Integrating Formal Schedulability Analysis into a Verified OS Kernel

Xiaojie Guo[1,2], Maxime Lesourd[1,2], Mengqi Liu[3], Lionel Rieg[1,3(✉)], and Zhong Shao[3]

[1] Univ. Grenoble Alpes, CNRS, Grenoble INP, VERIMAG, Grenoble, France
[2] Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG, Grenoble, France
[3] Yale University, New Haven, CT, USA
lionel.rieg@univ-grenoble-alpes.fr

Abstract. Formal verification of real-time systems is attractive because these systems often perform critical operations. Unlike non real-time systems, latency and response time guarantees are of critical importance in this setting, as much as functional correctness. Nevertheless, formal verification of real-time OSes usually stops the scheduling analysis at the policy level: they only prove that the scheduler (or its abstract model) satisfies some scheduling policy. In this paper, we go further and connect together Prosa, a verified schedulability analyzer, and RT-CertiKOS, a verified single-core sequential real-time OS kernel. Thus, we get a more general and extensible schedulability analysis proof for RT-CertiKOS, as well a concrete implementation validating Prosa models. It also showcases that it is realistic to connect two completely independent formal developments in a proof assistant.

Keywords: Formal methods · Proof assistant · Real-time scheduling · OS kernel · Schedulability analysis

1 Introduction

The real-time and OS communities have seen recent effort towards formal proofs, through several techniques such as model checking [16,22] and interactive theorem provers [7,14,17]. This trend is motivated by the high stakes of critical systems and the combinatorial complexity of considering all possible interleavings of states of a system, which makes pen-and-paper reasoning too error-prone.

Real-time OSes used in critical areas such as avionics and automobile applications must ensure not only functional correctness but also timing requirements. Indeed, a missed deadline may have catastrophic consequences. Schedulability analysis aims to guarantee the absence of deadline miss given a scheduling algorithm which decides which task is going to execute.

I. Dillig and S. Tasiran (Eds.): CAV 2019, LNCS 11562, pp. 496–514, 2019.
https://doi.org/10.1007/978-3-030-25543-5_28

In the current state of the art, the schedulability analysis is decoupled from the kernel code verification. This is good from a separation of concern perspective as both kernel verification and schedulability analysis are already complex enough without adding in the other. Nevertheless, this gap also means that both communities may lack validation from the other one.

On the one hand, schedulability analysis itself is error-prone, *e.g.,* a flaw was found in the original schedulability analysis [26,27,29] for the Controller Area Network bus, which is widely used in automobile. To tackle this issue, the Prosa library [7] provides mechanized schedulability proofs. This library is developed with a focus on readable specifications in order to ensure wide acceptance by the community. It is currently a reference for mechanized schedulability proofs and was able to verify several existing multicore scheduling policies under a new setting with jitter. However, some of its design decisions, in particular for task models and scheduling policies, are highly unusual and their adequacy to reality has never been justified by connecting them to a concrete OS kernel enforcing a real-time scheduling policy.

On the other hand, OS kernels are very sensitive and bug-prone pieces of code, which inspires a lot of existing work on using formal methods to prove functional correctness and other requirements, such as access control policies [17], scheduling policies [31], timing requirements, etc. One such verified OS kernel is RT-CertiKOS [21], developed by the Yale FLINT group and built on top of the sequential CertiKOS [9,13]. Its verification focuses on extensions beyond pure functional correctness, such as real-time guarantees and isolation between components. However, any major extension such as real-time adds a lot of proof burden.

In this paper, we solve both problems at once by combining the formal schedulability analysis given by Prosa with the functional correctness guarantees of RT-CertiKOS. Thus, we get a formal schedulability proof for this kernel: if it accepts a task set, then formal proofs ensure that there will be no deadline miss during execution. Furthermore, this work also produces a concrete instance of the definitions used in Prosa, ensuring their consistency and adequacy with a real system.

Contributions. In this paper, we make the following contributions:

- Definition of a clear interface for schedulability analysis between a kernel (here, RT-CertiKOS) and a schedulability analyzer (here, Prosa);
- A workaround for the mismatch between the notion of jobs in schedulability analysis (which contains actual execution time) and in OS scheduling through the scheduling trace;
- A way to extend a finite scheduling trace (from RT-CertiKOS) into an infinite one (for Prosa) while still satisfying the fixed priority preemptive (FPP) scheduling policy;
- A formally proven connection between RT-CertiKOS and Prosa, validating Prosa modeling choices and enabling RT-CertiKOS to benefit from the state-of-the-art schedulability results of Prosa.

Outline of the Paper. Section 2 introduces the Prosa library and its description of scheduling. In Sect. 3, we describe RT-CertiKOS, its scheduler, as well as the associated verification technique, abstraction layers. Section 4 then highlights the key differences between the models of Prosa and RT-CertiKOS, and how we resolve them. Finally, Sects. 5, 6, and 7, evaluate our work, present future work and related work before concluding.

2 Prosa

Prosa [7] is a Coq [25] library of models and analyses for real-time systems. The library is aimed towards the real-time community and provides models and analyses found in the literature with a focus on readable specifications.

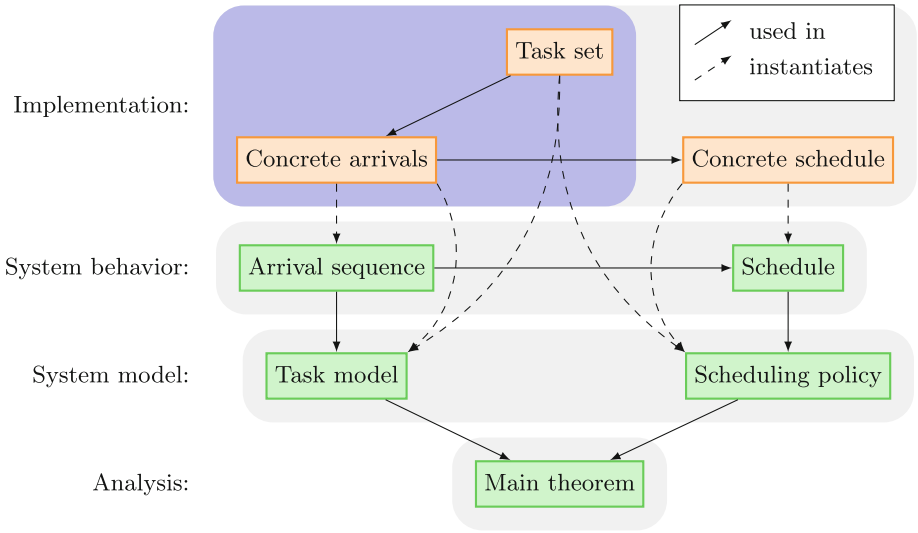

Fig. 1. An overview of Prosa layers

The library contains four basic layers, which are presented in Fig. 1:

System behavior. The base of the library is a model of discrete time traces as infinite sequences of events. We consider two such kinds of sequences: arrival sequences record requests for service called *job* activations and schedules record which job is able to progress.

System model. In order to reason about system behavior, jobs with similar properties are grouped into *tasks*. Based on system behavior, task models (arrival patterns and cost models) and scheduling policies are defined. These models are axiomatic in the sense that they are given as predicates on traces/schedules and not as generating and scheduling functions. In particular, a "FPP scheduler" (see Sect. 2.2) is modeled as "any trace satisfying the FPP policy".

Analysis. The library provides response time and schedulability analyses for these models.

Implementation. Finally, examples of traces and schedulers are implemented to validate the specifications axiomatized in the System model layer and to use the results proven in the Analysis layer. It is this part (more precisely, the top left dark block of Fig. 1) that is meant to connect with RT-CertiKOS.

2.1 System Behavior

The basic definitions in Prosa concern concrete system behavior. The notion of time used in the library corresponds to scheduling ticks: durations are given in number of ticks and instants are given as number of ticks from initialization of the system. For this paper, we focus on single-core systems[1] on which instances of a finite set $TaskSet$ of tasks are scheduled. To each task τ is associated a relative deadline D_τ which corresponds to the delay we want to guarantee between the activation of an instance of a task and its completion. We defer the definition of tasks (Definition 4) until their parameters are relevant and focus first on the modeling of system behavior in Prosa. The instances of tasks which are to be scheduled are called *jobs*.

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.

These jobs are used to describe the workload to be scheduled. This workload is defined by an *arrival sequence* which is a trace of job activations.

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 job can only appear once in an arrival sequence.

Since a job $\jmath$ can appear at most once in an arrival sequence ρ, we can define its *arrival time* $\mathbf{a}_\rho(\jmath)$ in ρ as the instant t such that $\jmath \in \rho(t)$.

We do not model the scheduler as a function, instead we work with *schedules* over an arrival sequence which are traces of scheduled jobs.

Definition 3 (Schedule). *A* schedule *over an arrival sequence ρ is a function σ which maps any time instant t to either a job appearing in ρ or $\bot$.*

The symbol $\bot$ is used for instants at which no job is scheduled. Given an arrival sequence ρ and a schedule σ over 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. A job $\jmath$ is said to be complete at time t if its service received up to time t is equal to its cost $\mathbf{c}_\jmath$ and $\jmath$ is said to be *pending at time t* if it has arrived before time t and is not complete at time t. From now on, we require schedules to only schedule pending jobs. A job $\jmath$ is said to be schedulable if it is complete by its *absolute deadline* $d_\jmath := \mathbf{a}_\rho(\jmath) + D_{\tau_\jmath}$.

[1] Multicore systems are handled by Prosa but we do not consider them here.

2.2 System Model

Task 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 *fixed priority preemptive* (FPP) scheduling policy.

Definition 4 (Sporadic FPP task). *A sporadic FPP task τ is defined by a deadline $D_\tau \in \mathbb{N}$, a minimal inter-arrival time $\delta_\tau^- \in \mathbb{N}$, a worst case execution time (WCET) $\mathbf{C}\tau$, and a priority $p_\tau \in \mathbb{N}$. When D_τ is equal to δ_τ^-, the deadline is said* implicit*.*

Sporadic Task Model. 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 δ_τ^-: an arrival sequence ρ is sporadic if for any two distinct jobs $j_1, j_2 \in \rho$ of the same task τ, $|\mathbf{a}_\rho(j_1) - \mathbf{a}_\rho(j_2)| \geq \delta_\tau^-$. Periodic arrivals are a particular case of this model where δ_τ^- is the period and jobs arrives exactly at intervals of δ_τ^-. This is sufficient for us as the schedulability analysis for FPP 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 $j \in \rho$, $\mathbf{c}_j \leq \mathbf{C}_{\tau_j}$.

FPP Scheduling Policy. The FPP policy is modeled in Prosa as two constraints on the schedule: it must be *work conserving*, that is, it cannot be idle when there are pending tasks; and it must respect the priorities, that is, a scheduled job always has the highest priority among pending jobs.

2.3 Analysis

Prosa contains a proof of Bertogna and Cirinei's [4] response time analysis for FPP single-core schedules of sporadic tasks, with exact bounds for implicit deadlines. The analysis is based on the following property of the maximum workload for these schedules.

Definition 5 (Maximum Workload). *Given a task $\tau \in TaskSet$ and a duration Δ, the maximum workload of the system* w.r.t. *τ within that duration is*

$$W_\tau(\Delta) := \sum_{\substack{\tau' \in TaskSet \\ p_{\tau'} \geq p_\tau}} \mathbf{C}_{\tau'} \times \left\lceil \frac{\Delta}{\delta_{\tau'}^-} \right\rceil$$

The maximum workload $W_\tau(\Delta)$ corresponds to the worst case activation pattern in which all tasks are simultaneously activated with maximum cost (WCET of their task) and minimal inter-arrival distance. It is an upper bound on the

amount of service required to schedule activations of the tasks with a priority higher than or equal to p_τ in any interval of size Δ. Based on this property, we can derive a response time bound for our system model if we can find a Δ larger than $W_\tau(\Delta)$.

Theorem 1 (Response Time Bound). *Given a sporadic taskset $TaskSet$ and a task $\tau \in TaskSet$ then for any $R > 0$ such that $R \geq W_\tau(R)$, any job $\jmath$ of task τ in an FPP schedule σ over an arrival sequence ρ is completed by $\mathbf{a}_\rho(\jmath) + R$.*

For instance, the smallest response time bound for a task $\tau \in TaskSet$ can be computed by the least positive fixed point of the function W_τ. Using this 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.

2.4 Implementation and Motivation for the Connection with RT-CertiKOS

The Prosa library includes functions to generate periodic traces and the corresponding FPP schedules, together with proofs of these properties and an instantiation of the schedulability criterion for these traces. This implementation was initially provided as a way to check that the modeling of the arrival model and scheduling policy are not contradictory and as such the implementation is as simple as possible. Although this is a good step in order to make the axiomatic definition of scheduling policies more acceptable, there is still room for improvement: these implementations are still rather ad-hoc and there is no connection to an actual system. This is where the link with RT-CertiKOS is beneficial to the Prosa ecosystem: it justifies that the model is indeed suitable for a concrete and independently developed real-time OS scheduler.

3 The RT-CertiKOS OS Kernel

RT-CertiKOS [21], developed by the Yale FLINT group, is a real-time extension of the single-core sequential CertiKOS [9,13],[2] whose functional correctness has been mechanized in the Coq proof assistant [25]. The sequential restriction greatly simplifies the implementation of the OS kernel. However, it does not support multicore, and the lack of kernel preemption can also degrade the responsiveness of the whole system. RT-CertiKOS proves spatial and temporal isolation (including schedulability) between components.

Both CertiKOS and RT-CertiKOS follow the same proof methodology, organized around the notion of abstraction layers that permits decomposition of the kernel into small pieces that are easier to verify.

[2] There is a multicore version of CertiKOS [14,15], but RT-CertiKOS is developed on top of the sequential version.

3.1 Abstraction Layers

Abstraction layers [13] are essentially a way to combine code fragments and their interface with simulation proofs. They consist of four elements: *(a)* a piece of code; *(b)* an *underlay*, the interface that the code relies on; *(c)* an *overlay*, the interface that the code provides; *(d)* a *simulation proof* ensuring that the code running on top of the underlay indeed provides the functionalities described in the overlay.

Implementation details of lower layers are encapsulated in higher layers, allowing to reason directly with the specifications rather than the implementation.

Notice that the underlay and overlay are specifications written in Coq and may be expressed using the semantics of several programming languages at once. This explains how CertiKOS (and RT-CertiKOS) manages to encompass both C and assembly code verification into a unified framework. Notice further that this notion of interface not only includes functions but also some *abstract state*, which exposes memory states of lower layers in a clean and structured way, and allows the overlay to access them only by invoking verified functions.

3.2 The Scheduler in RT-CertiKOS

RT-CertiKOS supports user-level fixed-priority preemptive scheduling. Its scheduler is invoked by timer interrupts periodically, dividing CPU time into intervals, which are called *time slots*, *time quanta*, or *time slices*.

Task Model. Each task in RT-CertiKOS is defined by a fixed priority, a period, and a budget (or WCET), the latter two being given in time slot units. Tasks are strictly periodic, with implicit hard deadlines, that is, the deadlines are the start of the next period and no deadline miss is allowed at all. While this is a restricted setting, it is enough to handle closed-loop control, used in control real-time systems. Furthermore, RT-CertiKOS only allows for fixed priorities in order to get maximum predictability, which is of utmost importance in critical systems. Finally, RT-CertiKOS also enforces budgets at the task level: in each period, a task cannot be scheduled for more than its specified budget.

Fixed-Priority Scheduler. The RT-CertiKOS scheduler maintains an integer array to keep track of time quantum usage for each task. Upon invocation, the scheduler first iterates over all tasks, replenishing quotas whenever a new period arrives. It then loops again and finds the highest priority task that has not used up its budget, followed by a decrement on the chosen task's current quota. Its abstraction is a Coq function that iterates over an abstract array of task control blocks, updates them, and returns the highest task identifier available for scheduling.

Yield System Call. Tasks do not always use up their budgets. A task can yield to relinquish any remaining quota, so that lower priority tasks may be scheduled earlier and more time slots may be dedicated to non real-time tasks.

3.3 Proof Methodology

Based on sequential CertiKOS, RT-CertiKOS [21] follows the idea of deep specifications[3] in which the specification should be rich enough to deduce any property of interest: there should never be any need to consider the implementation. In particular, even though its source code is written in both C and assembly, the underlay always abstracts the concrete memory states it operates on into abstract states, and abstracts concrete code into Coq functions that act as executable specification. Subsequent layers relying on this underlay will invoke Coq functions instead of the concrete code, thus hiding implementation details.

In the case of scheduling, there are essentially two functions: the scheduler and the yield system call. The scheduler relies on two concrete data structures: a counter tracking the current time (in time slot units) and an array tracking the current quota for each periodic task. The yield system call simply sets the remaining quota of the current task to zero. Both functions are verified in RT-CertiKOS, that is, formals proofs ensure that their C code implementations indeed simulate the corresponding Coq specifications.

3.4 Motivation for the Connection with Prosa

Upgrading an OS kernel into a real-time one is not an easy task. When one further adds formal proofs about functional correctness, isolation, and timing requirements, the proof burden becomes enormous. In particular, there is still room for future work on RT-CertiKOS, *e.g.,* a WCET analysis of its system calls.

In order to reduce the overall proof burden, it is important to try to delegate as much as possible to specialized libraries and tools. Thus, from the RT-CertiKOS perspective, the benefit of using Prosa is precisely to have state-of-the-art schedulability analyses already mechanized in Coq, without having to prove all these results.

Furthermore, the schedulability check of Prosa is only performed once while verifying the proofs, such that there is no runtime overhead and no loss of performance for RT-CertiKOS.

4 From RT-CertiKOS to Prosa: A Schedule Connection

Prosa definitions cannot apply to RT-CertiKOS directly. Indeed, the perspectives of Prosa and RT-CertiKOS on the real-time aspects of a system are not the same, which is reflected in the differences in their task models, their executions, and the information they need. In this section, we explain how we bridge these gaps to actually perform the connection. Table 1 summarizes the various definitions and proofs and how they relate to each other.

[3] https://deepspec.org/.

Table 1. Summary of the range of the various data between RT-CertiKOS and Prosa

RT-CertiKOS	Simplified Model	Interface	Prosa
scheduler quota array			
schedule prefix with batch tasks	schedule prefix		infinite schedule
	valid schedule prefix		valid infinite schedule
	FPP prefix		
			FPP
			schedulability analysis
			schedulable execution
schedulable prefix			

4.1 Interface Between RT-CertiKOS and Prosa

We design an interface to link RT-CertiKOS and Prosa, focusing on the precise amount of information that needs to be transmitted between them. The interface is shaped by the information Prosa needs to perform the schedulability analysis: a task set and a schedule, together with some properties.

Key Elements of the Interface. The task model we consider is the one of RT-CertiKOS, as it is more restrictive than the ones supported by Prosa. Tasks are defined by a priority level p, a period T_p and a WCET (more accurately a budget) C_p. Since we only allow one task per priority level, we identify tasks and priority levels and we write C_p, D_p, and T_p instead of C_τ, D_τ, and T_τ. In order for this setting to make sense, we assume the following inequality for each task p: $0 < C_p \leq T_p$. Notice that this is a particular case of Prosa's FPP task model (Definition 4). There is no definition of the jobs of a task as they can be easily defined from a task and a period number.

The second element Prosa needs is an infinite schedule. RT-CertiKOS cannot provide such an infinite schedule, as only a finite prefix can be known, up to the current time. Thus, we keep RT-CertiKOS's finite schedule as is in the interface and it is up to Prosa to extend it into an infinite one, suitable for its analysis.

Finally, Prosa needs two properties about the schedule: *(a)* any task receives no more service than its WCET in any period; *(b)* the schedule indeed follows the FPP policy. We refer to schedules satisfying these properties as *valid schedule prefixes*. Proving these properties falls to RT-CertiKOS.

Handling Service and Job Cost. 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 a schedule prefix: its value is the actual service received if the job has yielded and the WCET of its task otherwise. This definition relies on the computation of service in any period, which we also provide as part of the interface.

4.2 The RT-CertiKOS Side

Adding the Schedule in RT-CertiKOS. RT-CertiKOS only maintains the current state of the system, which the scheduler relies on, such as the current time and quota array. However, the interface requires a schedule trace. We introduce such a ghost variable in RT-CertiKOS, and update a few scheduling-related primitives to extend this trace whenever a task is scheduled.

This introduction adds absolutely no proof overhead, since it does not affect the scheduling decisions, thus existing proofs about the rest of the system still hold. Furthermore, it is a purely logical variable introduced through refinement, meaning that it does not exist in the C code, thus it causes no computation overhead.

Too Much Information in RT-CertiKOS. The full RT-CertiKOS model contains too much information compared to what the interface requires.

Firstly, services in RT-CertiKOS may affect a part of the state that is relevant to practical scheduling, but is of no interest to the scheduling model we want to verify, like batch tasks.

Secondly, due to the nature of *deep specification*, the abstraction of the whole scheduling operation contains more information than what is required for reasoning about real-time properties. For example, saving and restoring registers is essential for the correctness of context switches (thus, of the scheduler), but it is irrelevant to temporal properties.

Thirdly, specifications in RT-CertiKOS enumerate preconditions of the scheduler such as the correct configuration of the paging bit in the control register, the validity of the current stack and so on. These are required for other invariants of the kernel at other abstraction levels, but again they are irrelevant to scheduling.

Simplified Model of RT-CertiKOS. For all these reasons, we define a simplified scheduling model of RT-CertiKOS, with a much simpler abstract state containing only the data structures that are actually used in scheduling, from which the interface data and its properties must be derived. This simplified abstract state contains four fields:

ticks the current time, that is, the number of past time slots;
quanta a map giving the remaining quota for each priority;
cid the identifier of the running process (if it exists);
schedule the schedule prefix remembering past scheduling decisions.

This abstract state is not equivalent to the complete one, because it operates on a totally different abstract data type where all irrelevant fields are removed.

It is also more permissive: more transitions are allowed since it does not perform the sanity checks about preconditions such as being in kernel mode, host mode, etc. Nevertheless, we still have a simulation: any step in the full RT-CertiKOS is also allowed in the simplified version and results in the same scheduling decision and trace. This simulation is enough for our purposes as we are ultimately interested in the behavior of the full RT-CertiKOS.

Proving the Properties Required by Prosa. The interface requires two key properties: *(a)* the service received by each job is at most the WCET of its task; and *(b)* the schedule prefix follows FPP. These properties must be proven on the RT-CertiKOS side for any schedule that might be generated. This way, Prosa can rely on them through the interface.

Since RT-CertiKOS verification is based on state invariants rather than traces, we prove these properties using the following main invariants on the simplified scheduling model:

- the length of the schedule trace is the current time + 1 (the scheduler takes a decision for the *next* time slot);
- if a task has yielded in the current period, its remaining quota is 0;
- the service plus the remaining quota is equal to the job cost;
- the service received in any period is less than the WCET;
- pending jobs have two equivalent definitions (having positive remaining quota or having less service than their job cost);
- the current schedule follows FPP.

To prove that these statements are indeed invariants, we must prove that they are preserved by any step, that is, by the scheduler (triggered by the user-level timer interrupt) and by the yield system call (triggered by the user process), since all other kernel steps do not modify the scheduling data of the simplified scheduling model.

Simulation Between the Simplified Scheduling Model and RT-CertiKOS. To connect the full RT-CertiKOS model and the simplified one, we define a projection function *RData_proj* extracting the relevant fields from the full RT-CertiKOS state to build the simplified one.

As shown in Fig. 2, we prove that given a scheduler transition of RT-CertiKOS between the (full) states d and d', there is also a transition from their projections s and s' by invoking the simplified scheduler.[4] If the states d and s satisfy respectively the invariants for RT-CertiKOS and the simplified model, then so do d' and s' (they are invariants). As the states s and s' are projections of d and d', the invariants of s and s' also hold on the corresponding fields in d and d'. This allows us to utilize the invariants proved in the simplified model to establish properties on the full state of RT-CertiKOS. Notice that the schedulability property we study is a safety property (deadlines are never missed) and not a liveness one (everything is eventually scheduled).

[4] More precisely, we prove that `certikos_sched`(s) and *RData_proj*(d') are *extensionally* equal.

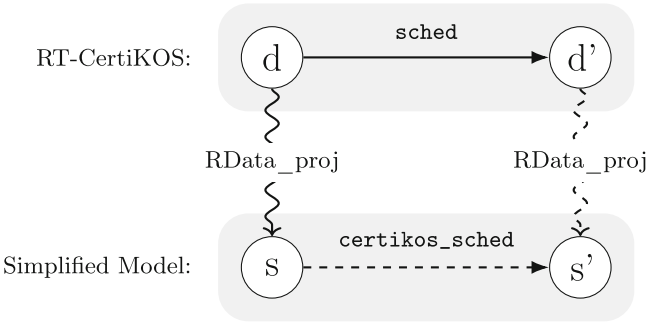

Fig. 2. Simulation between simplified scheduling and RT-CertiKOS

4.3 The Prosa Side

Proven Schedulability Analysis in Prosa. In order to use the response time bound of Sect. 2, we need to relate any finite schedule prefix from the interface to an arrival sequence and a schedule satisfying the model described in Sect. 2. We can then rely on any schedulability criterion (*e.g.,* the one described at the end of Sect. 2.3) to prove that the response time bound holds and deduce that any valid schedule prefix from the interface is indeed schedulable.

Bridging the Gap Between the Interface and Prosa. The interface provides Prosa with a task set, service and job cost functions, and a valid schedule prefix. We first build an arrival sequence from the schedule prefix where the n-th job ($n > 0$) for a given task p arrives at time $(n-1) \times T_p$ with the cost given by the interface. Note that jobs that do not arrive within the prefix cannot have yielded yet so that their costs is the WCET of their tasks: we assume the worst case for the future.

The arrival sequence is then defined by adding all jobs of each task p from $TaskSet$, that is, the arrival sequence at time t contains the $(\lfloor t/T_p \rfloor + 1)$-th job of p iff t is divisible by T_p.

Next, we need to turn the finite schedule prefix into an infinite one. There are two possibilities: either build a full schedule from the arrival sequence using the Prosa implementation of FPP, or start from the schedule prefix of the interface and extend it into an infinite one. The first technique gives for free the fact that the infinite schedule satisfies the FPP model from Prosa. The difficulty lies in proving that the schedule prefix from the interface is indeed a prefix of this infinite schedule. The second technique starts from the schedule prefix and the difficulty is proving that it satisfies the FPP model as specified on the Prosa side.

In this paper, we use the first strategy and prove that the prefix of the schedule built by Prosa is equal to the schedule prefix provided in the interface. To do so, we use the fact that two FPP schedule prefixes with the same arrival sequence and job costs (only known at runtime) are the same, provided we take care to properly remember when jobs yield.

Assuming that the task set is accepted by the schedulability criterion, we know that the Prosa schedule is schedulable and, since this implies that its prefix is also schedulable, we deduce that the valid schedule prefix given by the interface is schedulable.

5 Evaluation and Future Work

5.1 Evaluation

As the C and assembly source code of RT-CertiKOS was not modified at all, this connection does not introduce any overhead to its performance and there is no need for a new performance evaluation. Instead, we focus on the benefits this works brings and on the amount of work involved, described in Table 2.

Benefits for RT-CertiKOS and Prosa. The schedulability analysis already present in RT-CertiKOS was manually proved and took around 8k LoC to handle the precise setting described in this paper. By contrast, interfacing with Prosa requires 50% less proofs, is more flexible and can easily be extended (see Sect. 5.3). The introduction of a simplified scheduling model also reduced by 75% the size of proofs of invariants about the high-level abstract scheduler since we are freed from the unnecessary information described in Sect. 4.2.

On the Prosa side, having a complete formal connection with an actual OS kernel developed independently validates the modeling choices made for describing real-time systems. Indeed, seeing schedulers as predicates over scheduling traces is very general but one can legitimately wonder whether such predicates accurately describe reality.

Proof Effort. Designing a good interface allowed us to cleanly separate the work required on the RT-CertiKOS and Prosa sides.

On the RT-CertiKOS side, the design of the simplified scheduling setting was pretty straightforward, as was the correctness of the translation. Indeed, this translation is essentially a projection, except for batch tasks which are removed. Designing adequate inductive invariants to prove the two properties required by the interface was the most challenging part of this work and unsurprisingly, it took several iterations to find correct definitions.

On the Prosa side, building the arrival sequence and the infinite schedule is quite effortless given a prefix and a job cost function. The subtle thing was to find a good definition of the job cost function, which made the corresponding proofs significantly easier. Proving that the prefix of the built infinite schedule is the same as the interface prefix *w.r.t.* executions was troublesome for two reasons. First, the interface prefix contains an additional boolean representing whether the scheduled job yielded and which is used for computing job costs, whereas it does not exist in the built schedule. Second, the definition of the FPP property in the interface depends on a schedule prefix, while the one in Prosa depends on an infinite schedule.

Overall, we see the small amount of LoC required to perform this work as a validation of the adequacy of our method to the considered problem.

Table 2. Proof effort

Feature	Changes (LoC)
Adding a schedule field to RT-CertiKOS	15
Interface (with proofs)	380
Simplified scheduling	100
Proving the invariants about the simplified scheduling	950
Translation RT-CertiKOS → simplified scheduling	380
Conversion between ZArith and SSReflect	280
Translation interface → Prosa	1900
Using the schedulability analysis of Prosa	130
Total	4135

5.2 Lessons Learned

Beyond the particular artifact linking RT-CertiKOS with Prosa, what more general lessons can we learn from this connection?

First, using the same proof assistant greatly helps. Indeed, beyond the absence of technical hassle of inter-operability between different formal tools, it also avoids the pitfall of a formalization mismatch between both formal models and permits sharing common definitions.

Second, the creation of an explicit interface between both tools clearly marks the flow of information, stays focused on the essential information, and delimits the "proof responsibility": which side is responsible for proving which fact. It also segregate the proof techniques used on each side so as not to pollute the other one, either on a technical aspect (vanilla Coq for RT-CertiKOS *vs* the SSReflect extension for Prosa) or on the verification methods used (invariant-based properties for RT-CertiKOS *vs* trace-based properties for Prosa). This separation makes it unnecessary to have people be experts in both tools at once: once the interface was clearly defined, experts on each side could work with only a rough description of the other one, even though this interface required a few later changes. In particular, it is interesting to notice that half the authors are experts in RT-CertiKOS whereas the other half are experts in Prosa.

Third, the common part of the models used by both sides must be amenable to agreement: in our case, this means having the same notion of time (scheduling slots, or ticks) and a compatible notion of schedule (finite and infinite).

Finally, we expect the interface we designed to be reusable for other verified kernels wanting to connect to Prosa or for linking RT-CertiKOS to other formal schedulability analysis tools.

5.3 Future Work

Evolving with RT-CertiKOS. The existing implementation of the scheduler in RT-CertiKOS imposes a fixed priority scheduling policy with implicit deadlines. In the future, as RT-CertiKOS evolves and supports more task models, the interface connecting it with Prosa should also extend.

A straightforward extension is to allow *constrained deadlines*, that is, to have the deadline D_p be shorter than the period T_p (but greater than the WCET C_p) as the schedulability result we use from Prosa already supports it. This requires RT-CertiKOS to support an extended task model where a task is also specified by its deadline. Furthermore, RT-CertiKOS would also need to enforce budget at the deadlines, instead of at the beginning of the next period as it is currently the case.

Another extension would be to consider the Earliest Deadline First (EDF) scheduling policy which provides better utilization ratio. In addition to relaxing the current task model by not including priorities, the main proof effort would be to implement and verify this new scheduler in RT-CertiKOS.

Extensions to Prosa. Our experience connecting RT-CertiKOS and Prosa shows that Prosa's assumption of having an infinite schedule is quite impractical when verifying instances of real-time systems. This advocates for building reusable connections between Prosa's system model based on infinite traces and a model similar to the one used in the interface with RT-CertiKOS. Thus, one would prove analyses in the convenient setting of infinite traces and still be able to apply them to lower level models of real-time systems with finite traces.

6 Related Work

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 [20] presents a schedulability analysis technique for a simple system model described as a set of assumptions. Many later work [3,5,11,23,28] aim to capture more *realistic*[5] 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 [8,12,16], temporal logic [32,33], and theorem proving [10,30].

As far as we know, none of the above work has been applied to a formally verified OS kernel.

Verification of Real-Time OS Kernels. There is a lot of work about formal verification of OS kernels, see [18] for a survey. Therefore, we restrict our attention to verification of real-time kernels using proof assistants. We also do

[5] In terms of executions and arrival model.

not consider WCET computation, be it of the kernel itself (*e.g.,* [6,24]) or of the task set we consider. This is a complementary but clearly distinct task to get verified time bounds.

The eChronos OS [1,2] is a real-time OS running on single-core embedded systems. It stops its verification at the scheduling policy level, proving that the currently running task always has the highest priority among ready tasks. Xu et al. [31] verify the functional correctness of μC/OS-II [19], a real-time operating system with optimizations such as bitmaps. They also prove some high level properties, such as priority inversion freedom of shared memory IPC.

RT-CertiKOS [21] is a verified single-core real-time OS kernel developed by the Yale FLINT group, based on sequential CertiKOS [9,13]. It proves both temporal and spatial isolation among different components, where temporal isolation entails schedulability, etc. However, as explained in Sect. 5.1, its schedulability proof is longer whereas connecting to an existing schedulability analyzer is easier and more flexible.

7 Conclusion

Formal verification aims at providing stronger guarantees than testing. Real-time systems are a good target because they are often part of critical systems. Both the scheduling and OS communities have developed their own formally verified tools but there is a lack of integration between them. In this paper, we make a first step toward bridging this gap by integrating a formally proven schedulability analysis tool, Prosa, with a verified sequential real-time OS kernel, RT-CertiKOS. This gives two benefits: first, it provides RT-CertiKOS with a modular, extensible, state-of-the-art formal schedulability analysis proof; second, it gives a concrete instance of one of the scheduling theories described in Prosa, thus ensuring that its model is consistent and applicable to actual systems. We believe this connection can be easily adapted for other verified kernels or schedulability analyzers.

It also showcases that it is possible and practical to connect two completely independent medium- to large-scale formal proof developments.

Acknowledgments. This research has been partially supported by the following grants: PEPS INS2I JCJC 2019 Vefose, NSF grants 1521523, 1715154, and 1763399, DARPA grant FA8750-15-C-0082, as well as by the RT-PROOFS project (grant ANR-17-CE25-0016) and the CASERM project through the LabEx PERSYVAL-Lab (grant ANR-11-LABX-0025-01). The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

References

1. Andronick, J., Lewis, C., Matichuk, D., Morgan, C., Rizkallah, C.: Proof of OS scheduling behavior in the presence of interrupt-induced concurrency. In: Blanchette, J.C., Merz, S. (eds.) ITP 2016. LNCS, vol. 9807, pp. 52–68. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-43144-4_4
2. Andronick, J., Lewis, C., Morgan, C.: Controlled Owicki-Gries concurrency: reasoning about the preemptible eChronos embedded operating system. In: Proceedings Workshop on Models for Formal Analysis of Real Systems, MARS, pp. 10–24 (2015). https://doi.org/10.4204/EPTCS.196.2
3. Baruah, S.: Techniques for multiprocessor global schedulability analysis. In: Proceedings - 28th IEEE International Real-Time Systems Symposium (RTSS), pp. 119–128, December 2007. https://doi.org/10.1109/RTSS.2007.35
4. Bertogna, M., Cirinei, M.: Response-time analysis for globally scheduled symmetric multiprocessor platforms. In: 28th IEEE International Real-Time Systems Symposium (RTSS), pp. 149–160, December 2007. https://doi.org/10.1109/RTSS.2007.31
5. Bini, E., Buttazzo, G.C.: Schedulability analysis of periodic fixed priority systems. IEEE Trans. Comput. **53**(11), 1462–1473 (2004)
6. Blackham, B., Shi, Y., Chattopadhyay, S., Roychoudhury, A., Heiser, G.: Timing analysis of a protected operating system kernel. In: 2011 IEEE 32nd Real-Time Systems Symposium (RTSS), pp. 339–348, November 2011. https://doi.org/10.1109/RTSS.2011.38
7. Cerqueira, F., Stutz, F., Brandenburg, B.B.: PROSA: a case for readable mechanized schedulability analysis. In: 28th Euromicro Conference on Real-Time Systems (ECRTS), pp. 273–284 (2016). https://doi.org/10.1109/ECRTS.2016.28
8. Cordovilla, M., Boniol, F., Noulard, E., Pagetti, C.: Multiprocessor schedulability analyser. In: Proceedings of the 2011 ACM Symposium on Applied Computing, SAC 2011, pp. 735–741 (2011). http://doi.acm.org/10.1145/1982185.1982345
9. Costanzo, D., Shao, Z., Gu, R.: End-to-end verification of information-flow security for C and assembly programs. In: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pp. 648–664 (2016). http://doi.acm.org/10.1145/2908080.2908100
10. Dutertre, B.: The priority ceiling protocol: formalization and analysis using PVS. In: Proceedings of the 21st IEEE Conference on Real-Time Systems Symposium (RTSS), pp. 151–160 (1999)
11. Feld, T., Biondi, A., Davis, R.I., Buttazzo, G.C., Slomka, F.: A survey of schedulability analysis techniques for rate-dependent tasks. J. Syst. Softw. **138**, 100–107 (2018). https://doi.org/10.1016/j.jss.2017.12.033
12. Fersman, E., Mokrushin, L., Pettersson, P., Yi, W.: Schedulability analysis of fixed-priority systems using timed automata. Theor. Comput. Sci. **354**(2), 301–317 (2006)
13. Gu, R., et al.: Deep specifications and certified abstraction layers. In: Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), pp. 595–608 (2015). http://doi.acm.org/10.1145/2676726.2676975

14. Gu, R., et al.: CertiKOS: an extensible architecture for building certified concurrent OS kernels. In: 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI), pp. 653–669. USENIX Association (2016). https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gu
15. Gu, R., et al.: Certified concurrent abstraction layers. In: Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pp. 646–661 (2018). http://doi.acm.org/10.1145/3192366.3192381
16. Guan, N., Gu, Z., Deng, Q., Gao, S., Yu, G.: Exact schedulability analysis for static-priority global multiprocessor scheduling using model-checking. In: IFIP International Workshop on Software Technolgies for Embedded and Ubiquitous Systems, pp. 263–272 (2007)
17. Klein, G., et al.: seL4: formal verification of an OS kernel. In: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP), pp. 207–220 (2009). https://doi.org/10.1145/1629575.1629596
18. Klein, G., Huuck, R., Schlich, B.: Operating system verification. J. Autom. Reasoning **42**(2–4), 123–124 (2009). https://doi.org/10.1007/s10817-009-9126-9
19. Labrosse, J.J.: Microc/OS-II, 2nd edn. R&D Books, Gilroy (1998)
20. Liu, C.L., Layland, J.W.: Scheduling algorithms for multiprogramming in a hard-real-time environment. J. ACM (JACM) **20**(1), 46–61 (1973)
21. Liu, M., et al.: Compositional verification of preemptive OS kernels with temporal and spatial isolation. Technical report, YALEU/DCS/TR-1549. Department of Computer Science, Yale University (2019)
22. Nelson, L., et al.: Hyperkernel: push-button verification of an OS kernel. In: Proceedings of the 26th Symposium on Operating Systems Principles (SOSP), Shanghai, China, 28–31 October 2017, pp. 252–269 (2017). https://doi.org/10.1145/3132747.3132748
23. Palencia, J.C., Harbour, M.G.: Schedulability analysis for tasks with static and dynamic offsets. In: Proceedings 19th IEEE Real-Time Systems Symposium (RTSS), pp. 26–37. IEEE (1998)
24. Sewell, T., Kam, F., Heiser, G.: High-assurance timing analysis for a high-assurance real-time operating system. Real-Time Syst. **53**(5), 812–853 (2017). https://doi.org/10.1007/s11241-017-9286-3
25. The Coq Development Team: The Coq Proof Assistant Reference Manual. INRIA, 8.4pl4 edn. (2014). https://coq.inria.fr/distrib/8.4pl4/files/Reference-Manual.pdf
26. Tindell, K., Burns, A.: Guaranteeing message latencies on controller area network (CAN). In: Proceedings of 1st International CAN Conference, pp. 1–11 (1994)
27. Tindell, K., Burns, A., Wellings, A.: Calculating controller area network (CAN) message response times. Control Eng. Pract. **3**(8), 1163–1169 (1995)
28. Tindell, K., Clark, J.: Holistic schedulability analysis for distributed hard real-time systems. Microprocessing Microprogramming **40**(2–3), 117–134 (1994)
29. Tindell, K., Hanssmon, H., Wellings, A.J.: Analysing real-time communications: controller area network (CAN). In: Proceedings of the 15th IEEE Real-Time Systems Symposium (RTSS), San Juan, Puerto Rico, 7–9 December 1994, pp. 259–263 (1994). https://doi.org/10.1109/REAL.1994.342710
30. Wilding, M.: A machine-checked proof of the optimality of a real-time scheduling policy. In: Proceedings of the 10th International Conference on Computer Aided Verification (CAV), pp. 369–378 (1998)

31. Xu, F., Fu, M., Feng, X., Zhang, X., Zhang, H., Li, Z.: A practical verification framework for preemptive OS kernels. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9780, pp. 59–79. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41540-6_4
32. Xu, Q., Zhan, N.: Formalising scheduling theories in duration calculus. Nord. J. Comput. **14**(3), 173–201 (2008)
33. Yuhua, Z., Chaochen, Z.: A formal proof of the deadline driven scheduler. In: International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems, pp. 756–775 (1994)