Formal Model of Fixed-Priority Scheduling

Real Time Systems, Spring 2016, SNU

Kim Yoonseung
Seoul National University
yoonseung.kim@sf.snu.ac.kr

Abstract

We present a formal model of the fixed-priority real-time systems. We define the task model and the execution process in a formal way, and we also provide some useful library of proofs. We also show a possible way to use this for more complex purpose, such as proving optimality of RM. Our supplementary materials are shown at https://github.com/kim-yoonseung/RTS_term_paper.

1. Introduction

The soundness and reliability of job scheduling algorithms in real-time systems is a greatly desired property. Fundamentally, the most important concern in real-time systems is to guarantee to meet the deadline of each job by using an appropriate scheduling algorithms that controls the execution of jobs.

Among the bunch of various scheduling algorithms, the fixed-priority scheduling is widely used in the real-world applications. In contrast to dynamic scheduling, it provides simpler implementation, more predictablility, and easier ways to handling deadline misses. Also, the simple, rate-monotonic algorithm is already known to be the optimal among them.

Meanwhile, the formal verification methods are going more closer to the real use. The formal methods provides the most strongest credibility for proving the correctness of a system. The formal proofs for real world systems had been regarded as almost intractable, and even just bothersome, since it required a lot of human efforts and the gain seemed to be small. However, current machine-checked proof assistant softwares such as Coq([1]) are greatly improved, and several realistic formally-proved systems are already introduced([2]).

Especially, the real-time systems field seems to be very promising as an application of the formal verification methods. Many real-time systems handle very important and sometimes vital tasks. The predictability of the behavior of the system is one of the desirable property of a real-time system. The formal methods can provide the evidence of that a system satisfies its specification, with very high confidence.

In this paper, we formally model the fixed-priority scheduling of real-time systems as the start of applying the formal methods to the real-time systems. We also support some proofs related to the system, which can be used to prove any specification this system should satisfy, such as schedulability.

2. Backgrounds

In this section, we introduce the two topics related to this work: the fixed-priority scheduling and the formal method.

2.1 Fixed-Priority Scheduling

The priority-driven scheduling algorithms are popularly used in real-time systems. They are classified as an online scheduling since the actual schedule is determined on-line by examining the current status.

The fixed-priority scheduling and the dynamic-priority scheduing are the two parts that consist of the priority-driven scheduling. The dynamic-priority scheduling outperforms the fixed-priority in general. However, the fixed-priority scheduling is still widely used in real world since it is simpler and more predictable.

Rate-monotonic Among the fixed-priority scheduling algorithms, the algorithm called rate-monotonic(RM) is known to be optimal. The scheduling rule is simple: tasks with shorter period get higher priorities. Here the optimality implies that this algorithm can successfully schedule any tasks if it can be scheduled by any other fixed-priority schedulers. The optimality of RM is proven in earlier[3]. Giving a formal proof for RM itself is not very important or surprising. However, formally proving RM can be served as a test for applying the formal method to the real-time systems. Also, the works done in this job can be reused for more complicated algorithms later.

2.2 Formal Methods

In present, formal proofs are usually done with machine-checked theorem prover[]. They are also called proof assistants. The products with formal proofs provides the greatest reliability for users, since the proof is checked by machine.

Coq The Coq proof assistant[1] is the most widely used theorem prover. Coq is equipped with strong mathematical theories so we can strongly believe the results. Also this prover supports both usual programming in ML-style and mathematical theorem proving, so that programmers can write a program, write its specification in precise mathematical formulas, and then prove the property at the same time. The most representative product is CompCert[2], a verified real-world C compiler.

CompCert The CompCert compiler clearly shows the importance of the formal methods. Until now, compiler writers had been writing the compiler code relying on their intuition. However, it turned out that current compilers contain many bugs[4]. By testing many bugs are fixed during the developing stages, but still wrong compilation occurs in real world.

3. Formal Model of Workloads

Here we present our way of modeling the workload model: tasks and jobs. We defined their formal structure and desirable properties for them. The works described here can be found in `def.v` of the supplementari materials.

3.1 Tasks

A task is modeled as a tuple of `(task_id, start, period, exec_time)`. `task_id` is supposed to be unique among the task set

to which the task is belong. `start` is the start time of the task, or `phase` in other word. `period` and `exec_time` means as expected. We suppose that relative deadlines are equal to the periods for simplicity.

This is the actual definition in the Coq syntax:

```
Record task : Set :=
Task { task_id:nat; start:nat;
    period:nat; exec_time:nat }.
```

Figure 1. Definition of tasks in Coq

`Record` means this structure is a tuple. `nat` is the type of natural numbers. For example, `task_id:nat` means that the task-id is a natural number.

3.2 Jobs

A job is composed of a tuple `(release_time, deadline, job_task_id, time_left)`. The first two value represent the absolute release time and deadline. `job_task_id` is the task-id of the job. `time_left` shows the remaining time to complete the job's execution. When `time_left` becomes 0 before the deadline, the job is successfully completed.

Again we show the actual definition of jobs in the Coq syntax:

```
Record job : Set :=
Job { release_time:nat; deadline:nat;
    job_task_id:nat; time_left:nat }.
```

Figure 2. Definition of jobs in Coq

3.3 Properties

We also define several desirable properties of a task set. A set of task in a real-time system is simply modeled as a list of tasks.

```
Definition taskset := list task.
```

The order of tasks in the list indicates the priority. The leftmost (head) task has the highest priority over other tasks.

3.3.1 `valid_tset`

We want a taskset to have some good properties. Here we define the condition for a taskset to be valid. First, we want that task-ids are unique in the taskset. Also, we added another property that there is no duplicate tasks in the list. In fact, the uniqueness of task-id implies this, but we explicitly use this for simplicity. Here is the actual Coq code which is quite intuitive.

```
Definition valid_tset (tset:taskset) : Prop :=
tset_unique_id tset ∧ NoDup tset.
```

The detail definition of `tset_unique_id` is shown in the supplementary material, and `NoDup` is supported by the standard Coq library.

3.3.2 `valid_queue`

We also desire a property for a job queue. We assume that all jobs in the queue are sorted in the priority order. Like tasks, we defined a property `valid_queue` for a job queue.

```
Inductive valid_queue (tset:taskset) : queue -> Prop :=
| Q_empty: valid_queue tset []
| Q_one : forall j (n:nat),
    priority_of_tid j.(job_task_id) tset = Some n ->
    valid_queue tset [j]
| Q_many : forall j1 j2 jq n1 n2,
    priority_of_tid j1.(job_task_id) tset = Some n1 ->
    priority_of_tid j2.(job_task_id) tset = Some n2 ->
    n1 < n2 ->
    valid_queue tset (j2::jq) ->
    valid_queue tset (j1::j2::jq).
```

Figure 3. Definition of `valid_queue`

In the code above, `priority_of_tid (job) (taskset)` returns the priority of the job. This looks complicated, but if you carefully examine, you must realize that this just implies the sorted order of the queue.

3.4 Taskset Generator

We provide a taskset generator which is simple to use. Randomly-made taskset may violate the validity condition `valid_tset` by assigning same task-id to multiple tasks. Our task generator automatically assign the task-id when adding a new task. This generator also automatically generates the proof that the generated taskset satisfies the `valid_tset` condition so that we can use this proof in further proofs. See the `TASK_GEN` section of `def.v`

4. Formal Model of Execution

Here we describe how we formally model the execution procedure. These are defined in `exec.v`.

4.1 State

We model the execution of a system as a sequence of transitions of states. Each state is a tuple of `(taskset, current_time, job_queue, list_of_failed_jobs)`. The first three parts are intuitively understandable. The `fail_list` part stores the list of jobs that missed the deadline from the start. Here we assume that the execution just throws away the missed jobs.

```
Record state : Set :=
  State { st_tset: taskset;
          time:nat;
          job_q:queue;
          fail_list: queue
        }.
```

Figure 4. Definition of `state`

The initial state for a taskset is simply the state with time 0, the empty job queue, and the empty fail list.

```
Definition init_state (tset:taskset): state :=
    State tset 0 nil nil.
```

4.2 One-step Execution

We assume that the execution proceeds with discrete time sequence, for simplicity. Therefore the whole execution consists of the sequence of successive steps. Here is the formal model of the one-step execution. As we mentioned, each step is a transition from a state to another state.

```
Definition execute_one (cur:state) : state :=
  match cur with
  | State tset time jq fl =>
    let jq_at_start := add_new_jobs tset time jq in
    let jq_at_end :=
        match jq_at_start with
        | [] => []
        | job::rest =>
          let new_time_left := job.(time_left) - 1 in
          if (Nat.eqb new_time_left 0)
          then rest
          else (Job job.(release_time) job.(deadline)
                job.(job_task_id) new_time_left)::rest
        end
    in
    let (lives, fails) :=
         filter_fails jq_at_end (1 + time) in
    State tset (1+time) lives fails
  end.
```

Figure 5. Definition of one-step execution

Each step starts with adding new jobs. We calculate newly-released jobs from the taskset from the `add_new_jobs` function. Then `jq_at_start` is the sorted job queue that the execution is going to process. Now it just checks the queue and process the job at the head of the queue. Our model simply subtract one time unit from the head job's `time_left` field. If the time left for the job is 0, we can remove the job from the queue. After the processing, we check the queue to filter jobs who fail to meet the deadline.

4.3 N-step Execution

Now we can easily design the n-step execution. If $n = 0$, we execute zero step, i.e. return the initial state. Otherwise, just repeat the one-time execution n times.

```
Fixpoint execute_n (cur_state:state) (n:nat)
    : state :=
  match n with
  | O => cur_state
  | S n' => execute_n (execute_one cur_state) n'
  end.
```

Figure 6. Definition of n-step execution

4.4 Assigning Priority

As we mentioned, the priority of tasks is represented as the order of tasks in the `taskset` list. Therefore we can model the priority assignment as reordering the given `taskset`. We define the type of the priority assigner in a general way: functions from `taskset` to `taskset`, and then give a constraint to be a valid priority assigner.

```
Definition priority_assigner : Type :=
  taskset -> taskset.

Definition valid_priority_assigner
           (pr:priority_assigner) : Prop :=
  forall (tset:taskset) (t:task),
    valid_tset tset ->
    (In t tset <-> In t (pr tset)) /\
    valid_tset (pr tset).
```

Figure 7. Priority assigner as reordering function

The constraint is intuitive: if a priority assigner `pr` is valid, then it preserves the validity of the given task set and the elements of the task set.

4.5 Running a taskset from initial

Now we can define the top-level execution model. An execution requires a priority assigner, taskset, and the time bound. Its job is simple: just sort the task set according the priority and execute n steps from the initial state.

```
Definition run (pr:priority_assigner)
               (tset:taskset) (n:nat)
    : state :=
  let sorted_tset := pr tset in
  execute_n (init_state sorted_tset) n.
```

Figure 8. `run`: Toplevel execution model

Now we can define the 'schedulability' condition. We say a task set is schedulable with the given priority, if for any unbound length of executions, the resulting state doesn't have any failed jobs.

```
Definition schedulable (pr:priority_assigner)
                       (tset: taskset) :=
  forall n:nat, exists final_state,
      run pr tset n = final_state /\
      final_state.(fail_list) = nil.
```

Figure 9. Schedulability condition

5. More Formal Proofs Supported as Libraries

In addition to the modeling of execution behaviors, we provide useful proofs as a library for the further complex reasoning. Here we introduce some of the interesting theorems. Many of them seem to be obvious, but they are necessary to give formal proofs for further complex propositions, such as the optimality of RM. You can check the whole result from `priority_proof.v` and `exec_proof.v` in the supplementary materials.

5.1 Proofs for Priorities

5.1.1 `existent-task-priority`

We prove that, every task in the task set is given a priority.

```
Lemma existent_task_priority:
  forall (t:task) (tset:taskset),
  In t tset -> exists (n:nat),
    priority_of_tid t.(task_id) tset = Some n.
```

5.1.2 `priority-tid-unique`

In a valid task set, the priorities of two task-ids are identical, then the ids are the same.

```
Lemma priority_tid_unique:
  forall (tid1 tid2:nat) (tset:taskset) (n:nat),
    valid_tset tset ->
    priority_of_tid tid1 tset = Some n ->
    priority_of_tid tid2 tset = Some n ->
    tid1 = tid2.
```

5.1.3 `priority-unique`

Also, in a valid task set, if two tasks' priorities are identical, then the tasks themselves are the same.

```
Lemma priority_unique:
  forall (t1 t2:task) (tset:taskset) (n:nat),
    valid_tset tset ->
    In t1 tset -> In t2 tset ->
    priority_of_tid t1.(task_id) tset = Some n ->
    priority_of_tid t2.(task_id) tset = Some n ->
    t1 = t2.
```

5.2 Proofs for Execution

5.2.1 get-text-nth-of-task-spec

Internally, for each time step and for each task, we determine whether it's the time to release a job from the task. This theorem proves that if our internal function `get_next_nth_of_task` says that this is the time to release the n-th job of the task, then it really is.

```
Theorem get_next_nth_of_task_spec:
  forall (t:task) (time:nat) (nth:nat),
    get_next_nth_of_task t time = Some nth ->
    (nth_job_of_task t nth).(release_time) = time.
```

5.2.2 add-job-to-queue-preserve-valid-queue

As we saw earlier, each one-step execution first adds newly-generated jobs. We prove here that for a valid task set, if the generated job is from the task set and original queue is valid, and if there is no other job from the same task, then the queue after inserting the generated job is still valid.

```
Lemma add_job_to_queue_preserve_valid_queue:
  forall (tset:taskset) (j:job) (q:queue),
    valid_tset tset ->
    (exists n,
       priority_of_tid j.(job_task_id) tset
        = Some n) ->
    valid_queue tset q ->
    (forall j', In j' q ->
       j'.(job_task_id) <> j.(job_task_id)) ->
    valid_queue tset (add_job_to_queue tset j q).
```

5.2.3 filter-fails-spec

After the one-step execution, we filter the failed jobs. This theorem proves that the filtered jobs really miss their deadlines.

```
Lemma filter_fails_spec:
  forall q qt qf time,
    filter_fails q time = (qt, qf) ->
    (forall j:job, (In j qt)
       -> (time < j.(deadline))) /\
    (forall j:job, (In j qf)
       -> (j.(deadline) <= time)).
```

6. Further Try: Formal Proof of Optimality of RM

The formal models of execution and the proof libraries can be utilized to prove more complex and useful theorems. We provide a template to prove the optimality of RM. This proof has several holes so incomplete. However, with some efforts, one can fill the hole to complete the whole proof of the optimality of RM. Here we describe the template, defined in `rm_swap.v` and `rm_optimal.v`.

6.1 Swap

The main constructor of the optimality proof is the swapping operation. For any schedulable executions, we perform swapping until it becomes the RM schedule. Then we should prove that the swapped schedule is still schedulable.

The swapping is represented here as a higher-order function for priority assigners. In the previous section we saw that a priority assigner is modeled as a task-set-reordering function. Then, swapping can be modeled as swapping one adjacent pair of tasks in the task set after running the original priority assigner. Here, swapping is done only when it found a pair whose priorities are inverted according to their periods. Formally, we define swap by combining the two functions below.

```
Fixpoint swap_taskset (tset:taskset) : taskset :=
  match tset with
  | [] => []
  | h1::t1 =>
    match t1 with
    | [] => [h1]
    | h2::t2 =>
      if (Nat.ltb h2.(period) h1.(period))
      then
        h2::h1::t2
      else
        h1::(swap_taskset t1)
    end
  end.

Definition swap_priority_assigner (pr:priority_assigner)
  : priority_assigner :=
  fun (tset:taskset) => swap_taskset (pr tset).
```

Figure 10. Modeling swap

Here, `swap_taskset` gets an input of a task set. Then, it checks the task set from the beginning whether there is an inverted pair or not. If it finds such pair, it swaps them and return the task set.

Then, `swap_priority_assigner` gets a priority assigner and then returns another priority assigner, which is almost same as the input assigner, but one inverted pair of tasks are swapped.

We present a formally-proven property of this swap model below:

```
Definition tset_swapped
           (tset1 tset2:taskset) (task1 task2:task):=
  exists (tl_bef tl_aft:taskset),
    (tset1 = tl_bef ++ [task1; task2] ++ tl_aft) /\
    (tset2 = tl_bef ++ [task2; task1] ++ tl_aft) /\
    task2.(period) < task1.(period).

Lemma swap_spec:
  forall (pr:priority_assigner)
         (tset0 tset1 tset2:taskset),
    tset1 = pr tset0 ->
    tset2 = (swap_priority_assigner pr) tset0 ->
    (sorted tset1 /\ tset2 = tset1) \/
    exists (task1 task2:task),
      tset_swapped tset1 tset2 task1 task2.
```

Figure 11. Specification of swap

This property guarantees that, the two task sets - one sorted by the original priority assigner, and another sorted by swapped priority - are identical except only one swapped pair.

6.2 Optimality Proof

Here we present the proof of RM's optimality with some holes.

6.2.1 Rate-monotonic

We first formally define the RM property. A priority assignment is RM for the given task when for any pair of tasks, the task with shorter period gets the higher priority.

```
Definition rate_monotonic (pr:priority_assigner)
                          (tset:taskset) : Prop :=
forall (t1 t2:task) (n1 n2:nat) (new_tset:taskset),
  new_tset = pr tset ->
  In t1 new_tset ->
  In t2 new_tset ->
  t1.(period) < t2.(period) <->
  (exists (n1 n2:nat),
   priority_of_tid t1.(task_id) new_tset = Some n1 /\
   priority_of_tid t2.(task_id) new_tset = Some n2 /\
   n1 < n2).
```

Figure 12. Formal definition of Rate-monotonic

6.2.2 Matching states

Simulation Here we plan to prove the schedulability of the swapped tasks using the 'simulation' technique: ideally we run the two systems - with the original schedule and the swapped schedule - simultaneously, and if currently the two states of the systems are somehow 'matched'. Then, if the original system executes one step without deadline misses, then we show that the swapped system also executes one step without deadline miss, and the two resulting states are again 'matched'. If we prove this, we can reason this way: If the initial states of the two systems are 'matched', then for the successive steps the two systems keep running without deadline misses and matched, unless the original system misses a deadline.

Here we do not provide a concrete such 'match_state' predicate, but we clarify that at least it should imply the absence of deadline misses. In the code below, `match_state_int` is not yet defined and remained as a hole (`Parameter` in the Coq's syntax).

```
Parameter match_state_int :
  state -> state -> Prop.

Definition match_state (st1 st2:state): Prop :=
  match_state_int st1 st2 /\
   st1.(fail_list) = [] /\ st2.(fail_list) = [].
```

Figure 13. 'match-state' with a hole

6.2.3 Proof of one-step

Actually we have to prove the simulation for one-step case. However, it requires a concrete definition of `match_state` relation, so we leave this as a hole.

```
Lemma swap_execute_one:
      forall (sti1 sti2 stf1:state),
        match_state sti1 sti2 ->
        execute_one sti1 = stf1 ->
        exists stf2, execute_one sti2 = stf2 /\
                match_state stf1 stf2.
Proof.
Admitted.
```

Figure 14. one-step proof as a hole

6.2.4 Proof of n-step

However, if we believe the one-step simulation, we can prove the n-step simulation. The below is the proposition of n-step simulation proved without a hole.

```
Lemma swap_execute_n:
  forall (n:nat) (sti1 sti2 stf1:state),
    match_state sti1 sti2 ->
    execute_n sti1 n = stf1 ->
    exists stf2, execute_n sti2 n = stf2 /\
            match_state stf1 stf2.
```

Figure 15. n-step proof without a hole

6.2.5 Schedulability of Swapped Tasks

Then we can also prove the schedulability of swapped tasks.

```
Lemma swap_schedulable:
  forall (tset:taskset) (pr:priority_assigner),
    valid_tset tset ->
    valid_priority_assigner pr ->
    schedulable pr tset ->
    schedulable (swap_priority_assigner pr) tset.
```

Figure 16. Schedulability proof of swapped tasks

6.2.6 Schedulability of n-times-swapped Tasks

Then it's easy to prove that the tasks swapped multiple times are schedulable if the original tasks were schedulable.

```
Lemma swap_n_schedulable:
  forall (tasks:taskset) (pr:priority_assigner),
    valid_tset tasks ->
    valid_priority_assigner pr ->
    schedulable pr tasks ->
    forall (n:nat), schedulable
       (ntimes swap_priority_assigner n pr) tasks.
```

Figure 17. Schedulability proof of n-times-swapped tasks

6.2.7 Swapping becomes RM

If we swap multiple times, the priority assignment will eventually be the RM scheduling. Currently we leave it as a hole, we believe that this is provable without huge obstacles.

```
Lemma swap_eventually_rm:
  forall (tset:taskset) (pr:priority_assigner),
  exists (n:nat), rate_monotonic
    (ntimes swap_priority_assigner n pr) tset.
Proof.
Admitted.
```

Figure 18. Swapping eventually becomes RM, as a hole

6.2.8 Optimality of RM

If we blindly believe the holes, we can now prove the optimality of RM with very light effort. The Theorem looks like this:

```
Theorem rm_is_optimal:
  forall (tset:taskset),
  forall (pr:priority_assigner),
    valid_tset tset ->
    valid_priority_assigner pr ->
    schedulable pr tset ->
    exists (rm_pr:priority_assigner),
      rate_monotonic rm_pr tset /\
      schedulable rm_pr tset.
```

Figure 19. Optimality of RM

7. Conclusion

In this paper, we present the formal model of the fixed-priority real-time systems. We define the task model and the execution process in a formal way, and we also provide some useful library of proofs. We also show a possible way to use this for more complex purpose, such as proving optimality of RM.

Our model has several limitations. First the time is considered as discrete. This model may be successfully used in frame-based systems, but not general systems. Coq supports real numbers, so we hope this work can be extended into real-number time.

One another useful application of this works is to define a formally-proved schedulability checker. We can formally describe the critical instant. Then we can prove that, if a task set is scheculable at the critical instant then it is schedulable all time. Then we may define a checker that is formally verified: the checker's result is mathematically proved and the proof is checked by machine.

References

[1] Coq. `https://coq.inria.fr/`.

[2] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[3] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20 (1):46–61, 1973.

[4] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in c compilers. In *ACM SIGPLAN Notices*, volume 46, pages 283–294. ACM, 2011.