevant to the basic mechanisms of the protocol, and our PVS model ignores these aspects. Modeling tasks or processes more explicitly, as is done by Fowler and Wellings [14] or by Pilling et al. [28], 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 real 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 given by the definition of `priority`. There are only finitely many priorities and this ensures that there is always a job of top precedence among the ready jobs. Other finiteness assumptions are not necessary and 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.

The use of dependent types may seem an extra complication. The type of a program counter is constrained to be between zero and the length of the corresponding program. This forces us to prove the invariance of predicate `P` to ensure that the protocol specifications are well typed. The specifications could be written differently to avoid having to discharge a TCC. However, this would not provide any gain. Property `P` is a useful invariant that would have to be proved anyway. Avoiding dependent types would require us to extend the domain of program counters to arbitrary natural numbers but then their actual range would have to be established by other means, that is, by proving another invariant. Using subtypes and dependent types is a more natural approach in PVS and does not require extra work. The constraint on `q`pc(j)` is also known and used by the PVS decision procedures, which makes proofs simpler and more automatic.

3.3 Properties of the Protocol

3.3.1 Inductive Invariants

All the important properties of the protocol are invariant properties. They are proven using several auxiliary inductive invariants. The most important such invariant is specified by predicate `P2` in Figure 3.9. `P2` is a property of resource management components that holds in all reachable states of the model. 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. This property essentially ensures that any job that might be blocked by j cannot have any blocker other than j.

The proof of the invariance of P2 is decomposed in several lemmas shown in Figure 3.9. The main steps are alloc_P2, wakeup_P2, and release_P2, which give sufficient conditions for P2 to be preserved by the three resource management operations. The lemmas rely on an auxiliary predicate P3, which states that all the resources allocated to any job j have ceiling at least equal to the priority of j:

- P2 is preserved by a resource allocation step performed in r, provided P3 holds in r.
- P2 is preserved after reactivation (via wake_up) of a job j in state r, provided P3 holds in r and j is not blocked in r.
- P2 is preserved by any resource release step.

The proofs of these three properties are straightforward. They are obtained by case analysis after expansion of the definition of each of the three operations.

Lemma intransitive_blocking implies that, in a state that satisfies P2, the blocker of a job of top priority cannot be blocked. The proof is again a simple case analysis. Once this result has been established, the invariance of P2 is straightforward since the auxiliary property P3 holds in all good states. P2 is true in the initial state and is preserved by the transition relation T.

Predicate Q is another useful invariant that is straightforward to prove:

```
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)).
```

If j is ready in g, then jobs of lower or equal priority and that have been dispatched after j cannot own any resource.

3.3.2 Essential Protocol Properties

Proving the inductive invariants is the main part of the protocol analysis. The essential protocol properties are direct consequences of P2 and Q.

Absence of deadlocks follows from the fact that blocking is not transitive and that a job cannot block itself. We have already encountered related properties, such as step_P2_aux, which states that eligible jobs cannot be blocked. The absence of deadlocks is actually subsumed by the schedulability results to be presented in the sequel. Showing that all the jobs meet their deadlines implies that they all terminate and then that deadlocks cannot occur.

Mutual exclusion is a straightforward consequence of P2. Assume j and k are distinct jobs and, without loss of generality, that j has lower priority than k. Assume s is a semaphore owned by j. If s has ceiling as high as k's priority, then P2 ensures that k cannot own any resource. Otherwise, s has ceiling lower that k's priority and, in such a case, s is not accessed by k. Hence, the same semaphore cannot be allocated to both j and k: the protocol ensures mutual exclusion.

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

```
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))

P3(r): bool =
   FORALL j, s: member(s, r`alloc(j)) IMPLIES prio(j) <= ceil(s)

%-------------------------
%  Induction step for P2
%-------------------------

alloc_P2: LEMMA P3(r) AND P2(r) IMPLIES P2(alloc_step(r, j, s))

wakeup_P2: LEMMA
    P3(r) AND P2(r) AND not blocked(r, j) IMPLIES P2(wakeup(r, j))

release_P2: LEMMA P2(r) IMPLIES P2(release_step(r, j, s))


%---------------------------------
%  Auxiliary results:
%  - blocking is intransitive
%  - P3 holds in good states
%  - eligible job is not blocked
%---------------------------------

intransitive_blocking: LEMMA
      P2(r) AND member(j1, blk(r, k)) AND prio(j2) <= prio(k)
         IMPLIES NOT member(j2, blk(r, j1))

invar_P2_aux: LEMMA P3(g`rsrc)

invar_P2_aux2: LEMMA P3(wakeup(g`rsrc, j))

step_P2_aux: LEMMA
      P2(g`rsrc) AND eligible(g, j) IMPLIES not blocked(g`rsrc, j)


%--------------------
%  Invariance of P2
%--------------------

init_P2: LEMMA P2(init_rsrc)

step_P2: LEMMA P2(g1`rsrc) AND T(g1, g2) IMPLIES P2(g2`rsrc)
```

Figure 3.9: Main Inductive Invariant

```
%------------------------------------------------------------
%  Job of lower priority than j that can run when j is ready
%------------------------------------------------------------

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 3.10: Definition and Properties of Blockers

blocking bounds can be derived: a job can have at most one blocker, and this blocker is within a critical section of a level equal to `j`'s priority. The PVS developments actually include two variants of these properties. The definition of blockers and associated properties are shown in Figure 3.10.

The set `blockers(r, j)` contains the jobs of lower priority than `j` that own a semaphore of ceiling as high as `j`'s priority. As shown by lemma `unique_blocker`, either this set is empty or it contains a single element `k`. This follows immediately from invariant `P2`. By definition of `blockers`, and because of invariant `P`, the program counter for this job `k` is within a critical section of a level equal to `j`'s priority.

Although `blockers(r, j)` is defined in any state `r`, it is relevant only if `j` is ready to run in `r`. In such a case, `blockers(r, j)` contains the job that prevents `j` from entering a critical section. The eligibility conditions ensure that, in a state `r` where `j` is ready, either the active job has precedence over `j`[4] or it is the unique element of `blockers(r, j)`. Another important property is that the set of blockers of `j` in a successor state of `r` is a subset of the blockers of `j` in `r`. This ensures that once the set `blockers(r, j)` becomes empty, it remains empty until `j` terminates.

A generalization of the set of blockers of `j` is shown in Figure 3.11. The set `blockers(r, p)` contains the jobs of priority less than `p` that own a semaphore of ceiling at least `p`. Properties similar to those above are satisfied by `blockers(r, p)`. This

[4]As we have defined it, the precedence relation is reflexive, so the active job can be `j` itself.

```
%------------------------------------------------------------------
%  Jobs of priority less than p that can block jobs of priority p
%------------------------------------------------------------------

blockers(r, p): set[job] =
   { k | prio(k) < p AND
            EXISTS s: member(s, r`alloc(k)) AND ceil(s) >= p }

unique_blocker2: LEMMA
   P2(r) AND member(j1, blockers(r, p)) AND
      member(j2, blockers(r, p)) IMPLIES j1 = j2

blockers_in_cs2: LEMMA
   member(k, blockers(g`rsrc, p)) IMPLIES
      cs(prog(k), g`pc(k), p)

eligible_prio2: LEMMA
   (EXISTS j: prio(j) >= p AND ready(g, j)) AND eligible(g, k)
      IMPLIES prio(k) >= p OR member(k, blockers(g`rsrc, p))

blockers_step2: LEMMA
   (EXISTS j: prio(j) >= p AND ready(g1, j)) AND T(g1, g2)
      IMPLIES subset?(blockers(g2`rsrc, p), blockers(g1`rsrc, p))
```

Figure 3.11: Generalization of Blockers

generalized notion is actually more useful than `blockers(u, j)` when schedulability properties are being established.

3.3.3 Verification Effort

The protocol properties presented above are all fairly easy to prove with PVS. The only slight difficulty is determining the inductive invariants `P2` and `Q`. The streamlined specification of the protocol is crucial for simplifying the proofs. In particular, eliminating priority inheritance and avoiding the associated priority adjustment rules is an essential simplification. Similarly, the choice of using jobs as parameters makes the notion of “readiness” trivial and simplifies the analysis.

As a result of the PVS verifications, one can see that the protocol works under weaker assumptions than originally made by its authors. Sha et al. assume that the critical sections of a job are properly nested [34], but this requirement is actually unnecessary. All the important properties of the protocol are satisfied even if critical sections overlap.

Chapter 4

General Scheduling Properties

Various PVS developments support schedulability analysis. From the state-machine specifications, we first construct the traces of the protocol. Each trace is an infinite sequence of states representing a possible execution of the protocol. As previously, the traces are parameterized by a fixed set of jobs characterized by their priority, dispatch time, and program.

To obtain schedulability results that are independent of the particular state representation chosen, we map each trace to a more abstract notion of schedule. Schedules are sequences that record the successive active jobs in each state of a trace. Schedulability analysis examines the amount of processing time allocated to a job or a set of jobs in specific intervals of a given schedule.

4.1 Traces

A trace of the protocol is a sequence w of `good_states` such that $w(0)$ is equal to `init_sch` and two successive states of w are related by the transition relation `T`. The corresponding definition is shown in Figure 4.1; `trace` is a nonempty subtype of sequences of `good_states`. Nonemptiness is shown by constructing a sequence `tr` of `good_states` and proving that `tr` satisfies the constraints of `trace`.

The two lemmas `init_trace` and `step_trace` state the two basic properties of traces. Although these properties are only repeating the definition, it is convenient to restate them separately. The two lemmas can be used as rewrite rules and provide short cuts in proofs.[1] The other lemmas of Figure 4.1 state that `Q` and `P2` are satisfied in all the states of a trace `u`.

Some notions defined in `priority_ceiling` are reformulated in a more convenient form that hides the details of the state machine:

[1]For example, a single `(REWRITE "step_trace")` replaces a `(TYPEPRED ...)` command followed by propositional simplifications and quantifier instantiation.

```
%-----------------------
%  Existence of traces
%-----------------------

next_state_exists: LEMMA EXISTS g2: T(g1, g2)

tr(t): RECURSIVE good_state =
   IF t=0 THEN init_sch ELSE epsilon! g: T(tr(t-1), g) ENDIF
 MEASURE t

%----------
%  traces
%----------

w: VAR [nat -> good_state]

trace: NONEMPTY_TYPE =
   { w: [nat -> good_tate] | w(0) = init_sch AND FORALL t: T(w(t), w(t+1)) }
      CONTAINING tr

u, v: VAR trace

init_trace: LEMMA u(0) = init_sch

step_trace: LEMMA T(u(t), u(t+1))

%-------------------
%  Main invariants
%-------------------

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

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

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

Figure 4.1: Traces

```
pc(u, j, t): pc(prog(j)) = u(t)'pc(j)

ready(u, j, t): bool = ready(u(t), j)

blockers(u, j, t): set[job] = blockers(u(t)`rsrc, j)

busy(u, p, t): bool = EXISTS j: prio(j) >= p AND ready(u, j, t)

busy(u, p, t1, t2): bool =
   FORALL t: t1 <= t AND t <= t2 IMPLIES busy(u, p, t)

...
```

We also define a predicate that indicates whether job `j` is active in trace `u` at time `t`. It is trivial 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 that follow immediately from these definitions and the results established in `priority_ceiling` are also included. For example, the lemmas shown in Figure 4.2 summarize the important properties of blockers.

Redefining functions and predicates from `prio_ceiling` and restating the associated properties in a slightly different form is not absolutely necessary. However, it has the advantage of making the development of further properties of traces 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.

4.2 Schedules

A schedule records the sequence of active jobs and idle steps corresponding to a particular trace. The PVS definition relies on a parameterized datatype `some_or_none[T]` defined as follows:

```
some_or_none[T: TYPE] : DATATYPE
  BEGIN
    none: none?
    some(the_one: T): some?
  END some_or_none.
```

A schedule is a sequence `sch` of elements of type `some_or_none[job]`. The value `sch(t)` is either of the form `some(j)` to indicate that job `j` is active at time `t` or equal to `none` to indicate that no job is active at that time.

```
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)).


active_prio2: LEMMA
   busy(u, p, t) AND active(u, k, t) IMPLIES
      prio(k) >= p OR member(k, blockers(u, p, t))

single_blocker2: LEMMA
   member(j1, blockers(u, p, t)) AND member(j2, blockers(u, p, t))
      IMPLIES j1 = j2

blocker_in_cs2: LEMMA
   member(k, blockers(u, p, t)) IMPLIES cs(prog(k), pc(u, k, t), p)

blocker_step2: LEMMA
   busy(u, p, t) IMPLIES
      subset?(blockers(u, p, t+1), blockers(u, p, t))
```

Figure 4.2: Trace-level Formulation of Blocker Properties

4.2.1 Measuring Processing Time

The functions that are essential for scheduling analysis measure the amount of processing time allocated to a job or set of jobs in an interval $[t_1, t_2)$. The most general form of such functions is

```
process_time(sch, t1, t2, E): nat =
   sum(sch, t1, t2, { x | some?(x) AND E(the_one(x)) })
```

It relies on a generic `sum` function defined for sequences of arbitrary types. If `u` is of type `[nat -> T]` and `E` is of type `set[T]` then `sum(u, t1, t2, E)` counts the elements among `u(t1)`,..., `u(t2-1)` that belong to `E`. If `t2` is not larger than `t1`, the result is zero:

```
sum(u, t1, t2, E): RECURSIVE nat =
   IF t2 <= t1 THEN 0
      ELSIF E(u(t2-1)) THEN 1 + sum(u, t1, t2-1, E)
       ELSE sum(u, t1, t2-1, E) ENDIF
  MEASURE max(t2 - t1, 0)
```

Given a schedule `sch` and a set of jobs `E`, `process_time(sch, t1, t2, E)` is then the amount of processing time allocated to jobs of `E` between `t1` and `t2-1`.

Variants of this function are given for the case where `E` is a singleton and the case where `t1=0`:

```
process_time(sch, t, E): nat =
   process_time(sch, 0, t, E)

process_time(sch, t1, t2, j): nat =
   process_time(sch, t1, t2, singleton(j))

process_time(sch, t, j): nat =
   process_time(sch, t, singleton(j)).
```

In a similar way, function `idle_time` measures the amount of time where no job is active:[2]

```
idle_time(sch, t1, t2): nat = sum(sch, t1, t2, none?)

idle_time(sch, t): nat = idle_time(sch, 0, t).
```

4.2.2 Support Theories

A large part of the overall PVS developments provide various properties of the functions `process_time` and `idle_time`. A typical example of such properties is

```
E: VAR set[job]
A: VAR finite_set[index]
F: VAR [index -> set[job]]

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

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

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

[2]In PVS, sets are represented by predicates. The datatype recognizer `none?`, which is of type `[some_or_none -> bool]`, can be used as a set expression and it is equal to the singleton set {`none`}.

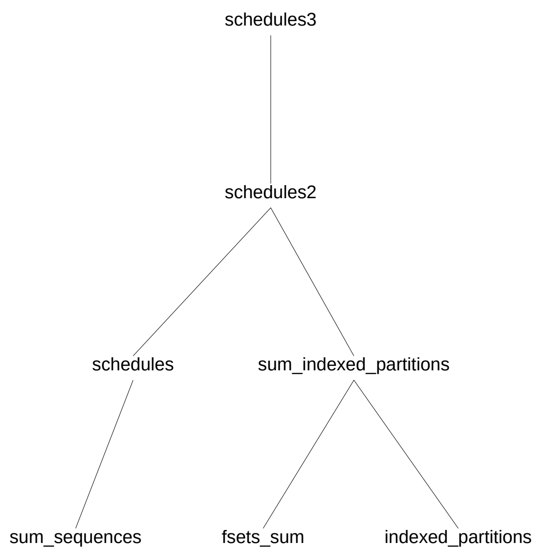

Figure 4.3: Support Theories

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 also provides elementary results, but they were not sufficient and the library had to be extended.[3] Other basic notions such as partitions and sums over sequences had to be defined from scratch.

An overview of the hierarchy of support theories is given in Figure 4.3. Each node in the graph represents a PVS theory, and the edges indicate theory dependencies. The theories shown contain a total of 134 lemmas and theorems. The most important properties are in theories `schedules`, `schedules2`, and `schedules3`. These three theories establish useful properties of function `process_time` that are very similar to `process_time_partition` above. Examples are shown in Figure 4.4.

These support theories have been developed in a systematic way and are intended to be general and reusable in different contexts. In particular, all the results related to finite sets and sums over sequences could be useful in completely different PVS applications. On the other hand, the support theories are not minimal and many theorems are not actually used in the schedulability analysis.

4.3 Bounds on Allocation of Processing Time

With a given trace `u` of the protocol model, one can immediately associate a schedule `sch(u)`. Given a time `t` and a job `j`, we have

[3]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 judgments that were as powerful.

```
total_cpu: LEMMA
   t1 <= t2 IMPLIES
      process_time(sch, t1, t2, fullset) +
         idle_time(sch, t1, t2) = t2 - t1

max_process_time: LEMMA
   t1 <= t2 IMPLIES process_time(sch, t1, t2, E) <= t2 - t1

split_process_time: LEMMA
   t1 <= t2 AND t2 <= t3 IMPLIES
      process_time(sch, t1, t2, E) +
         process_time(sch, t2, t3, E) = process_time(sch, t1, t3, E)

increasing_process_time: LEMMA
   t1 <= t2 AND t2 <= t3 IMPLIES
      process_time(sch, t1, t2, E) <= process_time(sch, t1, t3, E)

process_time_subset: LEMMA
   subset?(E1, E2) IMPLIES
      process_time(sch, t1, t2, E1) <= process_time(sch, t1, t2, E2)
```

Figure 4.4: Example Lemmas from Theory `schedule`

```
sch(u)(t) = some(j) IFF active(u, j, t),
```

and `sch(u)(t) = none` when there is no active job at time `t` in `u`. General results establish lower and upper bounds on the amount of processing time allocated to a job or a set of jobs in `sch(u)`. These bounds will allow us to establish schedulability conditions, that is, sufficient conditions to ensure that all the jobs will meet specified deadlines.

The following properties relate the amount of processing time allocated to a job `j` and the program counter of `j`:

```
process_time1: LEMMA process_time(sch(u), t, j) = pc(u, j, t)

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

Using these lemmas and the properties of blockers shown in Figure 4.2, we can bound the blocking time of a job `j`:

```
blocker(u, j): set[job] = blockers(u, j, dispatch(j))

...

blockers_in_critical_section: LEMMA
  ready(u, j, t) AND member(k, blocker(u, j)) AND t1=dispatch(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).
```

As defined previously, `blockers(u, j, t)` is the set of jobs of priority less than `j`'s and that own a semaphore of ceiling at least as high as `j`'s priority at time `t`. The bound on blocking is obtained by examining the set `blockers(u, j, t)` at times `t` where `j` is ready to run.

First, `blocker(u, j)` denotes the above set at the time when `j` is dispatched. Using the previous properties of `blockers`, either this set is empty or it contains a single element denoted by `the_blocker(u, j)`. From lemma `blocker_step`, one can also show that as long as `j` is ready, the set `blocker(u, j, t)` stays equal to `blocker(u, j)` or becomes empty. Since any element of `blockers(u, j, t)` is in a critical section of level `prio(j)` at time `t`, the blocking time for `j` cannot be longer than the longest critical section of `the_blocker(u, j)` of the same level.

A second lemma allows us to evaluate the amount of processing time allocated to a job j in an interval [t1,t2], where t1 is j's dispatch time and j is not completed at time t2.

```
H(j): set[job] = { k | k /= j AND precedes(k, j) }

process_time_ready_job: LEMMA
   ready(u, j, t2) AND t1 = dispatch(j) IMPLIES
      process_time(sch(u), t1, t2, j) = (t2 - t1)
        - process_time(sch(u), t1, t2, H(j))
        - process_time(sch(u), t1, t2, blocker(u, j)).
```

As long as j is ready, there is an active job. This job is either j itself, or the blocker of j, or a job that has precedence over j. The lemma above follows from this fact and elementary properties of sums that are given in support theories.

In conjunction with the bound on blocking, the latter lemma gives a lower bound on the amount of processor time allocated to a job j in the interval [t1, t2]. A similar result is proven in exactly the same way for the generalized notion of blockers:

```
K(p): set[job] = { j | prio(j) >= p }

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).
```

In this property, blocking(u, p, t1) refers to the longest critical section of level p of the unique job that belongs to blockers(u, p, t1). In the interval [t1,t2], there is always a job of priority at least p ready to run. The lemma gives a lower bound on the processing time allocated to jobs of priority at least p in this interval.

Chapter 5

Schedulability of Sporadic Tasks

We use the preceding results we have described to derive a sufficient schedulability condition for a set of sporadic tasks. A sporadic task τ consists of jobs separated by a minimal inter-arrival time T. Denoting by τ_1, $\tau_2, \ldots$ the successive jobs of τ and by $\texttt{dispatch}(\tau_i)$ the dispatch time of job τ_i, we have

$$\forall i,\ \texttt{dispatch}(\tau_{i+1}) - \texttt{dispatch}(\tau_i) \geqslant T.$$

A fixed deadline D is assigned to τ and the timing requirement is to ensure that every job of τ is completed no later than D time units after it is dispatched. Job τ_i must then be finished at time $\texttt{dispatch}(\tau_i) + D$ or earlier.

Sporadic tasks can be used to model interrupt-driven activities if the interrupt signals have a known maximal frequency. The deadline associated with a task specifies the urgency of the associated interrupt. Periodic activities can also be represented by sporadic tasks since a periodic task of period T can be considered as a special case of sporadic tasks where $\texttt{dispatch}(\tau_{i+1}) - \texttt{dispatch}(\tau_i) = T$.

We assume that a finite set of sporadic tasks is given. Each task is characterized by its job inter-arrival time and its deadline. The tasks have a fixed priority, and the length of all the jobs in a task is bounded. The tasks are scheduled according to the priority ceiling protocol. The problem is to obtain sufficient conditions to guarantee a priori that all the jobs meet their deadlines.

5.1 Task Model

In PVS, we denote by `nbtasks` the number of tasks. Each task is represented by a number in the interval $[0, \texttt{nbtasks} - 1]$. A task `i` in this interval is characterized by the following parameters:

- `prio(i)`: the task priority
- `T(i)`: the minimal inter-arrival delay between successive jobs of task `i`
- `D(i)`: the deadline of task `i`
- `C(i)`: the maximal length of the jobs of task `i`.

Since we use discrete time, the latter three constants are positive natural numbers.

We represent the jobs corresponding to a set of tasks by pairs of integers: a job $\mathtt{j} = (\mathtt{i}, \mathtt{n})$ is the `n`-th job of task `i`. As required by the priority ceiling model, each job has the following attributes:

- `prio(i, n)`: job priority
- `dispatch(i, n)`: dispatch time
- `prog(i, n)`: program

The priority of a job is the priority of the task to which it belongs, that is, `prio(i, n) = prio(i)`. Finally, there is a constant `B(p)` that bounds the blocking delay that jobs of priority `p` can experience.

The assumptions we make about these parameters are listed in Figure 5.1. The dispatch time between two successive jobs of task `i` is at least the delay `T(i)`. The length of a job of task `i` is no more than `C(i)`. The jobs of priority less than `p` have no critical section of level `p` longer than `B(p)`. The priority of jobs is consistent with the ceiling of semaphores and every job is well-behaved, that is, it releases all the semaphores it uses on termination.

We also assume that the total processor utilization is less than 1, that is,

$$\sum_{i=0}^{\mathtt{nbtask}-1} \frac{C(i)}{T(i)} \quad < \quad 1.$$

If the total utilization is greater than 1 then the task set requires more processing resource than is available, and some jobs never terminate. The task set can be feasible in case the total utilization is equal to 1, but only in very particular circumstances.[1] In the general case, it is reasonable to assume that the processor utilization is strictly less than 1.

We also use a more technical assumption [2] to simplify the PVS verifications. We assume that at least one task is given the highest priority possible:

```
topprio_is_used: ASSUMPTION EXISTS i: prio(i) = maxprio - 1.
```

This does not cause any loss of generality and avoids dealing with a special case.

5.2 Schedulability Analysis

Our model of the priority ceiling protocol defines the possible traces and schedules corresponding to the execution of a set of jobs satisfying the above assumptions. Using the above notations, the deadline for a job `j=(i, n)` is equal to `dispatch(j)+D(i)`. Task `i` is said to be feasible if all the jobs of task `i` terminate before their deadline in all possible executions. In many cases, the protocol is deterministic and there is actually only one execution possible. However, multiple traces are possible if several tasks have the same priority and contain jobs that are dispatched at the same time.

[1]This is possible only if the tasks are periodic, their periods are harmonic, and other conditions on job lengths are satisfied.

[2]Some people might call that a hack.

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

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

blocking: ASSUMPTION prio(j) < p IMPLIES max_cs(prog(j), p) <= B(p)


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

good_programs: ASSUMPTION  well_behaved(prog(j))


cpu_usage: ASSUMPTION sum(fullset[task], lambda i: C(i)/T(i)) < 1
```

Figure 5.1: Model of Sporadic Tasks

To determine whether a task i is feasible, one can compute its worst-case response time M(i), that is, the worst-case execution time of a job of task i. The task is feasible if $\mathtt{D(i)} \leqslant \mathtt{M(i)}$. This general approach to determining feasibility by computing response times was initiated by Joseph and Pandya [17] and by Harter [16]. Algorithms for computing M(i) have been proposed for different classes of jobs and with different assumptions about deadlines [3, 5, 6, 15, 35, 37]. In case D(i) is no more than T(i) and all the tasks have different priorities, M(i) is the smallest solution of the following equation:

$$\mathtt{C(i)} + \mathtt{B(p)} + \sum_{l \in \mathtt{H(i)}} \mathtt{C}(l) \times \left\lceil \frac{\mathtt{M(i)}}{\mathtt{T}(l)} \right\rceil \quad = \quad \mathtt{M(i)}, \tag{5.1}$$

where p is the priority of task i and H(i) is the set of tasks of higher priority than i.[3] The fact that the total processor utilization is less than 1 ensures that equation (5.1) has solutions. The smallest one can be obtained by computing the sequence $(u_n)_{n\in\mathbb{N}}$ defined by

$$\begin{aligned} u_0 &= \mathtt{C(i)} + \mathtt{B(p)} \\ u_{n+1} &= \mathtt{C(i)} + \mathtt{B(p)} + \sum_{l \in \mathtt{H(i)}} \mathtt{C}(l) \times \left\lceil \frac{u_n}{\mathtt{T}(l)} \right\rceil \end{aligned}$$

until a fixed point is reached. This fixed point is equal to M(i). Other initial values can be chosen for u_0, as long as the following condition is satisfied:

$$u_0 \quad < \quad \mathtt{C(i)} + \mathtt{B(p)} + \sum_{l \in \mathtt{H(i)}} \mathtt{C}(l) \times \left\lceil \frac{u_0}{\mathtt{T}(l)} \right\rceil .$$

The following sections present PVS proofs of these results. First, we examine equations of a form similar to equation (5.1) above. We show that these equations have solutions under the following assumption:

$$\sum_{l \in \mathtt{H(i)}} \frac{\mathtt{C}(l)}{\mathtt{T}(l)} \quad < \quad 1.$$

We also show that the fixed-point construction sketched above yields the smallest solution. This part of the analysis is independent of the protocol model but relies on several of the support theories used previously.

Using the general bounds on process time and blocking obtained before, we then establish a link between the solutions of equations of the previous form and the length of so-called busy periods. The schedulability condition mentioned above, namely, the fact that the smallest solution of equation (5.1) is the worst-case response time for task i in the case $\mathtt{D(i)} \leqslant \mathtt{T(i)}$, is an easy corollary.

5.3 Ceiling Equations

Figure 5.2 summarizes the main results related to the fixed-point construction described earlier. Given an arbitrary index type U (U is a parameter of the PVS theory), the expression

[3] $\lceil x \rceil$ denotes the ceiling of x, that is, the smallest integer at least equal to x. The same integer is denoted by ceiling(x) in PVS.

```
i: VAR U
A: VAR finite_set[U]
E: VAR non_empty_finite_set[U]
C, T: VAR [U -> posreal]
B: VAR nonneg_real
x: VAR nonneg_real

...

F(C, T, x)(i): nonneg_real = C(i) * ceiling(x / T(i))

...

G(A, C, T, x): nonneg_real = sum(A, F(C, T, x))

...

u(E, B, C, T)(n): RECURSIVE posreal =
    IF n=0 THEN B + sum(E, C)
      ELSE B + G(E, C, T, u(E, B, C, T)(n - 1))
    ENDIF
  MEASURE n

fixed_point: LEMMA
      (EXISTS n: u(E, B, C, T)(n + 1) = u(E, B, C, T)(n))
   OR (EXISTS c: FORALL n: u(E, B, C, T)(n) >= n * c + B + sum(E, C))

least_fixed_point: LEMMA
     B + G(E, C, T, z) = z IMPLIES FORALL n: u(E, B, C, T)(n) <= z

fixed_point_prop: LEMMA
     z < u(E, B, C, T)(n) IMPLIES z < B + G(E, C, T, z)

...

upper_bound1: LEMMA
    sum(E, lambda i: C(i)/T(i)) < 1 IMPLIES
       EXISTS z: G(E, C, T, z) + B <= z

...

smallest_solution2: LEMMA
    sum(E, lambda i: C(i)/T(i)) < 1 IMPLIES
       EXISTS z: G(E, C, T, z) + B = z AND
          (FORALL w: G(E, C, T, w) + B <= w IMPLIES z <= w)
```

Figure 5.2: Ceiling Equations

$G(A, C, T, x)$ is defined as the sum

$$\sum_{i \in A} C(i) \times \left\lceil \frac{x}{T(i)} \right\rceil ,$$

where `A` is a finite set of indices, and `T` and `C` are mappings from `U` to the positive reals. The function `G` is monotonic:

$$x \leqslant y \quad \Rightarrow \quad G(A, C, T, x) \leqslant G(A, C, T, y),$$

and it is bounded as follows:

$$G(A, C, T, x) \;\leqslant\; x \times \sum_{i \in A} \frac{C(i)}{T(i)} + \sum_{i \in A} C(i).$$

Equation (5.1) can be generalized slightly. For a fixed real `B` and a nonempty set of indices E, we consider equations of the form

$$G(E, C, T, z) + B \quad = \quad z, \tag{5.2}$$

where z is a non-negative number. To obtain the smallest solution of this equation if one exists, we can construct the sequence $(u_n)_{n \in \mathbb{N}}$ such that

$$u_0 \quad = \quad B + \sum_{i \in E} C(i)$$

and

$$\begin{aligned} u_{n+1} \quad &= \quad B + G(E, C, T, u_n) \\ &= \quad B + \sum_{i \in E} C(i) \times \left\lceil \frac{u_n}{T(i)} \right\rceil . \end{aligned}$$

Let c be the smallest of all $C(i)$ for $i \in E$; we have either $u_{n+1} = u_n$ or $u_{n+1} \geqslant u_n + c$. It follows that the sequence is either unbounded (i.e., $u_n \geqslant n.c + u_0$ for all n) or it reaches a fixed point (i.e., there exists n such that $u_{n+1} = u_n$). In case

$$\sum_{i \in E} \frac{C(i)}{T(i)} \;<\; 1,$$

the bound on G above implies that there exists z such that

$$G(E, C, T, z) + B \;\leqslant\; z.$$

By construction of u and since G is monotonic, it is easy to see that $u_n \leqslant z$ for all $n \in \mathbb{N}$. As a result, the sequence u is bounded and by the previous remark it must reach a fixed point. This fixed point is the smallest solution of equation (5.2).

The PVS definition of u and the lemmas corresponding to the main stages of this reasoning are shown in Figure 5.2. The function `ceiling` is predefined in the PVS prelude. The rest of the development relies on the generic theories of sums over finite sets and other auxiliary results, such as the fact that a nonempty finite set of reals has a minimal element.

5.4 Schedulability Conditions

Given a fixed set of sporadic traces with the attributes described in Section 5.1, let u be an arbitrary trace obtained by scheduling these tasks using the priority ceiling protocol. Let $[t_1, t_2]$ be an interval such that for any t between t_1 and t_2 there is a job of priority at least p ready to execute at time t in u. The important result used to derive schedulability conditions is lemma busy_time2 of Section 4.3: In the interval $[t_1, t_2]$, we have

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

where K(p) is the set of jobs of priority at least p. The constant blocking(u, p, t1) is defined by

```
blocking(u, p, t): nat =
   IF empty?(blockers(u, p, t)) THEN 0
     ELSE max_cs(prog(the_blocker(u, p, t)), p) ENDIF,
```

so B(p) is larger than or equal to blocking(u, p, t1). We then obtain the following inequality:

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

Now, let K(p, t1, t2) be the set of jobs of priority at least p that are dispatched between t1 and t2:

```
K(p, t1, t2): set[job] =
   { j | prio(j) >= p AND t1 <= dispatch(j) AND dispatch(j) < t2 }.
```

Using various properties of sums over sequences, and the assumptions about job inter-arrival time and job length, one obtains the following important lemma:

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

where A(p) is the set of tasks of priority p or higher. In more standard notations, this lemma states that the amount of processing time allocated in $[t_1, t_2]$ to jobs of priority at least p that started between t1 and t2 is bounded by

$$\sum_{i \in A(p)} C(i) \times \left\lceil \frac{t_2 - t_1}{T(i)} \right\rceil .$$

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. An interval $[t_1, t_2]$ is called a busy period of

```
busy_interval: LEMMA
   quiet(u, p, t1) AND t1 <= t2 IMPLIES
      process_time(sch(u), t1, t2, K(p)) =
          process_time(sch(u), t1, t2, K(p, t1, t2))

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)))

...

critical_interval: PROPOSITION
   quiet(u, p, t1) IMPLIES
      EXISTS t2: t1 < t2 AND t2 <= t1 + M(p) AND quiet(u, p, t2)

delay_to_quiet_time: LEMMA
   FORALL t: EXISTS t2: quiet(u, p, t2) AND t < t2 AND t2 <= t + M(p)

busy_period_length: LEMMA
   busy_period(u, p, t1, t2) IMPLIES t2 - t1 <= M(p)
```

Figure 5.3: Bounding the Length of Busy Periods

level p if t_1 and t_2 are quiet times, there is no quiet time between t_1 and t_2, and one job of priority at least p is ready at time t_1:

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

busy_period(u, p, t1, t2): bool =
  t1 < t2 AND busy(u, p, t1) AND quiet(u, p, t1) AND quiet(u, p, t2) AND
    FORALL t: t1 < t AND t < t2 IMPLIES not quiet(u, p, t)

busy_period_prop: LEMMA
  busy_period(u, p, t1, t2) AND t1 <= t AND t < t2 IMPLIES busy(u, p, t).
```

All along the interval $[t_1, t_2]$ there is a job of priority at least p ready to execute. All these jobs have started at or after t_1 and strictly before t_2 and all these jobs are finished at time t_2. The schedulability result presented above can now be proven by bounding the length of busy periods. Let `M(p)` be the smallest solution of the equation

```
 M(p) = B(p) + sum(A(p), lambda i: C(i) * ceiling(M(p) / T(i))).
```

Then a busy period of level p cannot have a length larger than `M(p)`.

The PVS proof is decomposed in several lemmas shown in Figure 5.3. The main step is lemma `critical_interval`, which shows that successive quiet points cannot be distant by a delay larger than `M(p)`. This immediately implies that busy periods cannot be longer than `M(p)`.

The proof of `critical_interval` is based on the following argument: Let t_1 be a quiet point and consider the interval $[t_1, t_1 + \mathtt{M(p)}]$. If there is a point t_2 in this interval where no job of priority `p` or higher is ready, then t_2 is a quiet point. Otherwise, for all t such that $t_1 \leqslant t \leqslant t_1 + \mathtt{M(p)}$, there is a job of `K(p)` ready to run at time t. This implies that

```
                                                              1
   process_time(sch(u), t1, t1+M(p), K(p)) >= M(p) - B(p).
```

Since t_1 is a quiet time, jobs of `K(p)` started before t_1 are finished at t_1 and then they do not use any processing time in the interval $[t_1, t_2]$. Since jobs that start after or at t_2 are not allocated processing time in $[t_1, t_2]$, we have

```
   process_time(sch(u), t1, t1+M(p), K(p)) =
      process_time(sch(u), t1, t1+M(p), K(p, t1, t2).
```

By lemma `process_time_K`, it follows that

```
                                                              2
   process_time(sch(u), t1, t1+M(p), K(p)) <=
       sum(A(p), lambda i: C(i) * ceiling(M(p)/T(i))).
```

By definition of `M(p)`, inequalities $\boxed{1}$ and $\boxed{2}$ give

```
   process_time(sch(u), t1, t1+M(p), K(p) =
       sum(A(p), lambda i: C(i) * ceiling(M(p)/T(i))).
```

This equality implies that a job $\mathtt{j} = (\mathtt{i}, \mathtt{n})$ of `K(p)` that started between t_1 and $t_1 + \mathtt{M(p)}$ has been allocated $\mathtt{C(i)}$ units of processing time in $[t_1, t_1 + \mathtt{M(p)}]$. This means that job `j` is finished at time $t_1 + \mathtt{M(p)}$ since the length of `j` is no more than $\mathtt{C(i)}$. As a consequence, $t_2 = t_1 + \mathtt{M(p)}$ is a quiet time: all the jobs of `K(p)` that started between t_1 and t_2 are finished at t_2, and, since t_1 is quiet, all the jobs of `K(p)` started before t_1 are finished at t_1 (and then at t_2).

Since zero is a quiet time, lemma `critical_interval` implies by induction that any instant t is between two quiet times t_1 and t_2 such that $t_1 \leqslant t < t_2$ and $t_2 - t_1 \leqslant \mathtt{M(p)}$. Since t_2 is a quiet point, any job of priority `p` that is dispatched at time t is finished at time t_2 and then before time $t + \mathtt{M(p)}$. This is sufficient to prove the schedulability result presented earlier. If all the tasks have different priorities, the deadlines `D(i)` are not larger than the delays `T(i)`, and for all task `i` there is $\mathtt{M} \leqslant \mathtt{D(i)}$ such that

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

then the set of sporadic tasks is schedulable. All the jobs meet their deadlines. This follows from the fact that such an M is a solution of the equation

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

where p is the priority of task i. M is then larger than or equal to M(p).

The main schedulability conditions we obtained are listed in Figure 5.4. Lemma termination1 is a very general result that holds for any set of sporadic tasks: a job of priority p dispatched at time t is finished at t+M(p). This gives a simple schedulability test stated as lemma schedulability1. This bound is sufficient to obtain directly the standard schedulability criterion in the case where tasks have different priorities and deadlines are not larger than periods (lemma schedulability_criterion).

5.5 Discussion

The overall approach we used for developing schedulability conditions for sporadic tasks in PVS is fairly standard. Our PVS proofs follow the well-known method based on the notion of busy periods introduced by Lehoczky [19]. However, a notable difference between our PVS approach and more traditional methods is to avoid the search for a worst-case scenario. Many results on real-time schedulability 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 [20]. In general, schedulability analysis can then concentrate on the worst possible case, where all the tasks are released at the same time t. Such a t is called a critical instant and the analysis is based on finding the response time for jobs released at a critical instant or the length of busy periods that start at a critical instant.

The notion of critical points is not adequate when jobs can share resources and block each other. The worst-case response time must take blocking into account, and assuming that all the tasks are released at the same time is not sufficient by itself. Furthermore, looking for worst-case scenarios is an unnecessary detour since the essential bound given by lemma busy_interval2 is satisfied by any busy period of level p whether it starts at a critical instant or not.

The results presented in this chapter were concerned essentially with the case where deadlines are less than or equal to job inter-arrival delays. This corresponds to a very important practical case. Still, many results necessary for handling the more general case, where $D(i)$ can be larger than $T(i)$, have already been established. The general case could then be formally analyzed in PVS with some extra work.

```
%---------------------------
%  First termination bound
%---------------------------

termination1: PROPOSITION
      prio(j) = p IMPLIES finished(u, j, dispatch(j) + M(p))


%-----------------------------------------------------
%  First schedulability criterion: M(prio(i)) <= D(i)
%-----------------------------------------------------

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


%-----------------------------------------------------
%  Schedulability criterion in the more standard case
%  - one task per priority
%  - deadline before period
%-----------------------------------------------------

l, l1, l2: VAR task

M: VAR posnat

H(i): set[task] = { l | prio(l) > prio(i) }

J(i): set[task] = { l | prio(l) = prio(i) }

B(i): nat = B(prio(i))

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

Figure 5.4: Sufficient Schedulability Conditions

Chapter 6

Related Work

We compare our PVS developments with related work on formalization and analysis of real-time scheduling algorithms and real-time kernels or scheduling protocols.

6.1 Formalization of the Priority Ceiling Protocol

Pilling et al. [28, 29] give formal specifications of two variants of the priority ceiling protocol written in Z [36]. The specifications were based on an early description of the protocol and clarified some aspects in the dynamic adjustment of priorities. The model used is a state-transition system where the transitions correspond to the dispatch of a job, the termination of a job, or the locking or unlocking of semaphores. The modeling is based on processes rather than jobs: the state of the system includes sets of active, inactive, and blocked processes, and timing aspects are not included. The model follows very closely the description given by Sha et al. [33, 34] and uses dynamic priorities and explicit priority inheritance.

Since timing issues are not modeled, the analysis performed is more limited than in this report. Schedulability tests or results such as the bound on blocking are not considered. Instead, the paper shows that the protocol ensures absence of deadlocks, mutual exclusion, and that a process can have at most one blocker. The developments do not use any tools; the various proofs, though written in the Z notation, remain largely informal.

Burns and Wellings [7] present a formalization of priority inheritance for concurrent real-time systems that communicate by message passing. The model is an automaton written in the Z notation and is similar to the model used by Pilling et al. [28]. The paper shows that the model satisfies simple properties. Unfortunately, the proofs are informal and the specifications contain several errors.

6.2 Verification of Scheduling Theorems

Wilding [38] presents a formal proof of the optimality of the earliest-deadline-first (EDF) scheduling algorithm. This is a formal specification and proof of one of Liu and Layland's theorems [20]. The formalization relies on the Boyer-Moore theorem prover Nqthm [4].

Although applied to a different scheduling policy and a slightly different problem, the approach is similar to the schedulability analysis for sporadic tasks presented in Chapter 5. The model relies on a notion of schedules represented as lists, and the analysis consists of showing that if a set of periodic tasks has total utilization less than or equal to 1, then an EDF schedule for these tasks ensures that all the jobs meet their deadline. This result holds provided task deadlines are equal to periods, that is, a job of task τ must terminate before the release of the next job of the same task.

Wilding's work illustrates the usefulness of precise specifications and mechanically assisted methods of verification. The original proof of the optimality of EDF published by Liu and Layland was flawed [38].

6.3 Modeling and Analysis of Real-Time Kernels

Fowler and Wellings [14] present a PVS specification of a real-time kernel intended to be the basis of an Ada95 runtime support system. The model is state-based and represents a fixed set of Ada tasks communicating via protected objects. Mutual exclusion is implemented using a simplified form of the priority ceiling protocol, and both sporadic and strictly periodic tasks can be considered. Timing properties are expressed using a real-time logic embedded in PVS.

Although the kernel model is fairly sophisticated and complete, only simple invariant properties are verified in [14]. More complex kernel properties with some timing aspects are proven in Fowler's thesis [13], but the emphasis is on using the formal specifications as a basis for developing a trusted kernel implementation. The PVS verifications focus on showing that successive refinements steps are correct, including from a temporal perspective.

Fidge et al. [12] formalize and verify a simple real-time scheduler using the Ergo theorem prover. The formalization is machine language code for an interrupt handler activated at regular intervals by an external clock. A finite set of tasks of the same period and priority is scheduled in a round-robin fashion by the interrupt handler. The analysis relies on a precise model of the microprocessor used. The verification consists of showing that the scheduler periodically allocates a minimal amount of uninterrupted processing time to each task. Although the task set and the scheduling algorithm are too simplistic for real applications, the analysis illustrates how processor characteristics can be taken into account in real-time analysis. Processor pipelines and caches are modeled, and context switching time is taken into account during the analysis.

More recently, Wilding et al. [39] formalize and analyze a similar round-robin scheduler using PVS. The model is very close to [12], but the verifications have a different objective. The kernel is intended to ensure task isolation, that is, to prevent a faulty task from adversely affecting other tasks in the system.

Several other examples of operating system specifications using formal notations can be found in the literature. Some examples include tool-assisted verifications. In most cases, real-time aspects and resource sharing protocols are not considered. A comparison and survey of these kernel specifications and verification is given in [13].

Chapter 7

Conclusion

This report presents 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. All the developments were performed with tool support, using the PVS specification and verification system.[1] This provides high assurance of correctness and ensures a complete and rigorous analysis.

The formalization is more precise, and we believe simpler, than traditional descriptions of the protocol. Priority adjustment rules can be avoided by using a slightly modified rule for selecting active jobs. Such a simplification may point to new ways of implementing the protocol, but it was mostly useful for reducing the verification effort. Other results emerged from the analysis, such as the fact that proper nesting of critical sections is not necessary.

Using the protocol model, we provided general results such as bounds on blocking time and on the amount of processing resource allocated to a job. These results form the essential basis for developing feasibility conditions for various sets of tasks. As an important application, we gave the example of sporadic tasks with deadline smaller than periods. Other classes of applications could be considered in a similar way including the more general case of sporadic tasks with arbitrary deadlines.

As a whole, our verifications mostly confirmed several results already known and published in the literature [34, 37]. We did not discover unexpected errors in these preexisting results. Our contribution was to apply mechanical theorem proving to a fairly challenging problem and to provide rigorous and detailed proofs of properties often presented very informally. As a whole, 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.

This work is a preliminary development toward the specification and analysis of real-time kernels ensuring strong partitioning. Developing such kernels is becoming increasingly important in avionics and other critical applications. The emerging integrated modular avionics architectures allow sharing of processors between independent avionics functions [1, 22, 23]. In this context, it is essential to ensure that independent tasks or processes that share a common processor cannot adversely affect each other. Real-time kernels must

[1]The theories are available at `http://www.csl.sri.com/~bruno/pvs/prio-ceiling.txt`.

then protect separate tasks from unwanted interference by implementing strong partitioning mechanisms [30]. Formal models addressing some of these issues have been proposed recently [10, 11, 39] but the results are still preliminary and incomplete.

We are particularly interested in temporal noninterference, that is, the property that a temporal failure of a task, such as overrun or nontermination, cannot prevent independent tasks from satisfying their timing requirements. Simple solutions have been proposed that rely on a very strict static scheduling [2, 39]. Since static schedulers are extremely inflexible [21], there is interest in developing kernels that implement priority-based scheduling while still ensuring strong partitioning and temporal noninterference.

In future work, we intend to extend our model of the priority ceiling protocol in two directions. First we need to make more realistic timing assumptions in the kernel model. In the real world, not all job instructions or kernel operations have the same duration, and process switching delays cannot be ignored. A second objective is to extend our model of the priority ceiling protocol with mechanisms that ensure temporal noninterference. Such mechanisms might rely on quotas or other time management techniques to enforce temporal partitioning.

Bibliography

[1] Design Guidance for Integrated Modular Avionics. ARINC Report 651, ARINC, Annapolis, MD, November 1991.

[2] Avionics Application Software Standard Interface. ARINC Specification 653, ARINC, Annapolis, MD, January 1997.

[3] 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.

[4] R. S. Boyer and J. S. Moore. *A Computational Logic Handbook*. Number 23 in Perspectives in Computing. Academic Press Inc., 1988.

[5] A. Burns. Fixed-Priority Scheduling with Deadlines Prior to Completion. Technical Report YCS-93-212, Department of Computer Science, University of York, York, UK, 1993. Available at `http://www.cs.york.ac.uk/rts/`.

[6] A. Burns. Preemptive Priority Based Scheduling: An Appropriate Engineering Approach. Technical Report YCS-93-214, Department of Computer Science, University of York, York, UK, 1993. Available at `http://www.cs.york.ac.uk/rts/`.

[7] A. Burns and A. Wellings. Priority Inheritance and Message Passing Communication: A Formal Treatment. *Real-Time Systems*, 3(1):19–44, March 1991.

[8] R. W. Butler. An Elementary Tutorial on Formal Specification and Verification Using PVS 2. NASA Technical Memorandum 108991, NASA Langley Research Center, Hampton, VA 23681, June 1995.

[9] J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas. A Tutorial Introduction to PVS. In *WIFT'95 Workshop on Industrial-Strength Formal Specification Techniques*, April 1995.

[10] Ben Di Vito. A Model of Cooperative Noninterference for Integrated Modular Avionics. In *Dependable Computing for Critical Applications (DCCA-7)*, pages 269–286, San Jose, CA, January 1999.

[11] B. Dutertre and V. Stavridou. A Model of Noninterference for Integrating Mixed-Criticality Software Components. In *DCCA-7, Dependable Computing for Critical Applications*, pages 301–316, San Jose, CA, January 1999.

[12] C. Fidge, P. Kearney, and M. Utting. Formal Specification and Interactive Proof of a Simple Real-Time Scheduler. Technical Report 94-11, Software Verification Research Center, University of Queensland, Queensland, Australia, April 1994.

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

[14] 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.

[15] L. George, N. Rivierre, and M. Spuri. Preemtive and Non-Preemptive Real-Time Uniprocessor Scheduling. Technical Report 2966, INRIA, Rocquencourt, France, September 1996. Available at `http://www.inria.fr/RRRT/publications-eng.html`.

[16] P. Harter. Response Times in Level-Structured Systems. *ACM Transactions on Computer Systems*, 5(3):232–248, August 1987.

[17] M. Joseph and P. Pandya. Finding Response Times in a Real-Time System. *The Computer Journal*, 29(5):390–395, May 1986.

[18] B. Lampson and D. Redell. Experience with Processes and Monitors in Mesa. *Communications of the ACM*, 23(2):105–117, February 1980.

[19] J. Lehoczky. Fixed Priority Scheduling of Periodic Task Sets with Arbitrary Deadlines. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 201–209, Lake Buena Vista, FL, December 1990.

[20] 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.

[21] D. Locke. Software Architecture for Hard Real-Time Applications: Cyclic Executives vs. Fixed-Priority Executives. *Real-Time Systems*, 4(1):37–53, March 1992.

[22] M. Morgan. Integrated Modular Avionics for Next-Generation Commercial Airplanes. *IEEE Aerospace and Electronic Systems*, 6(8):9–12, August 1991.

[23] A. Nadesakumar, R. Crowder, and C. Harris. Advanced System Concepts for Future Civil Aircraft - An Overview of Avionic Architectures. *Proceedings of the Institution of Mechanical Engineers. Part G: Journal of Aerospace Engineering*, 209(G4), 1995.

[24] 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.

[25] S. Owre, N. Shankar, J. Rushby, and D. Stringer-Calvert. *PVS Language Reference. Version 2.3.* Computer Science Laboratory, SRI International, July 1999.

[26] S. Owre, N. Shankar, J. Rushby, and D. Stringer-Calvert. *PVS Prover Guide. Version 2.3.* Computer Science Laboratory, SRI International, July 1999.

[27] S. Owre, N. Shankar, J. Rushby, and D. Stringer-Calvert. *PVS System Guide. Version 2.3*. Computer Science Laboratory, SRI International, July 1999.

[28] M. Pilling, A. Burns, and K. Raymond. Formal Specifications and Proofs of Inheritance Protocols for Real-Time Scheduling. Technical Report 100, Department of Computer Science, University of Queensland, St Lucia, Australia, December 1988.

[29] 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.

[30] J. Rushby. Ubiquitous Abstraction: A New Approach for Mechanized Formal Verification. In *Second IEEE International Conference on Formal Engineering Methods (ICFEM'98)*, pages 176–178, Brisbane, Australia, December 1998.

[31] J. Rushby and D. Stringer-Calvert. A Less Elementary Tutorial for the PVS Specification and Verification System. Technical Report CSL-95-10, Computer Science Laboratory, SRI International, August 1996.

[32] L. Sha, J. Lehoczky, and R. Rakjumar. Solutions for Some Practical Problems in Prioritized Preemptive Scheduling. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 181–191, New Orleans, LA, December 1986.

[33] L. Sha, R. Rajkumar, and J. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. Technical Report CMU-CS-87-181, Computer Science Department, Carnegie Mellon University, 1987.

[34] 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.

[35] 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.

[36] J. M. Spivey. *The Z Notation*. Prentice-Hall International, 1989.

[37] K. Tindell, A. Burns, and A. Wellings. An Extendible Approach for Analysing Fixed Priority Hard Real-Time Tasks. *Real-Time Systems*, 6(2):133–151, 1994.

[38] 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.

[39] M. Wilding, D. Hardin, and D. Greve. Invariant Performance: A Statement of Task Isolation Useful for Embedded Application Integration. In *Dependable Computing for Critical Applications, DCCA-7*, pages 287–300, San Jose, CA, January 1999.