RefinedProsa: Connecting Response-Time Analysis with C Verification for Interrupt-Free Schedulers

KIMAYA BEDARKAR, MPI-SWS, Germany
LAILA ELBEHEIRY, MPI-SWS, Germany
MICHAEL SAMMLER, ETH Zurich, Switzerland and ISTA, Austria
LENNARD GÄHER, MPI-SWS, Germany
BJÖRN BRANDENBURG, MPI-SWS, Germany
DEREK DREYER, MPI-SWS, Germany
DEEPAK GARG, MPI-SWS, Germany

There has been a recent upsurge of interest in formal, machine-checked verification of timing guarantees for C implementations of real-time system schedulers. However, prior work has only considered tick-based schedulers, which enjoy a clearly defined notion of time: the time "quantum". In this work, we present a new approach to real-time systems verification for *interrupt-free schedulers*, which are commonly used in deeply embedded and resource-constrained systems but which do not enjoy a natural notion of periodic time. Our approach builds on and connects two recently developed Rocq-based systems—RefinedC (for foundational C verification) and Prosa (for verified response-time analysis)—adapting the former to reason about timed traces and the latter to reason about overheads. We apply the resulting system, which we call *RefinedProsa*, to verify Rössl, a simple yet representative, fixed-priority, non-preemptive, interrupt-free scheduler implemented in C.

CCS Concepts: • **Computer systems organization → Real-time systems**; • **Theory of computation → Logic and verification**; **Separation logic**.

Additional Key Words and Phrases: Real-time systems, response-time analysis, verification, Iris, Rocq

ACM Reference Format:
Kimaya Bedarkar, Laila Elbeheiry, Michael Sammler, Lennard Gäher, Björn Brandenburg, Derek Dreyer, and Deepak Garg. 2025. RefinedProsa: Connecting Response-Time Analysis with C Verification for Interrupt-Free Schedulers. *Proc. ACM Program. Lang.* 9, PLDI, Article 150 (June 2025), 25 pages. https://doi.org/10.1145/3729249

1 Introduction

Cyber-physical systems (CPSs) are systems in which software interfaces with physical components that monitor and control their environment. Common, ubiquitous examples include cars, planes, trains, medical monitoring devices, and autonomous vehicles. For obvious reasons of safety, many CPSs are tasked with providing a "reliable response" to external stimuli, meaning that they must

Authors' Contact Information: Kimaya Bedarkar, MPI-SWS, Saarland Informatics Campus, Germany, kbedarka@mpi-sws.org; Laila Elbeheiry, MPI-SWS, Saarland Informatics Campus, Germany, lelbehei@mpi-sws.org; Michael Sammler, ETH Zurich, Zurich, Switzerland and ISTA, Klosterneuburg, Austria, michael.sammler@ist.ac.at; Lennard Gäher, MPI-SWS, Saarland Informatics Campus, Germany, gaeher@mpi-sws.org; Björn Brandenburg, MPI-SWS, Saarland Informatics Campus, Germany, bbb@mpi-sws.org; Derek Dreyer, MPI-SWS, Saarland Informatics Campus, Germany, dreyer@mpi-sws.org; Deepak Garg, MPI-SWS, Saarland Informatics Campus, Germany, dg@mpi-sws.org.

ACM 2475-1421/2025/6-ART150
https://doi.org/10.1145/3729249

respond with the right result within a fixed time limit (*e.g.,* a car swerving in time to avoid hitting a pedestrian). Such systems are referred to as *real-time systems (RTSs)*.

Seeing as they are frequently deployed in safety-critical environments, RTSs are a natural candidate for certification via rigorous formal verification. Indeed, owing to reports of errors in pen-and-paper RTS theory [17, 19, 26, 29], there has been a growing interest in formally verifying the theory with machine-checked proofs in an interactive proof assistant [6, 9, 11, 15, 16, 22–24, 26, 35, 36, 40]. In particular, the Prosa project of Brandenburg and collaborators [16, 40] focuses on the verification of schedulability analyses, specifically *response-time analyses* (RTAs). RTAs consider systems that process a stream of incoming *jobs* (each running a specified *task*) and schedule them for execution—the analyses take as input some assumptions on the "workload" (*i.e.,* the rate at which new jobs arrive and how long each takes to execute once it is dispatched), and yield a guaranteed bound on the "response time" (*i.e.,* the maximum time any given task can take to complete after it has arrived in the system). In developing Prosa, Cerqueira et al. [16] were motivated by the discovery of flaws in published pencil-and-paper proofs of such analyses; they set out to instill confidence in the formal foundations of the field by building a general framework for *mechanizing* these analyses in Rocq.

However, there remains a gap between the theory underlying RTSs and their actual implementations in low-level systems programming languages like C. In particular, as is common in the real-time scheduling literature, the RTAs verified in Prosa consider an abstract system model that does not expose implementation-specific details of an actual real-time scheduler, and as such, they are susceptible to the possibility that the implementation does not faithfully implement the model. This concern is not merely academic. For example, Cofer and Rangarajan [18] analyzed the timing behavior of the Deos real-time operating system in the presence of various advanced scheduling features. Among the errors they found were some caused by an incorrect C++ implementation of the system model, *e.g.,* wrong computation of the period of the idling task. More recently, the work by Teper et al. [41] provides counterexamples for two previously proposed (and on-paper verified) RTAs for the multithreaded executor of ROS2. In both cases, they show that there exists a task that is starved (and therefore has unbounded response times) even when the proposed RTAs claim that a response-time bound exists. And in both the now-refuted RTAs, the problem was not caused by an incorrect analysis of the system model under consideration, but rather by the fact that the system model did not accurately account for how the wait set (set of tasks waiting to be executed) was constructed by the system during execution.

Thus, to truly establish higher confidence in RTSs, it is important to connect formally verified RTAs with real implementations. To this end, Guo et al. [27, 28] took an important step forward in their work on ProKOS, wherein they showed how the Rocq-verified schedulability analysis of Prosa could be integrated into RT-CertiKOS [33, 34], a real-time extension of the Rocq-verified CertiKOS kernel [25], in order to establish a machine-checked response-time bound on its scheduler.

RT-CertiKOS's scheduler is "tick-based", meaning that it is invoked periodically by timer interrupts, which divide time up into units called "quanta"; at each interrupt, the scheduler decides what job should be scheduled during the next quantum. In this paper, we tackle a similar problem to the one that motivated ProKOS but for a different type of scheduler: an *interrupt-free scheduler*.

1.1 Our Problem: Response-Time Analysis of Interrupt-Free Schedulers

Unlike tick-based schedulers, interrupt-free schedulers do not get invoked after some periodic quantum. Rather, an interrupt-free scheduler runs in a loop: at each iteration, it polls for newly arrived jobs, decides which job to dispatch next, transfers control to that job, and then waits until the executing job yields (or terminates) and returns control to the scheduler. Interrupt-free schedulers do not have a means of regaining control from a runaway job (*i.e.,* one that takes

longer to execute than promised), but in return, each job executes *non-preemptively* (*i.e.,* without any disruptions), which increases both efficiency and temporal predictability. Most importantly, interrupt-free schedulers can be implemented with nothing more than regular function calls and can therefore be realized in any context and on any hardware platform.

Due to their minimal requirements, interrupt-free schedulers are common in deeply embedded systems deployed on extremely resource-constrained hardware platforms (*e.g.,* 8-bit microcontrollers) and severely energy-constrained systems (*e.g.,* wireless sensor nodes expected to last for months or even years on a single battery charge). For instance, the well-known TinyOS [32] and Contiki [21] operating systems for Internet-of-Things (IoT) devices both use interrupt-free schedulers by default. At the other end of the spectrum, interrupt-free schedulers are also commonly encountered in complex middleware systems in the form of in-process schedulers. For example, the default "executor" of the widely used robotics middleware ROS2 (which is typically deployed as a Linux process) uses interrupt-free scheduling to sequence the execution of callback functions [14] (which are the primary ROS primitive allowing robots to react to environmental stimuli).

Unfortunately, it is not clear how the approach of ProKOS (or other, more recent, approaches [42], see §6) can be adapted to support the verification of interrupt-free schedulers. The essential problem is that, whereas tick-based schedulers are structured around a clearly defined notion of time (the time quantum), interrupt-free schedulers are not. That is, in ProKOS, response-time guarantees are expressed simply in terms of the number of time quanta that occur between when a job arrives in the system and when it finishes executing—the verification *does not explicitly consider the time taken by the execution of the C scheduler code itself* but rather *assumes* it to be absorbed into a fraction of each quantum. In contrast, in an interrupt-free scheduler, there is no clearly defined notion of quantum and, hence, no clear way to absorb scheduling overheads. Furthermore, since an interrupt-free scheduler is only invoked *in between* the uninterrupted executions of different jobs, a pile-up of newly arrived jobs can lead to bursts of scheduling overhead, which must be carefully accounted for in order to achieve a reliable real-time guarantee.

1.2 Our Solution: RefinedProsa

In this paper, we propose **RefinedProsa**, a new methodology for formally verifying real-time guarantees for the C implementation of an interrupt-free scheduler in Rocq. Our approach leverages and connects two existing verification tools: **Prosa** and **RefinedC**. Concerning the core real-time theory, we take advantage of the fact that Prosa has already been used to formalize a wide range of schedulability analyses. For connecting these analyses to real C code, we build on the recently developed RefinedC tool [38]: it supports foundational, machine-checked proofs of correctness for C programs in Rocq but with a significant degree of automation. With RefinedProsa, we show how Prosa and RefinedC can fruitfully join forces, and we demonstrate the effectiveness of their union by using them to verify response-time bounds for **Rössl**, a simple yet representative, fixed-priority, non-preemptive, interrupt-free scheduler implemented in C.

At a high level, RefinedProsa adopts the same basic proof structure employed by ProKOS: we first establish key invariants on the trace of Rössl's actions (in our case using RefinedC), and then show how those invariants imply the prerequisites for an RTA formalized in Prosa. The key overarching challenge in developing RefinedProsa is that interrupt-free schedulers like Rössl do not have a built-in notion of time, and yet we must find a way to reason about response-time bounds and overheads regardless. To this end, RefinedProsa introduces several technical innovations.

Step 1: Dividing the trace into basic actions using marker functions. First, we observe that the execution of Rössl divides naturally into a series of logically distinct loop-free segments and OS-provided system calls, which we call *basic actions*. These basic actions include: reading from a

channel to observe a newly arrived job (both successful and failed reads), selecting a new job to execute, completing a job, idling, and more. Since C lacks a clear cost semantics, we have no means to reason precisely about the time taken by each basic action. However, given their simplicity, it is reasonable to assume the existence of *worst-case execution times* (WCETs) for these basic actions (to be determined experimentally or by static analysis), and make those WCET bounds a parameter of the verification.

We indicate these basic actions in the C code of Rössl through "ghost calls" to *marker functions*, which indicate the demarcation points between basic actions.[1] Then, we use RefinedC to establish a *functional correctness* property on these basic actions: namely, that they are sequenced according to a clearly defined *scheduler protocol*. This protocol is formalized as a state-transition-system invariant, which is proven to hold for all traces of marker functions that Rössl may produce.

Step 2: Incorporating time into the trace. The functional correctness property established by RefinedC says nothing about timing. To incorporate time into the verification, we *assume* we are given an *arrival sequence* (stipulating the time at which each job arrives, expressed in arbitrarily fine-grained units of time) representing the stimuli generated by the system's nondeterministic environment and a *list of timestamps* (mapping of marker function calls to the times at which they occur), and we assume that they are consistent with each other and with the trace exhibited by Rössl. Furthermore, we assume that this timing information is consistent with the aforementioned WCET bounds on basic actions. Based on these assumptions, together with the trace invariant established by the RefinedC verification, we then convert our timed trace of marker functions into a *schedule of processor states*, which follows the more abstract system model employed by previous work on RTAs including Prosa. Moreover, we establish that this schedule of processor states satisfies a set of *validity constraints* that are sufficient to perform the RTA in the final step.

Step 3: Formally verifying an RTA with overheads. Last but not least, we show how to extend prior work on verified RTAs to handle Rössl. The main challenge here is that none of the previously mechanized RTAs have taken explicit account of *overheads*, *i.e.,* the time spent by the scheduler itself when it is not executing jobs. Towards this end, we build on the recently developed aRSA technique [10], an abstract framework for verifying RTAs for non-ideal processors subject to "supply restrictions". Although the general notion of supply restrictions formalized by aRSA was originally not intended to model overheads, we show how to instantiate aRSA with an abstract system model of Rössl such that the validity constraints established in Step 2 imply the hypotheses of aRSA. Specifically, this involves defining a *supply bound function* $SBF(\Delta)$ (which characterizes the amount of potential overhead during an interval of length Δ) as well as modeling some implementation-specific delays using *release jitter* (which bounds the gap between a job's arrival and when the scheduler observes it). Finally, we obtain a timing correctness theorem (Thm. 5.1) which connects the response-time bound to the timed trace of marker functions from Step 2.

Structure of the paper. We begin in §2 with an overview of Rössl, as well as a high-level presentation of the three steps outlined above. The subsequent sections, §3 and §4, give a more detailed formal presentation of Steps 1 and 3, respectively. Lastly, §5 puts the pieces together to present our final timing correctness result, and §6 concludes with a comparison to related work.

2 An Overview of RefinedProsa

In this section, we give an overview of RefinedProsa, our approach to RTS verification, using Rössl as a major case study. Rössl is a prototype scheduler we designed to resemble, at a high level, the

[1] The use of "ghost calls" is not novel (*e.g.,* see Bengtson et al. [7]). However, we believe ours is the first work to use this technique, together with that of Step 2, to modularly integrate temporal reasoning into the verification of a real-time system.

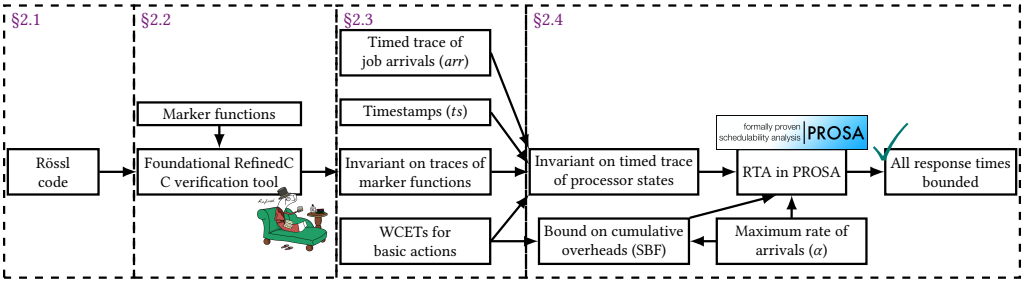


Fig. 1. A high-level proof overview.

ROS2 default executor. Rössl accepts new jobs arriving as messages over sockets and schedules them by dispatching a corresponding callback function. Our goal is to state and verify a bound on the time interval between a job arriving and the callback function for it finishing execution.

Fig. 1 gives a high-level overview of the various steps involved in this verification. We will discuss these steps in the rest of this section. First, §2.1 gives an overview of the implementation of Rössl. Then, §2.2 explains how we add *marker functions* to delimit different parts of the execution of Rössl called *basic actions*. We also integrate support for these marker functions into the foundational C verification tool RefinedC [38] and use RefinedC to prove a specific set of invariants on the traces of marker functions generated from executing Rössl. Next, in §2.3, we show we can enrich the trace of marker functions with timing information. Finally, §2.4 shows how we then use RTS techniques to analyze this timed trace to obtain guarantees about the overall timing behavior of Rössl.

2.1 The Rössl Scheduler

Rössl is designed to resemble callback-driven schedulers at the heart of systems like ROS2. At a high level, Rössl accepts new jobs arriving as messages over sockets and schedules them by dispatching a corresponding callback function. Rössl can be configured to support different types of jobs and corresponding callback functions. It then schedules jobs according to a *non-preemptive, fixed-priority* policy. A non-preemptive policy schedules all callbacks until completion, without preemption. A fixed-priority policy requires that (1) jobs have a statically assigned priority and (2) out of all the pending jobs at any time, Rössl always selects the highest-priority job to execute first.

Scheduling loop. The main scheduling loop of Rössl is shown in Fig. 2. For the moment, ignore the annotations in lightblue. Each iteration of the loop goes through three phases: the *polling phase*, which checks for newly arrived jobs, the *selection phase*, which selects the next job to execute, and the *execution phase*, which executes the selected job. Fig. 3 shows an example run of Rössl with two jobs on one socket. Let us go over the three different phases:

In the ***polling phase***, Rössl checks for new jobs by calling the `check_sockets_until_empty` function on line 3, which calls the `read` system call for all sockets in a loop. This loop terminates when there is one iteration where the reads on all sockets fail. Each received message corresponds to a new job that is added to the internal state of Rössl. This is shown in the example run in Fig. 3 where Rössl first reads the job j_1 that arrived earlier, then the job j_2 that arrived while reading j_1, and then stops since no more jobs are available (*i.e.,* the read failed).

Next, in the ***selection phase***, Rössl selects the next job to execute by calling `npfp_dequeue` on line 6. Following the fixed-priority policy, the function selects the pending job with the highest priority. In our example, we say that j_2 has a higher priority than j_1, so `npfp_dequeue` selects j_2.

In the ***execution phase***, Rössl executes the selected job using `npfp_dispatch` on line 11. In our running example, this executes the callback corresponding to j_2. After the callback finishes, there is

```
1  int fds_run(struct fd_scheduler *fds) {
2    while(1) {
3      check_sockets_until_empty(fds); // receive jobs on all sockets
4      selection_start();
5
6      struct job *j = npfp_dequeue(&fds->sched); // get highest-priority job
7      if (!j) {
8        idling_start(); // if there is no job, wait for new input
9      } else {
10       dispatch_start(j);
11       npfp_dispatch(&fds->sched, j); // execute the job
12       free(j); // release the memory
13     }}}
```

Fig. 2. The scheduling loop of Rössl. Annotations in lightblue.

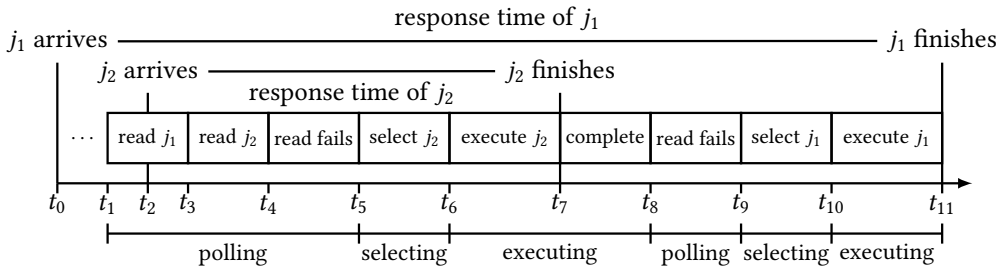


Fig. 3. An example run of Rössl with two jobs arriving on one socket.

some overhead for completing the loop iteration and starting again with the polling phase. In our running example, after finishing j_2, Rössl goes through all the phases again, selecting and executing j_1. If there is no pending job to select, Rössl enters the ***idling phase***.

Response times. The key property that we prove about Rössl's scheduling algorithm is that it guarantees certain bounds (which we define by analyzing Rössl's behavior) on the *response time* of each job. The response time of a job is the time between the job arriving in the system and its callback finishing execution. This span of time for j_1 and j_2 is shown in Fig. 3. In the next sections, we will see how we can prove a statically determined bound on the response times of jobs.

2.2 Reasoning About Timing by Partitioning the Code Into Basic Actions

To prove the desired response time bounds for Rössl, we first need a verification tool that can reason about C code. On the one hand, this tool should provide a high degree of automation to make the proof effort more manageable and maintainable. On the other hand, it should generate foundational, machine-checked proofs such that the verification can be formally—and with high confidence—connected to the formalized schedulability analysis of Prosa. For this reason, we build on RefinedC [38]: RefinedC uses a combination of refinement and ownership types to automate the foundational verification of C code to a high degree and thus provides the ideal basis for our work.

By building on a C verification tool like RefinedC, we can verify functional properties—for instance, that the `npfp_dequeue` function of Rössl always picks the job with the highest priority. However, C verification tools—including RefinedC—typically do *not* reason about the timing behavior of programs (*i.e.,* how many seconds the execution of a chunk of C code takes). This is

$$\text{basic\_actions} \triangleq \text{Read } \mathit{sock}\ j_\perp \mid \text{Selection } j_\perp \mid \text{Disp } j \mid \text{Exec } j \mid \text{Compl } j \mid \text{Idling}$$
$$\text{marker} \triangleq \text{M\_ReadS} \mid \text{M\_ReadE } \mathit{sock}\ j_\perp \mid \mid \text{M\_Selection} \mid \text{M\_Dispatch } j$$
$$\mid \text{M\_Execution } j \mid \text{M\_Completion } j \mid \text{M\_Idling}$$

Fig. 4. Basic actions and marker functions of Rössl. $j_\perp$ is either a job j or $\perp$.

because reasoning about the timing behavior of C code is extremely challenging since compiler optimizations might completely change the structure of the code and, furthermore, the execution time of the compiled code depends highly on architecture-specific details like cache sizes. Prior work on verified schedulers [27, 28, 42] sidesteps this problem by using a preemptive scheduler that schedules work in fixed slices of time (*i.e.,* quanta) and assuming that the execution of the scheduler code takes a negligible amount of time for each quantum. However, this approach does not work for Rössl since it is non-preemptive and event-driven and thus the timing behavior of the C code influences when jobs become available to the scheduler. In the rest of this section, we will see how we adapt RefinedC such that we can use it to reason about the timing behavior of Rössl.

Basic actions. Our key idea to reason about timing behavior of Rössl's C code is to *not* reason about the timing behavior of the C code directly using RefinedC (since reasoning about the timing of C code at the source level is difficult). Instead, we partition the execution of the scheduler into separate chunks that we call *basic actions*. Each basic action corresponds to the execution of a specific part of the C code. Then, we extend RefinedC such that it can prove which sequences of basic actions can be generated by Rössl. The key benefit of this approach is that the RefinedC-based verification is completely agnostic to the concrete timing behavior of the C code and the basic actions. Instead, we reason about the timing behavior in a separate step where we link the basic actions to dynamically measured or statically derived WCETs. This step can work at a much higher level of abstraction since it only needs to reason about the basic actions and not an intricate C semantics. But before we get there (in §2.3), let us first look at basic actions in more detail.

Fig. 4 shows the set of basic actions for Rössl, each corresponding to one logical operation done by the scheduler. In fact, we have already seen these basic actions implicitly in the example execution of Rössl in Fig. 3. The Read *sock* $j_\perp$ basic action corresponds to either reading a job j, or a failed read where no job is available ($j_\perp = \perp$). Selection $j_\perp$ similarly corresponds to either selecting j as the highest-priority pending job for execution, or failing to select a job because there are no pending jobs ($j_\perp = \perp$). Disp, Exec, and Compl are the actions for initiating, executing, and completing the callback for job j. Finally, Idling is the action taken when there are no pending jobs and the scheduler is idle.

Delimiting basic actions with marker functions. In order to partition the code of the scheduler into basic actions, we insert *marker functions* into the code, which mark the beginning of a new basic action. Marker functions do not affect the actual runtime behavior of Rössl (*i.e.,* they are a form of "ghost code" for verification purposes only). We place marker functions carefully throughout Rössl. Some of them are visible in the main scheduling loop shown in Fig. 2 highlighted in lightblue. For example, the function `idling_start` on line 8 marks the beginning of the Idling basic action, while the `dispatch_start` function on line 8 marks the beginning of the Disp j basic action.

The marker functions are shown in Fig. 4. Note that there is no 1-to-1 correspondence between basic actions and marker functions since marker functions identify the start of a new basic action, but in some cases it only becomes clear later which basic action it is. For example, the function `selection_start` on line 4 marks the beginning of either the Selection $\perp$ basic action or the Selection j

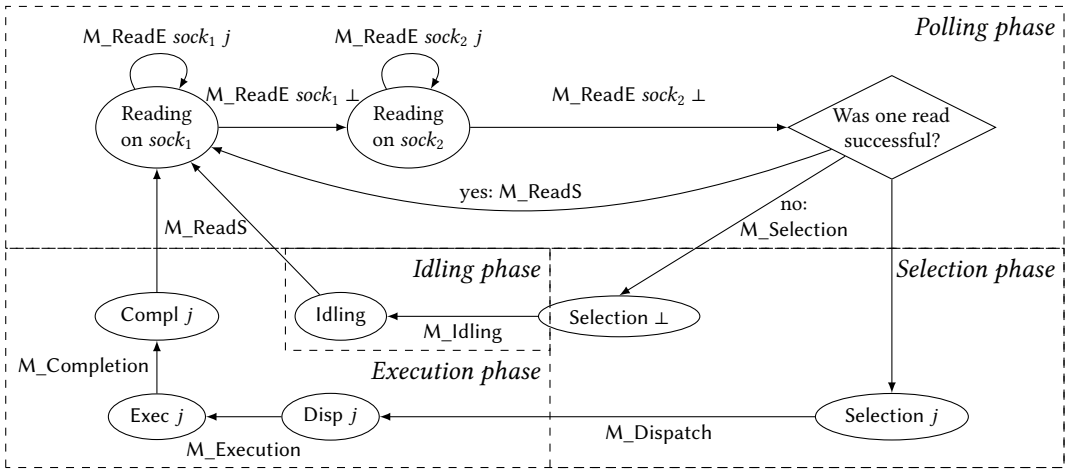


Fig. 5. The state-transition system used to reason about the trace of marker functions for two sockets.

basic action, depending on whether the selection fails and the next marker function is `idling_start` or the selection succeeds with job j and the next marker function is `dispatch_start(j)`.

Formally, the relation between marker functions and basic actions is described by the state-transition system (STS) in Fig. 5.[2] The nodes of the STS denote the basic actions while the edges show the different marker functions. There is one exception: Reading is split into the marker function M_ReadS and the "pseudo marker function" M_ReadE that corresponds to the result of the read system call. In the STS, these two are implicitly coalesced into a single Read basic action corresponding to the result of M_ReadE. To check whether a trace of marker functions is valid, one can see if there is a run that starts in Idling and accepts the trace. The sequence of states in the run gives the sequence of basic actions for the trace of marker functions.

Proving functional correctness of marker function traces. We built an extended version of RefinedC that can abstract Rössl into a set of traces describing its flow of execution. Concretely, we first adapt Caesium, the formal C semantics used by RefinedC, to emit the trace of the marker functions that are invoked during the execution. Second, we extend RefinedC to prove functional correctness properties about these traces (see §3 for a detailed explanation of these extensions). These extensions allow us to reason about the behavior of Rössl over time without requiring direct reasoning about timing properties using RefinedC. We leverage these capabilities to prove two classes of properties about Rössl: First, we prove that the trace of marker functions is accepted by the STS in Fig. 5, and thus can be seen as a sequence of basic actions. We call this property the *scheduler protocol*, as it describes the behavior of the scheduler. Second, we prove that the trace is *functionally correct*, *i.e.,* satisfies the following two properties: (1) selected jobs have the highest priority, and (2) Rössl only enters Idling if there are no pending jobs. As we will see in the following sections, these invariants enable us to reason about the timing correctness of Rössl.

[2]The presentation is simplified to use two sockets instead of being parametric in the number of sockets.

2.3 Incorporating Timing Information

The previous section showed how we abstract the behavior of Rössl to traces of marker functions. However, these traces do not contain any information about timing. This section shows how we enrich these traces with timing information to reason about the response-time guarantees of Rössl.

Introducing time. To see how we can introduce time into the verification, let us first consider what timing information we need for stating the property that we want to prove for Rössl. Intuitively, this property has the following form, *i.e.,* we want to prove that the completion time of each job j is bounded by its arrival time plus its response time bound:

$$\forall j.\ \textit{completion time of } j \leq \textit{arrival time of } j + \textit{response time bound for } j$$

To make this property more formal, let us first consider the completion time. (§2.4 describes how we compute the response-time bound.) Intuitively, the completion time of a job j corresponds to the end of the Exec basic action for the job. To be able to formally describe this time, we associate each marker function on the trace with a distinct timestamp, representing the point in time at which the marker function was called.[3] The resulting sequence of timestamps ts allows us to talk formally about the timing of the execution of Rössl. We call a trace tr of marker functions augmented with timestamps a *timed trace* (tr, ts). Crucially, by introducing these timestamps separately from the RefinedC verification, RefinedC does not need to be aware of the timestamps.

In addition to the times of the marker functions, we also need to reason about the arrival of jobs in the system. Since our system is event-driven, a job's arrival corresponds to the time when the message representing it is enqueued on a socket and becomes available to be read. Following the standard RTS approach, we model these arrivals as an arbitrary *arrival sequence arr* that determines an arrival time for every job. Concretely, in the context of Rössl, *arr* is a mapping from a socket and time to the set of jobs arriving on that socket at that time ($arr : sock \to \mathbb{T} \to list\ \text{Job}$).

Assumptions about time. The arrival sequence and the timed trace allow us to state Rössl's response-time guarantee. However, to be able to *prove* this guarantee, we need more information.

First, we need to bound the execution time of basic actions. We have designed the basic actions such that each corresponds to a part of the C code with a bounded runtime (*i.e.,* it does not contain infinite loops). Thus, for each basic action, we can assume a WCET that gives a bound on its runtime. There are various ways in which such WCET bounds could be computed. For instance, we could use one of a number of academic or industrial tools (*e.g.,* [1, 2, 4]) to determine these WCETs for the compiled code running on the target hardware. Additionally, research on WCET tools is constantly evolving, and further methods for computing WCETs have been proposed by Bonenfant et al. [8] and Zolda and Kirner [43]. To focus our efforts in this work, we therefore simply assume the WCET bounds on basic actions as a parameter of our verification.

For the Exec basic action, we assume different WCETs for each callback that the client provides, as different callbacks will typically have different execution times. Our timing correctness property holds for all executions where the actual run times of the basic actions and callbacks stay below their WCETs. For example, to express our assumption on the WCET of the Disp basic action, we assume the following property about the timed trace (tr, ts):

$$\forall i, j.\ tr[i] = \text{M\_Dispatch}\ j \implies ts[i+1] - ts[i] \leq \mathit{WcetDisp}$$

The full definition expressing all WCET assumptions can be found in [5].

Second, we need to know that the arrival sequence is consistent with the timed trace. The consistency of the timed trace with the arrival sequence is formulated as follows:

[3] The unit of timestamps is arbitrary and can be instantiated with any arbitrarily fine-grained units such as processor cycles.

Definition 2.1 (Consistency). A timed trace (tr, ts) is *consistent* with an arrival sequence *arr* if:

- Each job is read only after it has arrived:

$$\forall i, sock, j.\ tr[i] = \mathsf{M\_ReadE}\ sock\ j \implies \exists t_a.\ j \in arr_{sock}\ t_a \land t_a < ts[i]$$

- If a read fails because no data is available, there are no unread arrived jobs:

$$\forall i, sock, j, t_{arr} < ts[i].\ tr[i] = \mathsf{M\_ReadE}\ sock\ \bot \land j \in arr_{sock}\ t_{arr} \implies j \in \mathsf{read\_jobs}\ i$$

where $\mathsf{read\_jobs}(i) \triangleq \{j \mid \exists k\ sock, k < i \land tr[k] = \mathsf{M\_ReadE}\ sock\ j\}$

2.4 Response-Time Analysis of Rössl

While the timed trace from the previous section abstracts over the C implementation of Rössl, it is still too detailed relative to the abstract model used in standard RTAs, including those in Prosa. This section shows how we turn the timed trace into a *schedule* to which we can apply Prosa's RTA.

Schedule. Following standard convention of the real-time systems community, our RTA is based on an abstract model of system behavior called a *schedule*. A schedule maps each instant in time to a *processor state* describing what the system (scheduler being verified) is doing a that time. We define the set of possible processor states as:

$$\begin{aligned} \mathit{ProcessorState} \triangleq\ & \mathit{Idle} \mid \mathit{Executes}\ j \mid \mathit{ReadOvh}\ j \mid \mathit{PollingOvh}\ j \\ & \mathit{SelectionOvh}\ j \mid \mathit{DispatchOvh}\ j \mid \mathit{CompletionOvh}\ j \end{aligned}$$

They split into three categories: First, the system can be idle and waiting for new jobs to arrive (*Idle*). Second, the system can execute a job j (*Executes* j). Finally, the system can perform work that is not directly executing a job. Such work is called an *overhead*. Rössl's overheads include: selecting the next job j (*SelectionOvh* j), dispatching job j (*DispatchOvh* j), cleaning up after the execution of job j (*CompletionOvh* j), and polling for new jobs (*ReadOvh* j and *PollingOvh* j). (We will come back to the difference between *ReadOvh* j and *PollingOvh* j in a moment.)

Converting a timed trace to a schedule. Compared to the basic actions in Fig. 4, the processor states abstract over failed and successful reads as well as the concept of sockets. To convert a timed trace to a schedule, we need to bridge this gap. The main challenge is accounting for the time spent on failed reads. (Other basic actions map basically 1-to-1 to processor states.) In particular, for the RTA, it is important that each overhead is attributed to a job so that we can then bound the overall time spent in overhead states by bounding the number of jobs. However, it is *a priori* unclear what job we should attribute a failed read to. We address this problem by distinguishing three cases:

If a sequence of reads across all the sockets results in multiple failed reads followed by a successful read, we attribute the time of the failed read to the job that was successfully read, *i.e.,* both the time of the failed and the successful reads is mapped to *ReadOvh* j (where j is the job that was successfully read). This adds only a bounded amount of time to the overhead induced by job j since there can be at most as many failed reads as there are sockets before a successful read.

If all reads fail in one iteration of checking all sockets, the polling phase of Rössl concludes. If Rössl executes job j next, we assign the processor state *PollingOvh* j and attribute the overhead of the failed reads to job j. Again, we can bound this overhead using the number of sockets.

If there is no job to execute after the polling phase, the failed reads (and the following failed selection) are mapped to the *Idle* processor state.

Formally stating this conversion is tricky since we have to "look into the future" to see which processor state a failing read should be attributed to. (Technically, we solve this problem by defining the conversion function as a finite look-ahead parser on the timed trace of marker functions.)

Validity constraints. Just constructing the schedule from the timed trace is not sufficient to perform an RTA. Additionally, we need to design and prove a set of *validity constraints* that constrain the schedules Rössl produces. These validity constraints (a) enforce bounds on each discrete instance of a processor state (except *Idle*), (b) ensure that the schedule is consistent with the arrival sequence (jobs read in the schedule must come from the arrival sequence), (c) encode that Rössl is functionally correct (the selected job has the highest priority among all read jobs), (d) encode a version of the scheduler protocol for schedules, and (e) that all jobs have unique identifiers.

Let's take the validity constraint that bounds the WCET of polling as an example. Recall that the *PollingOvh* refers to the combination of failed reads across all sockets. Therefore, we define the WCET for it as: $PB \triangleq |input\_socks| \times WcetFR$ where *WcetFR* is the WCET for a failed read. Then, we prove that each distinct *PollingOvh* state in the schedule runs for at most its WCET. We state the property as an invariant on the Prosa-compatible schedule representation *sched* as:

Definition 2.2 (PollingOvh WCET respected).

$$\forall t_1, t_2, j.\, |\{t | t_1 \leq t < t_2 \wedge sched\ t = PollingOvh\ j\}| \leq PB \tag{1}$$

We prove this property using the WCET assumptions on timestamps.

RTA for Rössl. With the schedule and validity constraints at hand, we can finally perform the RTA for Rössl. In RTS terms, we can characterize Rössl as a fixed-priority (the job to be dispatched is chosen in order of non-increasing priority) and non-preemptive (callback functions do not preempt) scheduler (NPFP for short). NPFP schedulers are common in practice and several RTAs have been proposed to analyze the schedulability of this policy [11, 13, 20]. While RTAs for NPFP systems are generally well understood, it is the presence of overheads in Rössl that complicates our RTA: we do not just have to account for the delays due to other jobs in the system, but also for the delays from all the overhead states. This difficulty in dealing with overheads is further exemplified by the fact that none of the previously mechanized RTAs have reasoned about systems with overheads.

The key component of our analysis is the recently introduced aRSA technique [10, 40]. aRSA is an abstract framework for verifying RTAs for non-ideal processors subject to "supply restrictions", *i.e.,* timespans where the system cannot supply processing time that would allow jobs to progress in their execution. We model overheads as states without supply. aRSA enables a separation of concerns, allowing us to reason about the system while essentially ignoring the effects of overheads, using standard RTA techniques. However, there are two key challenges for applying aRSA to Rössl.

First, we need to be careful when proving that Rössl implements the NPFP scheduling policy correctly (*i.e.,* it always picks the job with the highest priority that has arrived but not been scheduled yet). For example, if a job arrives after Rössl finishes its polling loop but before it makes a scheduling decision, it might not actually schedule the highest-priority job that has arrived, and it might even decide to idle when new jobs just arrived. In particular, on the Rössl side, the NPFP policy is always implemented on the set of *read jobs* but, on the Prosa side, functional correctness is stated in relation to the set of *arrived jobs*. Therefore, the fact that the set of read jobs always lags behind the set of arrived jobs causes issues in applying Prosa theorems as-is to Rössl. To address this challenge, we use the modeling flexibility offered by the classic concept of *release jitter* [3, 12]. In particular, if a job is overlooked during a scheduling decision since it was not yet read, we say that the job is experiencing release jitter and thus is not ready to be scheduled yet. Then, after proving a bound on the maximum release jitter incurred by any job and its worst-case impact on scheduling delays, we can prove that any job's response time is offset by no more than the maximum release jitter incurred [3].

Since our final response-time bound is offset by the jitter bound, one may wonder whether the jitter bound can be inflated to such a degree that the final response-time bound becomes large

enough as to render our theorem vacuously true. This is not the case, as the release jitter bound is not freely chosen, but rather fixed in the proof and computed as a linear sum of the WCETs of basic actions (*i.e.,* it is a sum of low-level scheduling overheads). In a typical deployment of Rössl, the jitter bound amounts to just a few microseconds and thus does not undermine the final response-time bounds, which are typically on the order of tens to hundreds of milliseconds.

Second, aRSA requires a *supply bound function SBF*(Δ) that provides a lower bound on the overhead-free time (called supply) provided by Rössl. Note that we have to both come up with this function and prove that it is indeed a lower bound on the supply. To address this challenge, we define the *SBF* by bounding the maximum time the system can spend in overhead states in an interval of length Δ. For this, we attribute each overhead to a job as described before and then leverage a bound on the rate of arrivals to bound the maximum overhead time. Finally, the validity constraints imposed on the schedule allow us to prove this bound to be sound (see §4.4 for details).

With this, we can apply the aRSA RTA to Rössl to obtain a provably correct bound on its response times. However, this bound is phrased in terms of the schedule. Thus, we need to translate the bounds back to the terms of the Rössl C code, *i.e.,* to the timed trace of marker functions. Concretely, our final theorem (§5) proves that, if the timestamps, the bound on the rate of arrivals, and other inputs are valid, then, in any trace of marker functions generated by Rössl, each job's response time is bounded by the bound obtained from aRSA plus the offset to account for release jitter.

2.5 Summary of the Assumptions and Guarantees of a RefinedProsa Analysis

Our framework, RefinedProsa, is designed for proving response-time guarantees for jobs that arrive during any run of a concrete (C) implementation of an interrupt-free scheduler. The parameters of this analysis are: the WCETs of the basic actions, the WCETs of the callbacks, an arrival curve (a bound on the job arrival rate for each callback), and an arrival sequence (a description of the jobs in the run, with their arrival times, which is typically left $\forall$-quantified).

RefinedProsa *assumes* that these parameters satisfy the following assumptions: First, every basic action and every external callback is assumed to run within its WCET. Second, the actual rate of arrivals of jobs for each callback (as per the given arrival sequence) is assumed to be no more than that stipulated by the arrival curve. Third, the operating system is assumed to implement system calls like `read` correctly (according to their assumed RefinedProsa specification).

Under these assumptions, RefinedProsa's theorem *guarantees* a response-time bound for every job in the given arrival sequence. This response-time bound takes all possible delays into account, including overheads, ensures the absence of bugs in the scheduler's code and the schedulability analysis, and prevents mismatches in assumptions between the RTA and the implementation.

3 Verifying Program Properties of Rössl in RefinedC

The first step of our approach is verifying Rössl's implementation using RefinedC. This allows us to abstract the scheduler implementation into a set of traces that our schedulability analysis can reason about. This section explains how we specify and verify parts of the scheduler using extensions to RefinedC's separation logic (§3.1), how we instrument RefinedC's C semantics to emit a trace of marker functions and extend RefinedC's logic (§3.2), and how we adapt RefinedC's adequacy theorem to account for traces (§3.3).

3.1 Reasoning about Traces in RefinedC

As discussed in §2.2, we use RefinedC to prove two classes of properties about the trace of marker functions emitted by a run of Rössl: The scheduler protocol and functional correctness properties.

Definition 3.1 (Scheduler protocol). A trace of marker functions *tr* satisfies the scheduler protocol, written tr_prot *tr*, if it is accepted by the transition system in Fig. 5, starting in the Idling state.

Definition 3.2 (Functional correctness). A trace *tr* is functionally correct (tr_valid *tr*) if it satisfies:

- *Selected jobs have the highest priority:* For any index i, if $tr[i] = \mathsf{M\_Dispatch}\ j$, then $j \in \mathsf{pending\_jobs}(i)$ and for every other job $j' \in \mathsf{pending\_jobs}(i)$, the priority of job j is higher-than-or-equal to the priority of job j'.
- *Idling only if no jobs are pending:* For any i, if $tr[i] = \mathsf{M\_Idling}$, then $\mathsf{pending\_jobs}(i) = \emptyset$.
- *Jobs have unique identifiers:* If $tr[i] = \mathsf{M\_ReadE}\ \_\ j_1$ and $tr[k] = \mathsf{M\_ReadE}\ \_\ j_2$, then $j_1 = j_2$ only if $i = k$.

The set of all pending jobs at index i is defined as the jobs that have been read but not yet dispatched:

$$\mathsf{pending\_jobs}(i) \triangleq \{j \mid \exists k_r < i.\ tr[k_r] = \mathsf{M\_ReadE}\ \_\ j \wedge \forall k < i.\ tr[k] \neq \mathsf{M\_Dispatch}\ j\}$$

The first two functional properties have already been mentioned in §2.2. The third property states that each read creates a unique identifier for the job (even if multiple packets with identical data are received). We explain below how we assign unique identifiers.

Specifying marker functions. Let us now see how we use RefinedC to prove the invariants tr_prot *tr* and tr_valid *tr*. The key observation here is that most aspects of verifying C code stay unchanged since only the marker functions (and the read function) appear on the trace *tr*. So let us now focus on how we adapt RefinedC to handle these marker functions. First, we discuss how we enrich the RefinedC specification language such that we can express specifications for these marker functions that ensure that the invariants are maintained. Concretely, let us explain how we specify the marker functions based on the example of `idling_start()`:

$$\{\mathsf{current\_trace}\ tr * \mathit{last}\ tr = \mathsf{M\_Selection} * \mathsf{currently\_pending}\ js * js = \emptyset\}$$
`  idling_start()`
$$\{\mathsf{current\_trace}(tr +\!\!+ [\mathsf{M\_Idling}]) * \mathsf{currently\_pending}\ js\}$$

For the purposes of this discussion, RefinedC can be seen as separation-logic based verification tool and thus the specification is given as a separation-logic Hoare triple. The precondition starts with the assertion current_trace *tr* stating that the trace produced by the execution so far is *tr*. This trace *tr* is used to check that the last invoked marker function is M_Selection, thus ensuring that the scheduler protocol is maintained (see Fig. 5). Additionally, we need to check that there are no pending jobs when Rössl enters the idling state (Def. 3.2). For this, we introduce the currently_pending *js* assertion stating that *js* is the set of currently pending jobs and check that it is empty (*i.e.,* $js = \emptyset$). In the postcondition, we obtain that the trace *tr* has been extended by M_Idling via the current_trace ($tr +\!\!+ [\mathsf{M\_Idling}]$) assertion and the pending jobs have not been modified. Crucially, current_trace *tr* and currently_pending *js* are stateful separation logic assertions that can evolve during the verification: the caller of `idling_start()` has to give up the ownership of current_trace *tr* when proving the precondition and obtains the updated current_trace ($tr +\!\!+ [\mathsf{M\_Idling}]$) back, ensuring that it always reflects the current trace.

Verifying the Rössl code. Using the specifications for the marker functions, we verify Rössl's C code using RefinedC. Concretely, we define new RefinedC predicates to tie our assertions like currently_pending *js* to the state of the scheduler in C. Thanks to RefinedC's extensible nature, we can extend its automation with rules for automatically reducing separation logic entailments about currently_pending *js* or current_trace *tr* to pure side conditions.

$$id \in \text{job\_id} \triangleq \mathbb{N} \qquad data \in \text{msg\_data} \triangleq \text{list}\ \mathbb{Z} \qquad j \in \text{Job} \triangleq (\text{msg\_data} * \text{job\_id})$$

$$f_{tr} \in \text{TraceFn} \triangleq \text{TrReadS} \mid \text{TrSelection} \mid \text{TrDisp} \mid \text{TrIdling} \mid \text{TrExec} \mid \text{TrCompl}$$

$$e \in \mathit{Expr} \triangleq \ldots \mid \text{BinOp}\ op\ e_1\ e_2 \mid \text{ReadE}\ e_{fd}\ e_{data}\ e_{len} \mid \text{TraceE}\ f_{tr}\ e$$

$$\sigma \in \mathit{State} \triangleq \mathit{State}_{heap} * \mathit{State}_{trace}$$

$$\sigma_{trace} \in \mathit{State}_{trace} \triangleq \{\text{idx} : \text{job\_id};\ \text{id\_map} : \text{msg\_data} \xrightarrow{\text{fin}} \text{list Job}\}$$

READ-STEP-FAILURE

$$(\text{ReadE}\ \mathit{sock}\ l\ \mathit{len}, (\sigma_{heap}, \sigma_{trace})) \xrightarrow{\text{M\_ReadE}\ \mathit{sock}\ \bot}_{\text{h}} (-1, (\sigma_{heap}, \sigma_{trace}))$$

READ-STEP-SUCCESS

$$\frac{j = (\mathit{data}, \sigma_{trace}.\text{idx}) \qquad \sigma'_{heap} = \sigma_{heap}[l \leftarrow \mathit{data}] \qquad |\mathit{data}| \le \text{max\_length} \qquad \sigma'_{trace}.\text{idx} = 1 + \sigma_{trace}.\text{idx} \qquad \sigma'_{trace}.\text{id\_map} = \sigma_{trace}.\text{id\_map}[\mathit{data} \leftarrow \sigma_{trace}.\text{id\_map}[\mathit{data}] +\!\!+ [j]]}{(\text{ReadE}\ \mathit{sock}\ l\ \mathit{len}, (\sigma_{heap}, \sigma_{trace})) \xrightarrow{\text{M\_ReadE}\ \mathit{sock}\ j}_{\text{h}} (|\mathit{data}|, (\sigma'_{heap}, \sigma'_{trace}))}$$

TRACE-STEP-IDLING

$$(\text{TraceE TrIdling}\ v, (\sigma_{heap}, \sigma_{trace})) \xrightarrow{\text{M\_Idling}}_{\text{h}} (\text{void}, (\sigma_{heap}, \sigma_{trace}))$$

TRACE-STEP-DISPATCH

$$\frac{\sigma_{heap}[l] = \mathit{data} \qquad |\mathit{data}| = \mathit{len} \qquad \sigma_{trace}.\text{id\_map}[\mathit{data}] = j :: \mathit{js}}{(\text{TraceE TrDisp}\ (l, \mathit{len}), (\sigma_{heap}, \sigma_{trace})) \xrightarrow{\text{M\_Dispatch}\ j}_{\text{h}} (\text{void}, (\sigma_{heap}, \sigma_{trace}))}$$

Fig. 6. Extension of RefinedC's Caesium operational semantics (excerpt, simplified).

3.2 Extending RefinedC's Semantics

Now that we have seen how we verify the Rössl code against the specifications for the marker functions, let us discuss how we prove that these specifications imply the desired trace invariants over the execution of Rössl. For this, we first need to understand how RefinedC models C code. RefinedC reasons about C code by translating the code into Caesium, a deep embedding of a subset of C in Rocq. The execution of a Caesium program is defined via a small-step operational semantics. RefinedC's logic and proof automation then reason about the Caesium code and are proven foundationally sound against its semantics.

The existing version of RefinedC only proves that the C code does not have undefined behavior, but does not support any reasoning about traces. Therefore, we have to extend Caesium with a notion of traces, and expressions can add events to this trace (in particular, the marker functions and the `read` system call). Fig. 6 shows the extensions to Caesium we make: At a high-level, we first have to extend the set of language expressions *Expr* to feature a new expression for reads ReadE and TraceE for marker functions. In order to assign unique identifiers to jobs (Def. 3.2), we moreover have to extend the state *State* of Caesium with a new component State_{trace}. Finally, we add new inference rules to the small-step operational semantics of Caesium, one each for failed and successful reads, and one each for every marker function. All of these steps emit an event on the trace of marker functions. We explain these additions in more detail below.

Axiomatizing the read system call. In order to interface with external events, Rössl utilizes the `read` system call. To model this system call in RefinedC, we introduce a new expression ReadE $e_{sock}\ e_{data}\ e_{len}$ that takes the socket (*sock*) to read from, a memory location that the data

is written to, and a maximum length of data that is written. We give a semantics to ReadE by axiomatizing the behavior of `read`.[4] In particular, executing a read non-deterministically picks one of two rules: READ-STEP-SUCCESS for the case that a message is read and READ-STEP-FAILURE for the case that no message is available. Both rules operate on tuples of the program expression and the program state, and emit a M_ReadE event, which adds this marker to the trace.

To ensure that each read job j is unique, the operational semantics keeps track of an index σ_{trace}.idx that is incremented on every read and added to the job as a unique identifier. (We cannot rely on the data to be the identifier since multiple packets might contain identical data.) Additionally, the operational semantics maintains a map σ_{trace}.id_map that the marker functions use to look up the corresponding job j for the provided data.

Marker functions. We also add the marker functions to Caesium with a new expression TraceE f_{tr} e that takes as arguments the kind of marker to emit (*e.g.,* TrIdling) and a value that is used in a marker-function specific way. We have instrumented the RefinedC frontend to emit TraceE instructions for the implementation of the marker functions.

The semantics of the marker functions is relatively straightforward. For instance, the TrIdling marker function simply emits a M_Idling event (TRACE-STEP-IDLING). The TrDisp marker function has a more complicated semantics, as the M_Dispatch j event it emits is tagged with the dispatched job j. Therefore, TRACE-STEP-DISPATCH interprets the argument as a tuple of a memory location l and length *len* and reads *data* of length *len* from the location l. Then, it looks up the first possible identifier j that can be assigned to a message with this data in the σ_{trace}.id_map map.[5]

3.3 Adequacy

Finally, we can put everything together and show that Rössl code verified using the marker function specifications (§3.1) against the Caesium semantics (§3.2) satisfies the desired invariants (§3.1).

Linking the new assertions to the trace. In the first step, we link the new assertions from §3.1—in particular current_trace and currently_pending—to the trace emitted by Caesium. For this, we extend RefinedC's *state interpretation*. The state interpretation is a separation logic predicate on the Caesium state that holds at each step of the execution. Usually, it is used to tie the heap in Caesium's state to separation logic points-to assertions.

Following Sammler et al. [37], we parameterize the state interpretation additionally with the trace of the current execution. Concretely, we define the state interpretation as follows:

$$SI((\sigma_{heap}, \sigma_{trace}), tr) \triangleq \mathsf{heap\_interp}\ \sigma_{heap} * \mathsf{tr\_prot}\ tr * \mathsf{tr\_valid}\ tr * \mathsf{trace\_state\_inv}\ \sigma_{trace}\ tr * \mathsf{pending\_jobs\_auth}\ tr * \mathsf{current\_trace\_auth}\ tr * \ldots$$

Here heap_interp is the existing state interpretation of RefinedC that links the heap σ_{heap} to the points-to connective. Using tr_prot tr and tr_valid tr, the state interpretation ensures that the scheduler protocol and functional correctness properties hold for the trace tr at every step of the execution. The trace_state_inv invariant relates the trace to the state of the operational semantics σ_{trace} by requiring that the set of IDs in the id_map field of σ_{trace} are consistent with the read but unfinished jobs in the trace so far. Finally, pending_jobs_auth and current_trace_auth link the currently_pending and current_trace assertions to the trace tr. With this state interpretation, we can validate the specifications of the marker functions form §3.1 against the operational semantics from §3.2.

[4] We only model `read` for the specific case of non-blocking message-based I/O on datagram sockets as used by Rössl.
[5] Note that the concrete ID that is picked here is not relevant, as the execution behavior is indistinguishable if the *data* is the same. We just have to make sure to pick an ID that was read before and which hasn't been dispatched yet.

Definition 3.3 (Rössl client). A client program of Rössl provides the following logical definitions (Rössl is generic in these): the list of tasks (*i.e.,* callback functions) τ to handle, the list of file sockets *input_socks* to read from, a mapping *msg_to_task* that infers the task of a message given its data, and a mapping *task_prio* that determines the priority of each task. Moreover, a client of Rössl implements a C function `msg_identify_type`, which computes the task type of a message according to *msg_to_task*. Finally, the client's `main` function has to initialize Rössl by adding the necessary sockets *input_socks* and registering the tasks τ together with their priority according to *task_prio* and callback, before calling into Rössl's `fds_run` main loop.

Given such a client of Rössl, we verify:

Theorem 3.4 (Rössl Adequacy). *Assume a client P of Rössl according to Def. 3.3. Then:*

$$\frac{(P, \sigma_{init}) \xrightarrow{tr}{}^{*} (e_{final}, \sigma_{final})}{(e_{final}, \sigma_{final}) \text{ is not stuck} \qquad \text{tr\_prot tr} \qquad \text{tr\_valid tr}}$$

4 RTA in Prosa

Next, we describe our verified RTA for the abstract model of Rössl (§2.4). Instead of a finite representation as lists of processor states, we reason on traces represented as possibly infinite schedules of type $\mathbb{N} \to \mathit{ProcessorState}$. This representation is standard in RTS theory and used throughout Prosa.

We start by describing Prosa's abstract model of workloads and schedules (§4.1). Our RTA relies on an instantiation of this model for Rössl. Next, we explain the aRSA technique, which we use to establish response-time bounds in the presence of overheads (§4.2). §4.3 and §4.4 explain how we discharge the assumptions needed to apply aRSA. Specifically, §4.3 explains our use of release jitter to handle the discrepancies between the time a job arrives in the system and the time at which it is read, and §4.4 explains how we state and verify a so-called supply bound function (*SBF*), which is a lower bound on the non-overhead time in any interval of a given duration.

4.1 The Abstract Model

We describe the model of workloads and schedules used by Prosa that we instantiate to state an RTA for Rössl. We divide the model's presentation into *statics*—parameters that are fixed across runs of Rössl—and *dynamics*—parameters that are specific to a single run.

Statics. A set of n distinct task types or *tasks* $\tau = \tau_1 \dots \tau_n$ describe characteristics of jobs that the processor may be asked to execute. The task τ_i describes jobs that have the following common characteristics: a WCET ($C_i : \mathbb{N}$), a priority level ($P_i : \mathbb{N}$), and a bound on the rate of arrivals specified as a function $\alpha_i : \mathbb{N} \to \mathbb{N}$, called the *arrival curve*, which is an upper bound on the number of jobs of type τ_i that may arrive in the system in any time interval of duration $\Delta : \mathbb{N}$.

Dynamics. A *job* is a runtime instance of a task. We let $\tau_{i,j}$ denote the j-th job of task τ_i. Concretely, each job is a message (with a unique ID, see §3.2) that arrives on one of the sockets.

We model a system run using an *arrival sequence* and a (processor) *schedule*. The arrival sequence, denoted $arr : \mathbb{N} \to \mathit{list\ job}$, models the workload. It maps each time instant to the set of jobs that arrive at that instant. For job $\tau_{i,j}$, we let $a_{i,j}$ denote its arrival time according to *arr*, i.e., $a_{i,j}$ is the unique time instant t such that $\tau_{i,j} \in arr(t)$. Our analysis assumes that any arrival sequence respects the arrival curve α_i, *i.e.,* the number of arrivals in any interval Δ is bounded by $\alpha_i(\Delta)$:

$$\forall t, \forall \Delta, \ \left|\left\{\tau_{i,j} \;\middle|\; t \le a_{i,j} < t + \Delta\right\}\right| \le \alpha_i(\Delta). \tag{2}$$

The *schedule*, *sched* : $\mathbb{N} \to$ *ProcessorState*, is a mapping from time instants to processor states. This is an abstract representation of the processor's activity at a given time instant (see §2.4).

Objective. Given an abstract system model, the objective of an RTA is to establish a *response-time bound*, R_i, which we define below, for each task τ_i. In the rest of this section, our objective is to find response-time bounds for the abstract system model of Rössl.

Definition 4.1 (Response-Time Bound). R_i is said to be a response-time bound for a task τ_i if, for any arrival sequence *arr* satisfying Eq. 2, any schedule *sched* consistent with *arr*, any $\tau_{i,j}$ with arrival time $a_{i,j}$ in *arr*, and any time t, if $a_{i,j} + R_i \leq t$, then $\tau_{i,j}$ has been completed in *sched* by t.

4.2 Applying the Restricted-Supply Analysis of aRSA

Our RTA builds on the aRSA framework [10] to establish overhead-aware response-time bounds for Rössl. aRSA, a recently-developed RTA technique available in Prosa, allows for verifying RTAs for non-ideal processors subject to "supply restrictions"—restrictions on the amount of processor time available to actually run jobs. While aRSA was not designed to verify overhead-aware RTAs, we use it for this purpose by mapping Rössl's overheads to aRSA's supply restrictions. Although a complete description of aRSA is beyond the scope of this paper, in the following, we describe a relevant subset of aRSA briefly, focusing on its inputs and outputs, and the challenges we face in applying it to Rössl's NPFP scheduler.

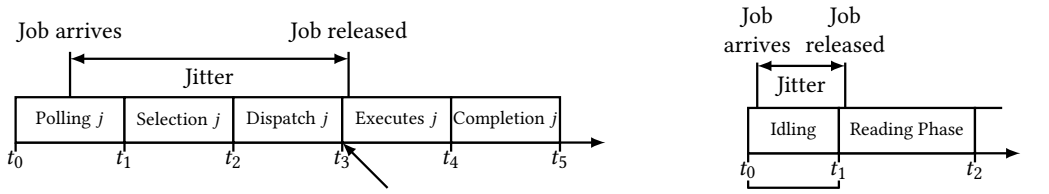
A primer on aRSA. aRSA is an abstract framework for verifying RTAs for schedulers subject to "supply restrictions". *Supply* is standard RTS terminology for the amount of system time available for executing the system's workload. The additive complement of supply is *blackout*, *i.e.,* the time when the system is *not* available for processing external workload (*i.e.,* the supply is restricted). In our case, we model all overhead states (*i.e., ReadOvh*, *PollingOvh*, *SelectionOvh*, *DispatchOvh*, and *CompletionOvh*) as blackouts.

aRSA is an abstract framework. Consequently, it is generic in the scheduling policy, the processor state model, the kind of workload, *etc.* Given an instantiation of these entities as well as proofs that the arrival sequence and the schedule are coherent according to the system model under consideration, aRSA yields a response-time recurrence for every task in the workload. aRSA establishes that the solution of this response-time recurrence is a bound on the response time of any job of the task under consideration.

To obtain response-time bounds for Rössl, we first concretely define Rössl's system model by defining the processor state and workload models under consideration. Then, we prove that Rössl satisfies aRSA's required properties and instantiate aRSA, which yields the desired bound. In the rest of this section, we first describe the schedule properties that aRSA assumes and the challenges we face in establishing them for Rössl. We then describe how we address these challenges.

Properties required by aRSA. First, aRSA requires that the schedule for any given arrival sequence be *priority-policy compliant* (w.r.t. the priority policy under consideration, *i.e.,* NPFP): If a job $\tau_{i,j}$ starts executing at time t, then $\tau_{i,j}$ has the highest priority among all jobs that have *arrived* but not *executed* before t. In Rössl's case, priority-policy compliance can actually be *temporarily* violated since jobs that arrive between the polling and job execution phases may not be read before the next job to execute is selected, and these newly arrived jobs may be of higher priority higher than the selected job. We bridge the gap between aRSA's definition of priority-policy compliance and Rössl's implementation using the standard RTS concept of *release jitter* [3, 12] in §4.3.

Second, aRSA requires that the schedule generated by the scheduler for an arrival sequence be *work-conserving*, which means that the processor idles only if all previously *arrived* jobs have been serviced. Again, this assumption can be briefly violated by Rössl's implementation because a job



(a) Ensuring priority-compliance (b) Ensuring work conservation

Fig. 7. Release jitter to ensure that the scheduling policy satisfies certain properties expected by aRSA

that arrives while the processor idles cannot be dispatched until the polling loop reads from the socket on which the job arrived. The processor state between the job arriving and it being read is *Idle* in Rössl's schedule, but aRSA requires it to be non-idling if there is a ready job. We bridge this difference using a second form of release jitter.

Finally, aRSA models supply restrictions generically using the concept of a *supply bound function* (*SBF*), which must be defined by the user. Specifically, $SBF(\Delta)$ must be a lower bound on the supply (*i.e.,* the time available for executing jobs) in any interval of length Δ. (Technically, aRSA requires *SBF* to bound the supply only in a task's "busy window", which is an interval where the processor is continuously non-idle, but we defer these details to the appendix [5].) *SBF* has to be defined by aRSA's user for the abstract system model under consideration. In §4.4, we define a suitable *SBF* for Rössl by establishing bounds on how much overhead can delay a job under consideration.

4.3 Priority-policy Compliance and Work Conservation Using Release Jitter

Next, we describe how we the obtain priority-policy compliance and work conservation properties for Rössl using release jitter.

Priority-policy compliance using release jitter. As explained earlier, Rössl's implementation may temporarily violate priority-policy compliance because jobs arriving between the polling phase and the execution phase are not considered when scheduling the next job. In fact, this effect is a typical example of the discrepancies between the idealized abstract models used in scheduling theory and the practical limitations arising from engineering constraints in real implementations. As such, we cannot apply aRSA to Rössl's schedules directly.

As a solution, we apply aRSA to the schedule and a modified arrival sequence—called the *release sequence*—in which we model the arrival of every job $\tau_{i,j}$ as delayed by a certain amount of time called the release jitter, denoted $jitter_{i,j}$. For a job that arrives between the polling and execution phases, we choose $jitter_{i,j}$ to be large enough to delay the arrival of the job past the start of the next execution phase. As a result, the release sequence makes the schedule priority-policy compliant. Fig. 7a illustrates this transformation of the arrival sequence. (The term "job release" in Fig. 7a refers to the conceptual "release" of the job to the system after it has been delayed by the jitter and is standard terminology [3].)

We explain momentarily how we recover a response-time bound w.r.t. the (original) arrival sequence from the response-time bound relative to the release sequence that aRSA gives us.

Work conservation using release jitter. As explained earlier, Rössl's scheduling policy may not be work-conserving because a real implementation cannot react instantaneously to incoming packets. We address this issue using jitter as well. Specifically, for a job $\tau_{i,j}$ that arrives in the *Idle* state in a schedule, we define $jitter_{i,j}$ to be large enough to push the job's arrival past the idling

state in the release sequence. Hence, the schedule is work-conserving w.r.t. the release sequence. Fig. 7b illustrates this use of jitter.

Tying it all together. Having obtained a release sequence as described above, we establish a supply bound function *SBF* for the schedule relative to this release sequence (the *SBF* is explained in §4.4). We then apply aRSA. This yields a response-time bound w.r.t. the release sequence. We recover a response-time bound w.r.t. the original arrival sequence using the following theorem [3].

Theorem 4.2 (Jitter-aware response-time bound). *If $\mathcal{J}_i$ is an upper bound on the release jitter applied to jobs of task τ_i and if R_i is a response-time bound for task τ_i w.r.t. the release sequence, then $R_i + \mathcal{J}_i$ is a response-time bound on the task τ_i w.r.t. the arrival sequence.*

What remains is to find a suitable upper bound, $\mathcal{J}_i$, on the jitter applied to jobs of type τ_i. To this end, we find upper bounds on the two types of release jitter above. The release jitter required to restore priority-policy compliance is upper-bounded by the sum of the durations of the states *PollingOvh*, *SelectionOvh*, *DispatchOvh*, while the release jitter needed to restore work conservation is upper-bounded by the duration of the *Idle* state immediately following a job's arrival. Recall from §2.4 that we calculate the upper bounds *PB*, *SB*, *DB* and *IB* on the durations of these states using WCET assumptions. Here, we use these upper bounds to define the $\mathcal{J}_i$ as:

Definition 4.3 (Maximum release jitter). $\mathcal{J}_i \triangleq 1 + \max(PB + SB + DB, IB)$.

The release curve. The release sequence may not satisfy the constraints of the arrival curve, α_i, since release jitter can have the effect of shifting jobs more closely together in time beyond what α_i stipulates. To handle this, we define a new arrival curve, called the *release curve* and denoted β_i, such that $\beta_i(\Delta) \triangleq 0$ if $\Delta = 0$ and $\beta_i(\Delta) = \alpha_i(\Delta + \mathcal{J}_i)$ otherwise. Given this definition, β is easily seen to be an upper bound on the rate of arrivals in the release sequence.

4.4 Defining the Supply Bound Function

In this subsection, we describe how we define the supply bound function, $SBF(\Delta)$, for Rössl's NPFP policy. To do this, for any task, we first define a suitable blackout-bound function, $BlackoutBound(\Delta)$. Crucially, our definition ensures that $BlackoutBound(\Delta)$ is an upper bound on the time spent in Rössl's overhead states in an interval of length Δ, hence modeling overheads as blackouts.

Specifically, we define two helper functions: 1) $TRB(\Delta)$, which bounds blackouts caused by instances of *ReadOvh*, and (2) $NRB(\Delta)$, which bounds blackouts caused by instances of *PollingOvh*, *SelectionOvh*, *DispatchOvh* and *CompletionOvh*. Then, we define $BlackoutBound(\Delta) \triangleq NRB(\Delta) + TRB(\Delta)$. We establish the correctness of $NRB(\Delta)$ and $TRB(\Delta)$ separately. For each, we use the bound on the rate of arrivals to bound the number of jobs that arrive in an interval of length Δ, and then use the scheduler protocol and WCETs of processor states to calculate a bound. The details are presented in the appendix [5].

Next, we define the supply bound function, $SBF(\Delta) \triangleq \max_{0 \le \delta \le \Delta}(\delta - BlackoutBound(\delta))$, and prove that this is a lower bound on the time devoted to job execution in any interval of length Δ. We take the max over all interval lengths from 0 to Δ to ensure that $SBF(\Delta)$ is monotonically non-decreasing, which aRSA requires ($\delta - BlackoutBound(\delta)$ may not be monotonic in δ).

Finally, we instantiate aRSA using the *SBF* defined above to obtain a response-time bound w.r.t. the release sequence as anticipated in §4.3.

5 Adequacy

This section describes the final adequacy theorem of Rössl. At a high level, we obtain this theorem by first applying our RefinedC adequacy theorem (Thm. 3.4), propagating the properties of the

timed trace of marker functions to properties of the timed trace of processor states, applying the response-time analysis in Prosa (§4), and then back-translating the proof of the response-time bound to the original trace of marker functions.

Theorem 5.1 (Timing Correctness). *For a client P of Rössl according to Def. 3.3, defining a set of tasks τ and the set of input sockets input_socks, fix the following:*

- *a set of valid arrival curves α_i (for each $\tau_i \in \tau$),*
- *the WCETs for basic actions (WcetFR, WcetSR, WcetSel, WcetDisp, WcetCompl, WcetIdling) of which WcetSel, WcetDisp, WcetCompl and WcetIdling are strictly positive and $1 <$ WcetFR and $1 <$ WcetSR, and*
- *the WCETs of callback executions C_i (for each task $\tau_i \in \tau$) with $0 < C_i$ for each i.*

Furthermore, assume a run of Rössl as follows:

- *an execution $(P, \sigma_{init}) \xrightarrow{tr}{}^{*} (e_{final}, \sigma_{final})$ of P in the instrumented Caesium semantics (§3.2) starting in the initial state σ_{init}, with trace tr,*
- *arrival sequences arr_{sock} (for every sock $\in$ input_socks) with arrivals bounded by the α_i's,*
- *a list of timestamps ts, marking the start times for each marker function, such that each basic action respects its WCET and such that the resulting timed trace (tr, ts) is consistent with arr (Def. 2.1), and*
- *a horizon t_{hrzn} up to which the scheduler is known to have run.*

Let R_i be the response-time bound w.r.t. the release sequence obtained from aRSA, which is precisely defined in the appendix [5] and parametric in the arrival curves, the WCETs, the task τ_i under consideration, and the number of input sockets input_socks. Then $R_i + \mathcal{J}_i$ is a response-time bound in tr and ts for the task τ_i:

$$\forall sock, j, t_{arr}.\ j \in arr_{sock}[t_{arr}] \land msg\_to\_task\ j = \tau_i \land t_{arr} + R_i + \mathcal{J}_i < t_{hrzn} \implies$$
$$\exists k.\ tr[k] = M\_Completion\ j \land ts[k] \le t_{arr} + R_i + \mathcal{J}_i$$

This theorem shows that, for each execution of Rössl, the response time of each job (of task τ_i) is bounded by $R_i + \mathcal{J}_i$. (Recall from Thm. 4.2 that we offset the response-time bounds obtained from aRSA by the maximum release jitter $\mathcal{J}_i$.) The $t_{arr} + R_i + \mathcal{J}_i < t_{hrzn}$ condition is necessary since we can guarantee only the completion of jobs whose response-time bound is within the time horizon, t_{hrzn}. Further details of the theorem are provided in the appendix [5].

Trusted computing base (TCB). Thm. 5.1 is proven in Rocq without additional axioms. In addition to the TCB of Rocq, we inherit the TCB of RefinedC, in particular its frontend and the Caesium C semantics (including our additions to Caesium like the axiomatization of `read` and the marker functions). Note that the definition of a client (Def. 3.3) is phrased in terms of the RefinedC program logic (and thus Iris). However, given a concrete client, the results of Thm. 5.1 do *not* depend on the RefinedC program logic nor Iris, so they are not part of the TCB.

The proof effort. We divide our proof effort as follows: (a) Our extension of RefinedC with marker functions and traces, including changes to the Caesium semantics and the corresponding modifications to RefinedC's adequacy theorem (2,150 lines of code or LoC); (b) The Rössl C source code (300 LoC); (c) Our specifications of Rössl (615 LoC); (d) RefinedC proofs of properties of Rössl's marker function traces, including Thm. 3.4 (4,300); (e) Gallina code to transform marker function traces to traces of timestamped processor states, and proving that properties of marker function traces carry over to traces of timestamped processor states (12,350 LoC); (f) Converting traces of timestamped processor states to finite schedules and then to Prosa's schedule representation, showing that properties are preserved along these transformations (11,700 LoC); (g) The RTA

including definitions and proofs of the SBF, the definition of the response-time bound and the instantiation of aRSA (4,000 LoC).

We believe that significant pieces of our proof effort are reusable. The 2,150 LoC in category (a) could be reused for any RefinedC verification effort that relies on marker function traces. The 27,050 LoC in categories (e) and (f) are language-agnostic and largely independent of the scheduler code. They could be reused for verifying other non-preemptive, polling-based schedulers written in any language. The 4,000 LoC in category (g) are largely specific to the Rössl scheduling policy, but some of that could transfer to other scheduling policies as well.

6 Related Work

The line of work most closely related to ours concerns ProKOS [27, 28], which integrates schedulability analysis into RT-CertiKOS, a verified single-core sequential real-time OS kernel [33, 34]. Like RefinedProsa, ProKOS uses Rocq to foundationally verify the schedulability of schedulers written in C, with ProKOS considering both FP and *earliest-deadline first* (EDF) policies, and the high-level structure of its proof has served as a reference point for that of RefinedProsa's. However, as we detail below, there are significant differences.

Interrupt-free vs. tick-based schedulers. In their initial work on ProKOS, Guo et al. [27] leveraged Prosa to verify the schedulability of RT-CertiKOS's fixed-priority, *tick-based* (preemptive) single-core scheduler. Guo et al. [28] later built on this work to verify the schedulability of RT-CertiKOS's EDF scheduler. In so doing, they proposed a generic interface for connecting the RTA of Prosa to the verified scheduler implementations of RT-CertiKOS. In another line of work, Liu et al. [33, 34] proved additional guarantees for RT-CertiKOS, including timing isolation and budget enforcement properties.

However, all the aforementioned verification efforts focus on *tick-based* schedulers and hence employ techniques that would not apply to interrupt-free schedulers like Rössl. First, tick-based schedulers can rely on the distance between two ticks as a unit of time and, hence, they do not need a separate treatment of time. In contrast, in our interrupt-free setting, there is no inherent notion of time, which is why we developed the multi-step approach described in §2 to incorporate for temporal reasoning. Second, ProKOS models overheads in a simple but highly abstract way—as a fixed percentage of the time between two ticks. It is not clear how to adapt such an approach to interrupt-free schedulers. Thus, in RefinedProsa, we instead model overheads at a much finer granularity: we account for multiple different sources of overhead, and obtain the total overhead by aggregating the individual overheads.

Finally, RT-CertiKOS makes the simplifying assumption that tasks are periodic. In RefinedProsa, we eliminate this periodicity assumption in the arrival sequence, which makes our model more general but also more complex to analyze. Related to this point: Vanhems et al. [42] also develop an approach to eliminating RT-CertiKOS's periodicity assumption. They verify an EDF scheduler implemented in Rocq and extract it to C using Digger [39]. However, their scheduler is still tick-based, and their verification accounts for overheads in a manner similar to RT-CertiKOS.

Structure of the verification effort. As noted in §1.2, the structure of RefinedProsa is similar to that of ProKOS: we start by proving invariants on traces of the scheduler (albeit in our case using RefinedC) and then use these to establish the properties on schedules of processor states (*e.g.,* work conservation, priority-policy compliance, *etc.*) that are required by our RTA in Prosa. In doing so, we encounter some of the same technical challenges as ProKOS did. Notably, we establish invariants on *finite* traces, whereas Prosa is built around a representation of schedules that can accommodate possibly *infinite* schedules. Like ProKOS, we therefore extend Rössl's traces by manually scheduling the completion of any pending jobs to fit Prosa's standard representation

and its associated invariants. (ProKOS further extends the trace to include all future job arrivals according to a periodic arrival sequence; as we demonstrate, such an infinite extension is not required, given that the end result is a response-time guarantee on the finite trace.)

However, due to our focus on an overhead-aware, interrupt-free scheduler, and our support for arbitrary arrival curves (as opposed to periodic arrivals), we also encounter other technical challenges not faced by ProKOS, in some cases requiring new and different solutions.

The root complication pertains to the nature of the scheduler actions that we are reasoning about. In ProKOS, due to the tick-based setting, there is no explicit reasoning about the time taken by the scheduler code itself. Consequently, it is possible to directly record a trace of processor states, and the notion of processor state is simple: at each instant, either the processor is executing some job or it is idle. In contrast, such a simple approach is not applicable to RefinedProsa since we need to reason temporally about the scheduler code itself.

Instead, the first step in RefinedProsa is to record an (untimed) trace of basic actions which accounts at a fine granularity for the different segments of the scheduler code, including the different kinds of overheads. Next, we introduce times into the trace and define an algorithm to convert the timed trace into a schedule of processor states. Correspondingly, our notion of processor states is much richer than that of ProKOS since (a) it must account for states when the processor is executing some overhead action and (b) it should also hide implementation-specific details which the RTA does not care about, such as the number of sockets. Having defined this algorithm, we then have to convert all the relevant properties on the trace of basic actions (which we have proved using RefinedC) into properties on the computed schedule of processor states. In this step, we not only have to prove properties such as work conservation and priority-policy compliance, but also some more complicated properties about the schedule of processor states—*e.g.,* that at the beginning of any *PollingOvh* or *Idle* period, all the jobs that have arrived into the system have been read. Finally, the handling of overheads also complicates the RTA on the Prosa side, necessitating the use of the aRSA framework. Specifically, we have to define and prove the correctness of an *SBF* and use the jitter modeling to establish work conservation and priority-policy compliance.

Comparison with VeriRT. In concurrent work, Kim et al. [30] present VeriRT, an end-to-end framework for verifying timing properties of distributed systems. They do not prove response-time bounds, which are the focus of our work. Like RefinedProsa, VeriRT relies on WCET assumptions about the durations of system calls and code sections between consecutive system calls, but the two frameworks differ in their model of time. VeriRT wraps the language's small-step semantics to track time and generate (symbolic) timestamps with each trace event. In contrast, RefinedProsa generates untimed traces first and adds timestamps afterward. Of the two approaches, VeriRT's approach is more expressive as it supports programs with *time-dependent control flow*. On the other hand, our model is arguably simpler (as we do not have to define the wrapping semantics), and it suffices for verifying nonpremptive, interrupt-free schedulers (as such schedulers do not have time-dependent control flow).

VeriRT has a "lifting theorem", which lifts the (relational) simulation relation of the language's untimed semantics to the simulation relation of the augmented timed semantics under the assumption that the syscall-to-syscall WCETs of the simulated program and the simulating program are equal. By applying this theorem to CompCert's existing source-target simulation, it is further shown that a C program simulates compiled code generated by the CompCert compiler [31]. We believe that a similar lifting theorem could be proved for RefinedProsa's timing model, applied to a compiler like CompCert, and combined with RefinedProsa's analysis to establish response-time bounds on compiled code running under machine semantics, not just C source code under RefinedC's semantics. However, we have not yet attempted such end-to-end proofs.

Acknowledgments

We would like to thank the anonymous reviewers for their helpful feedback.

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 803111).

Data Availability Statement

The Coq development and appendix for this paper can be found in Bedarkar et al. [5]. The current development version of RefinedProsa is linked from the project webpage at https://plv.mpi-sws.org/refinedprosa/.

References

[1] AbsInt Angewandte Informatik GmbH. 2024. Qualification Support for AIT. https://www.absint.com/ait/qualification.htm Accessed: 2025-03-22.

[2] AbsInt Angewandte Informatik GmbH. 2024. TimeWeaver: Timing Analysis for Task-Interaction Effects. https://www.absint.com/timeweaver/index.htm Accessed: 2025-03-22.

[3] Neil Audsley, Alan Burns, Mike Richardson, Ken Tindell, and Andy J Wellings. 1993. Applying new scheduling theory to static priority pre-emptive scheduling. *Software engineering journal* 8, 5 (1993), 284–292. doi:10.1049/SEJ.1993.0034

[4] Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. 2010. OTAWA: An Open Toolbox for Adaptive WCET Analysis. In *SEUS (Lecture Notes in Computer Science, Vol. 6399)*. Springer, 35–46. doi:10.1007/978-3-642-16256-5_6

[5] Kimaya Bedarkar, Laila Elbeheiry, Michael Sammler, Lennard Gäher, Björn Brandenburg, Derek Dreyer, and Deepak Garg. 2025. Artifact for "RefinedProsa: Connecting Response-Time Analysis with C Verification for Interrupt-Free Schedulers". doi:10.5281/zenodo.15185870 Project webpage: https://plv.mpi-sws.org/refinedprosa/.

[6] Kimaya Bedarkar, Mariam Vardishvili, Sergey Bozhko, Marco Maida, and Björn B. Brandenburg. 2022. From Intuition to Coq: A Case Study in Verified Response-Time Analysis of FIFO Scheduling. In *2022 IEEE Real-Time Systems Symposium (RTSS)*. 197–210. doi:10.1109/RTSS55097.2022.00026

[7] Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. 2011. Refinement types for secure implementations. *ACM Trans. Program. Lang. Syst.* 33, 2, Article 8 (Feb. 2011), 45 pages. doi:10.1145/1890028.1890031

[8] Armelle Bonenfant, Denis Claraz, Marianne De Michiel, and Pascal Sotin. 2017. Early WCET Prediction Using Machine Learning. In *WCET (OASIcs, Vol. 57)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 5:1–5:9. doi:10.4230/OASICS.WCET.2017.5

[9] Marc Boyer, Pierre Roux, Hugo Daigmorte, and David Puechmaille. 2021. A Residual Service Curve of Rate-Latency Server Used by Sporadic Flows Computable in Quadratic Time for Network Calculus. In *Proceedings of the 33rd Euromicro Conference on Real-Time Systems (ECRTS)*. 14:1–14:21. doi:10.4230/LIPICS.ECRTS.2021.14

[10] Sergey Bozhko. 2025, in preparation. *Rigorous and General Response-Time Analysis for Uniprocessor Real-Time Systems*. Ph. D. Dissertation. Saarland University, Germany.

[11] Sergey Bozhko and Björn B Brandenburg. 2020. Abstract Response-Time Analysis: A Formal Foundation for the Busy-Window Principle. In *Proceedings of the 32nd Euromicro Conference on Real-Time Systems (ECRTS)*. 22:1–22:24. doi:10.4230/DARTS.6.1.3

[12] Giorgio C Buttazzo. 2011. *Hard real-time computing systems: predictable scheduling algorithms and applications*. Springer. doi:10.1007/978-3-031-45410-3

[13] Giorgio C. Buttazzo, Marko Bertogna, and Gang Yao. 2013. Limited Preemptive Scheduling for Real-Time Systems. A Survey. *IEEE Transactions on Industrial Informatics* 9, 1 (2013), 3–15. doi:10.1109/TII.2012.2188805

[14] Daniel Casini, Tobias Blaß, Ingo Lütkebohle, and Björn Brandenburg. 2019. Response-time analysis of ROS 2 processing chains under reservation-based scheduling. In *31st Euromicro Conference on Real-Time Systems*. 1–23. doi:10.4230/LIPICS.ECRTS.2019.6

[15] Felipe Cerqueira, Geoffrey Nelissen, and Björn B Brandenburg. 2018. On strong and weak sustainability, with an application to self-suspending real-time tasks. In *Proceedings of the 30th Euromicro Conference on Real-Time Systems (ECRTS)*. 26:1–26:21. doi:10.4230/LIPICS.ECRTS.2018.26

[16] Felipe Cerqueira, Felix Stutz, and Björn B Brandenburg. 2016. PROSA: A case for readable mechanized schedulability analysis. In *Proceedings of the 28th Euromicro Conference on Real-Time Systems (ECRTS)*. 273–284. doi:10.1109/ECRTS.2016.28

[17] Jian-Jia Chen, Geoffrey Nelissen, Wen-Hung Huang, Maolin Yang, Björn Brandenburg, Konstantinos Bletsas, Cong Liu, Pascal Richard, Frédéric Ridouard, Neil Audsley, et al. 2019. Many suspensions, many problems: a review of

self-suspending tasks in real-time systems. *Real-Time Systems* 55, 1 (2019), 144–207. doi:10.1007/S11241-018-9316-9

[18] Darren D. Cofer and Murali Rangarajan. 2002. Formal Verification of Overhead Accounting in an Avionics RTOS. In *RTSS*. IEEE Computer Society, 181–190. doi:10.1109/REAL.2002.1181573

[19] Robert I Davis, Alan Burns, Reinder J Bril, and Johan J Lukkien. 2007. Controller Area Network (CAN) schedulability analysis: Refuted, revisited and revised. *Real-Time Systems* 35, 3 (2007), 239–272. doi:10.1007/S11241-007-9012-7

[20] Robert I. Davis, A. Zabos, and Alan Burns. 2008. Efficient Exact Schedulability Tests for Fixed Priority Real-Time Systems. *IEEE Trans. Computers* 57, 9 (2008), 1261–1276. doi:10.1109/TC.2008.66

[21] Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt. 2004. Contiki-a lightweight and flexible operating system for tiny networked sensors. In *29th annual IEEE international conference on local computer networks*. IEEE, 455–462. doi:10.1109/LCN.2004.38

[22] Pascal Fradet, Xiaojie Guo, Jean-François Monin, and Sophie Quinton. 2018. A generalized digraph model for expressing dependencies. In *Proceedings of the 26th International Conference on Real-Time Networks and Systems (RTNS)*. 72–82. doi:10.1145/3273905.3273918

[23] Pascal Fradet, Xiaojie Guo, Jean-François Monin, and Sophie Quinton. 2019. CertiCAN: A tool for the Coq certification of CAN analysis results. In *Proceedings of the 25th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 182–191. doi:10.1007/S11241-023-09393-2

[24] Pascal Fradet, Maxime Lesourd, Jean-François Monin, and Sophie Quinton. 2018. A Generic Coq Proof of Typical Worst-Case Analysis. In *Proceedigns of the 39th IEEE Real-Time Systems Symposium (RTSS)*. 218–229. doi:10.1109/RTSS.2018.00039

[25] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Newman Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. 2016. CertiKOS: An extensible architecture for building certified concurrent OS kernels. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 653–669.

[26] Arpan Gujarati, Felipe Cerqueira, Björn B Brandenburg, and Geoffrey Nelissen. 2019. Correspondence article: a correction of the reduction-based schedulability analysis for APA scheduling. *Real-Time Systems* 55, 1 (2019), 136–143. doi:10.1007/S11241-018-9315-X

[27] Xiaojie Guo, Maxime Lesourd, Mengqi Liu, Lionel Rieg, and Zhong Shao. 2019. Integrating Formal Schedulability Analysis into a Verified OS Kernel. In *CAV (2) (Lecture Notes in Computer Science, Vol. 11562)*. Springer, 496–514. doi:10.1007/978-3-030-25543-5_28

[28] Xiaojie Guo, Lionel Rieg, and Paolo Torrini. 2021. A generic approach for the certified schedulability analysis of software systems. In *RTCSA*. IEEE, 83–92. doi:10.1109/RTCSA52859.2021.00018

[29] Leandro Soares Indrusiak, Alan Burns, and Borislav Nikolic. 2016. Analysis of buffering effects on hard real-time priority-preemptive wormhole networks. *arXiv preprint arXiv:1606.02942* (2016). doi:10.48550/arXiv.1606.02942

[30] Yoonseung Kim, Sung-Hwan Lee, Yonghyun Kim, and Chung-Kil Hur. 2025. VeriRT: An End-to-End Verification Framework for Real-Time Distributed Systems. *Proc. ACM Program. Lang.* 9, POPL (2025), 1812–1839. doi:10.1145/3704897

[31] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115. https://doi.org/10.1145/1538788.1538814

[32] Philip Levis, Samuel Madden, Joseph Polastre, Robert Szewczyk, Kamin Whitehouse, Alec Woo, David Gay, Jason Hill, Matt Welsh, Eric Brewer, et al. 2005. TinyOS: An operating system for sensor networks. *Ambient intelligence* (2005), 115–148. doi:10.1007/3-540-27139-2_7

[33] Mengqi Liu, Lionel Rieg, Zhong Shao, Ronghui Gu, David Costanzo, Jung-Eun Kim, and Man-Ki Yoon. 2019. Virtual timeline: a formal abstraction for verifying preemptive schedulers with temporal isolation. 4, POPL (2019). doi:10.1145/3371088

[34] Mengqi Liu, Zhong Shao, Hao Chen, Man-Ki Yoon, and Jung-Eun Kim. 2022. Compositional virtual timelines: verifying dynamic-priority partitions with algorithmic temporal isolation. *Proc. ACM Program. Lang.* 6, OOPSLA2 (2022), 60–88. doi:10.1145/3563290

[35] Marco Maida, Sergey Bozhko, and Björn B Brandenburg. 2022. Foundational Response-Time Analysis as Explainable Evidence of Timeliness. In *Proceedings of the 34th Euromicro Conference on Real-Time Systems (ECRTS)*. 19:1–19:25. doi:10.4230/LIPICS.ECRTS.2022.19

[36] Pierre Roux, Sophie Quinton, and Marc Boyer. 2022. A Formal Link Between Response Time Analysis and Network Calculus. In *Proceedings of the 34th Euromicro Conference on Real-Time Systems (ECRTS)*. 5:1–5:22. doi:10.4230/LIPICS.ECRTS.2022.5

[37] Michael Sammler, Angus Hammond, Rodolphe Lepigre, Brian Campbell, Jean Pichon-Pharabod, Derek Dreyer, Deepak Garg, and Peter Sewell. 2022. Islaris: verification of machine code against authoritative ISA semantics. In *PLDI*. ACM, 825–840. doi:10.1145/3519939.3523434

[38] Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. 2021. RefinedC: automating the foundational verification of C code with refined ownership types. In *PLDI*. ACM, 158–174.

doi:10.1145/3453483.3454036
[39] The Digger team. 2024. Digger. https://github.com/2xs/digger.
[40] The Prosa team. 2024. Prosa. http://prosa.mpi-sws.org/.
[41] Harun Teper, Daniel Kuhse, Mario Günzel, Georg von der Brüggen, Falk Howar, and Jian-Jia Chen. 2024. Thread Carefully: Preventing Starvation in the ROS 2 Multi-Threaded Executor. In *2024 International Conference on Embedded Software*. doi:10.1109/TCAD.2024.3446865
[42] Florian Vanhems, Vlad Rusu, David Nowak, and Gilles Grimaud. 2022. A Formal Correctness Proof for an EDF Scheduler Implementation. In *RTAS*. IEEE, 281–292. doi:10.1109/RTAS54340.2022.00030
[43] Michael Zolda and Raimund Kirner. 2016. Calculating WCET estimates from timed traces. *Real Time Syst.* 52, 1 (2016), 38–87. doi:10.1007/S11241-015-9240-1