Foundational Response-Time Analysis as Explainable Evidence of Timeliness

Marco Maida
Max Planck Institute for Software Systems (MPI-SWS), Kaiserslautern, Germany

Sergey Bozhko
Max Planck Institute for Software Systems (MPI-SWS), Kaiserslautern, Germany
Saarbrücken Graduate School of Computer Science, Universität des Saarlandes, Germany

Björn B. Brandenburg
Max Planck Institute for Software Systems (MPI-SWS), Kaiserslautern, Germany

Abstract

The paper introduces *foundational* response-time analysis (RTA) as a means to produce strong and independently checkable *evidence of temporal correctness*. In a foundational RTA, each response-time bound calculated comes with an auto-generated *certificate* of correctness – a short and human-inspectable sequence of *machine-checked proofs* that formally show the claimed bound to hold. In other words, a foundational RTA yields *explainable* results that can be independently verified (e.g., by a certification authority) in a rigorous manner (with an automated proof checker). Consequently, the analysis tool itself does *not* need to be verified nor trusted. As a proof of concept, the paper presents POET, the first foundational RTA tool. POET generates certificates based on Prosa, the to-date largest verified framework for schedulability analysis, which is based on Coq. The trusted computing base is hence reduced to the Coq proof checker and its dependencies. POET currently supports two scheduling policies (*earliest-deadline-first, fixed-priority*), two preemption models (*fully preemptive, fully non-preemptive*), arbitrary deadlines, periodic and sporadic tasks, and tasks characterized by arbitrary arrival curves. The paper describes the challenges inherent in the development of a foundational RTA tool, discusses key design choices, and reports on its scalability.

2012 ACM Subject Classification Computer systems organization → Real-time systems; Software and its engineering → Formal software verification

Keywords and phrases hard real-time systems, response-time analysis, uniprocessor, Coq, Prosa, fixed priority, EDF, preemptive, non-preemptive, verification

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2022.19

Supplementary Material *Software (ECRTS 2022 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.8.1.7`

Funding 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), and from the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 391919384.

Acknowledgements We thank Pierre Roux for introducing us to CoqEAL and the members of the joint ANR-DFG project RT-PROOFS for fruitful discussions.

1 Introduction

The purpose of a *response-time analysis* (RTA) is to obtain safe bounds on the worst-case response times of all critical tasks in a real-time system. To this end, the system is described with a mathematical model, which typically comprises a workload model, a resource model, and a scheduling policy. The model is then analyzed to derive response-time bounds, which requires **(i)** a *theory* that rigorously justifies that the RTA correctly characterizes the worst-case scenario, and **(ii)** an RTA *tool* that executes the concrete calculations.

Both aspects are equally critical. An error in (i), such as an invalid over-generalization or a missed corner case, leads to a flawed theory. An error in (ii), which can be any significant bug in the tool, leads to a flawed implementation. Given the number of documented analysis mistakes in the real-time literature (e.g., [13, 18, 28, 34]), and the reality that complex tools are rarely bug-free, both the research community and industry have shown a growing interest in the application of computer-assisted formal verification (e.g., [11, 21, 23]), with safety standards also advising its use (e.g., ISO 26262, DO-178C).

However, applying the currently available proof assistants and verification tools to mechanize RTAs and formally verify RTA tools is neither an easy nor a cheap task. Once a theory – or the behavior of a program – is encoded in a proof assistant's specification language, it usually needs to be augmented with additional information (e.g., step-by-step proofs, program invariants) before the verification procedure can succeed. This process requires human intervention and usually takes a considerable amount of time. Moreover, developing and maintaining a verified tool requires advanced programming skills and specialized expertise. As a consequence of such cost and knowledge barriers, RTA tools in use today are typically not verified, and this is unlikely to change in the foreseeable future.

In this paper, we therefore explore the challenge of obtaining trustworthy response-time bounds without having to verify the RTA tool that produces them. To this end, we propose *foundational RTA* as means to produce evidence of timeliness that can be independently checked in a rigorous manner by an untrusting third party. As elaborated in Section 2, a foundational RTA tool must produce *proof-carrying response-time bounds*, that is, bounds that come with *certificates of correctness* that can be automatically checked by a proof assistant such as Coq. The key advantage of a foundational RTA is that the *trusted computing base* (TCB) is reduced to only the proof checker and its dependencies: the analysis tool itself does *not* need to be trusted because the correctness of its claims can be easily and independently verified. Therefore, the RTA tool's source code can be updated, modified, ported, and optimized like any other non-critical software.

The advantages of foundational RTA are obvious; its practical realizability and scalability, however, much less so. As a proof of concept, we present the design and implementation of the first foundational RTA tool. Our RTA tool, called POET (*Prosa Obsigned Evidence of Timeliness*), works in conjunction with the formally verified Prosa framework for real-time schedulability analysis [11], and in particular with its *abstract RTA* library [8]. When a problem instance (i.e., a concrete task set, scheduling policy, and preemption model) is given as input, POET generates, along with the usual response-time bounds, a set of formal proofs of correctness that can be automatically machine-checked by the Coq proof assistant [42]. The process is completely automated (i.e., users do *not* need to write definitions nor to prove theorems) and does not require any expertise with formal verification on behalf of the user. Consequently, POET is the first RTA tool that can assert the correctness of its results in a trustworthy manner without any need for the tool itself to be verified.

To summarize, this paper makes the following **contributions**:

- We introduce the notions of foundational RTA and proof-carrying response-time bounds (Section 2), along with the first foundational RTA tool, POET, and discuss key design and implementation challenges (Section 4);
- describe how mechanisms in POET and the overall workflow ensure the trustworthiness of the certificates (Section 5),
- discuss how POET computes the search space for each task's response-time bound in a manner that is both computationally efficient and verifiable (Section 6);
- explain how we enabled POET to scale to numerically large task parameters, despite Prosa's proof-oriented (but computationally inefficient) number representation (Section 7); and finally

- report on an empirical evaluation of POET on synthetic task sets of realistic complexity in terms of task count, utilization, and numerical magnitude of the parameters (Section 8).

Last but not least, POET makes it possible for users unfamiliar with Coq to benefit from the power of formal verification. POET will be released as an open-source project.

2 Design Space of Verified RTAs

To motivate the advantages of foundational RTA and to provide context, we begin with a brief survey of the space of possible alternatives. Given that the literature on real-time systems in general, and on RTAs in particular, targets mission- and safety-critical systems – such as cars, trains, aircraft, and spacecraft – RTA tools are an obvious candidate for verification. However, while the ultimate goal is clear – the computed bounds must never be optimistic – it is less obvious what it means to actually "verify an RTA tool" in practice. In fact, there are many ways in which formal verification could improve the trustworthiness of RTA tools, with differing trade-offs in terms of required effort and verification coverage.

Common flaws. To better understand the space of possible solutions, first consider the following (non-exhaustive) list of flaws threatening the validity of response-time bounds. The most obvious issue is that the *underlying RTA theory may be incorrect* (**F1**). Sadly, the published record shows that this is far from just a theoretical concern: many cases of flawed reasoning have been documented [12, 18, 28, 34, 49], Chen et al. [13] and Cerqueira et al. [11] give more examples of incorrect analyses of processor scheduling, Indrusiak et al. [31] list *eight* refuted RTAs for on-chip networks (NoCs), and new corrections still continue to appear on a regular basis (e.g., [29, 52]). At this point in time, it has become painfully clear that engineers would be foolish to trust an analysis to be correct just because it was peer-reviewed and published at a well-reputed venue. Formal verification is arguably the best, and perhaps the only way, to overcome this trustworthiness gap, especially in critical systems.

Moving on, even if the underlying analysis is not at fault, then a particular implementor may still *misinterpret the published analysis* (**F2**), or simply overlook *logic errors in the implementation* (**F3**) such as incorrect loop conditions or incomplete search-space traversals. From first-hand experience, such issues do not appear to be uncommon. In a similar vein, easy-to-overlook *programming mistakes resulting in silent errors* (**F4**), such as over- or underflow during unchecked integer arithmetic or floating-point precision issues, plague RTA tools just like they do applications in virtually all other domains.

Likely somewhat less common, but a real concern nonetheless, is the possibility that *software defects in other parts of the tool* (**F5**) corrupt the state of the analysis implementation, such as a dangling pointer in an XML or JSON parsing library, or a concurrency issue. This can also affect tools programmed in otherwise memory-safe languages (such as OCaml or Java) that link with native C libraries for common functionality.

Another concern related to I/O is that the *parsing of the workload and system description* (**F6**), especially if it is presented in a difficult-to-process format, may silently alter the input such that an otherwise correct analysis is run on a slightly, but critically different problem instance than intended. This is a particular concern for tools that do not produce *explainable* results, that is, for tools that produce output from which it is not possible for a human to understand how, or based on exactly which model, the result was obtained.

Last, and also decidedly least, is the concern that *transient errors*, such as bit flips due to cosmic background radiation or thermal noise, could affect the RTA tool during its execution (or the verifier, for that matter). While not unheard of, the likelihood of occurrence on an

engineer's workstation or laptop is extremely low. Furthermore, it can be safely assumed that any critical system is analyzed not just once, and not only on just one machine, at which point the risk of repeated transient errors at design time becomes negligible.

Clearly, an ideal "verified RTA" should mitigate all of these flaws (and more). Additional desiderata include that it should be reasonably fast and scale to workloads of industrially relevant size and parameter magnitudes (e.g., expressed in processor cycles or nanoseconds). Furthermore, an ideal RTA tool should be explainable in the sense that it justifies its results in a way that can independently checked by a third party (e.g., government regulators) for use in (non-formal) certification processes (e.g., as commonly encountered in the avionics and medical domains, and increasingly also in the automotive domain). How can these goals be achieved? While a full survey of the verification literature is beyond the scope of this paper, we can generally identify three fundamental approaches: *verify the tool* itself, augment an unverified tool with a verified *results validator*, and the proposed *foundational approach*.

Verified implementation. In principle, one could verify the entire RTA tool, in particular if it is implemented in a verification-friendly language such as OCaml or Haskell. To our knowledge, this has not been done to-date, and whether it is even worthwhile in the context of RTA tools is a point open for debate. Nonetheless, it is interesting to consider different levels of *specifications* that such an effort could target.

The weakest specification would be that the tool computes a particular mathematical expression, or finds a fixed-point of a certain structure, without any formal claims about the semantical implications of this. In other words, the tool would be verified to correctly compute a number, but no formal claim is made about the meaning of this number (i.e., that it is a bound). The advantage of this is that the underlying RTA does *not* have to be verified, i.e., this approach could be used with any RTA in the literature. While this requires proving the absence of F3- and F4-type flaws, it fails to mitigate F1- and F2-type flaws.

A step up would be to use the same specification for the implementation (of just the computation, without semantics), while also verifying the employed RTA *theory* in a separate effort (or using an already-verified one, e.g., [8, 11]). The resulting assurance is much improved, as this approach prevents the historically very problematic F1-type flaws. However, without an end-to-end verification, F2-type flaws still cannot be ruled out.

Ideally, the tool's specification should include *semantic guarantees*: if the tool claims a computed number to be a response-time bound, then there does not exist a schedule (i.e., a trace of the analyzed model) in which the bound is exceeded. Clearly, this would be the most desirable level to target as it completely eliminates the risk of any flaws in categories F1–F4.

The limiting factor is the implied amount of work: not only would such a semantically rich specification require a complete verification of the underlying theory (e.g., as done in Prosa [11]), but to also categorically avoid F5- and F6-type issues, a deep specification and verification of all "plumbing" code (such as the input format parser) would have to be carried out. With the contemporary software ecosystem and verification tools, this would be a daunting task of questionable feasibility. On top of this, a verified RTA tool would not inherently produce any additional evidence of correctness besides the computed bounds (i.e., its output is still not explainable). On the plus side, there is no reason to doubt that such a tool could be fast. However, to the best of our knowledge, there have been no attempts to-date to verify an entire RTA tool, likely due to the enormous effort that would be required.

Results validator. A different approach that side-steps the issue of having to verify the entire RTA tool with all its real-world complexities is to instead *check only the analysis results* [23, 36]. The high-level idea is to leave the actual RTA implementation unverified

(with all the software-engineering flexibility this brings), and to develop a *second*, verified tool that independently validates the results of the RTA tool. If it is fundamentally easier to check a solution than to find one, then this approach is beneficial in principle.

While much more attainable than a fully verified RTA implementation – as successfully demonstrated on industrially relevant network analyses [23, 36] – it comes at the price of having to develop the formally verified results validator, which is subject to all the aforementioned challenges related to verified code. Case in point, Fradet et al.'s CertiCAN [23] checking procedure is first verified with Coq and then automatically extracted from Coq as an OCaml file. However, to actually obtain a useful, runnable tool, the extracted OCaml code must still be combined with non-verified support code to handle "plumbing" tasks such as I/O and parsing. As a result, F5- and F6-type issues remain plausible concerns.

Proof-carrying bounds. In this paper, we seek to develop a way of obtaining trustworthy response-time bounds that avoids verifying or trusting the RTA tool altogether. To this end, inspired by Appel's seminal work on *foundational proof-carrying code* [3], we transfer the foundational approach to the RTA setting. To elaborate on the original concept, proof-carrying code is a (usually) low-level program, such as a sequence of instructions emitted by a compiler, that comes with an independently checkable proof establishing its properties [38]. A foundational proof-carrying code [3] has a proof that relies only on the foundations of mathematical logic, hence minimizing the number of additional components (ad-hoc type systems, verification condition generators, one-off output checkers, etc.) to be trusted.

Likewise, we require a *foundational RTA* to produce response-time bounds that come with a proof of correctness relying only on the mathematical logic implemented by the underlying proof assistant, i.e., a foundational RTA yields *proof-carrying response-time bounds*.

To demonstrate the practicality of this idea, we present the first such tool, POET, as a proof of concept. By analogy, POET behaves like a successful student during an exam: when faced with a tricky computational problem (i.e., finding the response-time bound of a task), POET not only yields the final solution (i.e., a number), but also justifies with a sequence of proof steps *why* the solution is correct. In other words, the solution is *explainable* in the sense that its derivation can be understood without interviewing the student about their thought processes (i.e., knowing POET's source code). POET thus does not have to be trusted at all, since its answer can be independently verified.

As a foundational proof, by definition, depends only on the foundations of mathematical logic, it necessarily includes a complete justification of both the underlying RTA and how it applies to the specific workload. POET hence categorically avoids all flaws of types F1–F5, and with light supervision (Section 5) sidesteps F6-type issues altogether, while inherently producing independently checkable evidence of timeliness.

The *Achilles' heel* of the foundational approach, however, is that its runtime is closely tied to that of the underlying proof assistant that checks the generated certificates, which, as we discuss in Sections 6–8, was a serious challenge in the realization of POET.

3 Background

POET relies on abstract RTA [8], which has been integrated into Prosa [40], a Coq [42] framework. We briefly review each of these building blocks in turn.

Coq [42] is a widely-used, mature interactive theorem prover based on the *calculus of inductive constructions* [15, 16, 39] that provides rich support for the *mechanization* (i.e., the formalization and machine-checked verification) of both nontrivial mathematical theories (e.g., [25, 26]) and large software systems (e.g., [35]). Coq provides primarily two languages:

Gallina, a formal, dependently typed specification language used to write mathematical definitions, state theorems, and implement functional programs, and Ltac, an untyped macro language used to steer the proof engine.

Coq is not a fully automatic theorem prover: while proof *checking* is automatic, proof *authoring* typically requires human intervention. Once a theorem is stated, it is necessary to provide a sequence of Ltac *tactic* applications (each of which can be seen as a single step of the proof) that, starting from the stated hypotheses, allows the proof engine to reach the claimed conclusion. As Coq allows the user to create new tactics via the Ltac language, it is possible to introduce domain-specific automation. POET heavily relies on this feature to completely automate the proof generation process.

Once a Coq source file (`.v`) has been written, it can be compiled into a lower-level representation (generating a `.vo` file) and finally verified by the standalone proof checker coqchk. The Coq compiler, coqchk, and their dependencies, form POET's entire TCB.

Prosa [40], the schedulability analysis framework underlying POET's certificates, uses Coq and its popular extension ssreflect [37]. Starting from classic real-time systems concepts, such as *job*, *task*, *processor*, and *arrival curve*, the contributors of Prosa both mechanized classical results (e.g., the optimality of the *earliest-deadline-first* scheduling policy) and developed new theories (e.g., [10, 22, 24]).

For our purposes, the most relevant theory in Prosa is *abstract RTA* (aRTA) [8], which formalizes the well-known busy-window principle to derive a generic RTA applicable to different types of workloads, scheduling policies, and preemption models. aRTA has been instantiated for two scheduling policies (*earliest-deadline-first, fixed-priority*) and four preemption models (*preemptive, non-preemptive, limited-preemptive, floating non-preemptive*) in every possible combination, yielding eight different fully verified RTAs [8].

Given that all proofs have been mechanized in Prosa, a high degree of trust may be placed upon the correctness of aRTA and its instantiations. However, a mechanized RTA theory, much like its traditional pen-and-paper counterpart, only describes and justifies *under which conditions* a claimed response-time bound is valid, but it is not, per se, an executable program that can yield numerical results given a concrete task set. Rather, the main aRTA proof follows an axiomatic approach by treating the scheduler, the tasks, and the claimed response-time bounds as abstract variables on which a number of assumptions are made. The abstract result is then derived from these assumptions.

The key idea at the heart of POET is that, by providing instantiations for all variables (i.e., by assigning concrete values), along with a proof that all of aRTA's assumptions are satisfied, aRTA can be put to practical use to verify precomputed response-time bounds. POET thus inherits the system model from aRTA, which we summarize next.

System model. aRTA assumes a discrete time model, where $\varepsilon \triangleq 1$ represents the smallest, indivisible unit of time (e.g., a processor cycle). The system is comprised of a set of n independent tasks $\tau = \{\tau_1, \ldots, \tau_n\}$ scheduled on a uniprocessor. Each task τ_i is characterized by its *worst-case execution time* C_i, its *relative deadline* D_i, and an *arrival bound* $\alpha_i(\Delta)$ that upper-bounds the number of new *jobs* (i.e., task activations) that arrive in any interval of length Δ. The two considered preemption models are expressed through each task's *run-to-completion threshold* RCT_i, where $RCT_i = \varepsilon$ in the case of fully non-preemptive tasks, and $RCT_i = C_i$ in the case of fully preemptive tasks. In the case of fixed-priority scheduling, each task also has a fixed priority π_i. As usual in schedulability analysis, any scheduling overheads are presumed to be negligible or already integrated into each task's cost C_i.

Over time, each task τ_i produces an infinite sequence of jobs $\{J_{i,0}, J_{i,1}, \ldots\}$. We let $\mathbb{J}$ denote the set of all jobs of all tasks. Each job $J_{i,j} \in \mathbb{J}$ has an arrival time $a_{i,j}$, an *execution time* $c_{i,j} \leq C_i$, and an *absolute deadline* $d_{i,j} = a_{i,j} + D_i$. The number of job arrivals in any interval is constrained by the arrival bound: $\forall \tau_i, \forall t, \forall \Delta,\ |\{J_{i,j} \mid t \leq a_{i,j} < t + \Delta\}| \leq \alpha_i(\Delta)$.

Finally, for ease of reference, the *arrival sequence* $a(t) \triangleq \{J_{i,j} \mid \forall i, j : a_{i,j} = t\}$ is a function that maps each instant t to the (possibly empty) set of jobs released at t. The arrival sequence and the jobs it contains are the only non-instantiated variables in POET's certificates, meaning that the response-time bounds are proven for all possible arrival sequences respecting the arrival curve and worst-case execution time (WCET) of each task.

Response-time bound. By design, aRTA is independent of specific scheduling policies and preemption models. This is achieved by formulating the RTA problem in an abstract way that captures the essential relationships between the task set, the scheduling policy, the preemption model, and the worst-case response time of a task under analysis [8]. We omit these general technical details here and instead focus on the specific cases relevant to POET.

Intuitively speaking, aRTA analyzes points in the schedule at which the system is *quiet*, which means that no potentially interfering workload is pending (w.r.t. the task under analysis τ_i). A job's *busy window* is the interval between the two closest quiet times enclosing both the job's arrival and completion (by definition, no quiet time occurs while the job is pending). The core of aRTA revolves around a worst-case analysis of the busy window of an arbitrary job $J_{i,j}$ of the task under analysis τ_i. As $J_{i,j}$'s busy window ends only when $J_{i,j}$ completes, a finite busy window implies a response-time bound.

To this end, an aRTA instantiation must provide (and prove correct) two essential inputs. First, there must exist a finite bound L on the maximum busy-window length of any job of τ_i, as otherwise the busy-window principle is not applicable. Second, aRTA requires a policy-specific *interference bound function* $IBF_i(A, \Delta)$ to be defined, with the following semantics [8]: if a job $J_{i,j}$ is released A time units after the beginning of its busy window (where $A \geq 0$, i.e., A is $J_{i,j}$'s relative release offset), then $IBF_i(A, \Delta)$ is an upper bound on the maximum amount of potentially interfering workload arriving in any interval of length Δ.

Given a task under analysis τ_i with a maximum busy-window length L and a policy-specific $IBF_i(A, \Delta)$, aRTA proceeds by considering every possible arrival offset $A \in [0, L)$. For each such offset A, a solution F to the fixed-point equation

$$A + F = RCT_i + IBF_i(A, A + F) \tag{1}$$

is required to exist. Intuitively, F is the maximum time it takes for a job with arrival offset A to receive sufficient service to certainly reach the run-to-completion threshold RCT_i, at which time, by definition, it cannot be preempted anymore. Therefore, the response time of a job with arrival offset A can be bounded by $F + (C_i - RCT_i)$. Task τ_i's overall response-time bound R is given by the maximum F encountered solving Equation (1) for each offset $A \in [0, L)$.

Search space. Practically speaking, it is impossible to check every possible arrival offset $A \in [0, L)$ since, for task sets specified with nanosecond resolution, L easily reaches magnitudes in the order of trillions. Fortunately, it is not necessary to check every single point in the interval, since the only varying term in Equation (1) is $IBF_i(A, A+F)$. Hence, aRTA defines the *search space* of the task under analysis τ_i as

$$\mathcal{A}_i \triangleq \{0\} \cup \{A \in (0, L) \mid \exists \Delta,\ IBF_i(A - \varepsilon, \Delta) \neq IBF_i(A, \Delta)\} \tag{2}$$

Listing 1 An example POET input file (in YAML format) specifying two tasks scheduled under the fully-preemptive fixed-priority policy. The lower-priority task (with ID 2) is periodic (Line 10) and has a deadline exceeding the period (Line 11). The higher-priority task (with ID 1) is a sporadic task characterized by an arrival-curve prefix (Line 5), which is specified by the length of the prefix (220) and the list of steps of the curve in the prefix: $\alpha_1(\Delta) = 1$ for $\Delta \in [1, 105)$ and $\alpha_1(\Delta) = 2$ for $\Delta \in [105, 220)$. The initial value $\alpha_1(0) = 0$ can be omitted by convention.

```
1  scheduling policy: fixed-priority
2  preemption model: fully-preemptive
3  - id: 1
4    worst-case execution time: 50
5    arrival curve: [220,[[1,1],[105,2]]]
6    deadline: 100
7    priority: 2
8  - id: 2
9    worst-case execution time: 10
10   period: 30
11   deadline: 100
12   priority: 1
```

The search space $\mathcal{A}_i$ restricts the analysis to such offsets A at which the interference bound function IBF_i changes in value, hence excluding all the plateaus of the function. In practice, this restriction results in a sparse search space, which is key to obtaining a practical runtime.

4 POET: Design and Workflow

POET is the first foundational RTA tool: it generates formal Coq proofs, i.e., *certificates*, establishing the correctness of its computed response-time bounds. As a proof of concept, it supports four concrete aRTA instantiations. Specifically, it supports real-time workloads comprised of recurrent, independent, arbitrary-deadline tasks under *fixed-priority* (FP) and *earliest-deadline-first* (EDF) scheduling with both the *fully-preemptive* (FP) and *fully non-preemptive* (NP) preemption models. Task activations may be *periodic* or *sporadic*, or defined by an arbitrary arrival curve. Due to POET's novel and unique combination of objectives and features, its design and implementation posed some unusual challenges. We begin with an overview of these challenges, key design decisions, and the resulting workflow, and then discuss central issues in more detail in the subsequent sections.

The first major requirement is **usability**. For foundational RTA to be successful it must have a low barrier to adoption, which means that POET must remain accessible to a general audience without any expertise in formal verification. In particular, users must *not* be expected to be proficient in authoring Coq proofs. We therefore designed POET to require only a human-readable YAML file specifying the task set, the scheduling policy, and the preemption model. Listing 1 shows an example. From this simple input, which does not differ significantly from that of other, unverified RTA tools, POET generates verified RTA results fully automatically, without any human interaction or need for verification expertise.

The second major requirement is **transparency**. Since the process of calculating the response-time bounds, generating formal proofs of their correctness, and then machine-checking the proofs is entirely automated, it is essential to prevent silent failures. It is thus necessary to ensure explainability of the results, that is, it must be readily possible

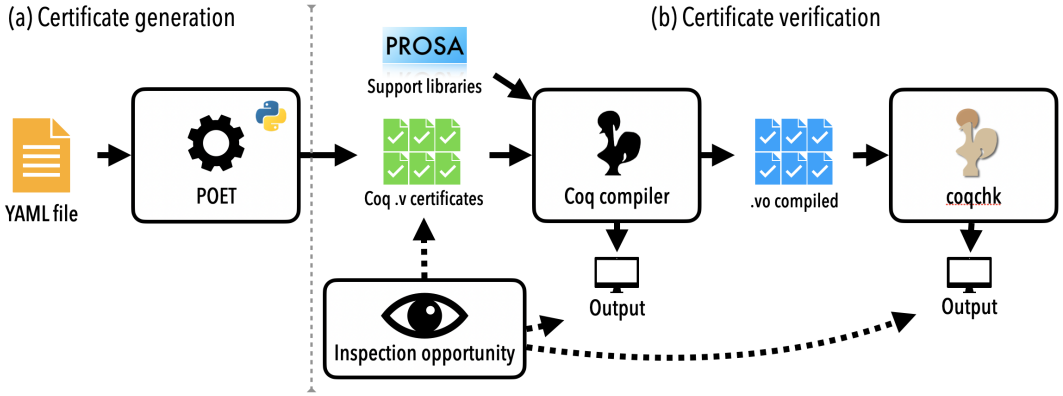

Figure 1 The POET workflow: (a) the YAML input file is used by POET to instantiate one certificate per task; (b) each certificate is compiled and verified using Coq. The procedure is fully automated, but open to human inspection at each step.

for a human to scrutinize in detail and fully understand the certificates. The certificates hence must be optimized for readability and any generated artifacts must be limited to a comprehensible size and scope. Moreover, there are ways in which machine-checked proofs might still not justify the intended conclusions. This issue is further discussed in Section 5.

A major challenge arises from POET's support of **arbitrary arrival curves**. By supporting not only the classic periodic and sporadic task models, but also any workload that can be characterized with arbitrary arrival curves, POET gains broad applicability to real-world workloads (e.g., bursty workloads, workloads with jitter, irregular arrival processes, trace-based empirical arrival bounds, etc.). However, as defined in Section 3, an arrival curve is a function on an infinite time domain. In the input to POET, such an arrival curve necessarily needs to be truncated to a finite *arrival-curve prefix* (e.g., Line 5 in Listing 1). Unfortunately, in the context of a Coq proof, representing and computing with arrival-curve prefixes is far more tricky than one might initially suspect. In Section 6, we discuss the strategies adopted in POET and the resulting accuracy *vs.* efficiency trade-offs.

Last but not least, as already mentioned in Section 2, the **computational efficiency** of the certification process proved to be a major hurdle in the implementation of POET, ultimately affecting its design. Specifically, numerically large computations are infeasible with the unary representation of numbers employed in ssreflect, and hence by extension also Prosa and aRTA. However, to be practical, POET's certificates need to support large-magnitude parameters because in real-world task sets periods, costs, and deadlines are often expressed in nanoseconds or processor cycles. We discuss the strategies that we adopted in POET to realize nonetheless acceptably fast calculations in Section 7.

4.1 Implementation and Workflow

The usability and transparency considerations lead to the workflow illustrated in Figure 1. The entire procedure comprises two phases, namely **(a)** the generation of certificates starting from an input file provided by the user, and **(b)** the Coq compilation and proof-checking.

The input file contains all necessary information about the task set, the scheduling policy, and the preemption model (recall Listing 1). Given an input file, POET produces one certificate (`.v` file) per task by instantiating a template specific to the given scheduling policy and preemption model. During this phase, Coq itself is not involved in any way.

POET is implemented as a simple Python tool and acts for the most part as a straightforward template engine. Certificate generation begins with a *template* similar to Listing 2 (discussed in Section 4.2 below) that, instead of containing specific data (e.g., task declarations, the value of L and R, etc.), has a uniquely named placeholder at each relevant position. POET simply replaces each placeholder with concrete data taken either from the YAML input file (e.g., task parameters) or computed on the fly (e.g., the value of R), and for each task stores the result as a new `.v` file containing the final proof script.

Using the same principle, POET also generates *data-dependent proof scripts*. For instance, each certificate needs a proof script showing that each point of the input-dependent search space can be paired with a fixed-point solution of Equation (1). However, the generation of data-dependent proof scripts is kept as simple as possible by implementing as much case-analysis logic as possible as generic Ltac tactics that automate the proof. Overall, the lion's share of the development effort has gone into the certificate templates and the supporting Coq libraries, whereas the Python component is relatively small and mundane in comparison.

Once the certificates are in place, POET triggers first the Coq compiler, which will produce compiled files (`.vo`) containing low-level *proof terms*, and finally coqchk, which verifies the proof terms. By design, phases (a) and (b) are independent and performed by different tools (POET and Coq). In particular, the second phase – compilation and verification of the certificates – may be performed repeatedly and on different machines.

The generated certificates establish the *soundness* of the bounds computed by POET, but do not make any *tightness* claims. A bug in POET could thus lead to the rejection of feasible task sets, or might cause pessimism in the claimed bounds, but it cannot result in incorrect bounds, as coqchk will reject any flawed certificates claiming unsafe bounds.

We chose to implement POET in Python because of our familiarity with the language and its convenient facilities for basic file handling and text manipulation. By design, a foundational RTA does not have to be verified itself, so the fact that Python is notoriously difficult to verify (being a dynamically typed scripting language) did not hinder us.

4.2 The Structure of a Certificate

Listing 2 shows the certificate generated by POET for the second task of the input file of Listing 1. The certificate starts by importing the correct support library in Line 1, which here is the fully-preemptive FP instantiation of aRTA in the Prosa open-source framework.

The actual task-specific content starts in Line 8, which opens a scope for the subsequent declarations. The certificate continues with straightforward declarations of the tasks in the task set (Lines 12–23), the task set (Line 25), the identity of the task under analysis (line 26), and the analysis results (Lines 28 and 29). Recall that L is a bound on the maximum busy-window length and R is the claimed response-time bound. Both L and R are computed by POET; the goal of the certificate is to prove that R is indeed an upper bound on the actual response time of any job of the task under analysis.

The first four lemmas proven by POET are auxiliary facts documenting that the user-provided parameters are valid (e.g., periods and task costs are positive, arrival curves are monotonic, etc.). Listing 2 shows one such example lemma in Lines 36–37. The next lemma in Lines 87 and 88 establishes that the bound on the maximum-busy interval calculated by POET is correct. Specifically, aRTA [8] requires L to be the solution to a fixed-point equation that depends on the scheduling policy and preemption model.

Next, in lines 96-101, the arrival sequence is introduced. Note that no concrete definition is given: the certificate treats the arrival sequence as a *variable*, i.e., the result is proven *for all possible arrival sequences* respecting the hypotheses $H_1, \ldots, H_5$ stated in lines 97-101. The

Listing 2 A simplified version of the certificate generated by POET for the second task of the workload given in Listing 1. All proofs and comments, and some intermediate lemmas and auxiliary definitions, have been omitted for brevity, and some details (e.g., lemma names) have been simplified. The notation `[::a;b;c]` is SSREFLECT syntax for a list comprised of three elements `a`, `b`, and `c`.

```
1   Require prosa.results.fixed_priority.rta.fully_preemptive.
      [...]
8   Section Certificate.
      [...]
12    Let tsk01 := {|
13        task_id := 1;
14        task_cost := 50;
15        task_deadline := 100;
16        task_arrival := Curve_Prefix (220, [::(1, 1);(105, 2)]);
17        task_priority := 2 |}.
18    Let tsk02 := {|
19        task_id := 2;
20        task_cost := 10;
21        task_deadline := 100;
22        task_arrival := Periodic 30;
23        task_priority := 1 |}.
24
25    Let ts := [::tsk01;tsk02].
26    Let tsk := tsk02.
27
28    Let L := 80.
29    Let R := 60.
      [...]
36    Lemma valid_arrival_curve :
37      ∀ task, task ∈ ts → max_arrivals tsk 0 = 0 ∧ monotone leq (max_arrivals tsk).
      [...]
87    Lemma L_fixed_point :
88      total_hep_rbf ts tsk L = L.
      [...]
96    Variable arr_seq : arrival_sequence Job.
97    Hypothesis H1 : arrival_sequence_uniq arr_seq.
98    Hypothesis H2 : all_jobs_from_taskset arr_seq ts.
99    Hypothesis H3 : arrivals_valid_job_costs arr_seq.
100   Hypothesis H4 : consistent_arrival_times arr_seq.
101   Hypothesis H5 : respects_max_arrivals arr_seq ts.
      [...]
106   Definition sched := uni_schedule arr_seq.
      [...]
126   Definition F_solutions := [::60;40;20].
127
128   Lemma R_is_maximum :
129     ∀ A, is_in_search_space tsk L A →
130       ∃ F, task_rbf tsk (A + 1) + total_ohep_rbf ts tsk (A + F) ≤ A + F ∧ F ≤ R.
      [...]
143   Theorem R_bounds_response_time :
144     task_response_time_bound arr_seq sched tsk R.
145   Proof.
        [...]
148     apply arta_response_time_bound_fp_fp.
        [...]
167   Qed.
      [...]
172   Corollary R_respects_deadlines :
173     task_response_time_bound arr_seq sched tsk R ∧ R ≤ task_deadline tsk.
      [...]
181 End Certificate.
182
183 Section AssumptionLessExample.
184     Definition concrete_arr_seq := concrete_arrival_sequence ts generate_jobs_at.
        [...]
186     Theorem R_bounds_response_time_concrete:
187       task_response_time_bound concrete_arr_seq (sched concrete_arr_seq) tsk R.
        [...]
200 End AssumptionLessExample.
```

hypotheses state that the arrival sequence contains each job only once (H_1) and only contains jobs of tasks in the task set (H_2). Further, each job present in the sequence has a positive cost that does not exceed its task's WCET (H_3) and its position in the arrival sequence is consistent with its arrival time (H_4). Finally, for each task in the task set, the cumulative number of job arrivals is bounded by the arrival curve of the task (H_5). Importantly, the definitions of $H_1, \ldots, H_5$ reside in Prosa, from which the certificate derives its semantics.

In Lines 126–130, the certificate shows that Equation (1) for fully-preemptive FP scheduling, as specified by the user in Listing 1, holds for every A in the search space. To this end, a sequence of solutions – for each A, the corresponding F in Equation (1) – is provided by POET in Line 126. The data-dependent proof script (here omitted) performs a case analysis for each A in the search space and presents the corresponding solution to the proof engine.

The overall correctness claim is stated in Lines 143–167, which states that R is indeed a response-time bound for the task under analysis. Crucially, to prove this fact, POET applies the main aRTA theorem for fully-preemptive FP scheduling [8] in Line 148.

Again, the definition of the predicate `task_response_time_bound` used in Line 144 resides in Prosa and is not specific to (nor in any way influenced by) POET. This is an important point: any formally verified result is useful only as far as the *specification* that is shown to hold is semantically meaningful. POET therefore does *not* provide its own semantic specification. Instead, it delegates the semantic modeling of real-time scheduling entirely to prior work and reuses an established specification, namely Prosa [11].

Finally, the corollary in Lines 172–173 explicitly confirms the desired result: the deadline of the task under analysis is always respected. As a safeguard against accidental omission of the main proof, the corollary repeats the claim from Line 144, which ensures that the corollary cannot be proven without first completing the main proof.

The main certificate ends in Line 181, which closes the scope of the non-instantiated variable declared in Line 96, namely the arrival sequence `arr_seq`. The remaining section in Lines 183–200 is another safeguard: it repeats the main theorem of the certificate (Line 144) in an *assumption-less* context (i.e., without `arr_seq` in scope), for a concretely defined arrival sequence. The purpose of this section is to demonstrate that the hypotheses $H_1, \ldots, H_5$ stated in lines 97-101 are free of contradictions, as we explain in more detail in Section 5.

Overall, the generated certificate resembles the flow of traditional pen-and-paper reasoning and is sufficiently short to be inspected manually (200 lines in total). In particular, all proofs, which are responsible for the bulk of the total line count, can be safely skipped since they are verified by coqchk. Importantly, the certificate is intentionally readable and simple, in the sense that only very little experience with Gallina is required to make sense of it.

5 Trustworthiness of the Procedure

By design, POET itself is not part of the TCB. In particular, any critical bugs in the tool that result in incorrect response-time bounds will not go unnoticed by Coq (which is the TCB) – any attempt to compile and check such corrupted certificates will result in obvious and unmistakable errors. In this section, we thus focus on issues that could lead to *silent failures*: how could auto-generated certificates still be misleading even if they are accepted as valid by the proof checker?

Incomplete proofs. Since POET is not assumed to be correct, it might conceivably generate an incomplete (or, in the extreme case, even completely empty) certificate that Coq would then successfully check: a cleanly truncated (or empty) file does not contain any incorrect proofs and hence does not give the proof checker any reason to reject it. Coq further lets the user *admit* theorems, i.e., to accept them as valid without proof, treating them as axioms.

Though these edge cases could easily be programmatically detected by POET itself, doing so would implicitly turn the tool into a trusted component. For the same reason, POET cannot be in charge of reporting the results of the verification attempt to the user. After the certificates have been generated, POET must not intervene in any way. Therefore, any action that needs to be executed after the creation of the certificates *is handled entirely in the Coq environment*. This includes printing of the results, which is done directly in Coq, and checking that no theorems have been *admitted* (using coqchk).

To entirely eliminate any need to trust POET, the certificates and the output they generate are designed to be human-readable. Although POET is fully automated, the process is transparent and open to human supervision. In particular, a user may **(i)** observe the outputs of the Coq compiler and coqchk to assess whether they succeeded, **(ii)** ensure that the input and the generated certificates match in terms of task set, task parameters, scheduling policy, and preemption model (e.g., the file in Listing 1 with Lines 1–25 of Listing 2), and **(iii)** to ensure that the response-time claim is included in the verified certificate (e.g., check whether the corollary in line 172 of Listing 2 is present).

Finally, the certificates themselves are completely transparent, too: it is easy for a user with Coq expertise to scrutinize (i.e., interactively step through) the complete list of proof steps that lead to the response-time bound. While we do not expect the typical user to inspect certificates that closely, striving for readability and making the certificates generally inviting increases their quality and ensures explainability of the results. Furthermore, it renders them suitable as *evidence of temporal correctness* since a third-party auditor or certification authority can independently study and dissect POET's certificates down to their fundamental definitions (as provided by Prosa) and the axioms of Coq's logic.

Although human supervision is always welcome, only step (i) – carefully reading the analysis outputs, a degree of supervision that *any* RTA tool requires – is essential. The rather hypothetical errors caught by steps (ii) and (iii) are unlikely to manifest accidentally, and are thus less relevant in practice. In any case, none of the steps is onerous, and each can be easily completed without in-depth verification expertise.

Contradictions. A second, more subtle type of error is related to the possibility of contradictory hypotheses in axiomatic theories such as aRTA: conclusions reached in a sound manner from contradictory premises may still be incorrect. This potential pitfall has been previously described in-depth by Cerqueira et al. [11]. As it is generally not possible for Coq to detect contradictory hypotheses automatically, it is necessary to establish the absence of contradictions on a case-by-case basis. Concretely, this requires demonstrating that it is possible to instantiate all variables such that all hypotheses are respected simultaneously.

POET's certificates generalize over only one variable, namely the arrival sequence `arr_seq` (Lines 96–101 of Listing 2). Recall from Section 3 that the arrival sequence yields, for any given time t, the sequence of jobs (each with a specific cost) that arrive at time t. The arrival sequence is intentionally left uninstantiated in the certificate's main section (Lines 8–181) as the response-time bound must hold for *any* possible pattern of arrivals that respects the workload constraints (e.g., that jobs arrive periodically).

Nonetheless, to formally prove the absence of contradictions, POET generates for each task, in addition to the general response-time bound (Lines 143–167), a *second* version of the theorem in a separate section devoid of *any* variables or hypotheses (Lines 183–200 in Listing 2). In this *assumption-less example*, the response-time bound is proven once more to hold by applying the general result to a concrete arrival sequence with fixed job costs, which establishes that the general result does not make any contradictory assumptions. Technically,

in Coq terminology, the type of the main result is shown to be inhabited. At this point, as no variable or hypothesis is in scope, only a soundness flaw in coqchk (which is part of the TCB) could allow an incorrect result to be proved.

To establish that the main result's hypotheses are contradiction-free, it actually suffices to instantiate *any* valid arrival sequence, including pathological ones in which no job ever arrives. However, from a user's perspective, it is more confidence-inspiring to use a nontrivial workload. POET therefore generates an arrival sequence that, at any time, greedily maximizes the number of arrivals of each task and their costs while respecting all workload constraints.

To summarize, neither POET nor the underlying RTA are part of the TCB, and hence do not have to be trusted, because **(1)** the results of the analysis are communicated from within the Coq environment (and not by POET itself), **(2)** any program defects in POET or flaws in the underlying RTA that lead to incorrect response-time bounds will be caught by Coq (the proof checker would fail), **(3)** the remote chance of mismatching assumptions in the certificate (e.g., wrong task parameters or a change in scheduling policy) or certificate truncation are immediately obvious to lightweight human supervision since POET's certificates are short and designed for readability, and **(4)** the risk of contradictory hypotheses is mitigated by the inclusion of an assumption-less example that exercises the general bound in a verified manner.

6 Supporting Arbitrary Arrival Curves

A major challenge affecting POET is the *representation* and *extrapolation* of arbitrary arrival-curve prefixes, and the fast computation of the aRTA search space $\mathcal{A}_i$ (recall Equation (2)), which is closely linked to it. To reiterate, support for arbitrary arrival curves in POET is desirable due to the flexibility it affords, enabling support for a wide range of real-world arrival processes that are neither perfectly periodic nor described well with a single, scalar minimum inter-arrival time. Fortunately, aRTA already supports arbitrary arrival curves [8], but only at an ideal mathematical, non-instantiated level (i.e., aRTA's arrival curves are functions on an infinite domain, and not some finite representation thereof). In contrast, POET needs to compactly represent and efficiently compute with *concretely defined* arrival-curve *prefixes*.

As sketched in Listing 1, POET expects each arrival-curve prefix to be compactly expressed with a *horizon* h and a sparse sequence of m *steps* $s_1, \ldots, s_m$. The horizon h defines the length of the prefix (i.e., the maximum Δ covered). A step $s_k = (\Delta_k, c_k)$ indicates that the arrival curve α_i "takes a step" at Δ_k: $\alpha_i(\Delta_k - \varepsilon) < c_k$ and $\alpha_i(\Delta_k) = c_k$.

Given an arrival-curve prefix with $h < L$, it becomes necessary to **extrapolate** during analysis. Let α^* denote the arrival curve extrapolated from the finite prefix, and let $s(t) \triangleq \max(\{0\} \cup \{c_k \mid 1 \leq k \leq m \ \wedge \ \Delta_k \leq t\})$ denote the result of looking up the number of arrivals in an interval of length $t \leq h$ in the given sequence of steps. Then:

$$\alpha^*(\Delta) \triangleq \lfloor \Delta / h \rfloor \cdot s(h) + s(\Delta \bmod h) \tag{3}$$

This choice represents a trade-off between the speed of extrapolation and analysis precision. Equation (3) does not guarantee an optimal extrapolation – depending on the given prefix, an extrapolation exploiting the arrival curve function's sub-additivity may be less pessimistic. Nonetheless, it provides the key advantages of being simple and fast to compute. In fact, each time the proof engine has to evaluate Equation (1), several arrival curves are evaluated. Since traversing the aRTA search space is one of the most expensive steps of certificate checking (as we show in Section 8), it is desirable to keep Equation (3) as simple as possible.

As shown in Listing 1, POET has built-in support for **periodic and sporadic tasks**. However, as a task τ_i with period or minimum inter-arrival time T_i can be easily described with an arrival curve $\alpha_i(\Delta) \triangleq \lceil \Delta / T_i \rceil$, POET's certificates work *exclusively* with arrival-curve prefixes. Specifically, a task with period or minimum inter-arrival time T_i is internally

represented with an arrival-curve prefix with horizon $h = T_i$ and a single step $s_1 = (1, 1)$. This conversion is lossless: in this important special case, Equation (3) does not introduce any pessimism since $\alpha^*(\Delta) = \lfloor \Delta/h \rfloor \cdot s(h) + s(\Delta \bmod h) = \lfloor \Delta/T_i \rfloor \cdot 1 + s(\Delta \bmod T_i)$, which is equal to $\frac{\Delta}{T_i} + 0 = \alpha_i(\Delta)$ if T_i divides Δ and equal to $\lfloor \Delta/T_i \rfloor + 1 = \lceil \Delta/T_i \rceil = \alpha_i(\Delta)$ otherwise.

Critically, the automatic conversion to arrival-curve prefixes is performed in POET's Coq libraries, and not in the Python part, hence introducing no verification gap whatsoever while still freeing the user from having to think about this detail.

As already mentioned, the **computation of the search space** is the primary bottleneck of POET. Recall from Equation (2) that aRTA defines the search space $\mathcal{A}_i$ for each task τ_i as the set of points at which the interference bound function IBF_i changes in value. Although aRTA correctly anticipates that a sparse search space will be necessary for practical use [8], it does *not* provide a way to compute it. However, the search space *must be computed by the proof engine* to validate POET's fixed-point solutions (e.g., in Line 126 of Listing 2).

To this end, we exploit the structure of Equation (3). Since the function IBF_i depends on the concrete scheduling policy, we discuss FP and EDF scheduling in sequence.

In the case of FP scheduling, Equation (2) reduces to the set of points at which the *extrapolated* arrival curve of the task under analysis τ_i changes in value in the interval $[0, L)$: $\mathcal{A}_i^{FP} = \{A \mid A < L \wedge \alpha_i^*(A) \neq \alpha_i^*(A + \varepsilon)\}$ [8]. As shown in POET's support libraries (which are checked as part of every certificate), $\mathcal{A}_i^{FP}$ can be easily over-approximated by repeatedly concatenating task τ_i's list of steps $s_1, \ldots, s_m$.

$$\mathcal{A}_i^{FP} \subseteq \{lh + s_1, \ldots, lh + s_m \mid 0 \leq l \leq \lfloor L/h \rfloor\} \tag{4}$$

Equation (4) defines a superset of the actual search space because some of the included points may exceed L (e.g., if $\lfloor L/h \rfloor \cdot h + s_m > L$). This does not affect the analysis's correctness as there is no harm in evaluating superfluous offsets [8].

In the case of EDF, the search space is far more complex since it involves every task in the task set: $\mathcal{A}_i^{EDF} = \{A \mid A < L \wedge \exists\, \tau_j \in \tau,\ \alpha_i^*(A + D_i - D_j) \neq \alpha_i^*(A + \varepsilon + D_i - D_j)\}$ [8]. For ease of computation, we decompose $\mathcal{A}_i^{EDF}$ on a per-task basis such that $\mathcal{A}_i^{EDF} = \bigcup_{\tau_j \in \tau} \mathcal{A}_{i,j}^{EDF}$, where $\mathcal{A}_{i,j}^{EDF} = \{A \mid A < L \wedge \alpha_i^*(A + D_i - D_j) \neq \alpha_i^*(A + \varepsilon + D_i - D_j)\}$. Note that $D_i - D_j$ is a constant on both sides of the defining inequality. If removed from both sides, we obtain exactly $\mathcal{A}_i^{FP}$; that is, $\mathcal{A}_{i,j}^{EDF}$ can be thought of as $\mathcal{A}_i^{FP}$ *shifted* by $D_j - D_i$ time units. Analogously to Equation (4), POET thus computes $\mathcal{A}_{i,j}^{EDF}$ as

$$\mathcal{A}_{i,j}^{EDF} \subseteq \{lh + s_1 + D_j - D_i, \ldots, lh + s_m + D_j - D_i\} \tag{5}$$

for $0 \leq l \leq \lfloor L/h \rfloor$. Again, this is an over-approximation; in particular, any negative points are simply ignored. Finally, Equations (4) and (5) demonstrate the key benefit of the fast extrapolation rule in Equation (3): the search space can be over-approximated easily and relatively quickly.

7 Scalability of the Certification Procedure

One of the major challenges we encountered during the development of POET is the poor computational performance of ssreflect's standard number representation. Without a working solution, this seemingly small detail can jeopardize the entire idea of foundational RTA – for which computation in the proof engine is essential.

POET's certificates depend on Prosa, and therefore implicitly on ssreflect, which uses a *unary* representation of natural numbers. The use of a unary number representation has clear advantages when writing proofs, as it simplifies inductive reasoning and case

Listing 3 Two working scripts that prove the same lemma. However, (a) is impacted by a large unary number, whereas in (b) no expensive computation is triggered in the proof.

(a)

```
Let x := 10000000000000000000.

Lemma Ex1 :
  forall y,
    1 + x − x + y = 1 + y.
Proof.
  by move⇒ y; vm_compute.
Qed.
```

(b)

```
Let x := 10000000000000000000.

Lemma cancel_n : forall n, 1 + n − n = 1.
Proof. by induction n. Qed.

Lemma Ex2 : forall y, 1 + x − x + y = 1 + y.
Proof. by move⇒ y; rewrite cancel_n.
Qed.
```

analyses. However, even a moderately large number (like one billion), when encoded in unary, takes considerable time to instantiate and can easily trigger stack overflows in the Coq compiler. Roughly, on our test machine (described in Section 8), the certification process stays somewhat feasible despite the unary representation (i.e., it takes "only" several hours) as long as costs, periods, and deadlines remain in the order of 10^6 (e.g., microseconds scale).

For instance, consider the task set in Listing 1, which uses milliseconds as its unit of measure. If we instead express all parameters in microseconds (i.e., multiply worst-case execution times, periods, and deadlines by 10^3), we do not, in principle, change the RTA's complexity. However, when using ssreflect's number representation, this has an enormous impact on performance, pushing the certification time from less than three seconds to around 15 minutes. A unary representation hence renders it impossible to express task parameters in nanoseconds or as processor cycle counts (and would thus severely limit POET's practicality).

To illustrate the challenge posed by unary calculations, consider the trivial example in Listing 3a. Despite its simplicity, and despite the fact that the given proof is correct, lemma Ex_1 cannot be compiled by Coq in practice (doing so would require an unreasonable amount of time and memory). More precisely, to reduce $1 + x - x$ by computation, Coq needs to build a unary representation of x, which is prohibitively expensive due to the magnitude of x.

Fortunately, it is also possible to proceed in another way that avoids the cost of unary computation, as shown in Listing 3b, where the magnitude of x does not impact the compilation time. In the proof of lemma Ex_2, the auxiliary lemma *cancel_n* is used to rewrite $1 + x - x + y$ into $1 + y$. Since *cancel_n* is valid for any number n, $1 + x - x$ is never evaluated, and the lemma is compiled by Coq virtually instantaneously.

The example shows that the proof engine is able to operate in the presence of large numbers, provided that no computation is triggered. Unfortunately, this principle alone is not enough for POET: any time a proof depends on the actual value of a number, a computation is still necessary. Unfortunately, it is inherent in the nature of a response-time bound to depend on specific, large numbers, which hence manifest in its proof.

The immediate solution to the scalable-computations problem would be to change the employed number representation entirely to binary. This is indeed possible in principle, but would imply rewriting most of Prosa and the parts of ssreflect it depends on (with substantial additional complexity due to the binary representation), and hence is undesirable.

Instead, POET's certificates employ a hybrid solution: every result is proven using the standard, unary definitions and lemmas provided by Prosa and ssreflect, but large computations are *isolated* and performed using a binary representation. This is easier said than done: since the aRTA library [8] expects unary numbers, trying to apply its theorems to binary-encoded inputs would result in type-checking errors.

Therefore, **(a)** a binary version of each unavoidable computation was implemented and integrated into POET's Coq support libraries, and **(b)** each such alternative implementation was related to its unary counterpart via an additional corpus of proofs so that the computed results can be *substituted* into the proof. More technically, for the proof engine to accept a rewriting step (i.e., to perform a substitution), the two involved functions must be proven to be extensionally equivalent. To this end, we applied CoqEAL [14], a Coq proof framework for changes in data representation.

To compute a unary-numbers function using a binary-numbers counterpart, CoqEAL requires proving a so-called *refinement*. Given the sets of unary numbers $\mathbb{N}_1$ and binary numbers $\mathbb{N}_2$, consider a conversion function $\Phi(x) : \mathbb{N}_2 \to \mathbb{N}_1$ that, given a binary number, yields its unary equivalent. Further, consider a unary-numbers predicate $p_1(x) : \mathbb{N}_1 \to \mathbb{B}$ and its binary-numbers counterpart $p_2(x) : \mathbb{N}_2 \to \mathbb{B}$. For the purposes of POET, a refinement can be seen as a proof that, for any binary number x, $p_1(\Phi(x)) = p_2(x)$. Analogous arguments can be made for predicates with multiple parameters and for predicates with higher-order parameter types based on unary numbers, such as lists and tuples over $\mathbb{N}_1$. With a refinement in place, a proof step triggering a computation of p_1 can be replaced, similarly as in lemma $Ex2$ in Listing 3b, with one involving a computation of p_2, which can be performed quickly.

In conclusion, for each numeric function computed by Coq when compiling POET's certificates, **(1)** an equivalent function defined on binary numbers was implemented in the support libraries, and **(2)** a refinement relating the two functions was proven. The total support code related to refinements is roughly 1800 LOC of definitions, proofs, and tactics, representing around 40% of the entire POET support code: enabling binary computations comes with an extra development cost. However, the switch to binary representation dramatically impacts runtime and memory needs and hence is essential.

Case in point, consider once again the task set in Listing 1 (expressed in milliseconds). With the binary representation in place, the total certification time is around five seconds on our testing machine (i.e., slightly slower than before), but stays roughly the same irrespective of whether task parameters are given in microseconds or nanoseconds.

Finally, it is worth emphasizing that CoqEAL is *not* part of the TCB since, as a Coq library, it is itself subject to full verification by coqchk. Therefore, speeding up the computation by translating to a binary encoding does not introduce any verification gaps.

8 Empirical Evaluation

To assess POET's runtime characteristics, we conducted an empirical evaluation using synthetic workloads, which we generated as follows. For a given number of tasks n and total utilization u, we used the Dirichlet-Rescale algorithm [27] to draw n utilizations values $u_1, \ldots, u_n$ summing to u. To exercise POET's versatility, we considered five different types of workloads: **(i)** sporadic workloads as commonly encountered in automotive systems, **(ii)** sporadic workloads with a log-uniform distribution of minimum inter-arrival times, **(iii)** sporadic workloads subject to job bursts, **(iv)** sporadic workloads subject to jitter, and **(v)** sporadic arrivals distributed according to a Poisson distribution. This selection covers a wide range from relatively well-structured arrival processes to less regular ones. Types (i)–(iv) equivalently represent periodic workloads since the sporadic task model generalizes the periodic one. We hence use the terms "inter-arrival time" and "period" interchangeably.

The different arrival bounds were generated as follows. Let τ_i denote the task for which an arrival bound is being generated. In the case of (i), the period T_i was chosen uniformly at random from $P \triangleq \{1, 2, 5, 10, 20, 50, 100, 200, 500, 1000\}$ms, a set of periods commonly

encountered in automotive systems [33]. For (ii), T_i was drawn log-uniformly from $[1, 100]$ms. For (iii), a bursty arrival process was defined by the maximum number of jobs in a burst $b_i = 4$ and the randomly chosen minimum intra-burst inter-arrival time $T_i^{in} \in [1, 100]$µs and the minimum inter-burst inter-arrival time $T_i \in P$. In the case of (iv), τ_i's period T_i was chosen from P and the maximum jitter was drawn uniformly at random from $[0.1T_i, 3T_i]$. Lastly, for (v), we considered a Poisson process with a mean arrival rate $r_i \triangleq \frac{1}{T_i}$, where T_i was drawn uniformly at random from P. For any m and Δ, a Poisson process has a non-zero (but rapidly diminishing) probability of yielding m arrivals in an interval of length Δ. To convert this to an arrival curve, we defined $\alpha_i(\Delta)$ such that, for any Δ, the probability of observing more than $\alpha_i(\Delta)$ job releases in an interval of length Δ is less than 10^{-3}.

Next, the generated arrival bounds were encoded in POET's input format. Cases (i) and (ii) are trivial as POET natively supports periodic and sporadic tasks. Cases (iii)–(v) require the arrival process to be expressed as a finite arrival-curve prefix, as discussed in Section 6. Recall that task τ_i's finite arrival-curve prefix consists of two parts: a horizon h_i and a sequence of m_i steps. In the case of (iii), bursty arrivals, the arrival bound can be encoded in a lossless manner by setting $h_i \triangleq T_i$ and defining $m_i = b_i$ equidistant steps with a separation of T_i^{in}. For both (iv) and (v), we simply truncated the arrival curve by choosing the maximum h_i containing at most $m_i = k$ steps, where k differed across experiments.

Finally, the relative deadline D_i was drawn from $[0.3T_i, 3T_i]$, and the task's WCET C_i was set to $\lceil u_i \cdot p_i \rceil$ in cases (i) and (ii), and to $\lim_{t\to\infty} \lceil (u_i \cdot t)/\alpha(t) \rceil$ in cases (iii)–(v). All parameters were given to POET in nanosecond resolution (i.e., $\varepsilon = 1$ns).

For each considered n, we let the total utilization u range from 0.5 to 0.9 in steps of 0.1. We evaluated each supported policy: fully-preemptive FP (FP-FP), fully non-preemptive FP (NP-FP), fully-preemptive EDF (FP-EDF), and fully non-preemptive EDF (NP-EDF). Under FP scheduling, tasks were assigned rate-monotonic priorities. We ran all experiments on a Linux host with two 2.50 GHz Intel Xeon "Platinum 8180" processors and 394 GiB RAM.

In the **first experiment**, we focused on classic sporadic tasks (i.e., type-(i) tasks). We varied the number of tasks n from 5 to 50 in steps of 5 and, for each combination of scheduling and preemption policy, generated 10 task sets for each cardinality (×10) and utilization (×5), resulting in 500 task sets per policy and 2000 in total. For each workload, we measured the end-to-end runtime of the entire workflow depicted in Figure 1 (including POET, the Coq compiler, and coqchk) while running *sequentially* on a single core.

As can be seen in Figure 2a, these workloads can be easily certified by POET. There is a clear difference between EDF and FP analyses, but no major difference between the two preemption policies. Small workloads ($n \leq 10$) are typically solved in seconds for both EDF and FP; larger workloads take significantly more time, with FP analyses being clearly faster. For $n = 50$, the mean runtime per task set was 4.2 minutes under FP-FP, 4.8 minutes under NP-FP, 119 minutes under FP-EDF, and 122 minutes under NP-EDF. Overall, across all cardinalities, the mean runtime per task set was 2.1 minutes under FP-FP, 2.4 minutes under NP-FP, 38 minutes under FP-EDF, and 35 minutes under NP-EDF.

The growth in certification time dependent on n evident in Figure 2a is due to two factors: adding a task obviously increases the number of certificates that must be generated and checked, while also increasing the complexity of the certificates of all prior tasks.

In the **second experiment**, we challenged POET with more demanding workloads by randomly choosing each task's arrival model among types (i)-(v). The resulting workloads are hence less structured and more complex, characterized by many nontrivial arrival curves.

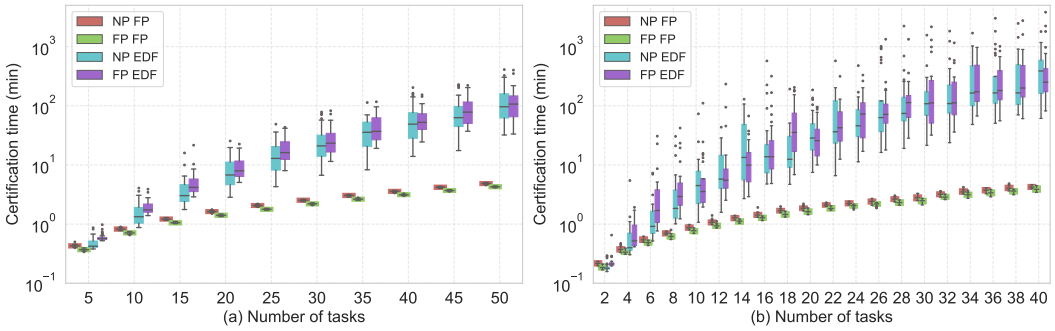

Figure 2 The end-to-end runtime of POET and Coq *vs.* the number of tasks in **(a)** for classic sporadic tasks (type (i)) and **(b)** mixed workloads (types (i)–(v)). Boxes range from the first to the third quartile; whiskers extend to 1.5 times the inter-quartile range (IQR).

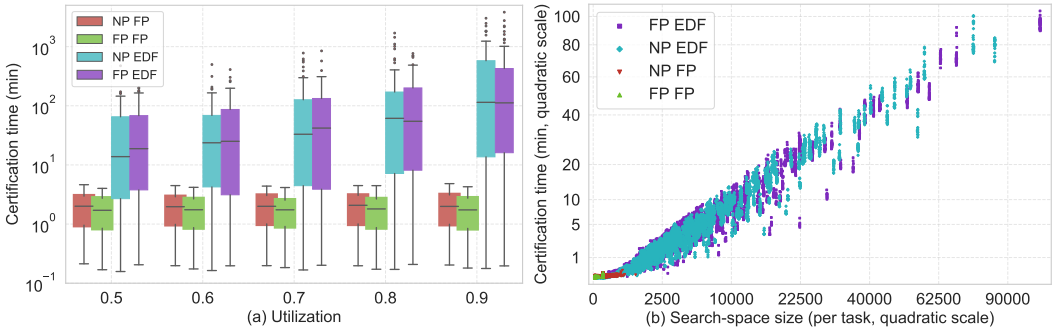

Figure 3 Based on the same data set as shown in Figure 2b, **(a)** the total certification runtime *vs.* total utilization and **(b)** the *single-task* certification runtime *vs.* the task's search-space size; note the quadratic scale of both axes in inset (b).

In this experiment, we varied the number of tasks n from 2 to 40 in steps of 2. For each combination of scheduling and preemption policy, we generated 5 task sets for each cardinality ($\times 20$) and utilization ($\times 5$), resulting in 500 task sets per policy and 2000 in total. The maximum length of arrival-curve prefixes was fixed to $k = 50$. As in the first experiment, we measured the end-to-end runtime of the entire workflow running sequentially on one core.

The results of the second experiment (depicted in Figure 2b) exhibit similar trends as in Figure 2a, but with an (expected) steeper growth in certification time. For $n = 40$, the mean runtime per task set was 3.9 minutes under FP-FP, 4.2 minutes under NP-FP, 568 minutes under FP-EDF, and 466 minutes under NP-EDF. Overall, across all cardinalities, the mean runtime per task set was 1.8 minutes under FP-FP, 2.1 minutes under NP-FP, 140 minutes under FP-EDF, and 145 minutes under NP-EDF. Clearly, POET's scalability is substantially reduced for such challenging workloads, but it still manages to certify workloads of nontrivial size. Notably, the most time-consuming aspect of POET, certificate checking, can be easily parallelized since task certificates are independent of each other. High runtimes for large n can thus be alleviated by letting POET run coqchk in parallel on up to n cores.

Next, to better understand the primary drivers of runtime growth, we plotted the data collected in the second experiment in two additional ways as shown in Figure 3. First, we explored the impact of the task set's **total utilization**. As can be seen in Figure 3a, in contrast to task set cardinality, total utilization influences certification time only slightly, and

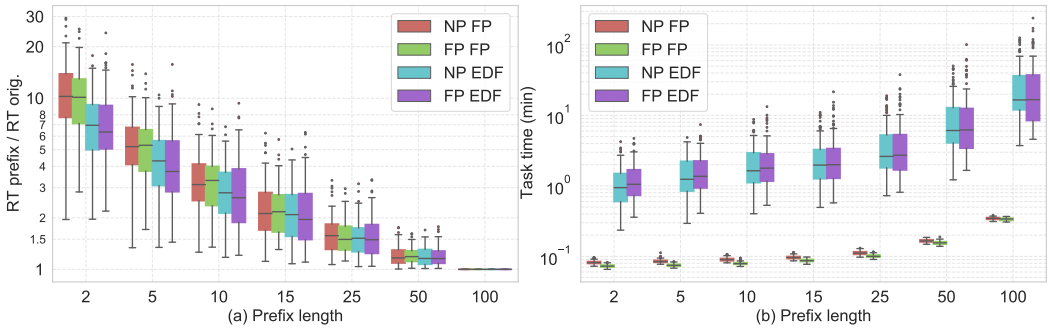

Figure 4 Number of steps in the arrival-curve prefix *vs.* **(a)** certification time of the most-preempted task and **(b)** the increase in response time for arrival bounds of types (iv) and (v).

only in the case of EDF. In fact, while high utilization can impact the maximum busy-window length in pathological cases, for the considered workloads, it did not make a significant difference in the expected case.

The major driver of certification runtime is instead the **search-space size**. In fact, as can be seen in Figure 3b, there is a clear linear correlation between the size of the search space of a task and its certification time. Note that, in contrast to the prior figures, Figure 3b shows the individual *per-task* certification time relative to the size of its search space. Certification finished in at most 5 minutes for most of the tasks (84%), and in less than 10 minutes for 92% of the tasks regardless of the preemption model and scheduling policy, peaking at 104 minutes under FP-EDF for a task with a search space containing 104547 points.

The search space and certification time of FP-FP and NP-FP are clustered in the bottom-left part of the plot. In contrast, FP-EDF and NP-EDF cover the entirety of the space, which is a result of the fact that the EDF search space is inherently larger and dependent on all tasks, as evident from Equations (4) and (5). In fact, 99% of tasks scheduled under an FP policy had a search space containing fewer than 104 points, whereas the 99^{th} percentile under an EDF policy is 57470 points. Hence, task sets usually take much less time to be verified under an FP policy, which explains the gap between the EDF and FP policies in Figure 2.

Finally, we performed a **third experiment** with the goal of evaluating the role of the number of steps in arrival-curve prefixes. For this experiment, we fixed the number of tasks to $n = 25$ and varied instead the prefix size of all tasks $k \in \{2, 5, 10, 15, 25, 50, 100\}$. For each combination of scheduling and preemption policy, we generated 10 task sets for each k ($\times 7$) and utilization ($\times 5$), considering task sets composed homogeneously of tasks with either arrival model (iv) or (v) ($\times 2$), resulting in 700 task sets per policy and 2800 in total. To arrive at a given prefix size k, we always started from an arrival curve with $m_i = 100$ steps, and then iteratively merged steps to gradually shrink the prefix. To maximize the effect of the loss of information that results from the shortening of the prefix, we focused on the *lowest-priority* task under FP scheduling, and on the task with the *largest deadline* under EDF. In both cases, we refer to this task as the *most-delayed* task.

First, we investigated the impact of k on RTA **accuracy**. Figure 4a shows the relative increase in the response-time bound of the most-delayed task relative to its *baseline response-time bound*, which was obtained using the full prefix with 100 steps. By reducing k from 100 to 2, the response-time bound for the most-delayed task reaches a staggering 20× increase, which (as expected) drops quickly as k is increased.

The results in Figure 4a should be seen in context of the corresponding **task certification times** that, as can be seen in Figure 4b, grow substantially with increasing k (note the log scale). EDF-scheduled task sets are once again substantially more expensive to analyze.

Overall, Figure 4 shows that, for complex workloads with irregular arrival processes, the number of steps of the arrival-curve prefix represents a major trade-off between certification time and analysis accuracy. It should be noted, however, that POET always managed to complete the certification, even for $k = 100$ steps. Furthermore, it is important to realize that full certification is expected only at the *final* stage of development – at which point runtimes in the order of several hours can be acceptable – and not during day-to-day development.

Overall, we consider POET to be a successful proof of concept. While the computational efficiency of the underlying proof assistant and libraries is (as expected) a major bottleneck, the experiments overall showed POET to cope with complex workloads and to scale to practical workload sizes. Foundational RTA is thus not only theoretically desirable, but also practical, and therefore worthy of further study, extension, and optimization.

9 Related Work

As already discussed in Section 2, POET draws inspiration and adopts terminology from Appel's classic work on foundational proof-carrying code [3], which has been highly influential. Since its publication two decades ago, it has been widely adopted in the area of program verification [9, 32], and continues to play a major role in state-of-the-art verification tools [41].

POET is closely linked to Prosa [11]. While Prosa is the to-date largest machine-checked framework for real-time schedulability analysis – and presently the only one with an implementation of aRTA [8] – it is neither the first nor the only attempt in this direction [7, 20, 21, 46, 51]. It is worth noting that a foundational tool like POET is not inherently related to Coq: some of the just-cited papers make use of different proof assistants, namely Nqthm [44], PVS [45], and Isabelle/HOL [43]. Though some are more suited than others, conceptually speaking, a foundational approach could be realized with any of these, and each would likely pose different challenges and trade-offs. In particular, the Lean proof assistant [19] is a modern alternative to Coq based on the same underlying logic [15, 16, 39]; Lean would likely be a viable alternative for use in foundational RTA tools.

Closest to POET in terms of objectives and approach are Mabille et al.'s results validator for network calculus [36] (using Isabelle/HOL) and Fradet et al.'s results validator CertiCAN [23] (using Coq and Prosa) for the CAN RTA implemented in RTaW-Pegase [6]. In contrast to POET, which generates proofs as explainable evidence, but is intentionally left unverified, these tools do not generate proofs nor other evidence, but are themselves verified.

Finally, an alternative way to approach the schedulability analysis problem is to validate the correctness of a bound with model-checking techniques. In this approach, the system under analysis is first reduced to a model comprising a network of discrete automata with timed semantics. Then, a model checker explores the state space of the model with the objective of covering every possible trace, including those in which the worst-case response time of a task is experienced. Generally speaking, model-checking has been a highly successful technique as shown by tools like Uppaal [5] and Kronos [50] (both based on timed automata [1]) as well as HyTech [30] (based on linear hybrid automata [2]). Compared to foundational RTA, model checking is an orthogonal technique with fundamentally different trade-offs and challenges. As a major advantage, a model checker requires no RTA theory to be developed or verified, since the worst case response-time is implicitly found during exploration of the state space. However, when compared to foundational RTA, model-checking requires a significantly larger, much more complex TCB. The reason is that practical model-checkers are typically large, nontrivial pieces of software that, due to model checking's well-known state-space explosion problem and the resulting scalability challenges (e.g., [48]), have large incentives to

be heavily optimized. This naturally leads to the development of advanced techniques to prune search trees [5], speed up computations via statistical techniques [17], and hardware acceleration [4]. Each optimization technique increases the size of the TCB and arguably renders it more fragile. While Wimmer and Lammich [47] developed a verified unreachability certificate checker for timed automata, they reported it to be an order of magnitude slower and significantly more memory intensive than the state-of-the-art tool Uppaal, which limits its practical use in the schedulability analysis of realistically sized task sets.

Regarding the explainability of results, model-checkers are capable of providing a counterexample leading to a worst-case scenario (e.g., a deadline is violated), but typically do not produce *evidence* that a property is *not* violated. Foundational RTA tools yield exactly the opposite: they do not give counterexamples, but do provide a sequence of machine-checked proofs that show the response-time bounds to be correct. In conclusion, both model checking and proof automation are important research directions, with diverse advantages and limitations. From a tool user's point of view, currently neither clearly dominates the other.

10 Conclusion

We have proposed foundational RTA as a means to obtain explainable, trustworthy evidence of temporal correctness and discussed the design and implementation of POET, the first foundational RTA tool. A foundational RTA produces proof-carrying response-time bounds that can be independently verified by a proof checker. Consequently, a foundational RTA tool does not have to be trusted and can be developed like any other application, while *its results* are trustworthy: fully explainable and verifiably correct.

While POET is an important first step demonstrating feasibility of the approach, for practical use, it will be necessary to go beyond ideal uniprocessor systems. In particular, it would be desirable t extend POET to more complex workloads (e.g., synchronization and precedence constraints), to more realistic system models (e.g., scheduling overheads), and to multiprocessor platforms (e.g., semi-partitioned scheduling).

References

1 Rajeev Alur and David L Dill. A theory of timed automata. *Theoretical computer science*, 126(2):183–235, 1994.

2 Rajeev Alur, Thomas A Henzinger, and Pei-Hsin Ho. Automatic symbolic verification of embedded systems. *IEEE Transactions on Software Engineering*, 22(3):181–201, 1996.

3 Andrew W Appel. Foundational proof-carrying code. In *Proceedings 16th Annual IEEE Symposium on Logic in Computer Science*, pages 247–256. IEEE, 2001.

4 Jiri Barnat, Luboš Brim, Milan Ceška, and Tomáš Lamr. CUDA accelerated LTL model checking. In *2009 15th International Conference on Parallel and Distributed Systems*, pages 34–41. IEEE, 2009.

5 Gerd Behrmann, Alexandre David, Kim G Larsen, John Hakansson, Paul Petterson, Wang Yi, and Martijn Hendriks. UPPAAL 4.0. In *Proceedings of the 3rd international conference on the Quantitative Evaluation of Systems*, pages 125–126, 2006.

6 Marc Boyer, Jorn Migge, and Marc Fumey. PEGASE-a robust and efficient tool for worst-case network traversal time evaluation on AFDX. Technical report, SAE Technical Paper, 2011.

7 Marc Boyer, Pierre Roux, Hugo Daigmorte, and David Puechmaille. A residual service curve of rate-latency server used by sporadic flows computable in quadratic time for network calculus. In *33rd Euromicro Conference on Real-Time Systems (ECRTS 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.

8 Sergey Bozhko and Björn B. Brandenburg. Abstract Response-Time Analysis: A Formal Foundation for the Busy-Window Principle. In *32nd Euromicro Conference on Real-Time Systems (ECRTS'20), July 7-10, 2020, Virtual Conference*, 2020.

9 Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W Appel. VST-Floyd: A separation logic tool to verify correctness of C programs. *Journal of Automated Reasoning*, 61(1):367–422, 2018.

10 Felipe Cerqueira, Geoffrey Nelissen, and Björn B Brandenburg. On strong and weak sustainability, with an application to self-suspending real-time tasks. In *30th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 26–1, 2018.

11 Felipe Cerqueira, Felix Stutz, and Björn B Brandenburg. PROSA: A case for readable mechanized schedulability analysis. In *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 273–284. IEEE, 2016.

12 Jian-Jia Chen and Björn B Brandenburg. A note on the period enforcer algorithm for self-suspending tasks. *Leibniz Transactions on Embedded Systems*, 4(1):01–1, 2017.

13 Jian-Jia Chen, Geoffrey Nelissen, Wen-Hung Huang, Maolin Yang, Björn Brandenburg, Konstantinos Bletsas, Cong Liu, Pascal Richard, Frédéric Ridouard, Neil Audsley, et al. Many suspensions, many problems: a review of self-suspending tasks in real-time systems. *Real-Time Systems*, 55(1), 2019.

14 Cyril Cohen, Maxime Dénès, and Anders Mörtberg. Refinements for free! In *International Conference on Certified Programs and Proofs*, pages 147–162. Springer, 2013.

15 Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76(2):95–120, 1988. doi:10.1016/0890-5401(88)90005-3.

16 Thierry Coquand and Christine Paulin. Inductively defined types. In *International Conference on Computer Logic*, pages 50–66. Springer, 1988.

17 Alexandre David, Kim G Larsen, Axel Legay, Marius Mikučionis, and Zheng Wang. Time for statistical model checking of real-time systems. In *International Conference on Computer Aided Verification*, pages 349–355. Springer, 2011.

18 Robert I Davis, Alan Burns, Reinder J Bril, and Johan J Lukkien. Controller Area Network (CAN) schedulability analysis: Refuted, revisited and revised. *Real-Time Systems*, 35(3), 2007.

19 Leonardo De Moura and Sebastian Ullrich. The Lean 4 theorem prover and programming language. In *International Conference on Automated Deduction*, pages 625–635. Springer, 2021.

20 Daniel de Rauglaudre. Vérification formelle de conditions d'ordonnancabilité de tâches temps réel périodiques strictes. In *JFLA-Journées Francophones des Langages Applicatifs-2012*, 2012.

21 Bruno Dutertre. The priority ceiling protocol: formalization and analysis using PVS. In *Proceedings of the 21st IEEE Conference on Real-Time Systems Symposium (RTSS)*, pages 151–160, 1999.

22 Pascal Fradet, Xiaojie Guo, Jean-François Monin, and Sophie Quinton. A generalized digraph model for expressing dependencies. In *Proceedings of the 26th International Conference on Real-Time Networks and Systems*, pages 72–82, 2018.

23 Pascal Fradet, Xiaojie Guo, Jean-François Monin, and Sophie Quinton. CertiCAN: A tool for the Coq certification of CAN analysis results. In *RTAS*, 2019.

24 Pascal Fradet, Maxime Lesourd, Jean-François Monin, and Sophie Quinton. A generic coq proof of typical worst-case analysis. In *2018 IEEE Real-Time Systems Symposium (RTSS)*, pages 218–229. IEEE, 2018.

25 Georges Gonthier. Formal proof–the four-color theorem. *Notices of the AMS*, 55(11):1382–1393, 2008.

26 Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi, and Laurent Théry. A machine-checked proof of the odd order theorem. In *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*, Lecture Notes in Computer Science, pages 163–179, 2013.

27 David Griffin, Iain Bate, and Robert I. Davis. Generating utilization vectors for the systematic evaluation of schedulability tests. In *41st IEEE Real-Time Systems Symposium (RTSS'20), December 1-4, Houston, TX, USA*, pages 76–88. IEEE Computer Society, 2020.

28 Arpan Gujarati, Felipe Cerqueira, Björn B Brandenburg, and Geoffrey Nelissen. Correspondence article: a correction of the reduction-based schedulability analysis for apa scheduling. *Real-Time Systems*, 55(1):136–143, 2019.

29 Mario Günzel and Jian-Jia Chen. A note on slack enforcement mechanisms for self-suspending tasks. *Real-Time Systems*, pages 1–10, 2021.

30 Thomas A Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. HyTech: A model checker for hybrid systems. *International Journal on Software Tools for Technology Transfer*, 1(1-2):110–122, 1997.

31 Leandro Soares Indrusiak, Alan Burns, and Borislav Nikolic. Analysis of buffering effects on hard real-time priority-preemptive wormhole networks. *arXiv preprint arXiv:1606.02942*, 2016.

32 Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. seL4: Formal verification of an OS kernel. In *SOSP*, 2009.

33 Simon Kramer, Dirk Ziegenbein, and Arne Hamann. Real world automotive benchmarks for free. In *6th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, 2015.

34 Karthik Lakshmanan, Dionisio de Niz, and Ragunathan Rajkumar. Coordinated task scheduling, allocation and synchronization on multiprocessors. In *RTSS*, 2009.

35 Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

36 Etienne Mabille, Marc Boyer, Loïc Fejoz, and Stephan Merz. Towards certifying network calculus. In *ITP*, 2013.

37 Assia Mahboubi and Enrico Tassi. *Mathematical Components*. Zenodo, 2021. `doi:10.5281/zenodo.4457887`.

38 George C Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 106–119, 1997.

39 Christine Paulin-Mohring. Introduction to the calculus of inductive constructions, 2015.

40 Prosa. `http://prosa.mpi-sws.org/`.

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

42 The Coq Proof Assistant. `https://coq.inria.fr`.

43 The Isabelle Proof Assistant. `https://isabelle.in.tum.de/`.

44 The Nqthm Theorem Prover. `https://www.cs.utexas.edu/users/moore/best-ideas/nqthm/index.html`.

45 The PVS Proof Assistant. `https://pvs.csl.sri.com/`.

46 Matthew Wilding. A machine-checked proof of the optimality of a real-time scheduling policy. In *International Conference on Computer Aided Verification*, pages 369–378. Springer, 1998.

47 Simon Wimmer and Peter Lammich. Verified model checking of timed automata. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 61–78. Springer, 2018.

48 Beyazit Yalcinkaya, Mitra Nasri, and Björn B Brandenburg. An exact schedulability test for non-preemptive self-suspending real-time tasks. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1228–1233. IEEE, 2019.

49 Maolin Yang, Jian-Jia Chen, and Wen-Hung Huang. A misconception in blocking time analyses under multiprocessor synchronization protocols. *Real-Time Systems*, 53(2):187–195, 2017.

50 Sergio Yovine. Kronos: A verification tool for real-time systems. *International Journal on Software Tools for Technology Transfer*, 1(1-2):123–133, 1997.

51 Xingyuan Zhang, Christian Urban, and Chunhan Wu. Priority inheritance protocol proved correct. In *International Conference on Interactive Theorem Proving*, pages 217–232. Springer, 2012.

52 Quan Zhou, Jihua Huang, Jianjun Li, and Zhi Li. Response time analysis for hybrid task sets under fixed priority scheduling. In *Proceedings of the IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 108–120. IEEE Computer Society, 2022.