Towards Synchronization in Prosa

Antonin Riffard
MPI-SWS
Kaiserslautern, Germany

Felipe Cerqueira
MPI-SWS
Kaiserslautern, Germany

Björn B. Brandenburg
MPI-SWS
Kaiserslautern, Germany

***Abstract*—We report on our ongoing work on the formalization of real-time resource sharing in Prosa, a framework based on the Coq proof assistant for the development of machine-checked schedulability analysis. We review the goals of the project, present a preliminary specification of a generic resource model, and then discuss the next steps and planned verification goals.**

I. Introduction

Certification standards call for the temporal requirements of safety-critical real-time systems to be thoroughly checked and validated. For this purpose, the real-time systems community has proposed a wealth of schedulability analysis techniques. However, the complexity inherent in state-of-the-art analyses has made it increasingly difficult to verify such analyses "by hand," and in fact there are ample recent examples that suggest that human error is a significant cause for concern [1, 2, 3, 4, 5, 6, 7]. This unfortunately raises a fundamental question: why should certification processes admit schedulability analysis results as evidence of temporal correctness if the underlying analysis methods may not be entirely sound?

To address this question and provide a *trustworthy* analytical foundation for the real-time systems of tomorrow, the Prosa project [8] seeks to develop rigorous, provably-correct schedulability analyses. Specifically, Prosa is an open-source framework based on the Coq proof assistant that provides a general and extensible *formal specification* of real-time scheduling theory and allows users to develop *machine-checked* schedulability analyses that are guaranteed to be correct.

To date, Prosa has been successfully used to reason about various aspects of real-time scheduling theory, including uniprocessor and multiprocessor scheduling, response-time analyses [8], arbitrary processor affinity (APA) scheduling [1, 2], sustainability in the context of self-suspending tasks [9], and work on the certification of analysis tools by Guo et al. [10].

Nevertheless, despite these promising initial results, Prosa still has limited applicability in the analysis of real-world systems since it offers no support for mutual exclusion. To address this issue, we are currently working towards support for the verification of real-time synchronization protocols as well as their associated blocking analyses in Prosa. In this paper, we first discuss the goals of this extension and specific challenges, then present our preliminary model of critical sections and mutual exclusion, and finally conclude with a summary of the next steps and planned verification goals.

This work was funded by Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) — 391919384.

II. An Initial Model of Mutual Exclusion

A. Goals and Challenges

The main goal of this project is to extend the current Prosa specification with a generic resource model. Initially, we will only target mutual exclusion, but plan to include additional types of relaxed concurrency control mechanisms in the future, such as reader-writer locks and k-exclusion protocols.

To formalize this resource model, we must define resources and critical sections, restrict the notion of a valid schedule to enforce mutual exclusion, and also adapt the properties of work conservation and priority compliance to be coherent with the resource model.

Those changes must be applied to both uniprocessor and multiprocessor scheduling. In case of the latter, we must additionally consider that jobs are allowed to spin (*i.e.*, busy-wait) when a resource is not available. Thus, when defining this new schedule state, we must be careful to maintain compatibility with existing definitions.

Moreover, to match the assumptions of existing analyses, we must define the various types of nesting of critical sections. For example, a restrictive, but also common, assumption is that a job can hold at most one resource at a time (*i.e.*, non-nested critical sections). Another variation is to only allow well-nested critical sections (*i.e.*, given two critical sections A and B, either A is completely inside B or they do not overlap). An even more general model, such as found in the Linux kernel, allows critical sections to be nested arbitrarily, as long as requests are ordered to prevent deadlocks. In Prosa, we seek a single model for critical sections that is able to express all types of nesting.

B. Prosa Foundations

Since Prosa already provides a foundation of common real-time scheduling concepts, we can focus directly on formalizing the resource model. To better understand the required changes, we begin with an overview of key definitions in Prosa.

The most basic notion in Prosa is that of jobs, which are represented by an opaque type called *Job*. Each *Job* is associated with certain parameters, modeled as functions, such as its actual execution cost:

```
Variable job_cost: Job → progress.
```

The above syntax expresses that each *Job* has a parameter called *job_cost* that maps a job to a value of type progress, which represents the amount of processor service required by a given job. The type progress, in turn, is just an alias of nat (the natural numbers), our model of discrete time.

Based on the notion of jobs, PROSA defines a uniprocessor schedule as a function that maps each time instant optionally to a job, represented by the type option *Job* (*i.e.*, either None or Some *j* with $j \in Job$), meaning that at any time, either the processor is idle or exactly one job is scheduled. The type time, in turn, is also an alias of nat.

```
Definition schedule := time → option Job.
```

Next, to model the progress and completion of execution, we define the central notion of service received by a job. Given a schedule *sched*, instantaneous and cumulative service of a job *j* are defined in PROSA as follows.

```
Definition service_at (j: Job) (t: time) : progress :=
  if sched t == Some j then 1 else 0.

Definition service (j: Job) (t: time) : progress :=
  \sum_(t' < t) service_at j t'.
```

When moving to the multiprocessor case, the only major change is the definition of schedule. Given the set of processors processor *num_cpus*, represented by integers in $[0, num\_cpus)$, PROSA defines a multiprocessor schedule as follows.

```
Definition schedule :=
  processor num_cpus → time → option Job.
```

Correspondingly, the definition of instantaneous service must also consider multiple processors. The definition of cumulative service, on the other hand, remains unchanged.

```
Definition service_at (j: Job) (t: time) : progress :=
    \sum_(cpu < num_cpus | sched cpu t == Some j) 1.
```

C. Formalization of Resource Sharing

Having presented an overview of the basic concepts in PROSA, we now introduce critical sections, add constraints to the schedule to respect mutual exclusion, and finally integrate the task and resource models.

1) Introducing Critical Sections: During execution, a job may be required to access certain shared resources in isolation. Each of those execution intervals that are subject to mutual exclusion is called a critical section.

We model a critical section *cs* as an execution interval $[cs\_start, cs\_end)$ that requires some resource *cs_resource*. More precisely, critical sections and resources are represented by opaque types *CriticalSection* and *Resource*, with parameters *cs_start*, *cs_end* (the bounds of the interval) and *cs_resource* (the targeted resource). In addition, for any job *j*, we ensure that $0 \leq cs\_start < cs\_end \leq job\_cost\ j$.

```
Variable cs: CriticalSection.
Variable cs_start: CriticalSection → progress.
Variable cs_end: CriticalSection → progress.
Variable cs_resource: CriticalSection → Resource.
```

Note that *cs_start* and *cs_end* are expressed in terms of progress, the service received by the job up to a given time.

Next, we associate with each *Job* a list of critical sections.

```
Variable job_critical_sections: Job → list CriticalSection.
```

Since jobs can have multiple critical sections, each with arbitrary start and end times, our model is very permissive and allows any type of nesting. Nevertheless, it can also accommodate more restrictive constraints. For instance, non-nested critical sections can be specified as follows.

```
Definition no_overlapping_critical_sections :=
  ∀ j, ∀ cs1 cs2,
    cs1 ∈ job_critical_sections j →
    cs2 ∈ job_critical_sections j →
    critical_sections_overlap cs1 cs2 → cs1 = cs2.
```

This predicate states that, if two critical sections *cs1* and *cs2* intersect, they must be the same. To test for intersection, we use the predicate critical_sections_overlap, which compares the boundaries of the critical sections.

2) Scheduling and Resource Allocation: Having defined critical sections, we now specify the conditions under which a schedule is valid assuming a given resource model.

First, we must identify critical sections in the schedule. We say that job *j* has entered a critical section *cs* by time *t* iff at an earlier time $t' \leq t$, **(a)** job *j* has received enough service to enter the left boundary *cs_start* of the critical section, and **(b)** job *j* is *effectively executing* at time *t'*.

```
Definition job_has_entered_section
                    (j: Job) (cs: CriticalSection) (t: time) :=
  ∃ t', t' ≤ t ∧
      service sched j t' ≥ cs_start cs ∧
      service_at sched j t' > 0.
```

Next, we say that job *j* has exited the critical section iff its received service is no longer within the right boundary *cs_end*.

```
Definition job_has_exited_section
                    (j: Job) (cs: CriticalSection) (t: time) :=
  service sched j t ≥ cs_end cs.
```

Using the predicates above, we define whether job *j* has entered but not yet exited a critical section at time *t*.

```
Definition job_in_section
                    (j: Job) (cs: CriticalSection) (t: time) :=
  job_has_entered_section j cs t ∧
  ¬ job_has_exited_section j cs t.
```

Considering that multiple critical sections can refer to the same resource, we also define whether job *j* is holding a resource *r* at time *t*.

```
Definition job_holds_resource
                    (j: Job) (r: Resource) (t: time) :=
  ∃ cs, cs ∈ job_critical_sections j ∧
      cs_resource cs = r ∧ job_in_section j cs t.
```

Finally, we formalize the key property of mutual exclusion, *i.e.*, at any time *t*, at most one job can access each resource.

```
Definition enforces_mutual_exclusion :=
  ∀ r t, ∀ (j1 j2: Job),
    job_holds_resource j1 r t →
    job_holds_resource j2 r t → j1 = j2.
```

Note that when formalizing reader-writer or k-exclusion synchronization in the future, this definition will no longer be valid, but it can be easily generalized to such cases.

3) Incorporating Spinning: Recall that in uniprocessor systems, jobs must suspend when waiting for a resource. In multiprocessors, however, it is sometimes more desirable to spin (*i.e.*, busy-wait) while waiting for a resource that is going be released shortly.

Since a job that spins does not make progress in terms of service, incorporating spinning requires changing the representation of a schedule as follows.

```
Inductive cpu_state :=
  Idle | Running of Job | Spinning of Job.
Definition schedule :=
  processor num_cpus → time → cpu_state.
```

In the definition above, the keyword `Inductive` indicates that cpu_state is an enumerated type with three possible values:

- Idle, when the processor has no job scheduled (this corresponds to None in the old definition);
- Running j, when the processor is executing a job $j \in Job$ (this corresponds to Some j in the old definition);
- Spinning j, when the currently scheduled job $j \in Job$ is busy-waiting for a resource.

Using the new processor state, we adapt the definition of service_at to count service only from Running processors:

```
Definition service_at (j: Job) (t: time) : progress :=
  \sum_(cpu < num_cpus | sched cpu t == Running j) 1.
```

Since the other definitions are built on top of service_at, no other changes are required and the property valid_resource_allocation remains valid.

4) Incorporating Resources into the Task Model: The remaining step is to incorporate job critical sections into the task model, so that we can later formalize blocking analyses.

For each task we define the maximum number and maximum length of critical sections (*task_num_cs* and *task_length_cs*, respectively). Then, we define two predicates to enforce such constraints for each individual job and critical section.

```
Variable task_num_cs: Task → Resource → nat.
Variable task_length_cs: Task → Resource → progress.

Definition num_critical_sections_is_bounded :=
  ∀ (j: Job), ∀ (r: Resource),
    count_mem r (map cs_resource (job_critical_sections j))
    ≤ task_num_cs (job_task j) r.

Definition critical_section_length_is_bounded :=
  ∀ (j: Job), ∀ cs, cs ∈ job_critical_sections j →
    cs_length cs
    ≤ task_length_cs (job_task j) (cs_resource cs).
```

In the above definition, the operation count_mem r (map ...) counts the number of sections of job j that access resource r.

III. Future work

As future work, we plan to extend the specification by defining other types of nesting and formalizing reader-writer and k-exclusion synchronization. In addition, we aim to define real-time synchronization protocols and important concepts such as the notion of priority inversion.

After concluding the specification, we seek to verify existing blocking analyses, so that they can be integrated with schedulability analysis frameworks in Prosa.

As a first step, we will focus on the uniprocessor case with the *stack resource policy* (SRP) and verify the existing blocking bounds. Next, we will consider an extension of the protocol to multiprocessor platforms and formalize results for spin-based protocols, namely the *multiprocessor SRP* (MSRP) [11]. After verifying the analysis developed by Gai et al. [11] for non-nested critical sections, we will focus on the improved bound based on linear programming by Wieder and Brandenburg [12]. Ultimately, we plan to formalize the blocking analysis for well-ordered nested critical sections by Biondi et al. [13], and if possible, generalize this result by incrementally removing or weakening nesting hypotheses, using Coq's abilities to detect and flag required changes in the proofs.

References

[1] A. Gujarati, F. Cerqueira, and B. Brandenburg, "Revised Version: Schedulability analysis of the Linux push and pull scheduler with arbitrary processor affinities, revision 1," Available at: https://www.mpi-sws.org/~bbb/papers/, 2015.

[2] ——, "Correspondence article: A correction of the reduction-based schedulability analysis for APA scheduling. To appear." *Real-Time Systems*, 2018.

[3] R. Bril, J. Lukkien, R. Davis, and A. Burns, "Message response time analysis for ideal controller area network (CAN) refuted," *Proceedings of the 5th International Workshop on Real-Time Networks (RTN'06)*, 2006.

[4] G. Nelissen, J. Fonseca, G. Raravi, and V. Nelis, "Timing analysis of fixed priority self-suspending sporadic tasks," in *Proceedings of the 27th Euromicro Conference on Real-Time Systems (ECRTS'15)*, 2015.

[5] J.-J. Chen, G. Nelissen, W.-H. Huang, M. Yang, B. Brandenburg, K. Bletsas, C. Liu, P. Richard, F. Ridouard, N. Audsley, R. Rajkumar, and D. de Niz, "Many suspensions, many problems: A review of self-suspending tasks in real-time systems," Department of Computer Science, TU Dortmund, Tech. Rep. 854, 2016.

[6] R. Davis, A. Burns, J. Marinho, V. Nelis, S. Petters, and M. Bertogna, "Global fixed priority scheduling with deferred pre-emption revisited," Univ. of York, Tech. Rep. YCS-2013-483, 2013.

[7] R. Devillers and J. Goossens, "Liu and Layland's schedulability test revisited," *Information Processing Letters*, vol. 73, no. 5, pp. 157–161, 2000.

[8] F. Cerqueira, F. Stutz, and B. Brandenburg, "PROSA: A case for readable mechanized schedulability analysis," in *Proceedings of the 28th Euromicro Conference on Real-Time Systems (ECRTS'16)*, 2016.

[9] F. Cerqueira, G. Nelissen, and B. Brandenburg, "On strong and weak sustainability, with an application to self-suspending real-time tasks," in *Proceedings of the 30th Euromicro Conference on Real-Time Systems (ECRTS'18)*, 2018.

[10] X. Guo, S. Quinton, P. Fradet, and J.-F. Monin, "Work In Progress: Toward a Coq-certified Tool for the Schedulability Analysis of Tasks with Offsets," in *Work in Progress Session, Real-Time Systems Symposium (RTSS'17)*, Paris, France, 2017.

[11] P. Gai, G. Lipari, and M. Di Natale, "Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip," in *Proceedings of the 22nd Real-Time Systems Symposium (RTSS'01)*, 2001.

[12] A. Wieder and B. Brandenburg, "On spin locks in AUTOSAR: Blocking analysis of FIFO, unordered, and priority-ordered spin locks," in *Proceedings of the 34th Real-Time Systems Symposium (RTSS'13)*, 2013.

[13] A. Biondi, B. Brandenburg, and A. Wieder, "A blocking bound for nested FIFO spin locks," in *Proceedings of the 37th Real-Time Systems Symposium (RTSS'16)*, 2016.