A generic approach for the certified schedulability analysis of software systems

Xiaojie Guo, Lionel Rieg, Paolo Torrini

A generic approach for the certified schedulability analysis of software systems

1st Xiaojie Guo
RealTime-at-Work
Grenoble, France
xiaojie.guo@realtimeatwork.com

2nd Lionel Rieg
VERIMAG
UGA, CNRS, Grenoble INP
Grenoble, France
lionel.rieg@univ-grenoble-alpes.fr

3rd Paolo Torrini
INRIA Grenoble – Rhône-Alpes
Grenoble, France
paolo.torrini@inria.fr

***Abstract*—Embedded systems often need to react in a timely manner. Life-critical or mission-critical ones require assurance that they comply with these real-time requirements. In particular, schedulability analysis is both essential and difficult to get right. Formal methods can help as they are a powerful tool for ensuring properties with the highest assurance level.**

We describe a case study for the FPP and EDF policies providing end-to-end assurance by connecting the schedulability analysis tool Prosa and the real-time OS kernel RT-CertiKOS, both using the Coq proof assistant to prove their results. Analyzing precisely the key ideas underlying this connection, we improve it to make it more generic and reduce the associated proof burden. We thus sketch a refined method which allows for providing formal schedulability guarantees to other OSes or low-level components with minimal effort.

***Index Terms*—Formal Methods, Proof Assistant, Real-Time Scheduling, OS Kernel, Schedulability Analysis.**

I. Introduction

A. Context and motivation

Decades after Liu and Layland's seminar 1973 paper, verification of real-time properties remains a very active research topic. One major reason for this is that systems, and the models used to analyze them, have become more and more complex over time. From uniprocessor systems of periodic tasks, we have moved to systems with multicore architectures, GPUs, NoCs, complex task models such as DAGs, etc. There is another reason though: real-time systems verification has almost exclusively used pen-and-paper proofs, and has often relied quite heavily on intuition and Gantt charts for key intermediate results in such proofs — typically, for identifying the activation scenario leading to the worst-case response time of a task. This has made it difficult to build on top of existing results when introducing a new feature in the system model. Although similar to the state of the art, a new proof had to be written from scratch. The similarity in the proof structure of many schedulability analysis results was both an advantage (deriving a new proof was not so difficult) and a drawback, because it proved misleading several times [1], [2]. Following a few independent attempts at providing a formal framework for real-time systems analysis, Prosa [3], which is written in the Coq [4]–[6] proof assistant, has established itself as the first substantial library of computer-verified proofs in this area.

Meanwhile, formally verified software systems have become a reality, with tools such as VST [7], [8] (for C code), Iris [9], [10] (separation logic), RustBelt [11], [12] (for Rust code) and achievements such as the verified optimizing C compiler CompCert [13], [14], the verification of the distributed consensus algorithms Paxos [15] and Raft [16] or the verification of cryptographic protocols by EasyCrypt [17].

For example, proof assistants such as Isabelle/HOL [18], [19] or Coq have been used for the verification of two microkernels, respectively seL4 [20] and mCertiKOS [21], [22]. While functional properties are at the core of the verification process of such systems, real-time properties are also needed. In the case of RT-CertiKOS (a real-time extension of mCertiKOS), a schedulability analysis for the Fixed Priority Preemptive (FPP) policy was obtained by connecting the kernel to Prosa [23]. This first effort towards applying Prosa to an actual system proved promising: it did not require major changes to the library of the kernel and only limited work was needed for interfacing both.

B. Contribution

In this paper, we investigate how to go further in this direction. Specifically, we connect the EDF scheduler of RT-CertiKOS to Prosa to evaluate how much work is required for such a rather straightforward extension. As we hoped, this shows that the approach proposed in [23] smoothly applies to other scheduling policies. In addition, this new connection works with the latest version of Prosa, which is more modular than the original one and is likely to serve as reference for the next few years.

Second, based on lessons learned from these first experiments, we propose a generic approach for connecting Prosa to other systems. Such an approach relies on an intermediate layer in Coq that is independent of the target system, thus providing generic proofs that can be reused for different kernels.

C. Outline of the paper

First, related work are presented in Section II. Section III presents Prosa, the schedulability analysis library which is in the background of all the work discussed in this paper. Section IV presents RT-CertiKOS, the certified real-time OS kernel of which the scheduler we are verifying is part. An overview of

the technique used for the connection is given in Section V. Then, Section VI presents the ProKOS approach to connect RT-CertiKOS and Prosa on a case study for two scheduling policies: fixed priority preemptive (FPP) and earliest deadline first (EDF). Section VII analyzes the benefits and drawbacks of this method and presents a novel, different approach to connect a concrete scheduler with Prosa, relying on a state-based scheduler which we expect to be more easily generalizable.

II. Related Work

The work presented in this paper relates to computer assisted proofs, to verification of OS kernels as well as to schedulability analysis of real-time systems.

A. Computer assisted proofs

Proof assistants are computer programs used to help formal proofs, which provide (a) a language for mathematical definitions and properties; (b) an interactive system assisting the user in writing all details of the proofs, thus ensuring their correctness by construction. They are well suited for building large libraries of shared definitions for models, problems, algorithms, and theorems, ensuring their mutual consistency, and reusability. In the long run, this makes it possible to build new and increasingly intricate but sound results.

In this paper, we use Coq [4]–[6], an interactive theorem prover based on a very expressive type system called the Calculus of (Co-)Inductive Constructions, whose implementation is based on an advanced functional programming language named Gallina. Coq provides strong support for reasoning about functional programs and supports program extraction to all-purpose functional languages such as OCaml and Haskell. SSReflect is an extension of Coq, used in particular by Prosa, which relies on the computational capabilities of the underlying language and on reflection to deal more efficiently with Boolean logic and decidable types.

B. Verification of real-time OS kernels

There is a lot of work about formal verification of software systems, see for instance [24] for a survey about OS kernels. Therefore, we restrict our attention to verification of real-time OS kernels using proof assistants. We also do not consider WCET computation, be it of the kernel itself (*e.g.,* [25], [26]) or of the task set we consider. This is a complementary but clearly distinct task to get verified time bounds.

The eChronos OS [27], [28] is a real-time OS running on single-core embedded systems with a certified proof that the scheduler always picks the highest-priority pending task. Xu et al. [29] verify the functional correctness of μC/OS-II [30], a real-time operating system with optimizations such as bitmaps. PikeOS [31] is a commercial OS kernel proving functional correctness of core parts of its implementation with respect to its API. Unlike most other verification projects, it starts from an existing kernel rather than building one specifically designed for verification. Pip [32] is a minimalistic separation kernel based on an executable Coq model, which delegates scheduling to userland and therefore could be easily integrated with different schedulers.

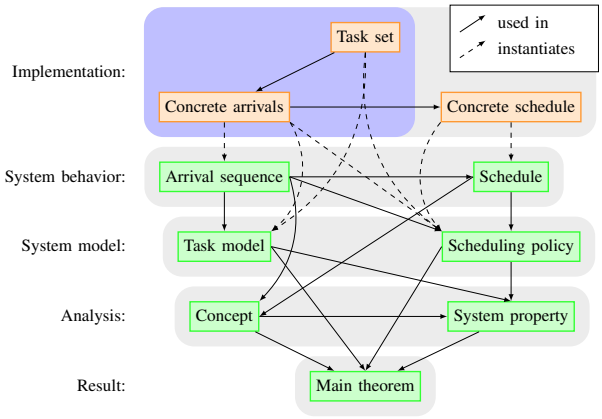

Fig. 1. An overview of Prosa layers

C. Schedulability analysis

Schedulability analysis as a key theory in the real-time community has been widely studied in the past decades. Liu and Layland's seminal work [33] presents a schedulability analysis technique for a simple system model described as a set of assumptions. A lot of research since then [34]–[38] has aimed to capture more realistic (in terms of executions and arrival model) and complex system models by generalizing those assumptions.

In order to provide formal guarantees to those results, several formal approaches have been used for the formalism of schedulability analyses, such as model checking [39]–[41], temporal logic [42], [43], and theorem proving [44], [45]. Still, existing work on formal proofs for real-time systems has remained rather scarce until the Prosa library was introduced.

III. Prosa

The Prosa library[1] [3], [48] was initially developed by Cerqueira and Brandenburg for formally proving existing and new schedulability analyses. The authors prioritize readability to make the project accessible to researchers familiar with real-time scheduling theories but without prior experience in formal methods. This feature makes the formal specifications in Prosa easily reviewed, understood and validated by the community. The library has been developed with a focus on modularity and extensibility, in accordance with its high-level character.

The library is now organized into five basic components as shown in Figure 1.

System behavior The base representation of system behavior is based on discrete time traces representing infinite sequences of events. There are two such kinds of sequences: arrival sequences which record requests for service called job activations and schedules which record processor states, *e.g.,* scheduled jobs, overhead. For instance, for an ideal uniprocessor (in which there is no overhead), a processor state denotes that there is a job scheduled or that the processor is idle.

[1]It is (being) further developed through a French project CASERM [46] and a French-German RT-Proofs project [47].

System model Based on system behavior, task models (arrival patterns and execution time models), processors, scheduling policies can be defined. A system model defines a set of possible behaviors that satisfy it. These models and policies are axiomatic in the sense that they are given as predicates on arrival sequences and schedules, not as generating and scheduling functions. In this component, some properties of these models are provided and proved.

Analysis This part of the library contains definitions needed for various analyses (*e.g.,* busy window, schedulability) as well as the actual proofs of response time and schedulability analyses.

Result An additional component, relying on the Analysis component, is used for collecting proven high-level analysis results. It actually contains results of EDF optimality and response time analyses for various policies such as EDF, FPP, or FPNP.

Implementation This library component contains implemented examples of arrival sequences and schedulers. The main purpose of this part of the Prosa library is to use concrete events/programs (*e.g.,* schedulers, concrete tasks, concrete arrivals) to validate the specifications axiomatized in the system model component. For instance, to validate the specification of the FPP scheduling policy, we implement a FPP scheduler program and prove that it satisfies indeed the properties described in that specification. This component can be an interface between concrete system schedules and the proven results in Prosa in order to benefit from each other.

A. System Behavior

The basic definitions in Prosa relate to concrete system behavior. Time is discrete and can be seen as scheduling ticks: duration is given in numbers of ticks and instants are given as numbers of ticks from the initialization. As key elements in both arrival sequences and schedules, jobs (*i.e.,* instances of tasks) are characterized as follows.

Definition 1 (Job)**.** *A job $\jmath$ is defined by a task $\tau_\jmath$, a positive cost $\mathbf{c}_\jmath$, and a unique identifier.*

We do not use the identifier directly, it is only used to distinguish jobs of the same task in traces. An arrival sequence is a sequence of job activations, from which the actual workload that must be scheduled at a given time instant can be deduced.

Definition 2 (Arrival sequence)**.** *An arrival sequence is a function ρ mapping any time instant t to a finite (possibly empty) set of jobs $\rho(t)$. A given job appears at most once in an arrival sequence.*

The arrival time (in ticks) of a job $\jmath$ appearing in an arrival sequence is written $\mathbf{a}_\jmath$. The fact that the job $\jmath$ appears at instant t in the arrival sequence ρ is formalized as $\jmath \in \rho(t)$, so that $\jmath \in \rho(t) \Rightarrow \mathbf{a}_\jmath = t$.

The scheduler is not modeled as a function, instead, we work with schedules: traces of processor states reflecting schedule information (*e.g.,* scheduled jobs, cores on which jobs are scheduled, overheads *etc.*).

Definition 3 (Schedule)**.** *A schedule is a function σ mapping any time instant t to a processor state.*

This definition is generic, and we obtain a schedule for a specific platform by specifying the processor states. For instance, a schedule for an ideal uniprocessor is defined as follows:

Definition 4 (Uniprocessor schedule)**.** *A uniprocessor schedule is a function σ mapping any time instant t to either the job scheduled at time t or $\bot$.*

This reflects the fact that for uniprocessors, the processor is either scheduling a job or idle (in other words, a uniprocessor cannot schedule more that one job at a time). Given an arrival sequence ρ and a schedule σ over ρ —that is, the scheduled jobs in σ are from ρ—, a job $\jmath \in \rho$ is said to be scheduled at an instant t if $\sigma(t) = \jmath$, the service received by $\jmath$ up to time t is the number of instants before t at which $\jmath$ is scheduled (this reflects the fact that all scheduling overheads are assumed to be zero). A job $\jmath$ is said to be completed at time t if the service it received up to time t is equal to its cost $\mathbf{c}_\jmath$. A job $\jmath$ is said to be pending at time t if it has arrived before time t and is not completed at time t. The response time of job $\jmath$ is defined as a duration between its arrival time and its completion time. For a task, its worst-case response time (WCRT) is the maximum value among all its jobs' response times. In a well-formed schedule, only pending jobs can be scheduled. From now on, we only consider well-formed schedules. A job $\jmath$ is said to be schedulable if it is completed before its absolute deadline. The absolute deadline $\mathbf{d}_\jmath$ of a job $\jmath$ is defined by its arrival time $\mathbf{a}_\jmath$ plus its task's relative deadline $\mathbf{D}_{\tau_\jmath}$: $\mathbf{d}_\jmath = \mathbf{a}_\jmath + \mathbf{D}_{\tau_\jmath}$.

B. System Model

In order to specify the behavior of the system we are interested in, Prosa introduces predicates on traces for which the response time analysis provides guarantees.

We now focus on the definitions related to the sporadic task model and the two simple scheduling policies: FPP and EDF.

Definition 5 (Sporadic task)**.** *A sporadic task τ is defined by a deadline $D_\tau \in \mathbb{N}$, a minimal inter-arrival time $T_\tau \in \mathbb{N}$, a worst case execution time (WCET) $\mathbf{C}\tau$, and a priority $k \in \mathbb{N}$. When D_τ is equal to T_τ, the deadline is said implicit.*

The sporadic task model is specified by a sporadic arrival model and a cost model.

In the sporadic arrival model, consecutive activations of a task τ are separated by a minimum distance T_τ: an arrival sequence ρ is sporadic if for any two distinct jobs $\jmath_1, \jmath_2 \in \rho$ of the same task τ, $|\mathbf{a}_{\jmath_1} - \mathbf{a}_{\jmath_2}| \geq T_\tau$. Periodic arrivals are a particular case of this model where T_τ is the period and jobs arrives exactly at intervals of T_τ. This is sufficient for us as the schedulability analysis for FPP and EDF yields the same bounds for sporadic and periodic activations.

The considered cost model is a constraint on activations: jobs in the arrival sequence must respect the WCET of their task, that is, for any $\jmath \in \rho$, $\mathbf{c}_\jmath \leq \mathbf{C}_{\tau_\jmath}$.

Both the FPP and EDF[2] policies are particular cases of the JLPP one, which is modeled in Prosa as two constraints on the schedule: it must be work conserving and respect the priority preemption.

Definition 6 (Job-level preemption scheduling policy)**.** *A schedule σ over an arrival sequence ρ respects the job-level preemption scheduling policy (JLPP) if and only if the two following conditions hold at any instant t:*

- *(Work conserving) If there is a pending job not scheduled at time t, then there must be another job scheduled at time t.*
- *(Priority preemption) If a job is scheduled at time t, then it has the highest priority among all pending jobs at time t. Specifically, for FPP the scheduled job is the job whose task has the highest priority among all pending tasks, while for EDF it is the job that has the smallest absolute deadline.*

C. Analysis

Prosa contains response time analyses of a set of sporadic tasks dispatched on a uniprocessor respecting the JLPP scheduling policy, hence also FPP or EDF.

We now focus on the specification of such analyses in Prosa. Consider a set S of sporadic tasks and a task τ_k of priority k from that set to analyze. The response time analysis computes the WCRT of τ_k by maximizing all workloads requested by tasks with a priority higher than or equal to k. The task workload bound is defined as follows:

Definition 7 (Task workload bound)**.** *Given a specific task $\tau \in S$ and a duration Δ, the maximum workload within Δ is*

$$wl^+_\tau(\Delta) := \lceil \frac{\Delta}{T_\tau} \rceil C_\tau$$

Theorem 1 (FPP WCRT bound)**.** *Given a sporadic task set S and a task $\tau_k \in S$, then for any $R > 0$ such that $R \geq wl^+_{S,k}(R)$, any job $\jmath$ of τ_k in a FPP schedule over an arrival sequence ρ is completed by $\mathbf{a}_\jmath + R$, where*

$$wl^+_{S,k}(\Delta) := \sum_{\substack{\tau_j \in S \\ j \leq k}} wl^+_{\tau_j}(\Delta)$$

Note that a response time bound R^+_k for a task $\tau_k \in S$ can be computed by the least positive fixed point of the function $wl^+_{S,k}(\Delta)$. It is the exact WCRT when deadlines are implicit, while it is an upper bound when deadlines are arbitrary. Using a response time bound, we can derive a schedulability criterion by requiring this bound to be smaller than or equal to the deadline of task τ_k.

The EDF WCRT bound is also provided in Prosa. We do not present it here in detail because this will not impact the following presentation of this paper. Interested reader can find more details here [49].

[2]In this paper, we focus on fully-preemptive EDF.

IV. The mCertiKOS & RT-CertiKOS OS Kernels

A. mCertiKOS & RT-CertiKOS

mCertiKOS [21], [22] is a single-core sequential OS kernel for the Intel x86 architecture whose functional correctness has been mechanically verified by the Coq proof assistant, ensuring that no incorrect behavior (w.r.t. the specification) can happen. In order not to jeopardize the formal guarantees provided by these proofs, the C code of the kernel is compiled using the CompCert compiler [13], [14], which also enjoys a mechanized formal proof of correctness ensuring that no miscompilation occurs and that the behavior of the generated assembly code is the same[3] than the one of the starting C code. Thus, the properties proved about the kernel are also valid at the assembly level, giving end-to-end verification.

RT-CertiKOS [50], [51] is a real-time extension of the mCertiKOS kernel, featuring preemptive interrupts, a timer, and preemptive scheduling. The sequential restriction greatly simplifies the implementation of the OS kernel. However, the lack of kernel preemption can degrade the responsiveness of the whole system.[4] In addition to the functional correctness inherited from mCertiKOS, RT-CertiKOS features mechanized proofs of both spatial and temporal isolation (including scheduling) between components (be it the kernel or user code).

Both these kernels are decomposed into small parts, called abstraction layers [52], that permits splitting the verification of the whole kernel into the verification of each feature independently of others. These layers are essentially a way to combine code fragments and their interface with correctness proofs. They consist of four elements:

(a) a piece of code (written in assembly or C);
(b) an underlay, the interface that the code relies on (written in Gallina, the programming language of Coq);
(c) an overlay, the interface that the code provides (written in Gallina);
(d) a simulation proof (done with the Coq proof assistant) ensuring that the code running on top of the underlay indeed provides the functionalities described by the overlay.

Thanks to this technique, all implementation details can be abstracted away and the reasoning is made on the interface model only.

B. Scheduling inside RT-CertiKOS

RT-CertiKOS supports user-level preemptive scheduling, with either fixed or dynamic priorities, following the FPP or EDF policy. Its scheduler is invoked by timer interrupts periodically, dividing CPU time into intervals, called time slots or ticks.

[3]It is actually not an equivalence but a refinement between the assembly code and the C code, meaning that every possible execution of the assembly code is allowed by the C code. This is due to several factors, such as optimization removing undefined behaviors or implementation-specific behaviors.

[4]This can be partly mitigated by delegating long interrupt-handling routines to user mode, the kernel mode being used only to schedule these routines.

a) Task model: Real-time tasks are defined by a fixed priority, a period and a budget (or WCET), the latter two being given in ticks. They are assumed to be strictly periodic, with hard implicit deadline, that is, the deadline is the start of the next period and no deadline miss is allowed at all. We only allow one task per priority level, so that priority levels can be used as task identifiers and, in the case of the FPP policy, that the schedule is deterministic. Following Prosa, the instance of a task in a period is called a job and it is defined by its task and a period index.

b) The RT-CertiKOS scheduler: By iterating over the jobs in decreasing priority order (this order being the only difference between the FPP and EDF policies), the scheduler selects the next job to execute by picking the pending one with highest priority. In order to enforce that no task can overrun, it keeps track of the budget left to each task, and refill these (to the WCET value) at the start of a new period. Its abstraction is a Coq function that iterates over an array of task control blocks (TCB), updates them, and returns the highest task identifier available for scheduling using the TCB array for FPP and a dedicated priority queue for EDF.

C. Simplified scheduling model

In order to simplify proofs related to scheduling, we create an intermediate Coq model in which all the parts of the kernel unrelated to scheduling are removed, such as context switching, memory management and so on. We prove that it faithfully represents the RT-CertiKOS scheduler by always generating the same schedule, thus ensuring that all results (such as schedulability analysis) on the schedule of this simplified model are also valid on the full RT-CertiKOS one.

This simplified scheduling model contains five elements:

ticks	the current time (in ticks);
cid	the identifier of the running process (if any);
schedule	the list of past scheduling decisions;
quanta	the remaining budget for each task;
queue	the priority queue containing pending jobs.

The queue is only required for EDF scheduling, we can do without it for FPP as the priorities are static. Notice that ticks and cid could be read off schedule (respectively as its length and first element) and, because the scheduling policy is simple (FPP or EDF), so could quanta and queue (albeit in a less straightforward way). Nevertheless, they are explicitly present in the RT-CertiKOS code because schedule is a logical variable having no existence in the actual code. Finally, one can remark that the schedule is always finite as it contains information only about past scheduling decision. This is a major difference with Prosa which considers infinite behaviors.

With this simplified scheduling model, the FPP scheduler is:

```
Definition scheduler (s: state) : state :=
  (** increase time *)
  let ticks' := ticks s + 1 in
  (** refill budgets at the start of new periods *)
  let quanta' := tick ticks' (quanta s) in
  (** find the highest priority pending task (if any) and schedule it *)
  match highest_pending quanta' with
  | Some id ⇒ (** schedule task id *)
    (** decrement id's budget *)
    let quanta'' :=
      ℤMap.set id (quanta'[id] - 1) quanta' in
    (** extend the schedule with id *)
    let schedule' := Some id :: schedule s in
    { ticks := ticks'; quanta := quanta'';
      cid := id; schedule := schedule' }
  | None ⇒ (** no task to schedule *)
    let schedule' := None :: schedule s in
    { ticks := ticks'; quanta := quanta';
      cid := default; schedule := schedule' }
  end.
```

The EDF scheduler is the same except for the queue which is used by `highest_pending` and must be updated.

This simplified scheduling model proves very valuable when proving invariants about the scheduler, as it frees us from handling all the details of an actual OS kernel that are irrelevant to scheduling.

V. Connecting RT-CertiKOS with Prosa

Our main goal is to use proven schedulability analysis from Prosa to formally verify RT-CertiKOS schedules built on sets of periodic tasks using a preemptive scheduling policy PP, which in the following may either be FPP or EDF. However, Prosa definitions cannot apply to RT-CertiKOS directly. Indeed, there are significant differences between the RT-CertiKOS representation of scheduling and the Prosa models of system behavior. The challenge we face is therefore to bridge this gap.

Leaving aside technicalities related to the use of SSReflect in Prosa, the more substantial differences are down to the fact that Prosa models have a high-level, axiomatic character which essentially builds on the notion of job, used as the fundamental abstraction. On the other hand, the RT-CertiKOS scheduler is defined as a stateful function. In RT-CertiKOS the notion of job does not play any fundamental role: it can be derived from the one of task, as each task only has one activation per period. Therefore, in RT-CertiKOS we can safely refer to activations by their task and omit jobs altogether, whereas the analysis in Prosa is entirely based on jobs, tasks being merely a way to create jobs.

The concept of schedule is central in Prosa as the scheduling policies are defined as properties over an (infinite) schedule. On the other hand, such a concept is computationally useless in RT-CertiKOS as the scheduler has an internal state. A closely related difference is the infinity of temporal representation. Prosa models are based on infinite traces and Prosa analysis only applies to them. On the opposite, even after adding the schedule as a logical variable to RT-CertiKOS, the scheduler will only give us a finite schedule at each time.

A. Modeling behavior in Prosa

In general, system behavior in Prosa is abstractly modeled as a relation between infinite arrival sequences and infinite schedules. By defining behaviors as pairs of arrival sequences and schedules, we may as well say that a Prosa model is a relation over behaviors. Both arrival sequences and schedules are functions of discrete time, where time units can be chosen,

in our case, to correspond to the time slots of our quantum-based scheduler.

The modeling relation is specified by properties which characterize the specific scheduling policy, as well as by some generic sanity conditions. In the case of PP, the specific property can be formalized in Coq as follows:

```
Definition PP_at (arr_seq: arrival_seq)
                 (sched: schedule) : Prop :=
  ∀ (t: time) (j: job),
    job_scheduled_at sched j t →
    ∀ (t': time) (j': job),
      t ≤ t' →
      job_arrived_at arr_seq j' ≤ t →
      job_scheduled_at sched j' t' →
      higher_or_equal_priority arr_seq j j'.
```

This can be read as saying that the job `j` scheduled at time `t` has higher or equal priority than any job pending at time `t`, that is, than any job `j'` which arrived before `t` and is scheduled at a later time `t'`. What is noticeable in this definition is its conceptual economy: it relies indeed on the knowledge of the infinite schedule (by referring to the future), but it does not make any assumption on the existence of a scheduler state. This is meant to ensure independence from implementation details and architectural aspects – a common feature of the properties defined in the Prosa library. Also notice that, given that `job_scheduled_at` and `job_arrived_at` are defined from an arrival sequence and a schedule, the only additional information required are the job deadlines.

Other conditions which a valid behavior should satisfy include a sanity property (only pending jobs are scheduled) and a work conservation one (as long as there are pending jobs, the processor cannot be idle). In the following, we will refer to PP compliance as the conjunction of these properties (and similarly for FPP and EDF). Given a system based on a set of sporadic tasks and its model as a set of behaviors that satisfy the compliance property, Prosa can carry out its schedulability analysis.

Especially in the case of EDF, in order to avoid non-determinism when two jobs have the same deadline, we further constrain priorities by defining them as the lexicographic order of deadlines and the task identifier. (Any other way to totally order tasks would work as well.) Thanks to this trick, we make sure that the schedule is deterministic, hence that the schedules in Prosa and RT-CertiKOS are necessarily the same. Notice that the problem does not arise with FPP, as we assume that any two tasks have distinct priorities and we prove that there is never more than one pending job for each task.

B. Defining the interface

In order to interface a concrete scheduler with Prosa, intuitively speaking, we need to show that, at each point in time, the scheduler satisfies PP compliance across a simple simulation relation which consists of extending the finite traces associated with the execution so far to infinite ones (as shown in Figure 2). A natural way to ensure this is by proving that, given an infinite arrival sequence `arr_seq` and a finite prefix `fin_arr_seq` of `arr_seq`, the finite schedule `fin_sched` obtained by the scheduler `S` from its finite input `fin_arr_seq` can be extended to an infinite one `sched` such that it satisfies the compliance relation `R` with respect to `arr_seq`. There are different ways in which the proof can be carried out. We are going to consider two distinct approaches, which differ essentially in the way they build the interface between Prosa and RT-CertiKOS.

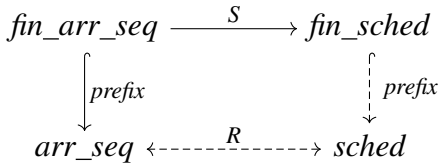

Fig. 2. Simulation diagram

The first approach, which we call ProKOS, consists in building a direct connection between RT-CertiKOS and Prosa through an abstract scheduling interface. This approach has been introduced for FPP in [23] (full details in Section VI), and it has been modified here to deal with EDF. The second approach consists in abstracting the concrete scheduler to an PP-compliant one, further defining a behavior interface with Prosa. This approach, detailed in Section VII, is less dependent on Prosa and it is meant to be more easily generalizable.

VI. The direct interface approach (ProKOS)

The basic rationale of ProKOS is to integrate RT-CertiKOS and Prosa through an interface which allows for translating to Prosa the outcome of each concrete scheduler execution as a finite behavior. Prosa library functions and lemmas are then used to extend finite traces to infinite ones and to prove compliance with the PP policy. This connection is as direct as possible, in the sense that it does not involve substantially any intermediate abstraction. The scheduling interface used in this approach answers to the purpose of clearly delineating which pieces of information flow between RT-CertiKOS and Prosa. It contains an inductive datatype that captures the PP property (either FPP or EDF) for a finite behavior, and we can use it as our representation in Prosa of the finite traces given by RT-CertiKOS. The extension of the finite schedule into an infinite one is performed by a logical scheduler implemented in Prosa.

The key issue when connecting RT-CertiKOS with Prosa is that the proven results in Prosa rely on two infinite traces of events as shown in Figure 1: arrival sequences and schedules, which RT-CertiKOS cannot provide as it only has access to past scheduling decisions and not future ones.

Specifically, the theorem proven in Prosa that we want to apply to RT-CertiKOS is:

Theorem 2 (Schedulability analysis for PP)**.** *Let S be a set of sporadic tasks, ρ be any infinite arrival sequence of S, and σ be any infinite schedule over ρ. If S passes the schedulability criterion presented at the end of Section III, if ρ respects the*

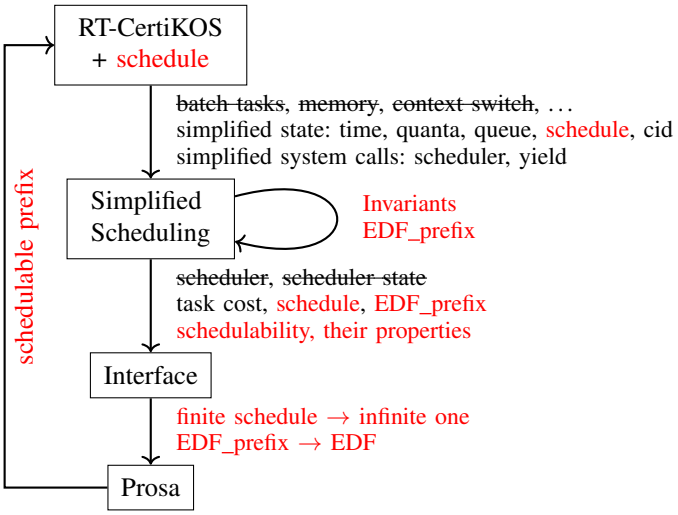

Fig. 3. Steps of the connection between Prosa and RT-CertiKOS for EDF. Logical elements are in red whereas code ones are in black.

sporadic model and σ satisfies the PP scheduling policy, then that schedule σ is schedulable. Formally,

$$\begin{aligned}&(\forall \tau_k \in S, R_k^+ \leq D_k)\\ &\implies \mathit{Sporadic}(\rho, S) \wedge \mathit{PP}(\rho, \sigma)\\ &\implies (\forall \jmath \in \rho, \forall \tau_k \in S, \tau(\jmath) = \tau_k \implies R_\jmath \leq D_k)\end{aligned}$$

where $R_\jmath$ is $\jmath$'s response time, R_k^+ is a τ_k's response time bound proven in Prosa for sporadic tasks scheduled with PP[5], and D_k is τ_k's deadline.

Note that the basis of the theorem is the two infinite traces ρ and σ. In RT-CertiKOS, we can only obtain a (finite) schedule prefix due to the fact that we do not have a priori information about the execution time of future jobs. Therefore, to apply Theorem 2 to the RT-CertiKOS schedule, we have to:

1) build an infinite arrival sequence and prove that it respects the sporadic model; and
2) build an infinite schedule using that arrival sequence and prove that it respects the PP scheduling policy.

Building an infinite arrival sequence is easy as tasks in RT-CertiKOS are assumed to be strictly periodic. The idea of building an infinite schedule, detailed in Section VI-C, is that we use the schedule prefix and assume the worst-case scenario for the future behavior.

A. Overview of the connection

This section and the next ones generalize the results of [23] to PP, that is, to both FPP and EDF. The overall architecture of the connection for EDF is depicted in Figure 3. The main steps can be summarized as follows:

- **On the RT-CertiKOS side.** The simplified scheduling model of RT-CertiKOS provides to the interface a finite schedule prefix σ^I_{pref} (the superscript I stands for interface) as well as the proof that it satisfies the PP-prefix property described below.
- **At the interface.** The job cost function is defined using the schedule prefix σ^I_{pref} in the past and the WCET for the future (worst-case scenario). The PP scheduling policy over a (finite) schedule prefix (Definition 9) is defined here. Note that this PP definition is different from Prosa's (Definition 6) which is based on an infinite schedule;
- **On the Prosa side.** We build an arrival sequence ρ^c using the schedule prefix σ^I_{pref} and the job cost function from the interface. Based on this arrival sequence and a PP scheduler from Prosa, we construct an infinite schedule σ^c with which we can apply the schedulability analysis. Then, we prove that σ^I_{pref} is a prefix of σ^c and is schedulable.

B. Handling main issues at the interface

Handling service and job cost. Prosa needs the arrival time and cost of each job. The former can be easily defined using its task period, but the latter is unknown. In RT-CertiKOS, and more generally in any OS, we only assume a bound on the execution time of a task, used as a budget. The exact execution time of each of its jobs is not known beforehand and can be observed only at runtime. On the opposite, Prosa assumes that costs for all jobs of all tasks are part of the problem description and thus are available from the start.

To fix this mismatch, we define a job cost function computed from the schedule prefix σ^I_{pref}.

Definition 8 (Job cost function)**.** *Let τ be a task, then the cost $c_n(\tau)$ of τ's n-th job is:*

$$c_n(\tau) = \begin{cases} \sum_{t=(n-1)T_\tau}^{nT_\tau} (\sigma^I_{pref}(t) = (\tau, \_)) \\ \qquad \mathit{if\ the\ job\ completes\ within}\ \sigma^I_{pref} \\ C \quad \mathit{otherwise} \end{cases}$$

where the Boolean expression $(\sigma^I_{pref}(t) = (\tau, \_))$ is implicitly converted to $\{0, 1\}$ with *true* mapped to 1. The job cost $c_n(\tau)$ is the service received during its n-th period if the job has completed (yielded) within σ^I_{pref}; otherwise, it is τ's WCET.

This definition relies on the computation of service in any period, which we also provide as part of the interface.

Handling infinite schedules. Prosa traces are based on an infinite schedule whereas in RT-CertiKOS, only a finite prefix can be known, up to the current time. Thus, we keep RT-CertiKOS's finite schedule σ^I_{pref} as is in the interface and it is up to Prosa to extend it into an infinite one, suitable for its analysis.

In order to use the schedulability analysis results of Prosa, two properties are needed: *(a)* the service received by each job is at most the WCET of its task; *(b)* the infinite schedule indeed follows the PP policy, that is, at each time, the scheduled job is the pending one with highest priority.

These two properties are proved on σ^I_{pref} in RT-CertiKOS. The former is easy. To prove the latter, we have to specify

[5]Note that FPP and EDF do not share the same response time bound R_k^+.

the PP scheduling policy over a schedule prefix (we call it PP-prefix for short) in the interface.

Definition 9 (PP-prefix)**.** *The PP scheduling policy is defined as an inductive predicate over a task set, an arrival sequence (implicit below), and a schedule prefix as follows:*

- *The empty trace is a PP-prefix.*
- *If you take any PP-prefix such that there is no job pending at its last time instant, then not scheduling any job at the next time instant will yield a PP-prefix.*
- *If you take any PP-prefix such that there are jobs pending at its last time instant, then one with the highest priority will be scheduled next.*

C. Connecting the interface with Prosa

The interface provides Prosa with a task set S, service and job cost functions c_n, and a schedule prefix σ^I_{pref} satisfying PP-prefix. We need to turn this information into the one used by Prosa for its analysis, that is, an arrival sequence and schedule.

1) The concrete infinite arrival sequence ρ^c: We first build a concrete (infinite) arrival sequence ρ^c from the schedule prefix σ^I_{pref} where the n-th job ($n > 0$) for a given task τ_k arrives at time $(n-1) \times T_{\tau_k}$ with cost $c_n(\tau_k)$. Note that jobs that do not arrive within the prefix cannot have yielded yet so that their costs are assumed to be the WCET of their tasks *i.e.,* we assume the worst case for the future. For instance, at time instant t, the $(\lfloor t/T_{\tau_k} \rfloor + 1)$-th job of task τ_k arrives if t is a multiple of T_{τ_k}.

2) The concrete schedule σ^c: Next, we need to turn the finite schedule prefix σ^I_{pref} provided by the interface into an infinite one. There are two possibilities: either build a full schedule from ρ^c using the Prosa PP scheduler, or start from σ^I_{pref} and extend it into an infinite schedule. The first technique gives for free the fact that the infinite schedule satisfies the PP policy from Prosa and the difficulty lies in proving that σ^I_{pref} is a prefix of this infinite schedule. The second technique is the opposite: σ^I_{pref} is by definition a prefix and the difficulty is proving that it satisfies the PP policy as specified by Prosa.

The key to both techniques is to prove the equivalence of both PP specifications: PP-prefix (Definition 9), defined in the interface; the other defined in Prosa on an infinite schedule (Definition 6 with the corresponding priority order).

Here, we present the first strategy: the concrete schedule σ^c is built using the arrival sequence ρ^c and a Prosa PP scheduler. For each time instant t, the Prosa PP scheduler computes all pending jobs using the concrete infinite arrival sequence and previous scheduling decisions up to t, and selects to execute the job with the highest priority among all pending jobs at t. Therefore, this concrete schedule σ^c naturally satisfies the following property: any job j from the concrete arrival sequence ρ^c is scheduled at time t if and only if it is pending and has the highest priority among all pending jobs. Formally:

$$\sigma^c(t) = j \iff Pending(j,t) \\ \wedge (\forall j' \in \rho^c, Pending(j',t) \implies p_j \geq p_{j'})$$

where $p_j \geq p_{j'}$ denotes that the priority of j is no less than that of j'. This property directly implies that σ^c satisfies *PP*.

3) σ^I_{pref} is a prefix of σ^c: In order to transfer the results of Prosa back to RT-CertiKOS, we prove via two steps that the schedule prefix σ^I_{pref} provided by the interface and the concrete infinite schedule σ^c match on the length of σ^I_{pref}, that is, σ^I_{pref} is a prefix of σ^c.

First, we prove that any prefix of the concrete schedule σ^c shorter than $len(\sigma^I_{pref})$ satisfies *PP-prefix*:

$$\forall l \in \mathbb{N},\ l \leq len(\sigma^I_{pref}) \implies \textit{PP-prefix}(S, \textit{Prefix}(\sigma^c, l))$$

Second, we use the fact that two PP schedule prefixes with the same arrival sequence are the same (Note that here we implicitly assume that all tasks have their own unique priority).

$$\textit{Prefix}(\sigma^c, len(\sigma^I_{pref})) = \sigma^I_{pref}$$

Finally, for a given task set accepted by the PP schedulability criterion (*i.e.,* $\forall \tau_k \in S, R^+_k \leq D_k$), we know that the computed infinite schedule σ^c is schedulable according to Theorem 2. Since σ^I_{pref} is a prefix of σ^c, we conclude that σ^I_{pref} is schedulable, as well as its extension to the end of the current period of any task.

VII. Discussion

A. Evaluation of the ProKOS approach for FPP and EDF

The ProKOS approach to connect Prosa and RT-CertiKOS provides a scalable mechanized formal proof of the correctness of the real-time scheduler of RT-CertiKOS. Furthermore, it does not incur any performance overhead as the C and assembly source code of RT-CertiKOS were not modified at all. The proof effort is also smaller compared to a dedicated schedulability proof within RT-CertiKOS (see [23]).

Switching the scheduling policy from FPP to EDF allows for assessing the adequacy and flexibility of this approach. Of course, some of the properties required by Prosa for EDF schedulability analysis are different from the ones for FPP so the interface should be adapted accordingly, in particular PP-prefix. Nonetheless, the definition of the interface can be almost kept as is, the only difference being in the computation of priorities, which are no longer statically determined by the task but dynamically computed depending on the absolute deadline, hence priorities depend on arrival times. Here is a summary of the changes:

1) The EDF scheduler also contains a priority queue to select the highest priority job to schedule;
2) The simplified scheduling model requires proving a coherence invariant between the queue and the quanta: a job is in queue iff its current quota is positive;
3) The PP-prefix property is updated to reflect the new policy (priorities depend on arrival times);
4) Proofs on the RT-CertiKOS and Prosa sides linking the various elements are updated.

Overall, apart from unrelated issues (changes internal to Prosa or RT-CertiKOS, mismatch over the Coq version between Prosa and RT-CertiKOS, porting our interface to Prosa's new architecture) the changes worked exactly as expected.

B. Lessons learned

The ProKOS approach is attractive for a number of reasons. On one hand, the proof that the EDF scheduling interface is refined by the RT-CertiKOS scheduler is made reusing the one for FPP with comparatively little change in the specification. We see this as a confirmation that the ProKOS method and abstractions are well-suited to this problem. Yet, the definition of the interface has a bottom-up feel: it is designed to be specific to a policy and we are embedding in Prosa finite traces produced by the concrete scheduler, delegating as much as possible to Prosa. Extending the traces is not trivial, but we can rely on a simple extension strategy (by stipulating that all the future jobs always execute up to the WCET of their task), and we can take advantage of a generic extension function which is already defined in Prosa. The high-level, modular character of Prosa makes our results quite abstract and independent of implementation details. Finally, by working in Prosa we can generally reuse many schedulability lemmas.

Nonetheless, we think there is room for improvement, in particular to avoid proof duplication. Indeed, we prove that the simplified scheduling model of RT-CertiKOS essentially satisfies the PP-policy (through invariants and the PP-prefix property), which also needs to be done about the Prosa PP-scheduler. One may even consider that having these two schedulers is one too many: only one should be required. Furthermore, all these proofs must be redone each time we switch to a new scheduling policy. Finally, converting between finite and infinite traces and between state-based and trace-based reasoning is not that easy.

Thus, a better approach might be to factor some of these efforts into more generic abstractions. One such solution could be to have the interface be a state-based abstract scheduler instead of a schedule.

C. The abstract scheduler approach

Reusing the terminology of Section V, this alternative approach consists in proving compliance of a scheduler independently of Prosa, then showing that this property is preserved by translating behaviors to Prosa. We do this by abstracting a concrete scheduler to a higher-level, yet state-based one.

The design of this abstract scheduler as a stateful recursive function is aimed at making the proofs as simple as possible by making all the current information available through the state. This is particularly true in proofs of simple time-dependent invariants, which are more naturally expressed as state properties rather than trace ones.

```
Record State := { scheduled : option job ;
                  service : job → nat ;
                  pending : list job }.
```

The state only needs to include information about the received service, and the list of pending jobs (specifically in order to avoid recomputation), in addition to the scheduled job (if any). In particular, here we choose to maintain the distinction between `pending`, as the list of jobs that have already arrived and are not yet complete (representing the information that is actually needed from the past schedule), and `service`, as a total function defined on every job (in the spirit of Prosa infinite traces). It would have been informatively equivalent to have `service` as a partial function only defined on the already arrived jobs.

We can directly define the scheduler input as a finite prefix of an infinite arrival sequence. Then, given its time-recursive definition, we naturally obtain an infinite schedule by iteration:

```
Fixpoint PP_sched (arr_seq: arrival_seq)
                  (t: time) : State :=
  match t with
  | 0 ⇒ initial_state arr_seq
  | S n ⇒ let st := PP_sched arr_seq n
          in state_update arr_seq (S n) st
  end.
```

In this way, the problem of converting finite traces into infinite one does not arise. It turns out quite easy to prove for every state the following state-based property stating that whenever there are pending jobs, the highest-priority one is scheduled.

```
Definition PP_correct (arr_seq: arrival_seq)
                      (st: State) : Prop :=
  ∀ j: job,
    In j (pending st) →
    exists j': job, scheduled st = Some j'
    ∧ In j' (pending st)
    ∧ higher_or_eq_priority arr_seq j' j.
```

As expected, from this property it is very easy to prove PP compliance. The last step on the Prosa side consists of showing that, under a natural translation of the infinite traces (only needed to bridge the gap between different types), the resulting ones satisfy PP compliance in Prosa, hence proving that our stateful scheduler actually refines the Prosa specification expressed by the abstract relation.

We experimented with this alternative approach for the EDF policy. Although here our abstract scheduler is specific to EDF, we believe that this approach can be naturally generalized by allowing for richer notions of state and policy parameters. The abstract scheduler is kept out of Prosa, both for technical reasons (no need of SSReflect) and conceptual ones. The kind of abstraction allowed by our design is not incompatible but comparatively orthogonal to Prosa, where the notion of processor state is intentionally kept minimal. On the other hand, the abstract scheduler involves a computational as well as declarative use of the state, and possibly the introduction of a generic state-based notion of scheduler correctness (with respect to which the `PP_correct` property in our example would boil down to a special case). The development of a generic scheduler model in Coq validating this approach is currently ongoing work by some of the authors.

VIII. Conclusion & Future Work

Critical systems require strong assurance of timely response, in particular for their schedulability analysis. The highest assurance can be obtained through formal methods, for instance via proof assistants such as Coq.

RT-CertiKOS is a real-time single-core OS-kernel whose correctness has been verified in Coq. Prosa is a state-of-the-art

schedulability analysis library proven with Coq. By connecting both, one can transfer the schedulability analyses of Prosa to RT-CertiKOS, ensuring that job deadlines are always met.

We detail a case study (partly taken from [23]) connecting Prosa and RT-CertiKOS for the FPP and EDF scheduling policies, highlighting its key steps and abstractions: a simplified scheduling model in RT-CertiKOS, an interface providing a schedule satisfying the policy under consideration, the extension of the schedule into an infinite trace.

Analyzing the similarities for these two policies, we observe that these abstractions are well-suited to prove a schedulability of RT-CertiKOS task sets, as most of the structure can be reused. Nevertheless, proof duplication and having to redo most proofs for each policy suggest other abstractions might alleviate the proof burden.

We sketch such an abstraction featuring an abstract scheduler rather than a schedule. This allows for a generic proof of compliance with the scheduling policy and permits to more easily provide certified schedulability analysis to other OSes. Indeed, once we have the generic proof of compliance, only a refinement proof between the implementation and this abstract scheduler is required. Future work is required to understand precisely the benefits of this new approach.

Acknowledgments: We would like to thank Sophie Quinton for insightful discussions and her comments and help with writing the paper.

REFERENCES

[1] R. I. Davis, A. Burns, R. J. Bril, and J. J. Lukkien, "Controller area network (CAN) schedulability analysis: Refuted, revisited and revised," Real Time Syst., vol. 35, no. 3, 2007.
[2] J. Chen, G. Nelissen, and W. Huang *et al.*, "Many suspensions, many problems: a review of self-suspending tasks in real-time systems," Real Time Syst., vol. 55, no. 1, 2019.
[3] F. Cerqueira, F. Stutz, and B. B. Brandenburg, "PROSA: A case for readable mechanized schedulability analysis," in Proc. ECRTS, 2016.
[4] Y. Bertot and P. Casteran, Coq'Art: The Calculus of Inductive Constructions. Springer, 2004.
[5] The Coq Development Team, The Coq Proof Assistant Reference Manual, 8th ed., INRIA, 2014.
[6] "The coq proof assistant," https://coq.inria.fr/.
[7] A. W. Appel, "Verified software toolchain," in Proc. NASA FM, ser. LNCS, vol. 7226, 2012.
[8] "The verified software toolchain," https://vst.cs.princeton.edu/.
[9] R. Krebbers, R. Jung, and A. Bizjak *et al.*, "The essence of higher-order concurrent separation logic," in ESOP, ser. LNCS, vol. 10201, 2017.
[10] "The iris project," https://iris-project.org/.
[11] R. Jung, J. Jourdan, R. Krebbers, and D. Dreyer, "Rustbelt: securing the foundations of the rust programming language," Proc. POPL, 2018.
[12] "The erc project rustbelt," https://plv.mpi-sws.org/rustbelt/.
[13] X. Leroy, "Formal verification of a realistic compiler," Communications of the ACM, vol. 52, no. 7, 2009.
[14] "The compcert C verified compiler," https://compcert.org/.
[15] L. Lamport, "Byzantizing paxos by refinement," in Proc. DISC, ser. LNCS, vol. 6950, 2011.
[16] J. R. Wilcox, D. Woos, and P. Panchekha *et al.*, "Verdi: a framework for implementing and formally verifying distributed systems," in Proc. PLDI, 2015.
[17] G. Barthe, F. Dupressoir, and B. Grégoire *et al.*, "Easycrypt: A tutorial," in FOSAD 2012/2013, ser. LNCS, vol. 8604, 2013.
[18] L. C. Paulson, "The foundation of a generic theorem prover," J. Autom. Reason., vol. 5, no. 3, 1989.
[19] "The isabelle proof assistant," https://isabelle.in.tum.de/.
[20] G. Klein, K. Elphinstone, and G. Heiser *et al.*, "seL4: formal verification of an OS kernel," in Proc. SOSP, 2009.
[21] R. Gu, Z. Shao, and H. Chen *et al.*, "Certikos: An extensible architecture for building certified concurrent os kernels," in Proc. OSDI, 2016.
[22] "Certikos: Certified kit operating system," http://flint.cs.yale.edu/certikos/.
[23] X. Guo, M. Lesourd, and M. Liu *et al.*, "Integrating formal schedulability analysis into a verified OS kernel," in Proc. CAV, 2019.
[24] G. Klein, R. Huuck, and B. Schlich, "Operating system verification," Journal of Automated Reasoning, vol. 42, no. 2-4, 2009.
[25] B. Blackham, Y. Shi, and S. Chattopadhyay *et al.*, "Timing analysis of a protected operating system kernel," in Proc. RTSS, 2011.
[26] T. Sewell, F. Kam, and G. Heiser, "High-assurance timing analysis for a high-assurance real-time operating system," Real-Time Syst., vol. 53, no. 5, 2017.
[27] J. Andronick, C. Lewis, and D. Matichuk *et al.*, "Proof of OS scheduling behavior in the presence of interrupt-induced concurrency," in Proc. ITP, 2016.
[28] J. Andronick, C. Lewis, and C. Morgan, "Controlled Owicki-Gries concurrency: Reasoning about the preemptible eChronos embedded operating system," in Proc. MARS, 2015.
[29] F. Xu, M. Fu, and X. Feng *et al.*, "A practical verification framework for preemptive OS kernels," in Proc. CAV, 2016.
[30] J. J. Labrosse, Microc/OS-II, 2nd ed. R & D Books, 1998.
[31] C. Baumann, B. Beckert, H. Blasum, and T. Bormer, "Lessons learned from microkernel verification – specification is the new bottleneck," in Proc. SSV, ser. EPTCS, vol. 102, 2012.
[32] N. Jomaa, P. Torrini, and D. Nowak *et al.*, "Proof-oriented design of a separation kernel with minimal trusted computing base," in Proc. AVOCS'18, 2018.
[33] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," J. ACM, vol. 20, no. 1, 1973.
[34] K. Tindell and J. Clark, "Holistic schedulability analysis for distributed hard real-time systems," Microprocessing and microprogramming, vol. 40, no. 2-3, 1994.
[35] J. C. Palencia and M. G. Harbour, "Schedulability analysis for tasks with static and dynamic offsets," in Proc. RTSS, 1998.
[36] E. Bini and G. C. Buttazzo, "Schedulability analysis of periodic fixed priority systems," IEEE Trans. on Comp., vol. 53, no. 11, 2004.
[37] S. Baruah, "Techniques for multiprocessor global schedulability analysis," in Proc. RTSS, 2007.
[38] T. Feld, A. Biondi, and R. I. Davis *et al.*, "A survey of schedulability analysis techniques for rate-dependent tasks," Journal of Systems and Software, vol. 138, 2018.
[39] E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi, "Schedulability analysis of fixed-priority systems using timed automata," Theor. Comp. Sci., vol. 354, no. 2, 2006.
[40] N. Guan, Z. Gu, and Q. Deng *et al.*, "Exact schedulability analysis for static-priority global multiprocessor scheduling using model-checking," in Int. Workshop on Soft. Tech. for Emb. & Ubiq. Sys., 2007.
[41] M. Cordovilla, F. Boniol, E. Noulard, and C. Pagetti, "Multiprocessor schedulability analyser," in Proc. SAC, ser. SAC '11, 2011.
[42] Z. Yuhua and Z. Chaochen, "A formal proof of the deadline driven scheduler," in Int. Symp. on Formal Techniques in Real-Time and Fault-Tolerant Systems, 1994.
[43] Q. Xu and N. Zhan, "Formalising scheduling theories in duration calculus." Nord. J. Comput., vol. 14, no. 3, 2008.
[44] M. Wilding, "A machine-checked proof of the optimality of a real-time scheduling policy," in Proc. CAV, 1998.
[45] B. Dutertre, "The priority ceiling protocol: formalization and analysis using pvs," in Proc. RTSS, 1999.
[46] "Caserm: Design and analysis of reconfigurable multi-view embedded systems," https://project.inria.fr/caserm/.
[47] "RT-Proofs: Formal proofs for real-time systems," http://rt-proofs.inria.fr.
[48] F. Cerqueira and B. Brandenburg, "Prosa: Formally proven schedulability analysis," http://prosa.mpi-sws.org.
[49] S. Bozhko and B. B. Brandenburg, "Abstract response-time analysis: A formal foundation for the busy-window principle," in Proc. ECRTS, 2020.
[50] M. Liu, L. Rieg, and Z. Shao *et al.*, "Compositional verification of preemptive OS kernels with temporal and spatial isolation," Dept. of CS, Yale University, Tech. Rep. YALEU/DCS/TR-1549, 2019.
[51] ——, "Virtual timeline: a formal abstraction for verifying preemptive schedulers with temporal isolation," Proc. POPL, vol. 4, 2020.
[52] R. Gu, J. Koenig, and T. Ramananandro *et al.*, "Deep specifications and certified abstraction layers," in Proc. POPL, 2015.