PROSA: A Case for Readable Mechanized Schedulability Analysis

Felipe Cerqueira Felix Stutz Björn B. Brandenburg
Max Planck Institute for Software Systems (MPI-SWS)

***Abstract*—Motivated by a string of recent errata, the paper argues that mechanized, yet readable schedulability proofs are desirable, feasible to create with current tools and with reasonable effort, and beneficial beyond the increase in confidence. To facilitate such mechanized analyses, PROSA, a new open-source foundation for formally proven schedulability analyses that prioritizes readability, is presented. The approach is demonstrated with a case study that mechanizes multiprocessor response-time analysis, including new variants for parallel jobs and release jitter.**

I. INTRODUCTION

For a field that prides itself in the development of analysis techniques for safety-critical systems—famously, *hard* real-time systems—there recently has been an alarming number of errata correcting or retracting unsound schedulability analyses.

Examples abound. Most famously, Bril et al. found the original *response-time analysis* (RTA) of the CAN bus [75] to be partially incorrect [23], more than a decade after initial publication, and well after industry adoption and deployment in commercial tools [32]. More recently, Nelissen et al. [67] discovered flaws in an analysis of self-suspending tasks [57], and a subsequent review of the literature on self-suspensions by Chen et al. [28] found many more issues (*e.g.*, [6, 7, 55, 59, 65]). One such flawed analysis [59] had particular impact, as it was reused by a number of other papers (*e.g.*, [21, 26, 52, 54, 79, 80, 82]), thus spreading the issue even further. In 2013, a well-received schedulability analysis of deferred preemptions [33] had to be revised to account for push-through blocking [31]. Shortly after presenting their work at ECRTS'15, von der Brüggen et al. found a mistake in a proof in the appendix of their paper [76], invalidating the originally claimed utilization bound. In fact, even Liu and Layland's seminal paper [62] contained incomplete and incorrect proofs [36]. And finally, to conclude this by no means exhaustive list with an example of particular relevance to us, a recent paper by our group [50] also contained a subtly, but thoroughly flawed analysis due to an incorrect generalization.

A. The Need for Mechanized Proofs

All of the above-mentioned papers were prepared with great care, by experts in the field, and passed peer review, in some cases even repeatedly (*e.g.*, the flaw in [50] is also present in a later journal version [51]). This suggests that these cases are not outliers; rather, they are symptoms of a structural problem.

The common feature of all these partially incorrect results is that they were established with conventional *pen-and-paper* proofs, where the trust in the correctness of a result derives solely from manual, human reasoning (by the authors, reviewers, and readers). However, the long (and still growing [28]) list of errata shows that this approach no longer scales to the complex, difficult, and at times tedious scheduling problems studied in contemporary real-time systems research.

***Note:** This paper has passed an Artifact Evaluation process. For additional details, please refer to http://ecrts.org/artifactevaluation.*

These issues further compound when results from different sources are combined: because pen-and-paper proofs tend to rely on implicit, informal assumptions, the composition of individually correct results may still be incorrect due to subtle differences in assumptions. Furthermore, the situation is not helped by the fact that conference papers often contain only proof sketches or abridged proofs, and that many proofs are given in appendices that may receive only limited attention.

In this paper, we instead argue in favor of a more rigorous approach based on formal, fully verified proofs: to eliminate the chance of human error, *schedulability proofs should be fully mechanized*, *i.e.*, they should be prepared and automatically checked with the help of a proof assistant such as COQ [1].

Of course, mature proof assistants of sufficient analytical power have been available for many years, and have been successfully employed both to formally verify large, complex software, such as compilers [60], OS kernels [56] and file systems [27], and to mechanize highly non-trivial mathematical proofs such as the Four Color Theorem [44] and the Odd Order Theorem [45]. Yet they have not been widely adopted by the real-time systems community (§V surveys prior efforts).

As argued in more detail in §II, we posit that this regrettable lack of adoption can be explained by: **(i)** a general lack of readability and clarity (thus excluding researchers without a background in formal methods); **(ii)** high startup costs (no suitable, sufficiently flexible foundation exists); and **(iii)** the (perceived) large cost of formalizing well-known, common analysis techniques such as fixed-point searches.

To overcome these barriers, we present PROSA [4], a flexible open-source foundation for formally **pro**ven **s**chedulability **a**nalyses that aims at enabling verification *without compromising readability or clarity*, and importantly, *with reasonable effort*.

B. Readable Mechanized Schedulability Analysis is Feasible

PROSA provides a beginner-friendly, readable, yet formally specified foundation for (multiprocessor) schedulability analysis using the COQ proof assistant [1] and the SSREFLECT extension library [2]. As a realistic case study, we mechanized Bertogna and Cirinei's RTA for global *fixed-priority* (FP) and *earliest-deadline first* (EDF) scheduling [17], as well as several novel extensions. Based on our experience, we claim that:

1) PROSA provides a flexible foundation that *covers a large section of existing real-time scheduling theory* (§III),
2) *mechanized schedulability proofs are feasible*, to the point that modern multiprocessor analyses can be formalized in a *reasonable time frame* (§IV),
3) our framework offers a systematic approach for *exploring neighboring results* (§IV-A), and
4) mechanized proofs help in nailing down a *minimal set of assumptions* for each proof and identifying why and when they are required (§IV-B).

Furthermore, while our primary focus is on PROSA itself, this paper also makes several novel contributions as part of the case study: we report on the first mechanized proof of a multiprocessor schedulability analysis (§IV), we present the first global RTA that accounts for release jitter (§IV-A), we present a global RTA for parallel tasks with unknown structure (§IV-B), and finally we identify a fixed-point search strategy for EDF RTA that dominates all others (§IV-C).

II. GOALS & PRINCIPLES

The goal of the PROSA project, and the scope of this paper, is *not* to prove correct any particular schedulability analysis, but to *build reusable foundations* for schedulability analyses that are *verifiably correct, extensible, and still easy to understand*.

Although we propose the use of proof assistants towards this end, we also recognize the potential drawbacks of this approach in terms of legibility, complexity, and required effort. In fact, it is relatively easy to (accidentally) build formal specifications and proofs that are incomprehensible to non-experts, and hence uncheckable, which makes them fundamentally no more trustworthy than traditional pen-and-paper proofs (*i.e.*, the trust rests to some extent in the author, and not just the proof itself).

To some extent, these issues arise because standard practice surrounding proof assistants today seemingly favors expressive and terse notation, complex logics, and advanced programming language techniques and abstractions. This has been the case also for some of the prior efforts to formalize real-time scheduling (reviewed in §V), which to date have not yielded a widely accepted formal basis for schedulability analysis.

PROSA takes a different route and makes *readability* the central goal. We believe that without an approach that involves the community and in which formal proofs are part of a social process [34], any attempt at formal schedulability analysis must fail due to a lack of relevance. We hence target our work explicitly at researchers without prior experience in mechanized proofs and establish the following principles.

A. Readability Is Essential

The correctness of a mechanized proof only goes as far as the correctness of the underlying specification, which must still be manually vetted and accepted as valid by the community at large. Hence, formal definitions and theorem statements should be as comprehensible as their pen-and-paper counterparts.

Therefore, our primary goal is to make the specification accessible to researchers familiar with real-time scheduling who do not necessarily have a background in formal methods. As part of our coding style, we favor the following guidelines.

a) Many lemmas, short proofs: To make proof strategies easy to follow, we split the proof of theorems into many short lemmas (so that each proof spans at most a few dozen lines of code). This makes it possible to understand the proof outline at a high level without diving into low-level proof scripts.

b) Long, verbose names: Our definitions and lemmas have verbose names to make statements self-contained and clear, so that the code can be followed even without comments. For example, the following code, which states that a subsequent proof assumes constrained deadlines, can be readily understood without prior exposure to COQ:

```
Variable ts: taskset_of sporadic_task.
Hypothesis H_constrained_deadlines:
  ∀ tsk, tsk ∈ ts → task_deadline tsk ≤ task_period tsk.
```

c) Heavy use of documentation: We exploit COQ documentation facilities (*e.g.*, *coqdoc*) to make the experience of reading a specification closer to that of reading a paper. Taking the example above, we aim for the following commenting style.

```
  (* Consider any task set ts ... *)
Variable ts: taskset_of sporadic_task.
  (* ... with constrained deadlines. *)
Hypothesis H_constrained_deadlines:
  ∀ tsk, tsk \in ts → task_deadline tsk ≤ task_period tsk.
```

That is, we intersperse paper-like explanations and matching formal statements to provide guidance to the reader. This is especially helpful in the few intermediary proof steps that require advanced COQ notation, as readers may skip the minutiae of the formalism and follow the appeal to intuition instead.

Importantly, such "skipping of detail" does *not* undermine the trustworthiness of the schedulability analysis: as each proof step is strictly checked by COQ, the trust in the validity of the claimed schedulability analysis rests *only* in the understanding of the specification (which we keep intentionally simple and accessible), and *not* in the ability to follow each proof step.

d) Redundancy, if it aids readability: Excessive generality and abstraction can harm readability, as it creates clutter, indirection, and numerous dependencies across many files and definitions. Therefore, in contrast to traditional software engineering guidelines, we favor redundancy and immediacy over reuse and abstraction in our definitions.

For instance, as discussed in §IV-A, we decided to completely isolate the basic model from the model with release jitter, even though the former is just an instantiation of the latter (by defining the upper bound on release jitter to be zero). For readers who are not interested in scheduling overheads and just want to develop a new schedulability analysis, references to jitter in every definition make the code less legible. On the other hand, those readers who are interested in proof details might like to compare the effects of introducing release jitter on the proofs (*e.g.*, how the definition of work conservation changes and whether this invalidates any prior assumptions).

Generally speaking, by favoring redundancy and immediacy over abstraction and terseness, we intentionally favor novice readers (who can focus on the subject matter, *i.e.*, scheduling and questions of timeliness) over experienced authors of specifications and proofs (who are forced to produce more code). This is a deliberate choice in line with the observation that formal proofs are of limited value if they are ignored by the community at large—to be successful, a formal specification must be read much more often than it is written.

B. We Maintain the Established Proof Culture

Instead of radically changing the way that proofs are carried out (*e.g.*, by switching to expressive, yet cryptic to the uninitiated temporal logics [70, 78, 81, 83, 84]), we believe that formal specifications should reuse the common notation and proof style familiar to real-time scheduling experts. Doing so not only

allows existing results to be more easily formalized, but also makes our formalization of scheduling concepts more accessible.

In our specification, we try not to use any complex logics or COQ features that go beyond common mathematical concepts (*e.g.*, we avoid records, inductive types, canonical structures, etc.), other than in a few exceptional cases for simplicity or compatibility with libraries. Rather, as highlighted in §III, we favor first-order logic, lists, functions and Peano arithmetic; basic concepts that are readily familiar to any computer scientist.

C. Some Proofs Are More Important Than Others

Given that the literature on real-time scheduling is vast, we focus our finite resources on those aspects of scheduling theory that are most relevant from a safety perspective. In particular, in schedulability proofs, we prioritize sufficient over necessary conditions and other secondary issues, such as algorithm termination and time complexity.

When deploying safety-critical systems, the scenario that must be prevented at all costs is the use of potentially unsafe analysis. In contrast, the failure modes of other possible issues such as a fixed-point search that fails to converge, accidental pessimism, or an erroneously derived time or memory complexity bound are either readily apparent to the engineer (*e.g.*, the analysis times out or unexpectedly claims a system to be unschedulable) or are of interest only to the academic community. Nevertheless, as briefly discussed in §IV-C, our framework is expressive enough to define and prove some of these additional properties.

In terms of generality, we also aim for a balance between specifying simpler, classical results and directly moving to the most generic scheduler model. First, starting with too restrictive assumptions makes it difficult to incorporate extensions without having to discard the entire codebase. For example, in this day and age, any reasonable formalization should permit reasoning about multiprocessors. On the other hand, it is also futile to try to anticipate every possible use case and come up with completely generic and reusable definitions.

Based on these considerations, we chose response-time analysis on identical multiprocessors [17] as our case study for two reasons. First, the problem is complex enough to include many concepts that are not present in uniprocessor scheduling such as parallelism, which in turn requires extensive background theories on counting, list operations, sums, etc. Second, we specifically seek to target state-of-the-art multiprocessor analysis techniques, which are more complex and not as well-understood as the canon of classic uniprocessor results.

D. The Promise of Provably Correct Schedulability Analysis

To enable peer review, and, we hope, ultimately community involvement, we have made the work described in this paper available as an open-source project [4]. As a long-term goal, we envision a shared repository of formal definitions and proofs for most major results in real-time scheduling, a vision that we believe to hold substantial promise.

In particular, as shared specifications of common concepts such as "sporadic tasks," "constrained deadlines," "identical multiprocessors," *etc.*, are reviewed and become widely accepted, mechanized proofs built on this foundation provide an opportunity for *trustworthy, non-disputable results*. By increasing the level of confidence in schedulability analysis, it is our hope that a formal foundation will eventually help in convincing practitioners to more readily adopt modern real-time resource management approaches, especially in the context of safety-critical systems subject to certification requirements.

Having made the case that formally verified, yet readable schedulability analysis is highly desirable, we dedicate the rest of this paper to showing that it is possible to realize this vision—with simple techniques, current tools, and reasonable effort.

III. A SPECIFICATION FOR SCHEDULABILITY ANALYSIS

To develop mechanized schedulability analyses, we first must precisely define all relevant real-time scheduling concepts. Such a formalization, however, is essentially arbitrary—there are many, fundamentally equivalent ways to model "real-time scheduling." Hence, the choice of specification is a matter of design, tradeoffs, and ultimately taste.

On the one hand, a good specification should be *powerful* and flexible: an overly simplistic or artificially constraining model will limit the kind of proofs that can be carried out with reasonable effort. Ideally, the adopted specification should cover as much of the existing real-time literature as possible.

On the other hand, it is desirable for the specification to be as *simple* as possible: all trust hinges on the specification; it should hence be as readable as possible (as argued in §II) and facilitate manual inspection. Furthermore, it shapes all subsequent proofs—a convoluted or heavily abstracted specification will multiply the effort required to prove anything but "toy results" and create avoidable barriers to adoption.

Based on these considerations, we drafted, experimented with, and iteratively refined several specifications before settling on the one presented herein. We believe that our final choice of specification is sufficiently powerful and flexible to support a large part of real-time scheduling theory, while also advancing a straightforward style that (we hope) will feel natural and elegant to real-time systems researchers.

In the following, we show key excerpts of our actual COQ development to convey the feel and design of PROSA. While showing each definition, we discuss the steps taken to build foundations that are both accessible and similar in style to seminal works in the area of real-time scheduling (Principles A and B in §II). Although in this initial work we restrict our platform model to identical multiprocessors (formalized in §III-F), a large part of our specification is already sufficiently generic to incorporate several extensions. Thus, whenever applicable, we also comment on how more general models found in the literature could be incorporated.

Before going into further detail, we begin with a short tutorial on the syntax of the proof assistant used in this project.

A. Background on COQ and SSREFLECT

Our specification and proofs have been developed with the COQ proof assistant [1] and the SSREFLECT extension [2].

COQ is a widely used proof assistant developed at INRIA that offers a higher-order logic with support for computations, which allows the user to define algorithms (as functional programs) and to prove facts about them. In the context of PROSA, this feature

is especially useful to define procedures such as iterative fixed-point searches. In future work, it could also be used to formalize task partitioning strategies or synchronization protocols and their associated blocking analyses, *etc.*

Coq specifications are organized into sections (which in turn are grouped into modules). Sections allow variables and lemmas to be defined, and hypotheses (*i.e.*, assumptions) to be explicitly stated. For example, in the sample code in Specification 1, we let x and y denote any natural numbers greater than one. Then, we prove a trivial lemma stating that $x \cdot y \geq x + y$.

Specification 1 Variables, hypotheses, and lemmas.

```
Section TrivialLemma.
  (* Given positive numbers x and y... *)
  Variable x y: nat.
  (* ... that are larger than 1, ... *)
  Hypothesis H_x_y_gt_one: x > 1 ∧ y > 1.

  (* ... we prove that x * y >= x + y. *)
  Lemma prod_ge_sum : x * y ≥ x + y.
  Proof.
  (* Omitted from paper for brevity. *)
  Qed.
End TrivialLemma.
```

Since low-level proof steps are of limited significance to the readers of a proof, we only show lemma statements in this paper. To reiterate, all trust rests solely in the validity of the specification and the Coq type checker, as any Coq code passes type-checking only if *every proof is correct* up to the axioms of the logic. As in other modern proof assistants, bugs in Coq's type checker itself are highly unlikely due to the use of a small certification kernel (see [68] for an introduction).

In Coq, functions can be defined by directly specifying their parameters, or more verbosely using sections, as shown in Specification 2 for the functions `double` and `S`, respectively. We tend to use the latter method to intersperse comments.

Specification 2 Defining functions.

```
(* Let double be the function f(x)=2*x. *)
Definition double (x: nat) := 2 * x.

Section VerboseDefinition.
  (* Given any natural number x, ... *)
  Variable x: nat.
  (* ... let S be the sum of all natural
numbers less than x. *)
  Definition S := \sum_(y < x) y.

End VerboseDefinition.
```

The concise and elegant, LaTeX-like notation "$\backslash sum$" for summation stems from SSReflect, an extension library for Coq provided by the *Mathematical Components* project [2], which provides support for large-scale mathematical proofs. Most famously, it has enabled mechanized proofs of the Four Color Theorem [44] and the Odd Order Theorem [45].

Finally, when applying functions with multiple parameters, Coq follows the standard ML syntax, *e.g.*, a function application $f(a, b, g(x))$ is written as `f a b (g x)`.

With the essential, straightforward Coq notation in place, we are now ready to discuss our formalization of real-time scheduling and the underlying design choices. We begin with its most fundamental concept: *time*.

B. Time Representation

For the sake of simplicity, we adopt a discrete time representation, where time is just an alias to natural numbers:

```
Definition time := nat.
```

There are several reasons for this choice. First, if time is discrete, properties that depend on the progression of job arrivals can be proven easily by induction on time. For example, by applying strong induction on time, one can obtain useful induction hypotheses such as "all jobs that completed before time t did not miss their deadline" and then analyze the finite schedule up to time t. This technique was used in all of our proofs of response-time bounds (§IV).

Second, apart from simplifying computations and proofs, this definition also fits actual scheduler implementations in practice, where all relevant times are necessarily defined by the processor's native cycle length (or lengths, in case of processors capable of running at multiple speeds).

Third, in terms of analytical power, assuming discrete time does not cause any major drawbacks: it has been shown for both uniprocessor [14] and multiprocessor [20] scheduling that, if arrival times are integer, any task set is feasible assuming dense time iff it is feasible assuming discrete time.

Finally, Coq and SSReflect provide better support for natural numbers, but representations of dense time could be explored in the future using a library for real numbers [19].

C. Jobs and Arrival Sequences

Next, we define the universe of all jobs, *i.e.*, the basic entities that can be scheduled:

```
Variable Job: eqType.
```

This means we simply assume that there is an opaque type `Job` that belongs to the class `eqType`, which corresponds to any type that supports equality. Being a completely generic definition, it does not impose *any constraints* on what a job is. For instance, it does not imply the lack of self-suspensions or critical sections. In fact, jobs can have as many properties as required, which however must be stated and defined separately.

We consider the need to state all assumptions explicitly, rather than defining a more high-level concept such as "sporadic task" as the basic concept, to be a feature. For one, it allows us to reason about mixed schedules of jobs of recurrent tasks and aperiodic jobs (that do not belong to any task). Furthermore, the need to make all assumptions explicit helps us to identify a minimal set of assumptions required to prove a given claim, as we discuss in more detail in §IV-B.

In order to reason about a particular collection of jobs, we define the notion of an *arrival sequence*, which is a mapping from time to finite sequences of jobs released at a given time:

```
Definition arrival_sequence := time → seq Job.
```

That is, an arrival sequence defines, for any time t, a (possibly empty, but finite) collection of arriving jobs. This definition covers both finite and infinite job sets. It does not, however, impose any arrival constraints, which must be introduced explicitly as additional hypotheses. For instance, it is possible to specify models such as periodic and sporadic tasks (see §III-H), generalized event streams [47, 69], *etc.*

Because time is discrete, and since the type "seq *Job*" describes *finite* job collections, this definition implies that the set of jobs forming an arrival sequence is countable. This makes it trivial to enumerate all jobs that arrive in a finite interval, which is particularly useful when defining functions that bound interference (see §III-J), load, and demand [13, 14].

Given an arrival sequence arr_seq, we define a special type called `JobIn`, which is simply a `Job` that is known to be a part of arr_seq. For such a job, $job\_arrival$ denotes its arrival time, which allows stating whether a job has arrived at time t:

```
Variable j: JobIn arr_seq.
Definition has_arrived (t: time) := job_arrival j ≤ t.
```

In summary, jobs are the basic, atomic notion of a "schedulable entity," and an arrival sequence captures the notion of a countable set of jobs with associated release times that are to be scheduled. We next formalize the notion of such a schedule.

D. Multiprocessor Schedule

Given the number of processors and an arrival sequence,

```
Variable num_cpus: nat,
Variable arr_seq: arrival_sequence Job,
```

a schedule for this arrival sequence is a mapping, for each processor and each point in time, to possibly a job (or none, in case the processor is idle). In Coq, this can be formalized as:

```
Definition schedule :=
  processor num_cpus → time → option (JobIn arr_seq),
```

where $processor\ num\_cpus$ denotes any natural number in the interval $[0, num\_cpus)$. The *option* type, in turn, is the standard way of encoding partial functions in Coq. Any variable of type $option\ T$ is either $Some\ x$, where x is of type T, or $None$. Hence, a schedule function either yields some job in the arrival sequence, or nothing to indicate idleness.

Based upon this basic mapping, we define a number of simple predicates to state whether a job is scheduled at all, on a specific processor, *etc.* For example, for a given schedule

```
Variable sched := schedule num_cpus arr_seq,
```

we let $scheduled\_on$ denote the predicate that, in the schedule $sched$, job j is assigned to processor cpu at time t:

```
Definition scheduled_on j cpu t :=
  sched cpu t == Some j.
```

For the sake of generality, our definition of a schedule only maps processors to jobs and carries no additional semantics. In particular, it does not imply job sequentiality (*i.e.*, multiple processors could schedule the same job), which allows formalizing parallel task models [53, 58]. It does, however, imply sequential processors (*i.e.*, at any time, at most one job per processor). For proofs that use the notion of a fluid schedule (*e.g.*, [15]), an appropriate definition can be easily added.

To express additional properties of a schedule, relevant hypotheses must be stated explicitly, as shown next.

E. Adding Schedule Constraints

Starting with a generic schedule maximizes flexibility and makes all assumptions explicit. By introducing additional hypotheses, we can precisely express when a job is allowed to execute, according to the underlying task and platform models. For example, given a schedule $sched$, we can *selectively* impose that jobs do not execute before their arrival:

```
Definition jobs_must_arrive_to_execute :=
  ∀ j ∀ t, scheduled sched j t → has_arrived j t,
```

where $scheduled\ sched\ j\ t$ denotes whether job j is assigned to any processor at time t. While it may at first seem surprising to require such a near-universal assumption to be stated explicitly, this constraint is in fact not appropriate when analyzing schedules subject to *early-releasing* (*e.g.*, [5, 63]). Making all assumptions explicit avoids any accidental loss of generality.

Similar constraints can be used to arbitrarily restrict the behavior of the schedule (*e.g.*, to impose migration models such as partitioned, clustered [9, 25], semi-partitioned [41], and APA scheduling [50], or restricted migrations [11]). For instance, dedicated interrupt handling [22, 72] prohibits jobs from executing on a processor reserved for interrupts. With Coq's syntax for logical negation (∼), we can easily state that no job is ever scheduled on some processor cpu_int:

```
Hypothesis H_cpu_reserved :
  ∀ j ∀ t, ∼ scheduled_on j cpu_int t.
```

In addition to being simple, such fine-grained schedule constraints are also highly composable and simplify the process of extending the specification, as further discussed in §IV-A.

F. Job Cost and Job Completion

Each job has some finite cost, which in real-time terminology denotes the job's *actual execution time* (ACET):

```
Variable job_cost: Job → time.
```

The execution requirement of a job is satisfied as it receives service. The instantaneous service received by a job j at time t is defined as the number of processors on which j is scheduled:

```
Definition service_at (t: time) :=
  \sum_ (cpu < num_cpus | scheduled_on j cpu t) 1.
```

Thanks to the LaTeX-like operators provided by SSReflect, instantaneous service is defined as a straightforward sum over the set of processors, quite literally matching its intuitive definition. Note that it transparently accounts for service received by a job in parallel on multiple processors (if any).

For the sake of simplicity, we focus on identical multiprocessors. However, uniform multiprocessors could be easily supported by using processor speeds as a factor in the definition of service (*e.g.*, Funk et al. [42] provide a suitable definition).

Next, to measure the progress in job j's execution, we define a notion of cumulative service received before time t':

```
Definition service (t': time) :=
  \sum_ (0 ≤ t < t') service_at t.
```

Based on the cumulative service received, we can impose another basic constraint on valid schedules:

```
Definition completed_jobs_dont_execute :=
  ∀ j ∀ t, service sched j t ≤ job_cost j.
```

Finally, we say that job j has completed by time t iff its cumulative service matches its cost:

```
Definition completed (t: time) :=
  service sched j t == job_cost j.
```

The constraints that jobs must arrive to execute, and that completed jobs do not execute, constitute the two most basic assumptions about a schedule. However, these two properties are trivially satisfied by an empty schedule (*i.e.*, one in which all cores are always idle), and hence are insufficient by themselves to prove meaningful bounds. In order to encode progress guarantees, we must define new constraints.

G. Work-Conserving Schedules

Central to any schedulability analysis is the notion of a "pending job," *i.e.*, any job j that has arrived but not yet finished:

```
Definition pending (t: time) :=
  has_arrived j t && ~ completed j t.
```

Because timeliness can only be proven for schedules in which pending jobs are eventually scheduled, actual schedulers are implemented to be *work-conserving*. Based on the notion of a *backlogged* job that is pending and not scheduled,

```
Definition backlogged (t: time) :=
  pending j t && ~ scheduled j t,
```

we say that a schedule is work-conserving iff, for any job j in an arrival sequence, at any time t, if j is backlogged, then every processor is busy scheduling some other job:

```
Definition work_conserving :=
  ∀ j, ∀t,
    backlogged job_cost sched j t →
    ∀ cpu, ∃ j_other,
      scheduled_on sched j_other cpu t.
```

Coq's type inference automatically determines that j_other must be a job in the *same* arrival sequence, based on the definition of *schedule* and the domain of the function *scheduled_on*, which highlights the benefit of using the special type `JobIn`.

Basic work conservation is only a weak progress guarantee. In a real-time context, we typically require also that schedules reflect job priorities: when a job is backlogged, not only should all processors be non-idle, but they should also execute jobs of at least equal priority. This expectation can be easily added akin to the above definition of work conservation.

However, it bears pointing out that any notion of expected progress is closely tied to the assumed platform and workload models. For example, when introducing self-suspensions, a relaxed definition of work conservation is required since suspended tasks should not count as backlogged, and when introducing job placement constraints (*e.g.*, partitioned scheduling or scheduling with arbitrary processor affinities [50]), the set of relevant processors with respect to each backlogged job changes. Similarly, introducing release jitter (see §IV-A) affects the definition of when a job is considered pending.

Nonetheless, all of these refinements are trivial to express with definitions just as simple as those shown here. In fact, taking a closer look at all definitions shown to this point, note that most are straightforward and do not use involved notation. The definition of *pending*, for example, closely resembles its actual meaning. And even in more complex definitions such as *work_conserving*, verbosity and clear syntax help in clarifying each statement (see Principle A in §II).

Next, to specify major results in real-time scheduling, we need a notion of recurring tasks.

H. Tasks, Task Sets, and the Sporadic Task Model

Similar to `Job`, a task is simply a generic, opaque type:

```
Variable sporadic_task: eqType.
```

In the sporadic task model [66], tasks are characterized by three parameters: *worst-case execution time* (WCET), *period* (*i.e.*, the minimum job inter-arrival time) and *relative deadline*:

```
Variable task_cost: sporadic_task → time.
Variable task_period: sporadic_task → time.
Variable task_deadline: sporadic_task → time.
```

When formalizing refined task models, additional parameters can be included as well (*e.g.*, in §IV-A we define the worst-case release jitter of a task).

In our specification, tasks are not scheduled directly. They only represent restrictions on certain sets of jobs (*e.g.*, they determine job inter-arrival times and maximum execution costs). Since jobs are the actual entities assigned to processors, it is possible to define schedules with jobs that do not belong to any task (and thus formalize aperiodic jobs, *e.g.*, [12, 71]). However, since our case study pertains to sporadic tasks only, for now we associate every job with a task:

```
Variable job_task: Job → sporadic_task.
```

This mapping allows enforcing constraints on job parameters. For example, a key constraint states that the ACET of any job j cannot be larger than the WCET of the corresponding task:

```
Definition job_cost_le_task_cost :=
  job_cost j ≤ task_cost (job_task j).
```

Similarly, for any two distinct jobs j_1 and j_2 of the same sporadic task, if j_1 arrives before j_2, then their arrival times are separated by at least one period of the task:

```
Definition sporadic_task_model :=
  ∀ (j1 j2: JobIn arr_seq),
    j1 ≠ j2 →
    job_task j1 = job_task j2 →
    job_arrival j1 ≤ job_arrival j2 →
    job_arrival j2 ≥ job_arrival j1 +
                     task_period (job_task j1).
```

Finally, tasks are grouped into (finite) task sets:

```
Definition taskset_of (Task: eqType) := seq Task.
```

By referring to task sets, it is easy to impose workload restrictions. For example, we can state that a task set ts has constrained deadlines simply as follows:

```
Variable ts: taskset_of sporadic_task.
Definition constrained_deadline_model :=
  ∀ tsk, tsk ∈ ts → task_deadline tsk ≤ task_period tsk.
```

In summary, by mapping jobs to tasks and by adding just a few more constraints, we defined the concept of sporadic tasks, without compromising clarity or readability, which speaks to the flexibility and extensibility of our basic model. More flexible workload models such as generalized event streams [69] or arrival curves [74] could be incorporated just as easily.

Next, we demonstrate how more complex concepts such as response-time bounds and interference can be defined.

I. Response-Time Bounds

A task's response-time bound determines how long any of its jobs remains pending. We say that R is a response-time bound of a given task tsk in a particular schedule iff every job of tsk completes within R time units of its arrival:

```
Variable R: time.
Definition is_response_time_bound_of_task :=
  ∀ (j: JobIn arr_seq),
    job_task j = tsk →
    job_has_completed_by j (job_arrival j + R),
```

where $job\_has\_completed\_by$ is just an alias of $completed$ for the sake of readability.[1] This definition already allows stating the correctness condition of response-time analyses. To prove such a result, however, we must reason about interference.

J. Job Interference

In a given time interval $[t_1, t_2)$, the total interference incurred by job j is the cumulative time during which it is backlogged:

```
Definition total_interference (t1 t2: time) :=
  \sum_(t1 ≤ t < t2) job_is_backlogged j t,
```

where $job\_is\_backlogged$ is an alias of $backlogged$. As it is the case with service (defined in §III-F), interference is a straightforward sum, which ensures readability. To bound the interference caused by a specific job, we also define per-job interference. Given some interfering job j_other,

```
Variable j_other: JobIn arr_seq,
```

the interference incurred by job j due to job j_other during the interval $[t1, t2)$ is defined as follows:

```
Definition job_interference (t1 t2: time) :=
  \sum_(t1 ≤ t < t2)
    \sum_(cpu < num_cpus)
      (job_is_backlogged j t &&
        scheduled_on sched j_other cpu t).
```

All processors are considered in case that jobs are parallel.

To conclude this section, however, one important question remains: having defined so many hypotheses, how do we ensure that they are actually contradiction-free?

[1] In the actual Coq source, the alias aids readability since it not only changes the name, but also binds some local parameters, which is not apparent here.

K. How to Avoid Contradictory Assumptions?

Recall that to capture the properties of a particular type of schedule (*e.g.*, "an EDF schedule"), we must explicitly declare all relevant hypotheses, which are then used in the proof of a particular schedulability analysis. However, as emphasized in §II-A, the correctness of formal proofs only goes as far as the validity of the specification—in particular, from contradictory assumptions, *anything* may be proven "correct."

In general, Coq cannot automatically detect contradictory assumptions. Hence, to ensure that our proofs are meaningful, not only must all assumptions be checked to be appropriate (*e.g.*, a proof about EDF schedules should not state assumptions on fixed priorities), they must also not contradict each other.

Given the granularity of assumptions in our specification, this would be a daunting, error-prone task if carried out manually. We instead address both issues by exploiting Coq's ability to implement functional programs within the logic framework.

Specifically, after proving that a schedulability analysis is sound based on a (large) number of assumptions, we implement a concrete model scheduler (*e.g.*, an EDF simulator) and prove that it generates schedules that actually satisfy *all* assumptions stated in the specification. More precisely, we instantiate the analysis for the model scheduler, so that Coq ensures that no assumptions are being overlooked.

Since absolutely no hypotheses are used in any of the proofs about the algorithm, proving that our model scheduler ensures all assumed hypotheses implies both that they are contradiction-free, and that there is no accidental mix-up of different models.

As a final remark, we note that the definitions presented in this section provide just a glimpse of our specification (albeit one carefully chosen to convey the key points). We have formalized many more concepts, such as deadline misses and schedulability, task precedence constraints, and the classification of fixed and dynamic priorities. In total, our foundations and simple lemmas span about 1900 LOC for the basic scheduler model and 930 LOC to introduce release jitter.

Concepts that we have not yet formalized can, in our experience, typically be defined in just a few lines of straightforward code, given the expressiveness of notations in Coq and the extensibility of our existing specification. Therefore, we conclude that *the developed foundations are sufficiently flexible and powerful to support a large fraction of the existing literature on real-time scheduling, without compromising readability* (**Claim 1**).

We next report on a case study showing how the developed foundations facilitate actual proofs of published analyses.

IV. Case Study: From Definitions to Proofs

An elegant specification is ultimately of little use if it does not lend itself to proofs of correctness. To ensure our specification is indeed practical, we developed several mechanized proofs with it (available in full online [4]):

1) correctness of the workload-based interference bound for work-conserving schedulers [16, p. 158, Eq. (17.3)] (∼600 LOC) and the EDF-specific interference bound [16, p. 161, Eq. (17.5)] (∼890 LOC);
2) definition and proofs of termination and correctness of Bertogna and Cirinei's RTA for FP scheduling [16, p. 167, Eq. (18.4)] (∼1050 LOC);

3) definition and proofs of termination and correctness of Bertogna and Cirinei's RTA for EDF [16, p. 160, Fig. 17.3] (∼1320 LOC);
4) implementation of a work-conserving JLFP scheduler to validate all stated assumptions (∼560 LOC);
5) extension of definitions and proofs for workloads with release jitter (∼5620 LOC); and
6) extension of definitions and proofs for workloads with parallel jobs (∼3030 LOC).

We chose Bertogna and Cirinei's multiprocessor RTA as the starting point for our case study not because of any particular doubts, but because it is an influential, recent *multiprocessor* result of manageable, but non-trivial scope. As RTA is based on a fixed-point search (as opposed to a simple inequality as in the density test [46]), it also provided an opportunity to explore proofs for this important, but more challenging technique.

An important outcome of this project is that, trusting the correctness of the proof assistant, and the validity of our specification (§III), *all above results are correct* and not at risk of lingering human errors necessitating later correction.

To date, we have invested roughly 8 person-months in this project, of which a large fraction was spent on learning how to use Coq and SSReflect, and in deriving a reasonable specification, which we rewrote several times from scratch in order to improve readability and to simplify proofs before finally arriving at the formalization presented in §III.

Given that future efforts can reuse and extend the foundations that we built, we conclude that, after some up-front time investment to learn the basics of Coq and SSReflect, *mechanized schedulability proofs are feasible, to the point that non-trivial multiprocessor schedulability analyses can be formalized in a reasonable time frame* **(Claim 2)**.

Nevertheless, end-to-end correctness is not the only advantage of mechanization. In the following, we highlight benefits and lessons that we discovered throughout the process.

A. Mechanization Allows Exploration of Neighboring Results

When it comes to extending specifications, proof assistants provide substantial advantages. Differently from pen-and-paper proofs, where any changes in assumptions are non-systematic and error-prone, a proof assistant automatically tracks assumptions, flags any proofs that are invalidated, and hence allows for an easy and *confident* exploration of neighboring results. To support this claim with a case study, we extended Bertogna and Cirinei's overhead-oblivious RTA to incorporate release jitter.

Although a jitter-aware RTA was proposed by Audsley et al. for uniprocessors more than 20 years ago [8], to the best of our knowledge, release jitter has not been considered in *any* (published) global schedulability analysis to date. While jitter was understandably omitted from the initial exploration of global scheduling, the task of incorporating jitter still has not attracted the attention of any publication, perhaps due to the restricted scope of the problem from a research point of view.

However, this places anyone seeking to apply RTA to actual systems in an extremely difficult situation: release jitter, especially in networked systems, is decidedly non-negligible in practice, but simply "hoping" that Audsley et al.'s classic uniprocessor result translates *without changes* to the multiprocessor case, more than two decades after the original publication, is risky at best, and would be highly irresponsible in the context of any critical system. Case in point, several recent corrections (*e.g.*, see [18, 28, 67]) surrounding the response-time analysis of self-suspending tasks on uniprocessors can be traced back precisely to an incorrect generalization [55, 65] of Audsley et al.'s release jitter theorem (*i.e.*, release jitter and self-suspension time are *not* interchangeable [18, 28]).

Fortunately, with a proof assistant, localized changes in assumptions can be easily incorporated and, once all now-failing proofs have been fixed, yield *guaranteed-correct* extensions of idealized results. We demonstrate this with the following case study, which introduces release jitter into our specification.

As discussed in §III, our specification favors generic, assumption-free definitions coupled with fine-grained hypotheses. Although having to assume the trivial fact that "jobs must arrive to execute" might initially seem counterproductive,

```
Definition jobs_must_arrive_to_execute :=
  ∀ j ∀ t, scheduled sched j t → has_arrived j t,
```

this choice is deliberate to simplify scheduler extensions. For example, by assuming that jobs incur initial release jitter,

```
Variable job_jitter: Job → time,
```

one can easily state whether the jitter has already passed for a given job j at time t:

```
Definition jitter_has_passed (t: time) :=
  job_arrival j + job_jitter j ≤ t.
```

In order to incorporate release jitter into the model, apart from changing the definition of pending job,

```
Definition pending (t: time) :=
  jitter_has_passed t && ∼ completed t,
```

we constrain jobs to execute only after the jitter has passed:

```
Definition jobs_execute_after_jitter :=
  ∀ j t, scheduled sched j t → jitter_has_passed j t.
```

Note that changing *only five lines* in the specification suffices to formalize the concept of release jitter. This reiterates the point that mechanized schedulability analysis does not have to compromise simplicity and readability (§II-A).

But even more important is that, after these changes, every proof fails *exactly* at the steps that depended on $jobs\_must\_arrive\_to\_execute$. Such a starting point for correcting existing proofs provides a systematic way of expanding the formalization to more realistic (and thus more complex and tedious) models, while maintaining end-to-end correctness.

Using this approach, we were able to incorporate release jitter into Bertogna and Cirinei's response-time analysis for FP and EDF scheduling with just a few days of work, yielding response-time bounds similar to the uniprocessor case. To illustrate the result, we state the theorem for FP scheduling. In the following, given a sporadic task T_i, let e_i be T_i's WCET, let p_i be T_i's period, and let d_i be T_i's relative deadline.

Theorem 1. *Let τ denote a set of sporadic constrained-deadline tasks scheduled on m processors. For any task $T_i \in \tau$, let hp_i denote the set of tasks with priority higher than T_i, let J_i be*

T_i's worst-case jitter, and let R_i denote a fixed point (if any) of the recurrence

$$R_i = e_i + \frac{1}{m} \cdot \sum_{T_k \in hp_i} \min\left(W_k'(R_i), \quad R_i - e_i + 1\right), \qquad (1)$$

where $W_k{}'(\Delta) = \eta_k{}'(\Delta) \cdot e_k + \min(e_k, (\Delta + R_k + J_k - e_k) \bmod p_k)$ and $\eta_k{}'(\Delta) = \lfloor (\Delta + R_k + J_k - e_k)/p_k \rfloor$. Under global FP scheduling, if $R_i + J_i \leq d_i$ and $R_k + J_k \leq d_k$ for every higher-priority task $T_k \in hp_i$, then T_i's maximum response time is bounded by $R_i + J_i$.

Proof. Omitted; formally verified in PROSA [3, result J5]. □

Given the short time required to prove the aforementioned bounds, we conclude that *our framework offers a simple and systematic approach for exploring neighboring results* **(Claim 3)**.

Most importantly, such extensions can be carried out with high confidence *without* having to go through the classic academic peer-review process. This makes extension efforts faster (no waiting for reviews), independent of community appeal (whether an extension is "worthy" of a paper is irrelevant), and actually trustworthy (confidence in the extension's correctness does not derive from human review of the proofs).

Next, we highlight another benefit: minimizing assumptions is easy as they are explicit and tracked by the proof assistant.

B. Nailing Down a Minimal Set of Assumptions

One of the basic assumptions in Bertogna and Cirinei's response-time analysis [17] is the fact that jobs are sequential:

```
Definition sequential_jobs :=
  ∀ j ∀ t ∀ cpu1 ∀ cpu2,
    scheduled_on sched j_other cpu1 t →
    scheduled_on sched j_other cpu2 t →
    cpu1 = cpu2.
```

Although (or perhaps because) this is a common assumption in the literature, it often remains unclear from a pen-and-paper-proof why exactly it is required—in fact, Bertogna and Cirinei's original proof does not even mention this standard assumption [17]. We decided to investigate precisely how the analysis is affected by job parallelism.

Since COQ tracks all hypotheses, this turned out to be trivial: only a few parts of the proof actually depend on *sequential_jobs* as a hypothesis. In particular, we discovered that the original workload and interference bounds no longer hold because parallel jobs can receive more than one unit of service at a time, which implies that interval lengths cannot be used to bound an interfering job's cumulative processor use.

However, we also found that the main proof remains intact, which allowed us to obtain the following bound for parallel jobs *with unknown structure* (*i.e.*, no assumption is made on the degree of parallelism actually present in each job).

Theorem 2. *Let τ denote a set of sporadic, **potentially parallel**, constrained-deadline tasks scheduled on m processors. For any task $T_i \in \tau$, let hp_i denote the set of tasks with priority higher than T_i, and let R_i denote a fixed point (if any) of the recurrence*

$$R_i = e_i + \frac{1}{m} \cdot \sum_{T_k \in hp_i} I_k(R_i),$$

where $I_k(\Delta) = \lceil (\Delta + R_k)/p_k \rceil \cdot e_k$. Under global FP scheduling, if $R_i \leq d_i$ and $R_k \leq d_k$ for every task $T_k \in hp_i$, then T_i's maximum response time is bounded by R_i.

Proof. Omitted; formally verified in PROSA [3, result P5]. □

Although less-pessimistic analysis for parallel jobs is clearly possible if each task's structure is known [61, 64], this bound may still be useful if such information is not known at analysis time (*e.g.*, in reservation-based environments where the "WCET" is actually an enforced budget). In any case, this example demonstrates how mechanized analysis improves upon current practice both by identifying precisely where an assumption is needed, and by making it easy to relax assumptions.

To summarize, we conclude that mechanized proofs aid in nailing down a *minimal set of assumptions* **(Claim 4)**.

Finally, the clarity of formal proofs yields additional insights.

C. Separating Correctness Proofs from Termination Proofs

Baruah et al. describe two strategies for performing EDF RTA [16], which differ on how slack bounds[2] are updated. Before starting the fixed-point iteration, *Strategy 1* [16, p. 160, Fig. 17.2] assumes the initial slack bounds to be zero (*i.e.*, a safe, but pessimistic assumption). On the other hand, *Strategy 2* [16, p. 160, Fig. 17.3] begins the fixed-point iteration with task costs as the initial response-time bounds, yielding initial slack bounds of $d_i - e_i$ time units for each task $T_i \in \tau$, which is clearly not a consistent initial state in the general case. Despite the differences in the two strategies, Baruah et al. do not discuss how this affects the correctness of the procedure.

In our proof of Bertogna and Cirinei's response-time analysis, we clearly separated correctness and termination criteria, which allowed us to show that the correctness of the response-time analysis does not depend on how the fixed point is computed. Because intermediate states of the computation can be safely ignored, both *Strategy 1* and *Strategy 2* yield correct results.

Moreover, by adapting lemmas about fixed points from the formalization of regular languages by Doczkal et al. [37], we were able to prove in PROSA that *Strategy 2* always yields the least fixed point [3, result B11] and thus *dominates any other strategy*, thereby obsoleting *Strategy 1*. To the best of our knowledge, this was previously unknown.

To conclude, these examples highlight that the effort invested into defining schedulability analysis more precisely can yield dividends on top of the main goal of guaranteed correctness.

V. RELATED WORK

This work is not the first attempt to advocate a more formal approach to schedulability analysis; however, to our knowledge, it is the first work to emphasize readability, in particular making the specification accessible to non-experts, and the first work to investigate mechanized proofs for multiprocessor real-time scheduling. In the following, we discuss the most relevant prior approaches and how they differ in comparison with our work.

Formalisms for schedulability analysis. One of the earliest attempts to formalize schedulability analysis dates back to

[2]Slack denotes the gap between a task's response-time bound and its deadline, which can be exploited to reduce the pessimism when bounding interference.

work by Yuhua and Chaochen, with a proof of EDF optimality on uniprocessors based on an interval logic called *Duration Calculus* (DC) [81]. Using a similar approach, Shuzhen et al. proved the sufficient schedulability condition of the Rate Monotonic (RM) scheduler [70]. Later, the proof for EDF was revised for clarity by Zhan [83]. More recently, Xu and Zhan derived simpler DC proofs of RM and EDF schedulability and published a comprehensive review [78].

Although these approaches share a motivation similar to ours and make use of formalism to reduce ambiguity, they rely on complex logics and manual proofs. This combination is inherently limited in terms of readability and correctness (see Principles A and D in §II), which however could be partly fixed with a COQ library for DC [29].

Earlier mechanized proofs. The effort of mechanizing schedulability analysis was started by Wilding, who developed a proof of EDF optimality on uniprocessors using the early Nqthm theorem prover (3 person-months) [77].

Dutertre [39] proved correct the uniprocessor Priority Ceiling Protocol and the corresponding schedulability analysis using the PVS proof assistant (∼2.5k LOC, 3 person-months). In a follow-up work, Dutertre and Stavridou [38] discuss the design decisions in the prior project and the importance of formal schedulability proofs. Recently, Zhang et al. proved the correctness of the blocking bound for the Priority Inheritance Protocol with the Isabelle/HOL proof assistant [85]. Using COQ, De Rauglaudre proved a schedulability condition for periodic tasks based on their phases and hyperperiod [35] (1.2k+ LOC). Most recently, Zhang et al. implemented EDF in a verification language based on *Propositional Projection Temporal Logic* (PPTL), and also provided an optimality proof using a COQ-based PPTL axiomatization [84].

In contrast to our work, none of the mentioned approaches considers readability as the primary goal (Principle A in §II). In some cases, readability is impaired either by the choice of tools available at the time (as in the Nqthm approach), by the complexity of the logic (in case of PPTL), or simply because it was not a priority and standard practice favors terseness.

Contrary to Principle C in §II, previous work focused only on restricted scheduling scenarios (*e.g.*, uniprocessor scheduling with periodic job arrivals). To the best of our knowledge, our work is the first to formalize a recent, highly influential multiprocessor schedulability analysis under the sporadic task model and to cover both FP and EDF policies with extensions (∼16kLOC, ∼8 person-months). Moreover, thanks to the syntax and lemmas from SSREFLECT, our formalization more closely resembles the definitions and proof style found in the real-time scheduling literature (Principle B in §II).

Schedulability analysis based on model checking. An alternative approach for provably correct schedulability analysis is the use of model checking techniques, which provide automated procedures for verifying (also temporal) properties.

In earlier work, Fersman et al. used timed automata to analyze real-time tasks under uniprocessor FP scheduling [40]. Later, Guan et al. proposed schedulability analysis for multiprocessor FP scheduling with periodic tasks, based on the NuSMV model checker [48], with a follow-up work that also covers EDF scheduling [49]. Also assuming periodic tasks, Cordovilla et al. proposed a timed-automata-based schedulability analysis using UPPAAL, applicable to static and dynamic priorities [30].

The periodicity assumption was first relaxed by Baker and Cirinei, who modeled the sporadic schedulability problem as a finite automaton reachability problem [10]. Geeraerts et al. later improved this approach with antichain techniques [43].

Bonifaci and Marchetti-Spaccamela proposed an exact schedulability analysis for global FP scheduling of sporadic tasks based on an exhaustive state-space exploration [20], which was intended primarily as a proof of concept and not as a practical analysis. This approach was later improved by Burmyakov et al. with state pruning techniques to reduce the search space [24].

More recently, Sun and Lipari used linear hybrid automata and pre-order simulation relations to model the schedulability problem using symbolic constraints instead of explicit discrete variables [73]. Differently from previous work, this approach does not artificially restrict task parameters to small integers (*e.g.*, task periods in the interval $[3, 10]$ combined with a low min/max period ratio as in Burmyakov et al.'s approach [24]).

Although we believe that model checking and timed automata are valuable techniques, all mentioned approaches suffer from state-space explosion issues and do not scale beyond small task and processor counts under more complex task models (*e.g.*, in case of sporadic tasks, no more than 10 tasks or 4 processors [24, 73]). Unless there is a major breakthrough in the field of model checking, we expect conventional schedulability analyses to remain a standard approach. Consequently, we do not see the need for conventional, but formally proven schedulability analysis to diminish in the foreseeable future.

VI. CONCLUSION AND FUTURE WORK

We have argued that mechanized, formally verified, and yet still readable schedulability proofs are desirable, feasible to create with current tools and with reasonable effort, and beneficial beyond the increase in confidence.

We have substantiated these claims with concrete case studies. While prioritizing readability, we formally specified a foundation for schedulability analysis (§III) and proved correct an influential multiprocessor response-time analysis (§IV). To demonstrate that our framework is flexible and extensible, we added release jitter (§IV-A), removed the assumption that jobs are sequential (§IV-B), and discussed how the formalization effort allowed us to prove a previously unknown dominance result (§IV-C).

In the next steps, we plan to extend our specification and proofs to other real-time scheduling problems, such as APA scheduling [50], analysis of self-suspensions, overhead accounting and event stream models [47, 69]. In the long term, it will be interesting to investigate how mechanized schedulability analysis can be integrated into commercial schedulability analysis tools such as RT-Druid, Rapid RMA, RTaW-Pegase, or SymTA/S.

To conclude, we believe that the time is right for *readable* mechanized schedulability analysis and that, despite the increased upfront proof effort, it is well worth the cost. Overall, formally verified schedulability analysis is an important, and perhaps even inevitable, step in the design of *provably-correct* real-time systems. PROSA is an open-source project [4] and we gladly welcome any involvement and contributions.

ACKNOWLEDGMENTS

We thank Sophie Quinton and Pascal Fradet of INRIA Grenoble – Rhône-Alpes and Jean-François Monin of Verimag for insightful discussions, and the MPI-SWS COQ user community, in particular Jan-Oliver Kaiser and Viktor Vafeiadis, for their help and valuable advice.

REFERENCES

[1] "The COQ Proof Assistant," project web site, https://coq.inria.fr.
[2] "Mathematical Components Library," project web site, http://math-comp.github.io/math-comp.
[3] "PROSA – Artifact Evaluation," supplemental material and formal proofs, http://prosa.mpi-sws.org/releases/v0.1/artifact.
[4] "PROSA: The Proven Schedulability Analysis Repository," project web site, http://prosa.mpi-sws.org.
[5] J. Anderson and A. Srinivasan, "Early-release fair scheduling," in *Proc. of the 12th Euromicro Conference on Real-Time Systems (ECRTS'00)*, 2000.
[6] N. Audsley and K. Bletsas, "Fixed priority timing analysis of real-time systems with limited parallelism," in *Proc. of the 16th Euromicro Conference on Real-Time Systems (ECRTS'04)*, 2004.
[7] ——, "Realistic analysis of limited parallel software/hardware implementations," in *Proc. of the 10th Real-Time and Embedded Technology and Applications Symposium (RTAS'04)*, 2004.
[8] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. Wellings, "Applying new scheduling theory to static priority pre-emptive scheduling," *Software Engineering Journal*, vol. 8, no. 5, pp. 284–292, 1993.
[9] T. Baker and S. Baruah, "Schedulability analysis of multiprocessor sporadic task systems," *Handbook of Real-Time and Embedded Systems*, 2007.
[10] T. Baker and M. Cirinei, "Brute-force determination of multiprocessor schedulability for sets of sporadic hard-deadline tasks," in *Proc. of the 11th International Conference on Principles of Distributed Systems (OPODIS'07)*, 2007.
[11] S. Baruah and J. Carpenter, "Multiprocessor fixed-priority scheduling with restricted interprocessor migrations," in *Proc. of the 15th Euromicro Conference on Real-Time Systems (ECRTS'03)*, 2003.
[12] S. Baruah and G. Lipari, "A multiprocessor implementation of the total bandwidth server," in *Proc. of the 18th International Parallel and Distributed Processing Symposium (IPDPS'04)*, 2004.
[13] S. Baruah, A. Mok, and L. Rosier, "Preemptively scheduling hard-real-time sporadic tasks on one processor," in *Proc. of the 11th Real-Time Systems Symposium (RTSS'90)*, 1990.
[14] S. Baruah, L. Rosier, and R. Howell, "Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor," *Real-time systems*, vol. 2, no. 4, pp. 301–324, 1990.
[15] S. Baruah, N. Cohen, C. G. Plaxton, and D. Varvel, "Proportionate progress: A notion of fairness in resource allocation," *Algorithmica*, vol. 15, no. 6, pp. 600–625, 1996.
[16] S. Baruah, M. Bertogna, and G. Buttazzo, *Multiprocessor Scheduling for Real-Time Systems*. Springer, 2015.
[17] M. Bertogna and M. Cirinei, "Response-time analysis for globally scheduled symmetric multiprocessor platforms," in *Proceedings of the 28th Real-Time Systems Symposium (RTSS'07)*.
[18] K. Bletsas, N. Audsley, W.-H. Huang, J.-J. Chen, and G. Nelissen, "Errata for three papers (2004-05) on fixed-priority scheduling with self-suspensions," CISTER, Tech. Rep. CISTER-TR-150713, July 2015.
[19] S. Boldo, C. Lelay, and G. Melquiond, "Coquelicot: A user-friendly library of real analysis for Coq," *Mathematics in Computer Science*, vol. 9, no. 1, pp. 41–62, 2015.
[20] V. Bonifaci and A. Marchetti-Spaccamela, "Feasibility analysis of sporadic real-time multiprocessor task systems," in *Proc. of the 18th Annual European Symposium on Algorithms (ESA'10)*, 2010.
[21] B. Brandenburg, "Improved analysis and evaluation of real-time semaphore protocols for P-FP scheduling," in *Proc. of the 19th Real-Time and Embedded Technology and Applications Symposium (RTAS'13)*, 2013.
[22] B. Brandenburg and J. Anderson, "On the implementation of global real-time schedulers," in *Proc. of the 30th Real-Time Systems Symposium (RTSS'09)*, 2009.
[23] R. Bril, J. Lukkien, R. Davis, and A. Burns, "Message response time analysis for ideal controller area network (CAN) refuted," *Proc. of the 5th International Workshop on Real-Time Networks (RTN'06)*, 2006.
[24] A. Burmyakov, E. Bini, and E. Tovar, "An exact schedulability test for global FP using state space pruning," in *Proc. of the 23rd International Conference on Real Time and Networks Systems (RTNS'15)*, 2015.
[25] J. Calandrino, J. Anderson, and D. Baumberger, "A hybrid real-time scheduling approach for large-scale multicore platforms," in *Proc. of the 19th Euromicro Conference on Real-Time Systems (ECRTS'07)*, 2007.
[26] A. Carminati, R. de Oliveira, and L. Friedrich, "Exploring the design space of multiprocessor synchronization protocols for real-time systems," *Journal of Systems Architecture*, vol. 60, no. 3, pp. 258–270, 2014.
[27] H. Chen, D. Ziegler, T. Chajed, A. Chlipala, M. F. Kaashoek, and N. Zeldovich, "Using Crash Hoare logic for certifying the FSCQ file system," in *Proc. of the 25th Symposium on Operating Systems Principles (SOSP'15)*, 2015.
[28] J.-J. Chen, G. Nelissen, W.-H. Huang, M. Yang, B. Brandenburg, K. Bletsas, C. Liu, P. Richard, F. Ridouard, N. Audsley, R. Rajkumar, and D. de Niz, "Many suspensions, many problems: A review of self-suspending tasks in real-time systems," Department of Computer Science, TU Dortmund, Tech. Rep. 854, 2016.
[29] S. Colin, V. Poirriez, and G. Mariano, "Thoughts about the implementation of the duration calculus with Coq," in *Proc. of the 4th International Workshop on the Implementation of Logics (IWIL'03)*, 2003.
[30] M. Cordovilla, F. Boniol, E. Noulard, and C. Pagetti, "Multiprocessor schedulability analyser," in *Proc. of 26th ACM Symposium on Applied Computing (SAC'11)*, 2011.
[31] R. Davis, A. Burns, J. Marinho, V. Nelis, S. Petters, and M. Bertogna, "Global fixed priority scheduling with deferred pre-emption revisited," Univ. of York, Tech. Rep. YCS-2013-483, 2013.
[32] R. Davis, A. Burns, R. Bril, and J. Lukkien, "Controller area network (CAN) schedulability analysis: Refuted, revisited and revised," *Real-Time Systems*, vol. 35, no. 3, pp. 239–272, 2007.
[33] R. Davis, A. Burns, J. Marinho, V. Nelis, S. M. Petters, and M. Bertogna, "Global fixed priority scheduling with deferred pre-emption," in *Proc. of the 19th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'13)*, 2013.
[34] R. De Millo, R. Lipton, and A. Perlis, "Social processes and proofs of theorems and programs," *Communications of the ACM*, vol. 22, no. 5, pp. 271–280, 1979.
[35] D. De Rauglaudre, "Vérification formelle de conditions d'ordonnancabilité de tâches temps réel périodiques strictes," in *Actes des 23e Journées Francophones des Langages Applicatifs (JFLA'12)*, 2012.
[36] R. Devillers and J. Goossens, "Liu and Layland's schedulability test revisited," *Information Processing Letters*, vol. 73, no. 5, pp. 157–161, 2000.
[37] C. Doczkal, J.-O. Kaiser, and G. Smolka, "A constructive theory of regular languages in Coq," in *Proc. of the 3rd International Conference on Certified Programs and Proofs (CPP'13)*. Springer, 2013.
[38] B. Dutertre and V. Stavridou, "Formal analysis for real-time scheduling," in *Proc. of the 19th IEEE Digital Avionics Systems Conference (DASC'00)*, 2000.
[39] B. Dutertre, "The priority ceiling protocol: formalization and analysis using PVS," in *Proc. of the 21st Real-Time Systems Symposium (RTSS'99)*, 1999.
[40] E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi, "Schedulability analysis of fixed-priority systems using timed automata," in *Proc. of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'06)*, 2006.
[41] S. Funk and S. Baruah, "Restricting EDF migration on uniform multiprocessors," in *Proc. of the 12th International Conference on Real-Time Systems (RTS'04)*, 2004.
[42] S. Funk, J. Goossens, and S. Baruah, "On-line scheduling on uniform multiprocessors," in *Proc. of the 22nd Real-Time Systems Symposium (RTSS'01)*, 2001.
[43] G. Geeraerts, J. Goossens, and M. Lindström, "Multiprocessor schedulability of arbitrary-deadline sporadic tasks: complexity and antichain algorithm," *Real-Time Systems*, vol. 49, no. 2, pp. 171–218, 2012.

[44] G. Gonthier, "A computer-checked proof of the Four Colour Theorem," 2005. [Online]. Available: http://research.microsoft.com/en-us/um/people/gonthier/4colproof.pdf

[45] G. Gonthier, A. Asperti, J. Avigad, Y. Bertot, C. Cohen, F. Garillot, S. Roux, A. Mahboubi, R. O'Connor, S. Ould Biha, I. Pasca, L. Rideau, A. Solovyev, E. Tassi, and L. Théry, "A machine-checked proof of the Odd Order Theorem," in *Proc. of the 4th International Conference on Interactive Theorem Proving (ITP'13)*, 2013.

[46] J. Goossens, S. Funk, and S. Baruah, "Priority-driven scheduling of periodic task systems on multiprocessors," *Real-time systems*, vol. 25, no. 2-3, pp. 187–205, 2003.

[47] K. Gresser, "An event model for deadline verification of hard real-time systems," in *Proc. of the 5th Euromicro Workshop on Real-Time Systems (EWRTS'93)*, 1993, pp. 118–123.

[48] N. Guan, Z. Gu, Q. Deng, S. Gao, and G. Yu, "Exact schedulability analysis for static-priority global multiprocessor scheduling using model-checking," in *Proc. of the 5th International Workshop on Software Technologies for Embedded and Ubiquitous Systems (SEUS'07)*, 2007.

[49] N. Guan, Z. Gu, M. Lv, Q. Deng, and G. Yu, "Schedulability analysis of global fixed-priority or edf multiprocessor scheduling with symbolic model-checking," in *Proc. of the 11th International Symposium on Object Oriented Real-Time Distributed Computing (ISORC'08)*, 2008.

[50] A. Gujarati, F. Cerqueira, and B. Brandenburg, "Schedulability analysis of the Linux push and pull scheduler with arbitrary processor affinities," in *Proc. of the 25th Euromicro Conference on Real-Time Systems (ECRTS'13)*, 2013.

[51] ——, "Multiprocessor real-time scheduling with arbitrary processor affinities: from practice to theory," *Real-Time Systems*, vol. 50, no. 1, pp. 1–44, 2014.

[52] G. Han, H. Zeng, M. Di Natale, X. Liu, and W. Dou, "Experimental evaluation and selection of data consistency mechanisms for hard real-time applications on multicore platforms," *IEEE Transactions on Industrial Informatics*, vol. 10, no. 2, pp. 903–918, 2014.

[53] S. Kato and Y. Ishikawa, "Gang EDF scheduling of parallel task systems," in *Proc. of the 30th Real-Time Systems Symposium (RTSS'09)*, 2009.

[54] H. Kim, S. Wang, and R. Rajkumar, "vMPCP: A synchronization framework for multi-core virtual machines," in *Proc. of the 35th Real-Time Systems Symposium (RTSS'14)*, 2014.

[55] I.-G. Kim, K.-H. Choi, S.-K. Park, D.-Y. Kim, and M.-P. Hong, "Real-time scheduling of tasks that contain the external blocking intervals," in *Proc. of the 2nd International Workshop on Real-Time Computing Systems and Applications (RTCSA'95)*, 1995.

[56] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish *et al.*, "seL4: Formal verification of an OS kernel," in *Proc. of the 22nd Symposium on Operating Systems Principles (SOSP'12)*, 2009.

[57] K. Lakshmanan and R. Rajkumar, "Scheduling self-suspending real-time tasks with rate-monotonic priorities," in *Proc. of the 16th Real-Time and Embedded Technology and Applications Symposium (RTAS'10)*, 2010.

[58] K. Lakshmanan, S. Kato, and R. Rajkumar, "Scheduling parallel real-time tasks on multi-core processors," in *Proc. of the 31st Real-Time Systems Symposium (RTSS'10)*, 2010.

[59] K. Lakshmanan, D. de Niz, and R. Rajkumar, "Coordinated task scheduling, allocation and synchronization on multiprocessors," in *Proc. of the 30th IEEE Real-Time Systems Symposium (RTSS'09)*, 2009, pp. 469–478.

[60] X. Leroy, "Formal verification of a realistic compiler," *Communications of the ACM*, vol. 52, no. 7, pp. 107–115, 2009.

[61] J. Li, Z. Luo, D. Ferry, K. Agrawal, C. Gill, and C. Lu, "Global EDF scheduling for parallel real-time tasks," *Real-Time Systems*, vol. 51, no. 4, pp. 395–439, 2014.

[62] C. Liu and J. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM (JACM)*, vol. 20, no. 1, pp. 46–61, 1973.

[63] C. Liu and J. Anderson, "Supporting sporadic pipelined tasks with early-releasing in soft real-time multiprocessor systems," in *Proc. of the 15th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'09)*, 2009.

[64] C. Maia, M. Bertogna, L. Nogueira, and L. Pinho, "Response-time analysis of synchronous parallel tasks in multiprocessor systems," in *Proc. of the 22nd International Conference on Real-Time Networks and Systems (RTNS'14)*, 2014.

[65] L. Ming, "Scheduling of the inter-dependent messages in real-time communication," in *Proc. of the 1st International Workshop on Real-Time Computing Systems and Applications (RTCSA'94)*, 1994.

[66] A. K. Mok, "Fundamental design problems of distributed systems for the hard-real-time environment," Ph.D. dissertation, Massachusetts Institute of Technology, 1983.

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

[68] B. Pierce, C. Casinghino, M. Gaboardi, M. Greenberg, C. Hriţcu, V. Sjoberg, and B. Yorgey, *Software Foundations*. Electronic textbook, 2015. [Online]. Available: http://www.cis.upenn.edu/~bcpierce/sf

[69] K. Richter and R. Ernst, "Event model interfaces for heterogeneous system analysis," in *Proc. of the 2002 Design, Automation and Test in Europe Conference and Exhibition (DATE'02)*, 2002.

[70] D. Shuzhen, X. Qiwen, and Z. Naijun, "A formal proof of the rate monotonic scheduler," in *Proc. of the 6th International Conference on Real-Time Computing Systems and Applications (RTCSA'99)*, 1999.

[71] B. Sprunt, L. Sha, and J. Lehoczky, "Aperiodic task scheduling for hard-real-time systems," *Real-Time Systems*, vol. 1, no. 1, pp. 27–60, 1989.

[72] J. Stankovic and K. Ramamritham, "The Spring kernel: a new paradigm for real-time systems," *IEEE Software*, vol. 8, no. 3, pp. 62–72, 1991.

[73] Y. Sun and G. Lipari, "A pre-order relation for exact schedulability test of sporadic tasks on multiprocessor global fixed-priority scheduling," *Real-Time Systems*, vol. 52, no. 3, pp. 323–355, May 2015.

[74] L. Thiele, S. Chakraborty, and M. Naedele, "Real-time calculus for scheduling hard real-time systems," in *Proc. of the 2000 IEEE International Symposium on Circuits and Systems (ISCAS'00)*, 2000.

[75] K. Tindell, H. Hansson, and A. J. Wellings, "Analysing real-time communications: Controller Area Network (CAN)," in *Proc. of the 15th Real-Time Systems Symposium (RTSS'94)*, 1994.

[76] G. von der Brüggen, J.-J. Chen, and W.-H. Huang, "Schedulability and optimization analysis for non-preemptive static priority scheduling based on task utilization and blocking factors," in *Proceedings of 27th Euromicro Conference on Real-Time Systems (ECRTS'15)*, 2015.

[77] M. Wilding, "A machine-checked proof of the optimality of a real-time scheduling policy," in *Proc. of the 10th International Conference on Computer-Aided Verification (CAV'98)*, 1998, pp. 369–378.

[78] Q. Xu and N. Zhan, "Formalising scheduling theories in duration calculus," *Nordic Journal of Computing*, vol. 14, no. 3, pp. 173–201, Sep. 2008.

[79] M. Yang, H. Lei, Y. Liao, and F. Rabee, "PK-OMLP: An OMLP based k-exclusion real-time locking protocol for multi-GPU sharing under partitioned scheduling," in *Proc. of the 11th International Conference on Dependable, Autonomic and Secure Computing (DASC'13)*, 2013.

[80] ——, "Improved blocking time analysis and evaluation for the multiprocessor priority ceiling protocol," *Journal of Computer Science and Technology*, vol. 29, no. 6, pp. 1003–1013, 2014.

[81] Z. Yuhua and Z. Chaochen, "A formal proof of the deadline driven scheduler," in *Proc. of the 3rd International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT'94)*, 1994.

[82] H. Zeng and M. Di Natale, "Mechanisms for guaranteeing data consistency and flow preservation in AUTOSAR software on multi-core platforms," in *Proc. of the 6th International Symposium on Industrial Embedded Systems (SIES'11)*, 2011.

[83] N. Zhan, "Another formal proof for deadline driven scheduler," in *Proc. of the 7th International Conference on Real-Time Computing Systems and Applications (RTCSA'00)*, 2000.

[84] N. Zhang, Z. Duan, C. Tian, and D. Du, "A formal proof of the deadline driven scheduler in PPTL axiomatic system," *Theoretical Computer Science*, vol. 554, pp. 229 – 253, 2014.

[85] X. Zhang, C. Urban, and C. Wu, "Priority inheritance protocol proved correct," in *Proc. of the 3rd International Conference on Interactive Theorem Proving (ITP'12)*, 2012.