Compositional Verification of Embedded Real-Time Systems

Mohammed Foughali, Pierre-Emmanuel Hladik, Alexander Zuepke

Compositional Verification of Embedded Real-Time Systems

Mohammed Foughali[a], Pierre-Emmanuel Hladik[b], Alexander Zuepke[c]

[a] *Université Paris Cité, CNRS, IRIF, F-75013, Paris, France*
[b] *Nantes Université, CNRS, LS2N, F-44000, Nantes, France*
[c] *Technical University of Munich, D-85748, Garching, Germany*

Abstract

In an embedded real-time system (ERTS), real-time tasks (software) are typically executed on a multicore shared-memory platform (hardware). The number of cores is usually small, contrasted with a larger number of complex tasks that share data to collaborate. Since most ERTSs are safety-critical, it is crucial to rigorously verify their software against various real-time requirements under the actual hardware constraints (concurrent access to data, number of cores). Both the real-time systems and the formal methods communities provide elegant techniques to realize such verification, which nevertheless face major challenges. For instance, model checking (formal methods) suffers from the state-space explosion problem, whereas schedulability analysis (real-time systems) is pessimistic and restricted to simple task models and schedulability properties. In this paper, we propose a scalable and generic approach to formally verify ERTSs. The core contribution is enabling, through joining the forces of both communities, compositional verification to tame the state-space size. To that end, we formalize a realistic ERTS model where tasks are complex with an arbitrary number of jobs and job segments, then show that compositional verification of such model is possible, using a hybrid approach (from both communities), under the state-of-the-art partitioned fixed-priority (P-FP) with limited preemption scheduling algorithm. The approach consists of the following steps, given the above ERTS model and scheduling algorithm. First, we compute fine-grained data sharing overheads for each job segment that reads or writes some data from the shared memory. Second, we generalize an algorithm that, aware of the data sharing overheads, computes an affinity (task-core allocation) guaranteeing the schedulability of hard-real-time (HRT) tasks. Third, we devise a timed automata (TA) model of the ERTS, that takes into account the affinity, the data sharing overheads and the scheduling algorithm, on which we demonstrate that various properties can be verified compositionally, i.e., on a subset of cores instead of the whole ERTS, therefore reducing the state-space size. In particular, we enable the scalable computation of tight worst-case response times (WCRTs) and other tight bounds separating events on different cores, thus overcoming the pessimism of schedulability analysis techniques. We fully automate our approach and show its benefits on three real-world complex ERTSs, namely two autonomous robots

and an automotive case study from the WATERS 2017 industrial challenge.

1. Introduction

1.1. Addressed Problem & Motivation

In an embedded real-time system (ERTS), the software, consisting of a set of *real-time tasks*, executes on an *embedded hardware*. The latter is usually a *multicore shared-memory* one, where "multicore" refers to a small number of cores (contrasted with a higher number of tasks) due to Size, Weight, Power and Costs (SWaP-C) considerations, and shared memory allows tasks to communicate the results of their computations. ERTSs are at the heart of *safety-critical* applications spanning various domains, such as automotive vehicles and mobile robots, where their failure may entail considerable economic losses and even human casualties. Unfortunately, classical *scenario-based testing* proved inefficient as a means to detect faulty executions and thus prevent an ERTS failure. For instance, Toyota vehicles with a major software bug behind the Unintended Acceleration failure, costing 89 human lives and billions of dollars to the automotive manufacturer, have nevertheless successfully passed millions of hours/miles of testing prior to their deployment [48].

It is therefore crucial to formally verify ERTSs as to guarantee their safe behavior. Such verification is typically carried out on an ERTS *model* vis-à-vis important *real-time properties*, such as bounded response and schedulability. For instance, in an autonomous drone, flight missions must obey stringent timing constraints to ensure that e.g., control algorithms perform fast enough to prevent collisions and crashes from happening. In order to obtain useful results, the verified model must take into account the actual hardware-software setting of the underlying ERTS, i.e., software timing parameters (tasks deadlines and execution times) and hardware specificities (concurrent access to data, number of cores). As an example, if the number of cores is ignored in the model, the verification results are only valid under the unrealistic assumption that the ERTS hardware has enough cores to run all tasks in parallel. Formal verification of ERTSs is thus a complex activity; it involves at least two research communities, namely the formal methods community and the real-time systems community.

On the one hand, the formal methods community provides rigorous approaches, such as (real-time) model checking [4, 23], to verify timed systems. In model checking, a mathematical model of the system under scrutiny is verified exhaustively against properties formalized in a logic. Powerful model checking algorithms are implemented within state-of-the-art tools, such as UPPAAL [53], based on timed automata (TA) [6, 43] and a fragment of the timed computational-tree logic (TCTL) [5] for modeling the system and formalizing the properties of interest, respectively. Unfortunately, in the case of ERTSs, model checking suffers from scalability issues due to the state-space explosion problem [24]. Non-exhaustive approaches, such as statistical model checking (SMC) [54] and runtime verification (RV) [9], are more scalable. In SMC, properties are no longer verified with certainty but up to some probability. However, obtaining

a sufficiently high probability (if defined) in a safety-critical setting poses the same scalability issues as SMC tends then towards exhaustive model checking (see e.g., the conclusions of [33, Sect. 7.2.2]). In RV, monitors check the properties satisfiability, and possibly react to their violation at runtime (e.g., à la Fault Detection, Isolation and Recovery FDIR [77, 69]), therefore avoiding the exhaustive exploration of the state space. Nevertheless, RV still faces major challenges in safety-critical settings, in particular the overhead of deploying monitors that may impede timely recovery from property violation (see e.g., the conclusions of [35]). Finally, real-time verification activities within the formal methods community seldom take the hardware actual constraints into account, which restricts the results' validity to unrealistic ERTSs (Sect. 8).

On the other hand, the real-time systems community has a long tradition with the analysis of ERTSs. In particular, schedulability analysis [20] is a well-anchored discipline, where one assesses whether some real-time tasks are *schedulable* (always finish executing before their deadlines), given a number of cores and a *scheduling algorithm*. Under the schedulability analysis umbrella, Integer Linear Programming (ILP) is extensively used to e.g., find an *affinity* (task-core allocation) that guarantees the schedulability of all tasks, or their subset of hard-real-time (HRT) tasks in an ERTS. Though solving an ILP problem is NP-hard [45], the community provides algorithms that are efficient in practice (see e.g., [80]). However, the more schedulability analyses are scalable, the more pessimistic they get in the periodic setting, as computationally efficient *schedulability tests* are typically sufficient but not necessary, i.e., if the schedulability test holds for some task then it is schedulable, but the converse is not true (more in Sect. 5, 7). Moreover, as its name indicates, schedulability analysis focuses on schedulability properties only. In addition, tasks models used in schedulability analysis are usually overly simplified compared to the ones found in real ERTSs, e.g., in automotive systems and mobile robots (Sect. 3).

To summarize, interdisciplinary formal verification approaches for ERTSs, that promote both realistic modeling and scalable verification, are needed.

1.2. Contributions & Outline

In this paper, we propose a scalable approach to formally verify real-time properties in real-world ERTSs. In particular, our approach joins efforts from both communities to enable compositional model checking, thus alleviating the state-space explosion problem. Moreover, formal models, on which properties can be verified compositionally, are automatically generated from any ERTS following a periodic-task model specified in a high-level format. Our contribution is thus twofold: we (i) provide a fully automated and generic framework to verify real-world ERTSs and (ii) enable scalability through compositional verification. Taming the state-space size comes with the direct advantage of verification feasibility on real-word ERTSs w.r.t. a number of properties where model checking allows, in addition, to compute tight upper bounds of the worst-case response times (WCRTs) of tasks as opposed to the pessimistic ones provided by schedulability tests.

The rest of this paper is organized as follows. First, we introduce formalisms and techniques used in this paper (Sect. 2). Second, we formalize a realistic ERTS, where tasks contain an arbitrary number of jobs and job segments, and hardware specificities are taken into account, then expose the main challenge hindering the compositional verification of such ERTS, namely tasks dependency w.r.t. cores and shared data. We present our overall approach as we show that this challenge can be efficiently tackled under a state-of-the-art scheduling algorithm, namely partitioned fixed-priority (P-FP) with limited preemption (Sect. 3). Consequently, we generalize techniques from the real-time systems community that incorporate data sharing overheads in tasks independently (Sect. 4), then devise an ILP algorithm that computes an affinity guaranteeing the schedulability of all tasks (or at least all HRT tasks) in a real ERTS (Sect. 5). We provide then a TA model for any ERTS under the above scheduling assumption, integrating data sharing overheads (from Sect. 4) and combined with the affinity (computed in Sect. 5), on which we demonstrate that important properties can be verified compositionally (Sect. 6). We validate the scalability of our approach on three real-world case studies featuring two autonomous robots and an automotive system, and show further how compositionality allows a scalable computation of tight WCRTs, therefore overcoming a posteriori the pessimism of the ILP algorithm (Sect. 7). Finally, we compare our approach with related research (Sect. 8) and conclude with future work (Sect. 9).

2. Preliminaries

In this section, we provide definitions of the formalisms and techniques serving as a basis for the approach presented in this paper.

2.1. Data Sharing

Concurrent execution in an ERTS requires *coordination*. A *task model* describes recurring operations in the ERTS, and a real-time *scheduler* ensures the timely execution of tasks based on their timing requirements (Sect. 3). Tasks access shared data, so the task model also defines *synchronization* concepts. *Consistent access* to shared data happens when executing some job segments, typically non-preemptible pieces of code, i.e., the scheduler never preempts a task inside a job segment. The dependency of tasks on data can often be modeled using *producer-consumer* patterns. The resulting synchronization allows *exclusive* (read-write, producer) or *shared* (read-only, consumer) data access, or provides *consistent snapshots* of logically coherent data. ERTS can use *locks*[1], e.g., fine-grained reader-writer locks [60], which causes *blocking* to other tasks, or *wait-free algorithms* [44]. Wait-free algorithms can be further classified as *retry-based* approaches, e.g., sequence locks in Linux [58], or use *multiple buffers*, e.g., Simpson's four-slot algorithm [74], to exploit parallelism. In either case, *data sharing overheads* refer to the maximum time a job segment is *delayed* by

[1] See [18] for in-depth details on locking protocols.

other job segments accessing data concurrently. This corresponds for instance to blocking in lock-based approaches and the *latency* due to retries in retry-based approaches (more in Sect. 4).

2.2. Integer Linear Programming (ILP)

A linear programming (LP) problem consists in *optimizing* (maximizing or minimizing) a *linear objective function subject to linear constraints*. A classical formulation for of an LP is:

$$\text{Maximize: } \mathbf{c}^T\,\mathbf{x}, \text{ subject to: } A\,\mathbf{x} \leq \mathbf{b}, \mathbf{x} \in \mathcal{X}$$

where $\mathbf{c}^T$ is the transpose of matrix $\mathbf{c}$ and $\mathbf{c}^T\,\mathbf{x}$ represents the objective function to optimize with $\mathbf{x}$ a vector of m unknown variables over the domain $\mathcal{X}$, $\mathbf{b}$ a vector of known coefficients and A a known matrix of coefficients. The constraints are expressed by the relations $A\,\mathbf{x} \leq \mathbf{b}$ and $\mathbf{x} \in \mathcal{X}$ that specify the polytope over which the objective function is to be optimized. Given this polytope, an LP *solver* finds a point where the objective function has the smallest or largest value through the polytope vertices. LP algorithms are known to be exact. If the unknown variables are all real, i.e., $\mathcal{X} = \mathbb{R}^m$, then the problem can be solved using the classical Simplex method [66].

The LP problem becomes an ILP problem if the variables are all required to be integers, i.e., $\mathcal{X} = \mathbb{Z}^m$ (binary variables[2] are considered as integers in the domain $\{0, 1\}$). Though ILP solving is NP-hard [45], many algorithms (e.g., branch-and-bound and branch-and-cut) are mature and well integrated in powerful solvers, such as CPLEX and Gurobi.

2.3. Timed Automata (TA)

Timed automata (TA) [6] extend Büchi automata with real-valued *clocks*. A simpler yet equivalent version with location *invariants* instead of accepting locations was introduced in [43] and is at the heart of modern model checkers, such as the state-of-the-art UPPAAL [53] used in this paper. We first define *synchronization-free networks of TA* (TA composed in parallel without synchronizations), mostly simplified from [13].

2.3.1. Synchronization-free Networks

Notation. Let X be a set of real-valued clocks and $\mathcal{B}(X)$ the set of *clock constraints* over X. Each clock constraint in $\mathcal{B}(X)$ is a (possibly empty) conjunction of atomic constraints of the form $x \sim k$ with $x \in X$, $\sim \in \{<, >, \leq, \geq\}$ and $k \in \mathbb{N}$. Let $v : X \mapsto \mathbb{R}_{\geq 0}$ be a *valuation* function. We write $v \in c$ to denote that $v(x)$ for each $x \in X$ satisfies $c \in \mathcal{B}(X)$, and $v + d$ ($d \in \mathbb{R}_{\geq 0}$) to denote the valuation $v(x) + d$ for each $x \in X$. Moreover, for $\lambda \subseteq X$, $[\lambda \mapsto 0]v$ denotes the valuation 0 for each $x \in \lambda$ and $v(x)$ for each $x \in X \backslash \lambda$. Finally, $\mathcal{V}_i$ denotes the i^{th} element of a vector $\mathcal{V}$ and $\mathcal{V}[\mathcal{V'}_i/\mathcal{V}_i]$ the vector $\mathcal{V}$ in which $\mathcal{V}_i$ was substituted by $\mathcal{V'}_i$.

[2] These can be used to represent Boolean variables or simply the values 1 and 0 to activate or deactivate a constraint (Sect. 5).

Syntax. In a synchronization-free network of TA $N = (A_1||..||A_n)$, $n \in \mathbb{N}$, each A_i is a tuple $\langle L_i, l_{i0}, X_i, E_i, I_i \rangle$ where:

- L_i is a finite set of locations,
- $l_{i0} \in L_i$ is the initial location,
- X_i is a finite set of clocks,
- E_i is a finite set of edges, each edge of the form (l, g, λ, l') with $l, l' \in L_i$, $g \in \mathcal{B}(X_i)$ a *guard*, and $\lambda \subseteq X_i$ a subset of clocks to be *reset*,
- $I_i : L_i \mapsto \mathcal{B}(X_i)$ a function that assigns an *invariant* to each location.

Note that a guard/invariant can be a tautology, i.e., an empty conjunction of clock constraints $\top$. A location is called *invariant free* if its invariant is $\top$. Such guards/invariants and resets over the empty set are often omitted in the remainder of this paper.

Semantics. Let $\mathcal{X} = \bigcup_{i \in 1..n} X_i$ be the set of all clocks in the network. The semantics of a synchronization-free network of TA $(A_1||...||A_n)$ is given over a transition system (TS) $\langle Q, q_0, \rightarrow \rangle$ where:

- Q is the set of states of the form $(\mathcal{L}, v)$, with $\mathcal{L} \in L_1 \times ... \times L_n$ a vector of locations and $v : \mathcal{X} \mapsto \mathbb{R}_{\geq 0}$ a clock valuation function,
- q_0 is the initial state $(\mathcal{L}^0, v_0)$, with $\mathcal{L}^0 = (l_{10}, \ldots, l_{n0})$ and $v_0(x) = 0$ for each $x \in \mathcal{X}$,
- $\rightarrow$ is the transition relation. A transition is *enabled* at some state q if it can be *taken* at q. Taking a transition changes the state as follows: (i) delay transitions $(\mathcal{L}, v) \rightarrow (\mathcal{L}, v + d)$ for some $d \in \mathbb{R}_{>0}$ if $v \in \bigwedge_{i \in 1..n} I_i(\mathcal{L}_i)$ and $(v + d) \in \bigwedge_{i \in 1..n} I_i(\mathcal{L}_i)$, (ii) discrete transitions $(\mathcal{L}, v) \rightarrow (\mathcal{L}[\mathcal{L'}_i/\mathcal{L}_i], v')$ if exists an edge $(\mathcal{L}_i, g, \lambda, \mathcal{L'}_i) \in E_i$, $v \in g$, $v' = [\lambda \mapsto 0]v$ and $v' \in I_i(\mathcal{L'}_i)$.

Note that the above definitions also apply to a synchronization-free network of networks, i.e., a network $\mathcal{N} = (N_1|| \ldots ||N_n)$, $n \in \mathbb{N}$ where each N_i is a network of TA, and there is no synchronization between any couple of TA belonging to two different networks[3] N_i and N_j in $\mathcal{N}$. Indeed, each N_i can be viewed as a single TA resulting from the (possibly synchronized) product of all the TA it contains [3], then the semantics above applies to the composition similarly.

[3]Here, the TA involved in the same N_i can be synchronized, in which case a set of *actions* (*channels* in UPPAAL) needs to be defined for these TA.

2.3.2. Timelocks

Timelocks are pathological phenomena that reflect modeling flaws in TA [78, 16]. In other words, the presence of a timelock in the underlying TS of a network of TA is a sign of a modeling mistake. This is because a timelock corresponds to an unrealistic scenario where global time may not evolve beyond some bounded value as we will explain in more details and provide an example next. More formal definitions and proofs are given in Sect. 6.

A network N has a timelock iff it contains at least one *timelock state*. A timelock state $(\mathcal{L}, v)$ is a state from which (i) no infinite sequence of transitions exists or (ii) each infinite sequence of transitions is bounded in time, i.e., time converges toward some integer. In the case (ii), each infinite sequence of transitions is called a *zeno* run, and $(\mathcal{L}, v)$ a *zeno-timelock* state. More informally, a network N has a timelock iff there exists a state in the underlying TS from which time may not progress beyond some finite value. Notice that, in [78], only zeno timelocks are called timelocks. Our more generic definition and examples stem from [16], where the impossibility of time progress beyond some global time t is considered a timelock, regardless of zenoness.

Example. Consider the singleton network $N = A$ where A has one location l_0 and one clock x.

Consider the case where A has no edges and $I(l_0) = (x < 2)$. Here, A timelocks because at least one state in the underlying TS is a timelock state (actually, all states here are timelock states). For instance, the initial state $(l_0, 0)$ is a zeno-timelock state, since the sum of delays corresponding to any infinite succession[4] of transitions from it is bounded by 2.

Consider now the case where A has no edges and $I(l_0) = (x \leq 2)$. Here, A timelocks, because there is at least a timelock state in the underlying TS: $(l_0, 2)$, for instance, is a (non-zeno) timelock state. Indeed, at such state, no infinite sequence of transitions in possible (actually, no transition is possible at all). The underlying TS is thus "frozen" at state $(l_0, 2)$.

Zeno timelocks can stem from discrete transitions as well. Consider the case of A where $I(l_0) = (x \leq 2)$ and A has one edge $(l_0, x \leq 2, \emptyset, l_0)$. Here, at $(l_0, 2)$, all possible infinite sequences of transitions contain only discrete transitions (the discrete transition corresponding to the only edge in A is always enabled, and time may no longer progress because of the invariant at l_0). $(l_0, 2)$ is thus a zeno-timelock state.

Timelocks are clearly unrealistic (time always evolves beyond some point in reality). Timelocks (including zeno ones) have unexpected consequences on networks semantics, and they typically result from modeling errors. Networks of TA need to be exempt of these phenomena, as we will see further in Sect. 6.

[4] Due to the strict $<$ in the invariant and delays taking their values in $\mathbb{R}_{>0}$, an infinite succession of delay transitions is possible from this state without ever reaching the value 2.

2.3.3. UPPAAL TA

In this paper, we deliberately avoid the semantics of synchronization and other extensions, such as priorities, for simplicity. Instead, we will directly explain how these extensions work in UPPAAL, the model checker that we use (some details on UPPAAL TA semantics are provided in [13]). Examples of UPPAAL TA are given in Sect. 6.

Channels. UPPAAL allows synchronizations through *handshake* and *broadcast* channels. The former (resp. the latter) are blocking and pairwise (resp. non blocking and multiparty), that is in a handshake (resp. a broadcast) channel, the *sender* synchronizes with only one *receiver* (resp. as many receivers as possible). A synchronization implies taking all edges involved in it simultaneously.

Priorities. Channels may have priorities. Priorities affect the semantics of discrete transitions. In brief, if channel c' has a higher priority than channel c, then at any state of the underlying TS, all discrete transitions involving c are disabled as long as there exists an enabled transition involving c'.

Data variables and functions. UPPAAL supports integer and Boolean data variables, whose values may be used in guards, and updated in discrete transitions (together with clock resets). Variables can be local (to a TA) or global. A small subset of C-like functions is also supported to ease writing complex updates.

Committed locations. Committed locations implement both an *urgency* and a priority. That is, in the underlying TS, if at some state $(\mathcal{L}, v)$ there exists $\mathcal{L}_i \in \mathcal{L}$ such that $\mathcal{L}_i$ is committed, then a discrete transition starting from a committed location must be taken immediately, i.e., (i) time may not progress, and (ii) only transitions of the form $(\mathcal{L}, v) \rightarrow (\mathcal{L}[\mathcal{L'}_i/\mathcal{L}_i], v')$ where $\mathcal{L}_i$ is committed are enabled. Committed locations are therefore handy to describe sequences of timeless actions that need to be realized immediately regardless of the other actions possible at the same time. This is for example the case of scheduling decisions such as activation and release (Sect. 6).

2.4. Timed Computational Tree Logic (TCTL)

TCTL is a logic that is both *temporal* and *timed*, allowing to reason on the order of, and the amount of time separating the satisfaction of some formulae, respectively. We focus on the fragment of TCTL that UPPAAL supports. A *state formula* is a propositional formula over locations, clocks and data variables, to be evaluated at a TS state. For instance, $A_i.x < 3$ holds in all states of the TS where the valuation of clock x in TA A_i is strictly less than 3. *Path formulae* enable quantifying over TS traces. Path formulae in UPPAAL use the operators A (for all paths) and E (there exists a path) combined with the modalities $\Box$ (necessity, all states) and $\diamond$ (possibility, some state). For instance, the property $A \diamond \phi$ (where ϕ is a state formula) translates to "for each path, there exists a state that satisfies ϕ" whereas $E\Box\phi$ reads "there exists a path in which all states satisfy ϕ".

Next are the three path formulae (out of five supported by UPPAAL) used in this paper (ϕ and ψ are state formulae):

- $A\Box\phi$: ϕ holds in all states of the TS (*safety*),
- $E \diamond \phi$: there exists a reachable state that satisfies ϕ (*reachability*),
- $A\Box(\phi \Rightarrow A \diamond \psi)$, denoted using the shorter syntax $\phi \rightsquigarrow \psi$: whenever ϕ holds, ψ eventually holds (*leadsto* & *bounded response*).

For bounded response, we are also interested in quantifying the maximum amount of time separating the satisfaction of ϕ and ψ. For this, we use the formula $sup\{A_i.l\} : A_i.x$ that computes the maximum valuation of clock x at location l in TA A_i. This will be handy in computing WCRTs and other important bounds (practical examples are given in Sect. 6, 7).

3. Challenges & Overall Approach

Formal modeling and the feasibility of verification are intimately related: the more the formal model is abstract, the more the verification is likely to scale. However, abstractions should not come at the cost of unrealistic modeling. Formal models should be the closest possible to the reality of the underlying system, i.e., the real hardware-software setting in the case of ERTSs. In this section, we provide a formal description of a realistic ERTS under a couple of state-of-the-art scheduling assumptions (Sect. 3.1), and explain why such a model can be verified in a scalable manner (Sect. 3.2). Then, we present our overall approach accordingly (Sect. 3.3).

3.1. ERTS Model

An ERTS is classically modeled as a set of real-time tasks executing on a set of cores following a *scheduling algorithm* [20]. Though models within the real-time systems community take more and more into account the dependency between tasks, e.g., vis-à-vis data sharing [17, 67, 68], they remain typically simple with each task having one *job* associated to a *Worst-Case Execution Time* (WCET), a *deadline* and possibly a *period*. In real ERTSs, a task is noticeably more complex, it can have several jobs each consisting of a sequence of *segments* (pieces of code). This is the case, e.g., in mobile robotics and automotive systems, where job segments are called, respectively, *runnables* [62, 63] and functions/codels [47, 40, 34] (actual examples are given in Sect. 7). As we have shown in [37, 34, 38], this complexity prevents the reuse of *schedulability tests* from the literature. We need therefore a realistic ERTS model which we present next. Our model relies on *partitioned fixed priority* (P-FP) scheduling with *limited preemption*. In brief, P-FP scheduling assigns statically each task to one and only one core (partitioned "P") and tasks priorities are fixed beforehand, i.e., they do not depend on the execution dynamics (fixed priority "FP"). As for limited preemption, it refers to the fact that preemption, i.e., putting the execution of a job in a lower priority task temporarily on hold in order to execute

a job in a higher priority task on the same core, is allowed but can happen only at some time instants relative to the execution of the job in the lower priority task. P-FP scheduling with limited preemption is an efficient and widely used scheduling algorithm in ERTS as we will explain further, with the appropriate references, in Sect. 3.2.

Syntax. An ERTS is made of a set of tasks T, a set of shared data L and a set of cores C ($|T| > |C| > 0$). We assume that all tasks in T are periodic (we discuss in Sect. 7 the implications of this simplification) and the deadlines to be the periods themselves (a classical assumption with periodic tasks).

Each task $\tau \in T$ is a finite-state machine (FSM) $\langle S_\tau, act_\tau, end_\tau, tr_\tau \rangle$ where S_τ is the set of states with act_τ and end_τ the special *activation* and *termination* states, respectively, and the remaining states $S_\tau \backslash \{act_\tau, end_\tau\}$ represent the job segments set JS_τ, and $tr_\tau \subset S_\tau \times S_\tau$ is the transition relation. We write $s \rightarrow s'$ if $(s, s') \in tr_\tau$ and $s \not\rightarrow s'$ otherwise. The state act_τ (resp. end_τ) has no predecessors (resp. no successors), formally $\forall s \in S_\tau : (s \not\rightarrow act_\tau \wedge end_\tau \not\rightarrow s)$. We call a *maximal run* in τ each run in the FSM starting at act_τ and ending at end_τ, denoted $act_\tau \rightarrow \cdots \rightarrow end_\tau$. All runs in the FSM are finite, i.e., there are no loops. A *job* is the ordered set of states appearing in a maximal run, excluding act_τ and end_τ. Accordingly, J_τ is the set of jobs in τ and $J = \bigcup_{\tau \in T} J_\tau$ (resp. $JS = \bigcup_{\tau \in T} JS_\tau$) is the set of all jobs (resp. job segments) in the ERTS. In the remainder of this paper, the shorthand terminology "segment" refers to a job segment, and we drop $_\tau$ subscripts when they are unneeded or understood from the context. The priority function $\pi : T \mapsto \mathbb{N}$ assigns a priority to each task. The set of HRT tasks $T_h \subseteq T$ comprises the tasks that are not allowed to miss their deadlines, e.g., defined by the user. Each core $c \in C$ is associated with a scheduler $\mathcal{H}_c$ and a prioritized queue Q_c to sort tasks waiting for their release (see the behavior below).

Next, we define data/core usage and timing constraints. $dw : JS \mapsto \mathcal{P}(L)$ (resp. $dr : JS \mapsto \mathcal{P}(L)$), with $\mathcal{P}(L)$ the powerset of L, associates each segment with the elements of data it writes (resp. reads). Note that $dr(s) \cup dw(s)$ may be empty; our definition remains therefore generic on whether a given segment uses shared data or not. The affinity function $aff : T \mapsto C$ associates each task to one core, and dually, a partitioning function $prt : C \mapsto \mathcal{P}(T)$ associates each core c to the set of tasks allocated to it (i.e., the *partition* $\{\tau | aff(\tau) = c\}$). The function $p : T \mapsto \mathbb{N}_{>0}$ associates each task with its period, and the function $wcs : JS \mapsto \mathbb{N}_{>0}$ (resp. $bcs : JS \mapsto \mathbb{N}$) returns for each segment its WCET (resp. Best Case Execution Time BCET). Naturally, the inequality $bcs(s) \leq wcs(s)$ is verified for every $s \in JS$. Finally, the function $\rho_r : L \mapsto \mathbb{N}_{>0}$ (resp. $\rho_w : L \mapsto \mathbb{N}_{>0}$) assigns each data a reading (resp. writing) penalty, i.e., the maximum time needed to read (resp. write) such data without overhead (i.e., without interference from concurrent readers/writers). Without loss of generality, we assume $wcs(s)$, for any segment s in JS, to include the reading (resp. writing) penalty of each data in $dr(s)$ (resp. $dw(s)$), which corresponds to the maximum time needed to read (resp. write) such data in the absence of contention with other readers/writers.

These definitions coincide with the informal description of a complex task

with an arbitrary number of jobs, where each job is a finite ordered set of segments, and fall under partitioned scheduling (the codomain of the affinity function *aff* is C, and thus each task is allocated to only one core). Note that our model does not assume a knowledge of BCETs; they can simply be set to 0, if unknown, without loss of generality[5]. Note also that the assumption on the finiteness of each run in a task FSM (absence of loops) is a natural condition since tasks are real-time and therefore must execute each job in a finite (bounded) amount of time. This absence of loops should not be mistaken for a restriction as it induces no loss of generality: the FSM definition above coincides also with real-time tasks with a loop behavior (e.g., implementing control algorithms), as we will show further in Sect. 7.

Behavior. A task τ is *activated* at global time 0 (when the ERTS starts)[6] then at each period $p(\tau)$. Activating a task τ resets its current state[7] to act_τ. If c, the core to which τ is allocated ($aff(\tau) = c$) is free, then τ is *released* immediately, otherwise $\mathcal{H}_c$ inserts its identifier in Q_c (by abuse of terminology, we call the identifier also task τ). The insertion algorithm guarantees that tasks in Q_c are always ordered according to their priorities (resp. their activation time) for tasks with different priorities (resp. same priority). Whenever c is free, $\mathcal{H}_c$ pops the task at the head of Q_c and releases it. Upon release, τ executes one of its jobs, say ι, e.g., chosen at runtime from J_τ, by traversing the path from act_τ to end_τ that corresponds to ι.

Let v and v' be two variables in the domain $prt(c)$ that continuously store the task at the head of Q_c and the task currently executing a job on c, respectively (if any, otherwise they take arbitrary values in their domain). By abuse of notation, we confound in the following these variables with their values at a given time. At each time t an insertion in Q_c happens[8], $\mathcal{H}_c$ compares $\pi(v)$ with $\pi(v')$. If the latter is less than the former, then v' is marked for preemption. The preemption happens as soon as the segment of the job v' was executing at time t, say s, ends, that is at a time t' comprised between t and $t + wcs(s)$ (+ possibly some data sharing overhead). $\mathcal{H}_c$ then suspends the job v' is executing, inserts v' back in Q_c, pops v from Q_c and releases it. Notice that the task released at t' may be different from the one that caused the preemption of v' at t, as the value of v may have changed in between due to an insertion of a higher priority task. A preempted task, when released, resumes at the next segment of the suspended job. The semantics of execution is further formalized in Sect. 6 using TA. Notice that we do not specify here a model for concurrent access to data. The overhead of sharing data is characterized in Sect. 4 as multiples of the reading/writing

[5] Though it is still recommended to evaluate BCETs for a better accuracy of the analysis.

[6] While activation at global time zero is not mandatory, it is appropriate here since we do not consider jitters or offsets in this paper.

[7] If such current state is end_τ, otherwise there is a deadline miss and the behavior depends on the task's criticality, Sect. 6.

[8] We recall that the insertion of a task captured by v happens only if there is another task captured by v' currently executing on c, otherwise the former would be released immediately.

penalties given by the functions ρ_r and ρ_w defined above.

3.2. Justifying the Scheduling Choices

Say that we have an ERTS modeled correctly as an UPPAAL network of TA (Sect. 2.3) on which we want to verify a TCTL property (Sect. 2.4). Even if the property is local (e.g., bounded response within one task), the verification must be carried out on the whole UPPAAL model since tasks behaviors depend on one another due to (i) data sharing and (ii) the possibility for a task to migrate between cores. Our goal is to break this bi-dimensional dependency as to enable compositional verification while preserving the correctness of the model. This goal is achievable under P-FP scheduling with limited preemption.

First, partitioned (P) schedulers prevent tasks migration and thus eliminate the second dimension of the dependency. The good news is, P scheduling is widely used, it is even the *de facto* choice in ERTSs [80, 81]. However, it requires computing an affinity beforehand that satisfies some schedulability requirements, which typically boils down to the bin-packing NP-hard problem [45]. Second, limited preemption makes it computationally tractable to integrate data sharing overheads in the WCETs of segments of each job independently, thus getting beyond the first dimension of the dependency. Fortunately, limited preemption is a natural and efficient choice in ERTSs. First, it may be used to prevent preemption inside a job segment (as in our model above), necessary for consist access to shared data (Sect. 2.1). Second, it is proven that, schedulability-analysis-wise, limited preemption is a better choice than both *full preemption* and *full non preemption* under P-FP scheduling [21].

3.3. Overall Approach

To summarize, compositional verification of ERTS is possible under P-FP scheduling with limited preemption. This requires however (i) a correct integration of data sharing overheads in each task independently and (ii) tackling the (NP-hard) affinity problem beforehand. We present accordingly our approach in Fig. 1. The input of the problem is a realistic ERTS model (Sect. 3.1) in a simple format (more in Sect. 7). Step 0 is optional, it consists in writing translators from given ERTS domains to our input format (we provide two such translators for some robotic and automotive applications, Sect. 7). Step 1 feeds the ERTS model to the ILP algorithm that we devise in Sect. 5 to compute an affinity guaranteeing the schedulability property for all tasks, or, if no such (total) affinity is found, a partial affinity such that at least all HRT tasks are schedulable (Step 2). Then, given the input model, the affinity, and the formal TA model of ERTS that we define in Sect. 6, an UPPAAL model is automatically generated (Step 3). Both the ILP algorithm and the UPPAAL model take into account all information from the input model and the semantics of P-FP scheduling with limited preemption, and integrate data sharing overheads based on the bounds we compute in Sect. 4. In Step 4, important properties are verified compositionally on the generated UPPAAL model, including a tight computation of tasks WCRTs that enables the possibility to counterbalance, a posteriori, the pessimism of the ILP pass (Sect. 6, Sect. 7).

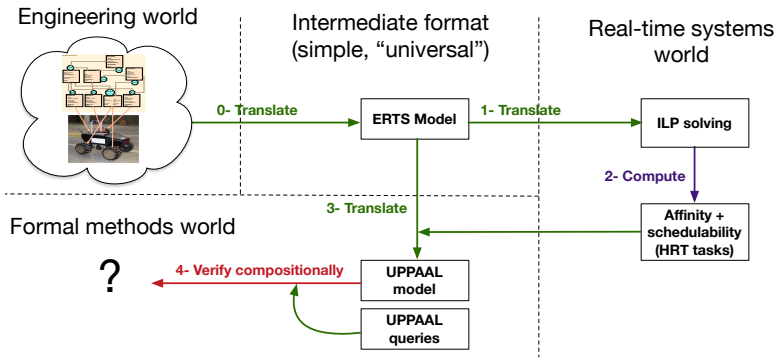


Figure 1: Overall Approach

4. The Data Sharing Problem

In an ERTS, we can model data sharing as producer-consumer patterns comprising m writers and n readers for each data $l \in L$. Based on the syntax of ERTS in Sect. 3.1, m and n can be defined formally for some data $l \in L$ as follows: $m(l) = |\{s | s \in JS \wedge l \in dw(s)\}|$ (the number of segments in the ERTS that write l) and $n(l) = |\{s | s \in JS \wedge l \in dr(s)\}|$ (the number of segments in the ERTS that read l). We distinguish two cases: *single writer* (where for each data l in L, all $m(l)$ writers belong to jobs of the same task) and *multiple writers* (otherwise). Under P-FP scheduling with limited preemption, we can already see that there are certain beneficial scenarios where there will be no blocking at all, i.e., if both writers and readers execute on the same core. We can also see that writers (resp. readers) will update (resp. read) data frequently, but not permanently. In the single writer case, we can therefore assume that a write operation interferes with specific readers at most once, i.e., while the reader is reading, new data arrives. With multiple writers, we can observe similar simplifications. When all writers of data l are allocated to the same core, the write operations will be serialized by the non-preemptive scheduling for free. Only when writers are allocated to different cores, write operations can happen concurrently and therefore must be protected, e.g., by using a spinlock.

For this type of synchronization problem, *sequence locks (seqlocks)* [42, 50, 58] are a good fit as we explain throughout this section. Seqlocks tend to favor read-heavy workloads under the optimistic assumption that no update/write of the shared data happens at the same time as readers read the data [52]. Otherwise, the read operation has to be retried, and for real-time use cases, the number of retries has to be bounded. We first compute the *total cost* of reading/writing a data, i.e., the maximum time needed to read/write a data in the presence of delays induced by concurrent access to the same data by other writers/readers. This is done through bounding the number of retries (Sect. 4.2, 4.3). Then, we deduce the data sharing overheads (Sect. 4.4). The overheads are for an ERTS under single writer seqlocks (that is, as explained above, each

Algorithm 1: Sequence lock usage (see Section 4.2)

```
1  sequence lock state:
2      s : spin lock = uncontended                          ◁ spin lock
3      c : unsigned int = 0                                 ◁ sequence counter
4      d : array of bytes = ...                             ◁ shared data
5  writer:
6      lock (s)
7      c ← c + 1
8      memory barrier: order write → write
9      d ← ...                                              ◁ update shared data
10     memory barrier: order write → write
11     c ← c + 1
12     optional artificial delay (see Section 4.3)
13     unlock (s)
14 reader:
15     repeat
16         repeat
17             c_1 ← c
18         until c_1 is even                                ◁ spin on write
19         memory barrier: order read → read
20         tmp ← d                                          ◁ read shared data
21         memory barrier: order read → read
22         c_2 ← c
23     until c_1 = c_2
24     return tmp
```

data l is written by one task at most, i.e., all segments that write l belong to jobs in the same task). Finally, we justify the use of single writer seqlocks, as we (i) compare the computed overheads to those of other state-of-the-art locking protocols and (ii) further explain the fitness of the single writer assumption to ERTS in practice. Our approach remains nonetheless generic, as the overheads from the table given at the end of this section can be used instead for any ERTS using a different data sharing assumption (Sect. 4.5).

4.1. Sequence Locks (Seqlocks)

Seqlocks are a state-of-the-art locking mechanism for read-heavy consistent access to shared data in Linux [42, 50, 58]. Seqlocks were conceived to solve starvation issues with reader-writer locks in certain use cases, such as updating the current system time from an interrupt handler [42]. Alg. 1 shows an implementation. Seqlocks use a *sequence counter* for reader-writer synchronization and

a spinlock for synchronization of multiple writers[9]. The sequence counter, e.g., a 32-bit unsigned integer initialized to zero, implements the following protocol. At the start of a write operation, the writer first increments the counter, then writes or updates the data, and finally increments the counter again. With this, an odd counter value indicates an ongoing write operation, and the upper bits of the counter define a generation counter. The read side first reads the counter, then reads the data, and then reads the counter again. If both counter values are equal and the value is even, the data was read consistently. Otherwise, the reader repeats. Also, we let the reader spin on the first counter access while the value is odd, i.e., a write operation is currently ongoing. Further, on hardware architectures with weak memory ordering, *memory barriers* are required: two read barriers on the reader side and two write barriers on the writer side to order the reads from writes to resp. the sequence counter and the data accesses.

4.2. Bounding Seqlocks

The two obvious problems of seqlocks w.r.t. predictability are that the reader side can be starved by successive back-to-back writes, and that the reader side can read stale data due to the *ABA problem* when the counter repeats after 2^{31} write operations [29]. Though both are unlikely to happen in real systems, we need to eliminate them to abide by safety-critical standards. To bound the number of retries on the reader side, Kopetz and Reisinger suggest a *minimum inter-arrival time* δ_w between two successive write operations [49].

In the following, we denote by $\widehat{\rho}_r(l)$ (resp. $\widehat{\rho}_w(l)$) the maximum time of the *total cost* of reading (resp. writing) data l in the presence of overheads. With non-preemptible segments (Sect. 3.1), a write operation of some data $l \in L$ takes therefore $\widehat{\rho}_w(l)$ time for a single writer (Lines 7 to 11 in Alg. 1), which is mainly driven by the time to access the shared data (Line 9). Further, we compose the individual steps of a read operation as the spinning part $\rho_s(l)$ (Lines 16 to 18) and the reading part $\rho_r(l)$ (Lines 19 to 22).

Under the condition that $\delta_w > \rho_r(l)$, we can construct the following worst case with at most one retry of the outer loop (Lines 15 to 23). Assume a read operation is successfully ongoing and reaches the second sequence counter check (Line 22). At the same time, a write operation starts and increments the counter (Line 7). This causes a counter mismatch on the reader side (Line 23), with spinning for the parallel write operation to finish (Lines 16 to 18) for at most $\rho_w(l)$ time, therefore $\rho_s(l)$ is upper-bounded by $\rho_w(l)$. Afterwards, the reader successfully reads the new data and the loop terminates. This takes at most $\widehat{\rho}_r(l) = \rho_r(l) + \rho_w(l) + \rho_r(l)$. Under the assumption that both $\rho_r(l)$ and $\rho_w(l)$ depend on the actual size of the shared data, i.e., the number of accessed cachelines, and that both reading and writing take the same time on modern computer architectures, we can set $\rho(l) = \rho_w(l) = \rho_r(l)$ and consequently simplify $\widehat{\rho}_r(l) = 3\rho(l)$, $\widehat{\rho}_w(l) = \rho_w(l) = \rho(l)$ and derive a minimum bound for $\delta_w > \rho(l)$.

[9] The spinlock can be omitted in the single writer case.

4.3. Implication on Task Set

The minimum inter-arrival time δ_w between two writes solves both initial issues of bounding the loop and the ABA problem, but requires a guarantee that it is always superior to the minimum bound $\rho(l)$. The worst case scenario with a single writer for some data l is then two segments s and s', both writers of l ($l \in dw(s)$ and $l \in dw(s')$), executing back to back within the same task τ. This happens for example when s, the last segment in ι, finishes executing at exactly the end of a period of τ, followed by immediate activation and release, in the next period, of ι' in which s' is the first segment to execute (with ι and ι' two jobs of τ). In this case, to remain generic (i.e., independent from the execution scenarios), the solution is to *inflate* the execution time of s by $\rho(l)$, e.g., by inserting an artificial delay at the end of job ι. With multiple writers, the problem gets even more complex (see the overheads summary in Sect. 4.5).

4.4. Deducing Data Sharing Overheads

We can now deduce the overhead for each data l that we need to integrate in the WCETs of segments to account for the delay that they may incur while concurrently accessing l (such overhead is for instance called *blocking bound* in [19]). We introduce thus two new functions $B_w : L \mapsto \mathbb{N}$ and $B_r : L \mapsto \mathbb{N}$ that we may define simply as $B_w(l) = \widehat{\rho}_w(l) - \rho(l)$ and similarly $B_r(l) = \widehat{\rho}_r(l) - \rho(l)$. That is, we simply subtract $\rho(l)$, the maximum time needed to read/write data l without contention (already included in segments' WCETs, Sect. 3.1), from $\widehat{\rho}_w(l)$ (resp. $\widehat{\rho}_r(l)$), the total cost of writing (resp. reading) l with overheads.

4.5. Comparison to Spinlocks and Reader-Writer Locks

We compare seqlocks to other spinning synchronization mechanisms. We assume a system where readers and writers are distributed on all $|C|$ cores. Table 1 shows the individual overheads of read and write operations as multiples of $\rho(l)$ for some data l.

Task-fair spinlocks, e.g., ticket locks or MCS locks [59], serialize all read and write operations in FIFO order, so an operation has to wait at most for $|C-1|$ other operations to complete. We can consider this to be a worst-case baseline.

Task-fair reader-writer locks order arriving requests in FIFO order, but allow adjacent read requests to form a *concurrent group* until the next write request arrives [60], and the writer has to wait $\rho(l)$ time for previous readers to finish. Later readers have to wait for the same time plus the writer's execution time.

Phase-fair reader-writer locks are another state of the art technique [19]. Here, requests are queued in either read or write request queues, and reader

Table 1: Comparison of data sharing overheads in dependency on $\rho(l)$ (extending [49, 19]).

Lock Type	Single writer		Multiple writers	
	Write $B_w(l)$	Read $B_r(l)$	Write $B_w(l)$	Read $B_r(l)$
Sequence lock	$\rho(l)$	$2\rho(l)$	$2(\lvert C\rvert-1)\rho(l)$	$2\rho(l)$
Task-fair locks	$(\lvert C\rvert-1)\rho(l)$	$(\lvert C\rvert-1)\rho(l)$	$(\lvert C\rvert-1)\rho(l)$	$(\lvert C\rvert-1)\rho(l)$
Task-fair reader-writer lock	$\rho(l)$	$2\rho(l)$	$(\lvert C\rvert-1)\rho(l)$	$(\lvert C\rvert-1)\rho(l)$
Phase-fair reader-writer lock	$\rho(l)$	$2\rho(l)$	$2(\lvert C\rvert-1)\rho(l)$	$2\rho(l)$

and writer phases alternate. Then, on a phase switch to readers, all waiting readers are released, so readers have to wait at most two phases of $\rho(l)$ time. This improves the throughput of read requests at the cost of write requests. Note that the timing behavior of phase-fair reader-writer locks with a single writer is the same as for task-fair reader-writer locks.

Summarized, seqlocks behave similar to state-of-the-art locking-based approaches in the worst case, but with an improved best case timing towards the reader side. Further, the single writer assumption is reasonable in practice. This is the case for instance of state-of-the-art frameworks designed for real-time robotics, such as OROCOS [47] and MAUVE [40], and for all of the real-world case studies evaluated in Sect. 7. In the remainder of this paper, we rely therefore on overheads for single writer seqlocks, but any overhead from Table 1 may be used for the corresponding assumptions.

5. Solving the Affinity Problem

In this section, we devise an ILP algorithm to compute the affinity $\mathit{aff}(\tau)$ for each task $\tau \in T$ in an ERTS, such that all tasks in T are schedulable. If no solution is found, the algorithm relaxes the constraints by removing some non-HRT task τ from $T \setminus T_h$ and retrying on the subset of T obtained thereof. The process is repeated until a partial affinity is found or the subset of T contains only HRT tasks, in which case it terminates with a failure. Note here that removing some non-HRT tasks does not mean that they will be removed from the application (if a partial affinity is found by the ILP algorithm, we will use it as a basis to try and find a total affinity, relying on tight computations of WCRTs, Sect. 7).

More formally, the algorithm works following the steps given below, where $\mathit{sched} : T \mapsto \mathbb{B}$ is a function that returns the truth value *True* iff its input is a schedulable task according to some schedulability test. In sum, the algorithm either succeeds, if a total affinity (with all tasks schedulable) or else a partial affinity (with some tasks, including all HRT tasks, schedulable) is found, or fails, otherwise (more in Sect. 7).

- A. Compute an affinity such that $\forall \tau \in T : \mathit{sched}(\tau)$,
  - A.1. If a solution is found, terminate with success and return the affinity,
  - A.2. Else, check the equality $T = T_h$ (T_h is the set of HRT tasks, defined in Sect. 3.1),
    - A.2.1. If the equality holds (there are only HRT tasks left), terminate with a failure,
    - A.2.2. Else, update T with $T \setminus \{\tau\}$ where τ is selected randomly from $T \setminus T_h$ and repeat at (A).

To specify the ILP algorithm, we need therefore to devise the schedulability tests to characterize the function *sched*. Further, we need to express all timing

constraints as linear constraints in order to enable modeling the affinity problem as an ILP one. We achieve both goals in the remainder of this section, but first we introduce some notations.

Notations. We denote by $hp(\tau) = \{ \tau' \mid \tau' \in T, \pi(\tau') > \pi(\tau) \}$ the set of tasks with priorities higher than $\pi(\tau)$ and $sp(\tau) = \{ \tau' \mid \tau' \in T \setminus \{t\}, \pi(\tau') = \pi(\tau) \}$ the set of tasks (excluding τ) the priorities of which are equal to $\pi(\tau)$. Similarly, $lp(\tau) = \{ \tau' \mid \tau' \in T, \pi(\tau') < \pi(\tau) \}$ denotes the set of tasks with priorities lower than $\pi(\tau)$. For a job $\iota \in J$, we denote by S_ι the set of segments of ι; its WCET $wj(\iota) = \sum_{s \in S_\iota} wcs(s)$; and the WCET of its final (last) segment $final(\iota)$. We also denote $J_\iota^{-1} \in T$ the task to which the job $\iota \in J$ belongs. Further, we abuse the membership relation: we say that a job segment $s \in JS_\tau$ belongs to task τ. A task $\tau \in T$ is characterized by its WCET equal to the WCET of its longest job, that is $wt(\tau) = \max \{ wj(\iota) \mid \iota \in J_\tau \}$, its utilization factor $u(\tau) = \dfrac{wt(\tau)}{p(\tau)}$, and its maximum segment WCET $\overline{w}_s(\tau) = \max \{ wcs(s) \mid s \in JS_\tau \}$ (the WCET of its longest job segment).

5.1. Ignoring Data Sharing Overheads

5.1.1. Schedulability Test

In order to facilitate the understanding of the method, we start by modeling the problem without data sharing overheads. We can therefore, under P-FP with limited preemption, consider the schedulability problem on each core independently (this assumption is obviously false when taking data sharing overheads into account). As a first step, we will therefore consider schedulability as a single-core problem. Davis and Burns propose in [27] a monoprocessor linear upper bound for the WCRT of sporadic tasks scheduled with limited preemption with arbitrary deadlines and release jitters. It is easy to adapt this test to our model, and, by considering that the sum of task utilisation factors is less than 1, a task $\tau \in T$ is schedulable if the next inequality is verified for each of its jobs $\iota \in J_\tau$:

$$\begin{aligned} &blocking(\tau) + wj(\iota) + \textstyle\sum_{\tau' \in sp(\tau)} wt(\tau') \\ &\qquad + \textstyle\sum_{\tau' \in hp(\tau)} [wt(\tau') + u(\tau') \cdot (p(\tau) - final(\iota) - wt(\tau'))] \leq p(\tau) \end{aligned} \quad (1)$$

with $blocking(\tau) = \max \{ \overline{w}_s(\tau') \mid \tau' \in lp(\tau) \}$.

In brief, the inequality above compares an upper bound on the WCRT of τ (left-hand operand) to the deadline of τ (i.e., period of τ, right-hand operand). While the original bound in [27] is exact for sporadic tasks, it becomes pessimistic for periodic tasks, and therefore for our case. Notice how, for instance, $blocking(\tau)$, which is an upper bound on the maximum time needed to preempt a lower priority task, is defined as equal to the maximum segment WCET of all tasks with priorities lower than $prio(\tau)$. This stems from the fact that schedulability tests are based on analytical formulae where one seeks linearity for practical ILP implementations. This results in computationally efficient schedulability tests that are nevertheless pessimistic in the periodic setting.

5.1.2. ILP Formulation & Implementation Considerations

Since the schedulability test is linear, the affinity problem can be easily expressed as an ILP. To represent the affinity, we use $|T| \cdot |C|$ *binary decision variables* $\alpha(\tau, c)$ such $\alpha(\tau, c) = 1$ if $\tau \in prt(c)$ ($\alpha(\tau, c) = 0$ otherwise). The constraints follow directly from the unicity of the task affinity (P scheduling) and the schedulability test (Inequality (1)). To represent that a task can be allocated to only one core, we use the constraint:

$$\forall \tau \in T : \sum_{c \in C} \alpha(\tau, c) = 1 \tag{2}$$

and, to represent that the utilization factor on each core is limited to 1, we use:

$$\forall \tau \in T : \sum_{c \in C} \alpha(\tau, c) \cdot u(\tau) < 1 \tag{3}$$

The only element that is not linear in Inequality (1) is the *blocking* term. To avoid this, we decompose the schedulability test given in Inequality (1) into as many constraints as tasks in $lp(\tau)$ to get the following inequality for each job ι of τ ($\forall \iota \in J_\tau$), for each task τ'' with a lower priority than τ ($\forall \tau'' \in lp(\tau)$) and for each core c of C ($\forall c \in C$):

$$\begin{aligned} &\alpha(\tau'', c) \cdot \overline{w}_s(\tau'') + wj(\iota) + \sum_{\tau' \in sp(\tau)} \alpha(\tau', c) \cdot wt(\tau') \\ &+ \sum_{\tau' \in hp(\tau)} \alpha(\tau', c) \cdot [wt(\tau') + u(\tau') \cdot (p(\tau) - final(\iota) - wt(\tau'))] \leq p(\tau) \end{aligned} \tag{4}$$

That is, we obtain Inequality 4 by decomposing Inequality (1) through considering $\overline{w}_s(\tau'')$ separately in each lower priority task, then multiplying each term pertaining to some task ζ by $\alpha(\zeta, c)$ to cancel such term if $\zeta \notin prt(c)$ (i.e., $\alpha(\zeta, c) = 0$).

However, the constraints defined in (4) must be verified only if task τ is allocated to core c. In other words, they are true if τ is not allocated to c, i.e., $\alpha(\tau, c) = 0$. A classical way to deal with this problem is to introduce a term to guarantee that the above condition is always true. This can be achieved by adding $(\alpha(\tau, c) - 1) \cdot M$ to the left-hand operand of the constraints (from (4)) with a huge value M (known as *big-M constraints*). So if $\alpha(\tau, c)$ is equal to 0, the constraint is always verified, and if $\alpha(\tau, c)$ is equal to 1, the effect of M is null. Here, we do not use this technique. Indeed, modern ILP solvers have efficient approaches to model this kind of behavior. As we use Gurobi[10], we can simply add a condition to activate a constraint; here the condition is simply $\alpha(\tau, c) = 1$. More implementation details are given in Appendix A.

5.2. Taking Data Sharing Overheads Into Account

5.2.1. Schedulability Test

As explained in Sect. 4, the WCET of a segment can increase if it is not allocated to the same core as other segments with which it shares data. To model this, we need to be careful as to obtain the tightest possible bounds on overheads and avoid unnecessary overpessimism. We first define a *conflict function* $cf : JS \times L \mapsto \mathcal{P}(T)$ that returns for each segment s and data l the

[10] https://www.gurobi.com

tasks to which other segments s' in conflict with s vis-à-vis l belong. The "in conflict" relation is defined similarly to the one in [38] but with regards to each data apart, i.e., two segments s and s' (in two different tasks) are in conflict vis-à-vis data l iff they both use l and at least one of them writes it (simultaneous readings are allowed). Formally, the conflict function is defined as follows (where $d(-) = dw(-) \cup dr(-)$):

$$\forall s \in JS, l \in L : ((\tau' \in cf(s,l)) \Leftrightarrow \exists s' \in JS_{\tau'}, \tau' \neq \tau : (l \in dw(s) \wedge l \in d(s')) \vee (l \in dw(s') \wedge l \in d(s))) \quad (5)$$

Under the single-writer assumption, we can safely replace each $d(-)$ with $dr(-)$ thus alleviating the membership checks.

Once the conflict function is defined, we add a conditional time overhead per data l. This overhead will be computed and integrated in the WCET of a segment s in JS_τ iff the affinity of some task τ' in $cf(s,l)$ is not identical to the affinity of τ. In other words, there is no need to consider the overheads related to some data l if all segments in conflict with s vis-à-vis l are executed on the same core as s. An overhead $B_s(l)$ is thus associated with a segment $s \in JS_\tau$ and the set $cf(s,l)$ such that $wcs(s)$ is increased by $B_s(l)$ iff the affinity of τ is not the same as all tasks in $cf(s,l)$, i.e., $\exists \tau' \in cf(s,l) : aff(\tau') \neq aff(\tau)$. We denote this condition $CON(s,l)$. Accordingly, we define formally the set $d_{con}(s) \subseteq d(s)$, the set of data written or read by s for which $B_s(l)$ must be integrated in $wcs(s)$, as follows: $d_{con}(s) = \{l | l \in d(s) \wedge CON(s,l)\}$, which is simply the definition given by (5) with the further constraint that at least one task in $cf(s,l)$ has an affinity different than that of the task s belongs to, that is:

$$\forall s \in JS, l \in L : ((l \in d_{con}(s)) \Leftrightarrow \exists s' \in JS_{\tau'}, aff(\tau') \neq aff(\tau) : (l \in dw(s) \wedge l \in d(s')) \vee (l \in dw(s') \wedge l \in d(s))) \quad (6)$$

Then, the value of the total overhead is the sum of all overheads $B_s(l)$ for each data l in $d_{con}(s)$, formally:

$$wcs^\star(s) = wcs(s) + \sum_{l \in d_{con}(s)} B_s(l) \quad (7)$$

With $B_s(l)$ equal to $B_w(l)$, $B_r(l)$ (Sect. 4) or their sum, depending on whether s only writes l, only reads l, or writes and reads l, respectively. Notice here that the overheads added to the WCET as devised by Equation 7 may still be not exact. For instance, it is possible that some data l is in $d_{con}(s)$, and therefore its overhead is added to the WCET of s (Equation 7). But, when analysing all behaviors of the ERTS, it turns out that s never incurs any delay when accessing l (i.e., all other segments that use l in parallel with s always use l before or after s uses it). Equation 7 is therefore another source of pessimism.

However, we recall, through the same Equation 7, that we add the overhead of data l to the WCET of a segment s iff s is effectively in conflict vis-à-vis l with another segment running on a different core, therefore Equation 7 gives us the tightest possible analytical bounds on overheads (i.e., without analyzing all the ERTS possible executions).

Accordingly, time values associated to a job ι become $wj^\star(\iota) = \sum_{s \in S_\iota} wcs^\star(s)$, $final^\star(\iota) = wcs^\star(s_{|S_\iota|})$, and for a task τ $wt^\star(\tau) = \max\{\, wj^\star(\iota) \mid \iota \in J_\tau \,\}$, $blocking^\star(\tau) = \max\{\, \overline{w}_s{}^\star(\tau') \mid \tau' \in lp(\tau) \,\}$, $\overline{w}_s{}^\star(\tau) = \max\{\, wcs^\star(s) \mid s \in JS_\tau \,\}$.

Since $wcs^\star(s)$ can be written as a linear expression of $CON(s,l)$ then $wj^\star(\iota)$, $final^\star(\iota)$, $blocking^\star(\tau)$, $wt^\star(\tau')$ are also linear. Rewriting the schedulability test (1), however, the inequality becomes quadratic due to the $-wt^\star(\tau') \cdot (final^\star(\iota) + wt^\star(\tau'))$ term. By considering $wt^\star(\tau') \geq wt(\tau')$, we can derivate a linear sufficient schedulability test:

$$blocking^\star(\tau) + wj^\star(\iota) + \sum_{\tau' \in sp(\tau)} wt^\star(\tau') + \sum_{\tau' \in hp(\tau)} wt^\star(\tau') \cdot (1 + \tfrac{p(\tau)}{p(\tau')}) - \sum_{\tau' \in hp(\tau)} \tfrac{wt(\tau')}{p(\tau')} \cdot (final^\star(\iota) + wt^\star(\tau')) \leq p(\tau) \quad (8)$$

The new schedulability test given by Inequality (8) comprises therefore another source of pessimism due to replacing $wt^\star(\tau')$ by a lower bound in a negative term. All sources of pessimism in the ILP algorithm will be summarized and further discussed at the end of this section.

5.2.2. ILP Formulation & Implementation Considerations

To model the problem with shared data as an ILP, we start by defining the binary variables $\beta(s,l,c)$, for each segment $s \in JS_\tau$ and each data l such that $cf(s,l) \neq \emptyset$. Then, $\beta(s,l,c) = 1$ if and only if the task τ that contains s and all tasks in $cf(s,l)$ are allocated to the same core c:

$$\beta(s,l,c) = \alpha(\tau,c) \wedge_{\tau' \in cf(s,l)} \alpha(\tau',c) \quad (9)$$

This means that if $\beta(s,l,c) = 1$ then the data l does not induce additional overhead, i.e., $B_s(l)$ should not be considered[11].

We also introduce new continuous variables $\gamma(s,c)$ to represent the WCET of a segment $s \in JS_\tau$ on a core c. With Gurobi, we can use conditions to activate a constraint, and so $\gamma(s,c)$ is constrained by:

$$\gamma(s,c) = \begin{cases} wcs(s) + \sum_{\{l \mid cf(s,l) \neq \emptyset\}} (1 - \beta(s,l,c)) \cdot B_s(l), & \text{if } \alpha(\tau,c) = 1 \\ 0, & \text{otherwise} \end{cases} \quad (10)$$

For each core c, to model $wj^\star(\iota)$ (the worst-case execution time of a job ι on core c) and $final^\star(\iota)$ (its final segment execution time), we introduce the continuous variables $\epsilon(\iota,c)$ (resp. $\lambda(\iota,c)$) constrained by:

$$\epsilon(\iota,c) = \sum_{s \in S_\iota} \gamma(s,c) \quad (11)$$

$$\lambda(\iota,c) = \gamma(s_{|S_\iota|},c) \quad (12)$$

[11] Remark that logical operators can be easily translated into linear constraints, e.g., $a = b \wedge c$ is equivalent to the system $0 \leq b + c - 2a \leq 1$ and in general $y = x_1 \wedge x_2 \wedge ... \wedge x_n$ is equivalent to the constraint $0 \leq \sum_{i=1}^{n} x_i - y \leq n - 1$. As we use the Gurobi solver, there exists a general constraint that facilitates the writing of logical operators. For instance, the constraint $r = and\_\{x_1, \ldots, x_n\}$ states that the binary variable r equals 1 iff all of the binary variables $x_1, \ldots, x_n$ are equal to 1. Thus, for $s \in JS_\tau$ and for all l such that $cf(s,l) \neq \emptyset$, we can simply rewrite in Gurobi the constraint on $\beta(s,l,c)$ as $\beta(s,l,c) = and\_\{\, \alpha(\tau,c), \alpha(\tau',c) \mid \tau' \in cf(s,l) \,\}$.

For modeling $blocking^{\star}(\tau)$ and $wt^{\star}(\tau)$, we introduce the intermediate continuous variables $\nu(\tau, c)$, resp. $\delta(\tau, c)$, that represent the blocking time (resp. the worst-case execution time) of a task τ on a core c. These new variables are constrained by:

$$\nu(\tau, c) = max\_\{\, \gamma(s, c) \mid \; s \in JS_{\tau'}, \tau' \in lp(\tau) \,\} \tag{13}$$

$$\delta(\tau, c) = max\_\{\, \epsilon(\iota, c) \mid \; \iota \in J_{\tau} \,\} \tag{14}$$

where the global constraint $max\_$, defined in Gurobi, sets a decision variable to the max of a list of decision variables.

With all these new intermediate variables, Inequality (8) becomes (under the conditional activation $\alpha(\tau, c) = 1$):

$$\begin{aligned}\nu(\tau'', c) + \epsilon(j, c) + \textstyle\sum_{\tau' \in sp(\tau)} \delta(\tau', c) + \sum_{\tau' \in hp(\tau)} \delta(\tau', c) \cdot (1 + \frac{p(\tau)}{p(\tau')}) \\ - \textstyle\sum_{\tau' \in hp(\tau)} u(\tau') \cdot (\lambda(j, c) + \delta(\tau', c)) \leq p(\tau)\end{aligned} \tag{15}$$

5.3. Summary and Discussion

We obtain therefore a complete ILP formulation of the schedulability test that takes into account fine-grained data sharing overheads. This test populates the definition of the *sched* function in the overall algorithm described at the beginning of this section. However, as explained earlier, linear schedulability tests remain pessimistic in the periodic setting. We identify at least three sources of pessimism in the final schedulability test given by Inequality 8:

- Upper-bounding the blocking term, stemming from Inequality 1, as explained under Sect. 5.1.1,
- Lower-bounding the negative term in the left-hand operand to linearize the test, as explained under Sect. 5.2.1,
- Upper-bounding data-sharing overheads, stemming from Equality 7, as explained under Sect. 5.2.1.

We will eliminate a posteriori all sources of pessimism except for the last one (due to upper-bounded overheads) by exploring all the possible behaviors of the ERTS through compositional model checking (Sect. 6, 7). The last source of pessimism remains because such exploration is based on a model where data sharing overheads are integrated in segments' WCETs following Equality 7 (Sect. 6). However, in practice, this last source of pessimism (based on a fine-grained model as explained in Sect. 4 and throughout this section) has a limited effect on WCRTs (and therefore on schedulability) compared to the first two sources of pessimism (more details in Sect. 7). We argue that this is a fair price to pay in order to enable compositional, scalable verification (Sect. 7.4).

6. TA Model & Compositional Verification

In this section, we develop a TA model in UPPAAL of a realistic ERTS (following the definitions given in Sect. 3) on which various real-time properties

can be verified compositionally. In Sect. 6.1, we detail our UPPAAL model taking data sharing overheads (Sect. 4, 5) and an affinity (Sect. 5) into account. Then, we show how properties can be verified on our model compositionally (Sect. 6.2). In the remainder of this paper, we simply write $wcs(s)$ to denote the WCET of segment s in which data sharing overheads (Sect. 4, 5) have been integrated. In other words, we use $wcs(s)$ to denote $wcs^{\star}(s)$ defined in Sect. 5.

6.1. UPPAAL Model

A network $N = (||_{c \in C} N_c)$ models the ERTS. Each $N_c = (\mathcal{H}_c||_{\tau \in prt(c)} TA_\tau)$ is a network with $\mathcal{H}_c$ (resp. each TA_τ) a TA modeling the scheduler of (resp. a task allocated to) core c. We give a generic description of any N_c with a scheduler and an arbitrary number of tasks allocated to c. We first describe TA_τ apart, i.e., outside N_c. Then, we explain how they are composed, using synchronizations via channels and data variables, with $\mathcal{H}_c$ to get N_c ($\mathcal{H}_c$ is universal so there is no need to present it outside of N_c) as to comply with the behavior in Sect. 3.1.

6.1.1. Task TA_τ (Without Synchronizations)

For each task $\tau = \langle S_\tau, act_\tau, end_\tau, tr_\tau \rangle$ (Sect. 3.1) in $prt(c)$, we want to generate a timed automaton $TA_\tau = \langle L_\tau, l_{\tau 0}, X_\tau, E_\tau, I_\tau \rangle$. For simplicity, we drop τ subscripts and write $l \to l'$ instead of (l, g, λ, l') when the values of g and λ are unimportant or discussed further in the text. We also write $l \to$ (resp. $\to l$) to denote all outgoing (resp. incoming) edges of l. We first give formal definitions interspersed with informal explanations, then provide an example to illustrate.

Definition 1 (TA_τ: Locations & edges)**.** Locations and edges of TA_τ are obtained by applying the following rules to τ:
(1) Locations: each $s \in S$ is mapped to a location with the same name in L. Location *end* is committed. Additional locations are *start* (committed), *wait*, and, for each $s \in S \backslash \{act, end\}$ s.t. $\exists (s, s') \in tr, s' \neq end$, a location s_{pr}. The initial location is *start*.
(2) Edges: each transition $(s, s') \in tr$ is mapped to an edge $s \to s'$ in E. Additional edges are (i) $start \to act$, $wait \to act$, $end \to wait$, and (ii) for each locations' couple (s, s_{pr}), $s \to s_{pr}$ and $s_{pr} \to s'$ for each $s' \neq end$ successor of s.

TA_τ preserves the structure of τ. Added location *start* (resp. *wait*) and its outgoing edge to *act* models activation at time 0 (resp. at each period).

Edge $s \to s_{pr}$ (resp $s_{pr} \to s'$) models preemption (resp. resuming after preemption) where s' is a successor of s in the underlying FSM. The former is taken after executing s (see below). Locations s_{pr} are unneeded when τ terminates after executing s (hence the condition of having at least one successor different from *end*, rule (1)). For similar reasons, *end* is excluded from the successors of s_{pr} (rule (2), see a more detailed explanation on this aspect in the example below). Note that locations s_{pr} are superfluous for tasks having the highest priority in $prt(c)$ (as they are never preempted).

Definition 2 (TA_τ: Clocks & timing constraints)**.** Clocks, guards and invariants in TA_τ (Definition 1) are defined as follows:
(1) Clocks: $X = \{x, y\}$.
(2) Guards and resets: For edges (s, g, λ, act), $g = \top$ ($g = (x = p(\tau))$ if $s = wait$) and $\lambda = \{x\}$. For edges (s, g, λ, s_{pr}), $g = (y \geq bcs(l))$ and $\lambda = \emptyset$ and for edges (s_{pr}, g, λ, s'), $g = \top$ and $\lambda = \{y\}$. For the edge $(end, g, \lambda, wait)$, $g = (x \leq p(\tau))$ and $\lambda = \emptyset$. For the remaining edges (s, g, λ, s'), $g = (y \geq bcs(l))$ and $\lambda = \{y\}$.
(3) Invariants: $I(wait) = (x \leq p(\tau))$. Locations act and $\{s_{pr} | s \in L\}$ are invariant free. Each location mapping a segment from $s \in S \backslash \{end, act\}$ (Definition 1) is associated with the invariant $y \leq wcs(s)$.

Clock y models the execution of a segment s through residing in location s for an amount of time between $bcs(s)$ and $wcs(s)$. Locations s_{pr} are invariant free as they model preemption for an *a priori* unknown amount of time. Constraints over x ensure that τ is activated at exactly each period. Location end is committed to allow reactivation as soon as possible.

Notice here that a timelock is possible at end if TA_τ misses its deadline: time may not elapse (end is committed) and the edge $end \rightarrow wait$ may not be taken (its guard is not satisfied by the valuation of x). This is normal as the model in Sect. 3.1 does not specify the behavior in such case. Below we give a new definition to handle deadline misses and eliminate these timelocks.

Definition 3 (TA_τ : Deadline misses)**.** TA_τ is obtained by applying Definition 1 and Definition 2. Then, two new locations *overshoot* (invariant $I(overshoot) = (x \leq na \cdot p(\tau))$, with na the *next activation* integer variable) and *error* (invariant free), are added with the following edges. $(end, g, \lambda, error)$, where $g = (x > tc \cdot p(\tau))$ and $\lambda = \emptyset$ (tc is the *tolerance constant* strictly positive integer); $tc - 1$ edges $(end, g, \lambda, overshoot)$, where in each n^{th} edge, $n \in 1 \ldots tc - 1$, $g = (x > n \cdot p(\tau) \wedge x \leq (n + 1) \cdot p(\tau))$, $\lambda = \emptyset$ and the non-clock update $na \leftarrow n + 1$ is associated; and $(overshoot, g, \lambda, act)$, where $g = (x = na \cdot p(\tau))$ and $\lambda = \{x\}$.

In brief, to handle deadline misses, we assign a tolerance constant tc, a strict upper bound on the number of consecutive deadlines a task can miss, and add two locations and $tc + 1$ edges to reflect the fact that only deadline misses respecting this tolerance are accepted (in which case location *overshoot* is reached), otherwise an error is raised (through reaching location *error*). Reaching *overshoot*, through taking one of the $tc - 1$ edges $end \rightarrow overshoot$, will determine the earliest possible activation[12] of TA_τ by updating the value of

[12]This kind of modelling is needed when subsequent behavior depends on the actual value of a clock, which cannot be read due to its symbolic nature. The method used here is called the *interval-test method* and is more practical in our case than the binary-search one due to the fact that the value of tc is typically small. Details on both methods are given in [37].

variable na; TA_τ will be activated accordingly at exactly $na \cdot p(\tau)$.

Therefore, for each $\tau \in prt(c)$, TA_τ is built by applying Definition 3 (which uses in turn Definitions 1, 2). Below we give an example illustrating this. In our examples hereafter, we upperbound tc by the value of 2 to simplify the figures.

Example. Fig. 2 (bottom) shows TA_τ obtained through applying Definition 3 to the FSM of a task τ (top). Notice how segment *s4* does not need a preemption location $s4_{pr}$ since the task can only terminate after executing *s4* (Definition 1, rule (1)). Notice also how location *s3_pr* is not a predecessor of *end* even though *end* is a successor of *s3* in the underlying FSM, which stems from the fact that *end* is excluded from preemption locations successors (Definition 1, rule (2), additional edges). This is because if TA_τ is marked for preemption at *s3*, then either TA_τ will be put on hold and resumed later at *s4*, or simply transit to *end* and terminate since *end* is a possible successor (and such transition is already accounted for by the edge $s3 \rightarrow end$, Definition 1, rule (2)).

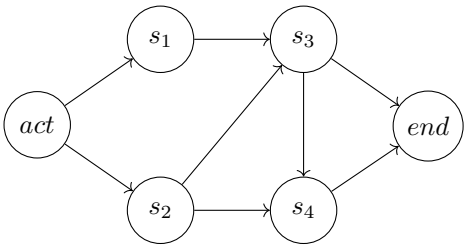
Location *error*, if reachable, allows to drop the verification if there is a deadline miss beyond the tolerance tc. Location *overshoot* permits to activate the task again starting at the next period if the deadline is missed but the tolerance is respected, i.e., the deadline is missed by an amount of time comprised between 0 excluded and $tc \cdot p(\tau) - p(\tau)$ ($p(\tau)$ is denoted by the constant integer P in the figure). For HRT tasks, tc is equal to 1 (no tolerance for deadline misses) and thus location *overshoot* can simply be removed.

Note on tasks dependency. It is important to recall here that TA_τ, the TA of task τ as given by Definition 3, already includes the effect of tasks dependencies due to data sharing. Indeed, as mentioned earlier, $wcs(s)$ refers throughout this section and beyond to the WCET of segment s in which the overheads due to data sharing, as computed in Sect. 5, has been already integrated, therefore breaking data dependency (faithfully representing it without explicitly modeling it). Although these overheads may be not exact, tightening them in a fine-grained manner, as shown in Sect. 5, leads to a limited effect on WCRTs (we will precisely quantify this effect in Sect. 7 on real-world case studies). As mentioned in Sect. 3, breaking data and core dependency is a cornerstone of our approach to enable compositional verification.

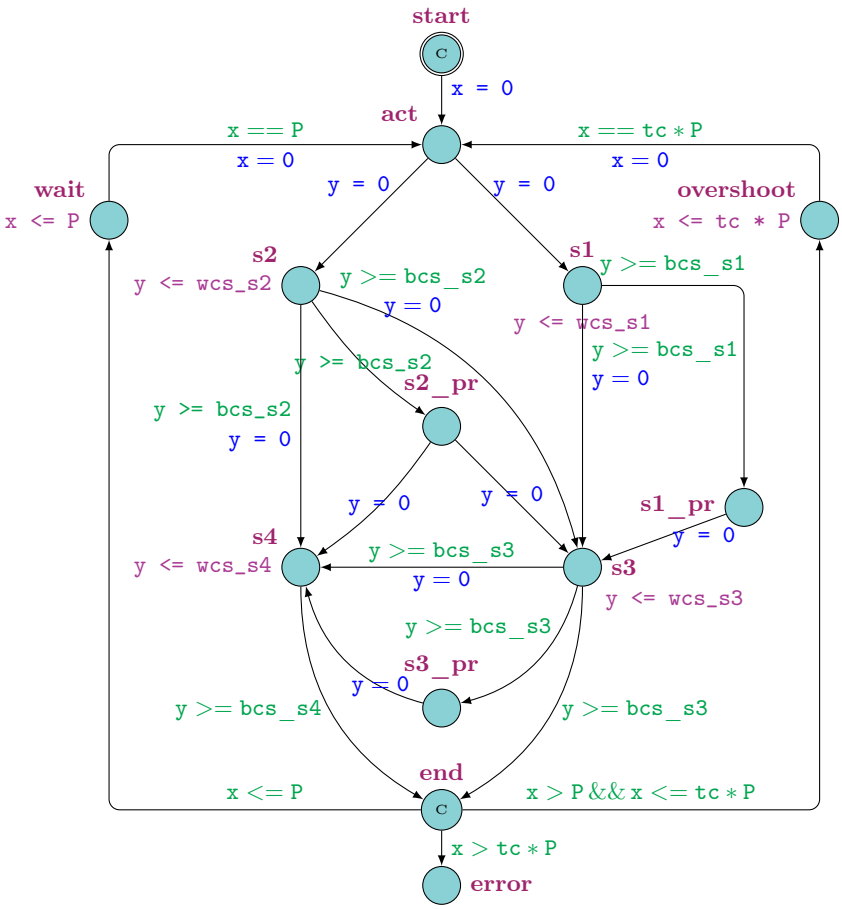
6.1.2. Gluing the TA

Definition 3 provides the means to generate TA_τ for any task $\tau \in prt(c)$. These tasks, however, need to communicate with the scheduler $\mathcal{H}_c$ in order to obtain the network N_c modeling some core c. In the following, we first define data variables/constants/functions, channels and priorities used to realize such communication, then present the network N_c and explain through an example how its behavior complies with the one described in Sect. 3.1.

Data variables/constants. For each TA_τ, we define three local constants: *id* (a distinct integer in $0 \ldots |prt(c)| - 1$), *pi*, the priority of τ, and P, its period; and two local variables *left* and *right* for $\mathcal{H}_c$. Q_c, c's queue, is modeled as an array



(a) τ FSM



(b) TA_τ (Definition 3)

Figure 2: FSM and resulting TA model (in UPPAAL notation) for a task τ. P is a constant equal to $p(\tau)$. Note that $x = 0$ on edge *start* $\rightarrow$ *act* is superfluous since *start* is both committed and initial (this edge is taken at $x = 0$ anyways, see the synchronized model in Fig. 3). Note also that tc is upperbounded by 2, i.e., this representation is valid for $tc = 1$ (in which case location *overshoot* is superfluous) or $tc = 2$, and therefore variable na is not needed and there is only one edge from *end* to *overshoot*.

shared between $\mathcal{H}_c$ and each TA_τ, where each cell contains an id and a priority. For simplicity, we say, when TA_τ inserts its id and pi in Q_c, that TA_τ is inserted in Q_c. Finally, $\mathcal{H}_c$ uses an array of Booleans B_c, indexed over ids, to track the only TA_τ currently executing, if any.

Functions. A number of user-defined functions are used in updates over edges/-transitions. In particular, function *add* allows inserting a task in Q_c, function *sort* sorts tasks in Q_c (when an insertion happens) according to their priorities (if different) or to their activation time (if equal), function *resort* resorts the already sorted Q_c when a preemption takes place, and function *shift* shifts each element of Q_c, starting at $Q_c[1]$, one position to the left (i.e., a pop operation on the queue). These functions are explained briefly later in this section and in more details in the public artefacts of this paper (link provided in Sect. 7.3).

Handshake channels. We define ins_c, $rel_c[prt(c)]$ (an array of channels), pre_c and ter_c to, respectively, insert, release, preempt and terminate a task.

Broadcast channels. We define cmp_c (resp. exe_c and $toend_c$) for internal computations of $\mathcal{H}_c$ (resp. each TA_τ). Both cmp_c and exe_c have no receivers.

Priorities. For correctness, two priorities are necessary. First, if a lower-priority task is marked for preemption, it must be preempted as soon as it finishes executing the current segment instead of continuing that is $exe_c < pre_c$. Second, if some tasks are activated at the same time, they must all be inserted in Q_c at their activation date before performing any further computations, that is $cmp_c < ins_c$.

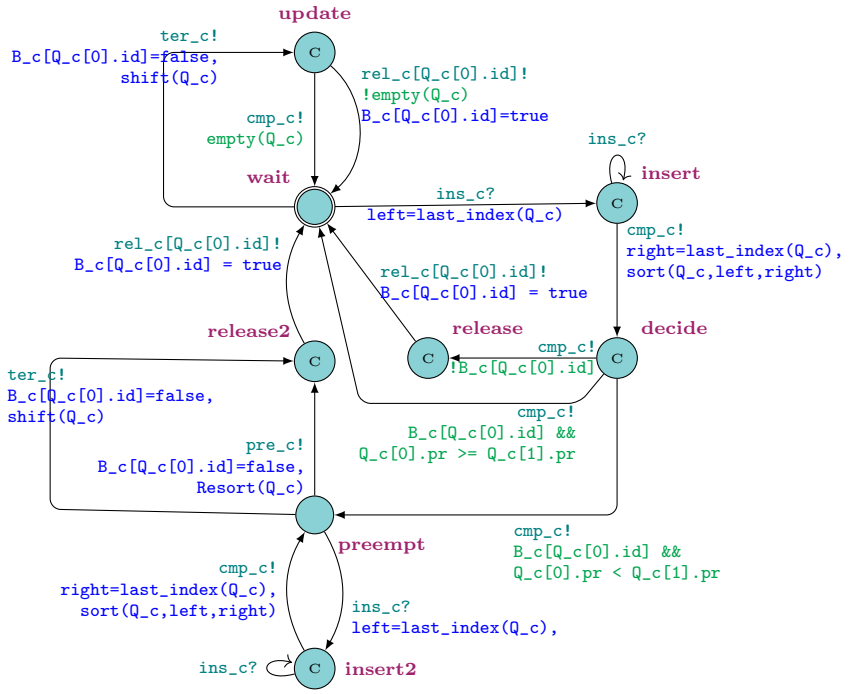
Network behavior. Fig. 3 shows TA_τ from Fig. 2 composed with $\mathcal{H}_c$. In general, tasks TA_τ are updated as follows:

- Edges $\rightarrow$ *act* are synchronized (sender) on ins_c and augmented with the update *add*,
- Edges *act* $\rightarrow$ and $s_{pr} \rightarrow$ are synchronized (receiver) on $rel_c[id]$,
- Edges $\rightarrow s_{pr}$ are synchronized (receiver) on pre_c,
- Edges *end* $\rightarrow$ are synchronized (receiver) on ter_c,
- Edges $\rightarrow$ *end* are labelled with $toend_c$ (sender),
- Remaining edges are labeled with exe_c (sender).

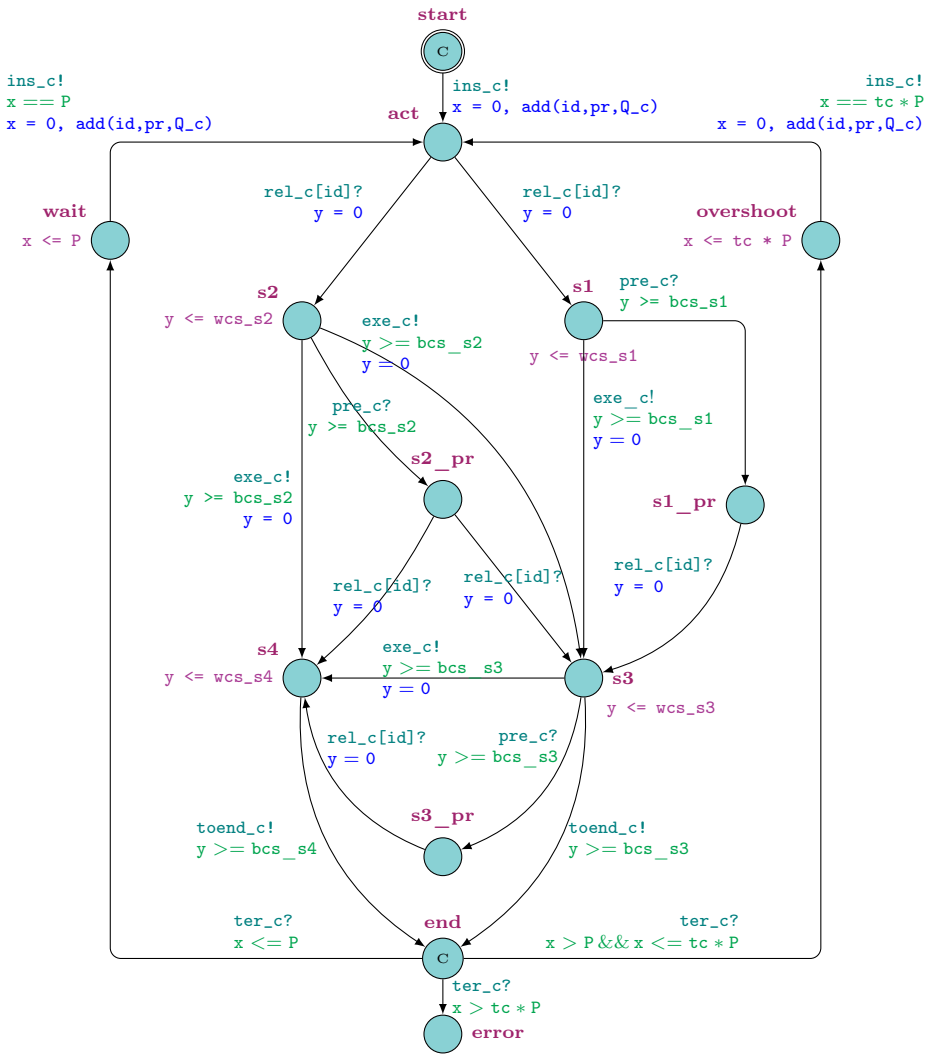
We sketch in the following the behavior of a network N_c, with an arbitrary number of TA_τ (we still rely on Fig. 3 as a support), and explain how it obeys the behavior described in Sect. 3.1. In particular: (i) only one task executes at a time, (ii) a lower-priority task is preempted properly, and (iii) tasks in Q_c are correctly ordered.

Let t be some point in time s.t. $\mathcal{H}_c$ is at *wait*. *As soon as* some TA_τ is activated, say at time t', $\mathcal{H}_c$ and TA_τ synchronize on ins_c and transit simultaneously to *insert* and *act*, respectively, as TA_τ is inserted in Q_c (function *add*). The "as soon as" urgency is enforced either by the invariant at *wait* and the guard on *wait* $\rightarrow$ *act*, or the fact that location *start* is committed, depending on whether TA_τ is at *wait* or *start*, respectively. Channel ins_c is triggered again until all TA_τ activated at t' (self-loop at committed location *insert* in $\mathcal{H}_c$) are inserted in Q_c (ins_c has the a higher priority than cmp_c). At *decide* (committed), tasks at positions $pos \ldots |prt(c)| - 1$ in Q_c are ordered through the function *sort* (edge *insert* $\rightarrow$ *decide*), which sorts tasks activated at t' (in the portion *left*... *right* of Q_c), then allows only tasks with higher priorities to cut ahead tasks with strictly lower priorities that were already in Q_c, thus complying with the ordering rules in Sect. 3.1.

If Q_c was empty before t' (i.e., $left = 0$) then $pos = 0$ and the task at $Q_c[0]$ is released (path *decide* $\rightarrow$ *release* $\rightarrow$ *wait*). Otherwise, *pos* equals 1 as the task at $Q_c[0]$ remains in place. From *decide*, either preemption is needed (guard on *decide* $\rightarrow$ *preempt*) or not (guards on *decide* $\rightarrow$ *wait* and *decide* $\rightarrow$ *release*). If no preemption is needed (the task at $Q_c[0]$, say TA_τ, has the highest priority in Q_c), either *decide* $\rightarrow$ *release* (if TA_τ is not released yet) or *decide* $\rightarrow$ *wait* (otherwise) is taken. In the former case TA_τ is released, synchronizing on $rel_c[id]$, as $\mathcal{H}_c$ transits back to *wait* and TA_τ to some location where it starts traversing a path corresponding to some job, by emulating the execution of each segment s



(a) $\mathcal{H}_c$



(b) TA_τ

Figure 3: TA_τ (Fig 2) and $\mathcal{H}_c$ with synchronizations. Notice the committed locations in $\mathcal{H}_c$ to enforce sequences of timeless actions corresponding to scheduling decisions. For instance, location *insert* is committed to force inserting all tasks activated at the same time one after another without interference from other actions, thus removing unnecessary interleaving.

in such path through residing in location s between $bcs(s)$ and $wcs(s)$.

If preemption is needed, $\mathcal{H}_c$ transits to *preempt*. While waiting for the preempted task, say $TA_{\tau'}$, to finish executing its current segment, say s, activated tasks after t' are inserted and sorted in Q_c in the same fashion. Since $pre_c > exe_c$,

the next edge $TA_{\tau'}$ takes is either $s \to s_{pr}$ or $s \to end$ (non-deterministically[13] if they both exist, e.g., if preemption happens at $s3$). In the first case, synchronizing on pre_c, $TA_{\tau'}$ is preempted as $\mathcal{H}_c$ removes $TA_{\tau'}$ from the head of Q_c and reinserts it according to its priority (function *resort*) and transits simultaneously to *release2*. When $TA_{\tau'}$ is released again, it resumes by transiting to one of the successors of s. In the second case, $TA_{\tau'}$ terminates as $\mathcal{H}_c$ removes it from Q_c, function *shift*). In either case, $\mathcal{H}_c$ transits back to *wait* (from *release2*) immediately as it releases the task at $Q_c[0]$. Finally, edge *wait* $\to$ *update* in $\mathcal{H}_c$ corresponds to some task TA_τ terminating while $\mathcal{H}_c$ is idling. After such termination, $\mathcal{H}_c$ transits back immediately to *wait* taking one of the two edges *update* $\to$ *wait*: either there is nothing to do (Q_c is empty after removing TA_τ) or the task at $Q_c[0]$ is released (otherwise).

Therefore, in compliance with Sect. 3.1, preemption is correctly handled, and only one task executes at a time, as (i) only the task at $Q_c[0]$ is released and (ii) no task is released unless the last released task is preempted or terminated.

6.2. Compositional Verification

So far, we have devised a TA model for any ERTS following the assumptions in Sect. 3. The ERTS model is thus $N = (||_{c \in C} N_c)$, a synchronization-free network (Sect. 2.3). Indeed, all synchronizations in N are local to each N_c, i.e., there is no synchronization (using channels or shared variables) between any two TA belonging to N_c and $N_{c'}$ in the network N such that $c \neq c'$. N can be thus treated as a synchronization-free network of TA by considering each N_c as a single TA resulting from the synchronized product of $\mathcal{H}_c$ and TA_τ for each $\tau \in prt(c)$. We show next that TCTL properties (Sect. 2.3) can be verified compositionally on N, i.e., without considering all N_c networks, thus reducing the underlying state-space size.

Notations. Let N be a TA network $N = (||_{i \in 1..n} A_i)$ (Sect. 2.3) and $R(N)$ the set of its possibly infinite runs of the form $(\mathcal{L}^0, v_0) \to \ldots$. Similarly, $R(A_i)$ is the set of runs of TA A_i evolving outside the network, i.e., in the singleton network A_i. We denote by $R_i(N)$ the set of projections of each run in $R(N)$ on $R(A_i)$, i.e., each $(\mathcal{L}^0, v_0) \to \ldots$ in $R(N)$ where (i) each discrete transition not involving A_i (i.e., $(\mathcal{L}, v) \to (\mathcal{L}[\mathcal{L'}_j/\mathcal{L}_j], v)$ with $j \neq i$) is removed and (ii) $(\mathcal{L}, v)$ are replaced with $(\mathcal{L}_i, v(X_i))$, their projections on locations and clocks of A_i.

Definition 4 (Independent TA)**.** In a network $N = (||_{i \in 1..n} A_i)$, each A_i is called independent iff $R(A_i) = R_i(N)$.

Definition 4 states the condition under which the behavior of a TA (be it a single TA or a network) is not affected by that of its neighbors in the composition, i.e., its runs remain the same whether evolving inside or outside the network.

[13] Remark the use of the distinct channel $toend_c$, with no priority ordering with either exe_c or pre_c, on $s3 \to end$ to allow a correct modeling of such non-determinism.

Proposition 1. *In a synchronization-free network $N = (||_{i \in 1..n} A_i)$, if each A_i is timelock-free, then each A_i is independent.*

We explain next the reasoning behind Proposition 1. From the semantics in Sect. 2.3, if a network $N = (||_{i \in 1..n} A_i)$ is synchronization free, a discrete transition corresponds to one edge, thus each A_i evolves independently at the discrete level. The only dimension that can be problematic is thus at the time progress level, namely if conditions over the evolution of time result in a semantical pathology, i.e., global time may not evolve beyond some instant t, resulting in preventing some A_i from realizing some of their behaviors past t when inside N, which invalidates the independence condition (Definition 4). These ill-formed cases correspond to timelocks (Sect. 2.3). More formally, if some A_j in N has a timelock and some A_i in N is timelock free, then A_i is not independent in N. Indeed, if some A_j timelocks at some global time t, i.e., time may not evolve beyond t (because, for instance, no delay or discrete transition can be taken, otherwise $I_j(\mathcal{L}_j)$ or $I_j(\mathcal{L}'_j)$ would be violated, Sect. 2.3), then the corresponding run in $R_i(N)$ for any timelock-free A_i is truncated of all transitions in $R(A_i)$ after time t, and therefore $R_i(N) \neq R(A_i)$. A more informal, intuitive account on timelocks being the only time-wise pathological behavior preventing independence is given in [16].

In brief, even though our network $N = (||_{c \in C} N_c)$ is synchronization free, we need to make sure that each N_c comprises no timelock to guarantee independence[14].

Proposition 2. *The network $N_c = (\mathcal{H}_c||_{\tau \in prt(c)} TA_\tau)$ is timelock free.*

Proof. The proof is given in Appendix B. □

Proposition 3. *In $N = (||_{c \in C}(N_c))$, each N_c is independent.*

Proof. N is synchronization free, and by Propositions 2, each N_c is timelock free. Therefore, by Proposition 1, each N_c is independent. □

By Proposition 3, if a property holds in N' outside of N (where N' is the composition of some N_c appearing in N), then it will hold in N, and conversely. We can therefore verify properties *l*ocal to N' (properties involving networks in N' only) on N' alone, thus reducing the state space.

6.2.1. Properties of Interest

Properties local to a core. The first property to verify in some N_c is $A\Box$ not $(\bigvee_{\tau \in prt(c)} TA_\tau.error)$ (*error* locations are never reached) as a quick check; if it is false, then the verification is dropped as at least one task violates its deadline (beyond the tolerance tc for non HRT tasks).

Second, if the ILP algorithm does not schedule some tasks (Sect. 5, 7), we are interested in verifying their schedulability. This is done using the query

[14] And actually, we need to make sure our models are exempt of timelocks anyways because timelocks reflect modeling flaws regardless of independence.

$A\Box$ not ($TA_\tau.overshoot$). However, this is not enough, as TA_τ can starve in an s_{pr} location after preemption. We need therefore to verify also the properties $TA_\tau.start \rightsquigarrow TA_\tau.wait$ (TA_τ does not starve during the first execution), and $TA_\tau.wait \rightsquigarrow TA_\tau.end$ and $TA_\tau.end \rightsquigarrow TA_\tau.wait$ (TA_τ does not starve during subsequent executions)[15]. Since (compositional) model checking is exhaustive and exact, this step may reveal that some tasks deemed unschedulable by the ILP pass are in fact schedulable, thus overcoming a posteriori the pessimism of the schedulability tests implemented within the ILP algorithm (Sect. 5, 7).

Third, we also want to go further and quantify the WCRT of each task τ using the query $sup\{TA_\tau.end\} : TA_\tau.x$ (Sect. 2.4). This allows us to compute tight WCRTs, thus squeezing the pessimistic bounds computed by the ILP pass.

All these properties are local to the core c to which the task of interest is allocated to, i.e., we can verify the schedulability and compute the WCRT of some task τ on N_c only, s.t. $\tau \in prt(c)$.

Properties local to a set of cores. Another important bound to quantify is the time separating the occurrence of two events w and r (e.g., writing and reading a data). This can be done locally on $N^{rw} = (||_{c \in C^{rw}} N_c)$, where $C^{rw} \subseteq C$ is the subset of cores that produce w or r. Typically, to compute such a bound, we use the *sup* query on a location of an *observer* that we compose with N^{rw} (Sect. 7).

Scalability. In all cases, our approach allows a correct verification on only a portion of the whole network representing the ERTS, therefore reducing the state space to explore. This reduction is maximal for properties local to one core c, where the verification is carried out only on the subnetwork N_c. For properties local to a set of cores, the worst case is when $C^{rw} = C$, where verification is no longer compositional (all of the network must be taken into account); this case is unlikely to happen in practice (Sect. 7). This scalability comes at the cost of an acceptable pessimism due to upperbounding data sharing overheads, as we will thoroughly show in Sect. 7.

7. Evaluation

We first introduce technical details on our fully automated toolchain (Sect. 7.1). Then, we evaluate our approach on three real-world complex case studies (Sect. 7.2). Afterwards, we discuss the experimental results and, accordingly, the benefits and limitations of our approach (Sect. 7.4). Our experiments are publicly available and fully reproducible in an automated manner (Sect. 7.3). All details on scalability (memory consumption and verification time) in this section are reported by UPPAAL on a mid-range computer featuring an Intel Core i7 processor and 8GB of RAM.

[15] Actually, as a positive side effect, the last three properties (absence of starvation) also prove the absence of timelocks in the ERTS under scrutiny which provides another handy sanity check (more explanation in the artefact accompanying this paper, link under Sect. 7.3)

7.1. Technicalities

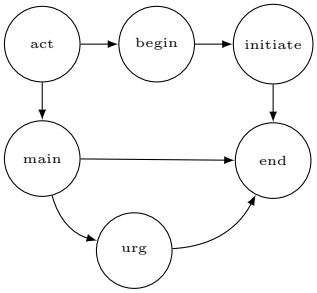
Toolchain. The entry of our chain is a .rt file, with a simple grammar where FSMs are expressed through states and their successors (see below). To further automatize the process, we wrote two more translators corresponding to the optional Step 0 in Fig. 1. The first (resp. second) translator generates from any robotic application written in GenoM3 [57, 36], say file.gen (resp. from the XML description of the automotive case study, say file.xml) the corresponding file.rt file. Following Steps 1 and 2 in Fig. 1, the ILP solver processes file.rt and generates a total affinity, a partial affinity or a failure message, via the algorithm explained in Sect. 5 (these steps are skipped in the automotive industrial challenge experiments as the affinity is already provided with the case study). Affinities are generated in file.sol. Finally, file.rt and file.sol are fed to another translator that generates an UPPAAL model for each core c in file_c.xta (Step 3 in Fig. 1). Steps 1-3 take into account data sharing overheads as computed in a fine-grained fashion (Sect. 4, 5).

Handling loop behaviors. A typical pattern in periodic control tasks is to resume at some control loop state at each subsequent period (e.g., periodic visual servoing). Fig. 4 (top) shows a simplified version of task *control* specification in GenoM3 from the drone case study (Sect. 7.2.1) implementing a similar behavior (*ether* is a GenoM3 keyword for task termination). Such specification underlies an *extended* FSM, where the keyword *pause* preceding e.g., the transition $(initiate, main)$ is interpreted as a special type of termination; it denotes that task *control* terminates after executing *initiate*, and resumes at the next period by executing segment *main*. This model is amenable to the generic FSM model given in Sect. 3.1 (as mentioned within the same section), by simply removing each pause transition (s', s) and adding (i) a transition from s' to *end*, to denote that the task terminates when taking the removed pause transition and (ii) a transition from *act* to s, to denote that the task may start executing at s (Fig. 4, bottom-left). The UPPAAL model of task *control* implements therefore the generic FSM following Definition 3. Note, however, that the generic FSM contains more behaviors than that of the extended FSM underlain by the GenoM3 specification in Fig. 4 (top), e.g., *control* can start executing at *main* since the beginning, i.e., even before any pause transition to *main* takes place. To eliminate this, we simply use local Booleans p_s to track taking pause transitions (s', s), thus allowing to resume at s only after a pause transition to s has been taken in the last execution. This is shown in Fig. 4, bottom-right (simplified version without synchronizations, WCET/BCET constraints and preemption) where p_main is initialized to false. Notice the two edges out of *urg*, one corresponding to the special pause termination (with the update $p\_main = true$) and the other to non-pause termination.

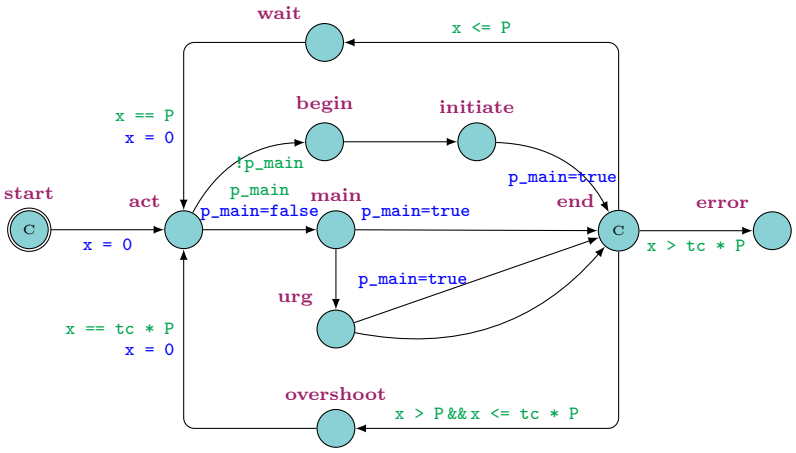
The .rt format. The .rt format follows a simple grammar providing real-time information of some ERTS while abstracting away any information that is not relevant to our real-time analysis. The keyword $task\_i$ permits enumerating the tasks in the ERTS, through providing the name of one distinct task after each

```
task control {
   period P;
   state<begin> transition_to initiate;
   state<initiate> transition_to pause::main;
   state<main> transition_to pause::main, urg;
   state<urg> transition_to pause::main, ether;
};
```

(a) Specification of task *control* (simplified)



(b) FSM of task *control*



(c) UPPAAL TA of task *control*

Figure 4: Handling loop behaviors. The GenoM3 description is simplified such that only states and transitions appear. In the FSM, the transition *urg* → *end* resulting from the removal of the pause transition *urg* → *main* is omitted (since the FSM transitions are not interpreted and there was already such transition in the FSM). In the UPPAAL model, all synchronizations, preemption locations, invariants, and constraints over clock y are removed for readability.

indexed keyword over $i \in 1..|T|$. Similarly, s_i_j is to enumerate segments in the task whose keyword is $task\_i$, and $sName\_succ\_k$ to enumerate the successors of segment *sName*, captured through its unique keyword s_i_j. Therefore, there is no need to enumerate jobs as the FSM for each task is accordingly specified. In a similar manner, we use keywords to:

- specify $p(\tau)$, $\pi(\tau)$, and to denote whether τ is an HRT task (i.e., whether $\tau \in T_h$ for each task τ),
- specify the number of cores,
- specify $wcs(s)$, $bcs(s)$, $dr(s)$ and $dw(s)$ for each segment $s \in JS$,
- specify $\rho(l)$ for each data $l \in L$.

The keywords *act*, *end* and *loop* (for loop behaviors using pause-like transitions as shown above) are reserved.

7.2. Experiments

We evaluate the scalability and benefits of our approach on two real autonomous robots (Sect. 7.2.1) and a real automotive case study from an industrial challenge (Sect. 7.2.2). In both cases, we verify important properties including the computation of WCRTs and other bounds between events occurring on different cores. For each task τ, we distinguish between two types of WCRT:

- pessimistic WCRT, denoted $pesW(\tau)$, as reported by the ILP pass (i.e., using Inequality 8),
- tight[16] WCRTs, denoted $tW(\tau)$, as computed by UPPAAL using compositional model checking on the model devised in Sect. 6.

According to the summary given in Sect. 5.3, $pesW(\tau)$ corresponds to the schedulability test suffering from at least three sources of pessimism whereas $tW(\tau)$ comprises only one possible source of pessimism due to including analytical data sharing overheads, albeit in a fine-grained fashion, in segments' WCETs.

We want however to go further and quantify the effect of fine-grained data sharing overheads on WCRTs and schedulability (we recall that these overheads are the tightest that can be obtained analytically, Sect. 5). For this, we also compute the optimistic WCRT of τ, denoted $optW(\tau)$, corresponding to the most optimistic scenario of data sharing, i.e., when there are no overheads at all. For this, we rely on compositional model checking, using UPPAAL, on a model where segments' WCETs are the original ones (Sect. 3.1).

Accordingly, the exact WCRT of task τ is comprised between its optimistic WCRT and its tight WCRT, that is in the interval $[optW(\tau), tW(\tau)]$. We will see that, thanks to the careful fine-grained computation of data sharing overheads, $tW(\tau)$ is typically very close to $optW(\tau)$, with no effect on schedulability in our case studies, whereas the purely analytical $pesW(\tau)$ leads to a lower schedulability ratio, notably in the drone case study under Sect. 7.2.1.

7.2.1. Autonomous Robots

The robotic case studies, originally described in [32, 33], consist in a drone and a terrestrial robot, each embedding a four-core hardware. The drone (resp. terrestrial robot) is illustrated in Fig. 5 top-left (resp. right), where each box is a *component* with one or two tasks. Overall, there are nine tasks and 57 segments in the drone and ten tasks and 61 segments in the terrestrial robot. Both applications do not scale with a global model checking approach [33, 34].

Segments' BCETs are set to zero and their original WCETs are experimentally evaluated by taking the maximum execution time of each segment following a

[16] Note the slight difference in terminology with the standard one in real-time systems' research. Here, tight does not mean exact; but rather something between exact and pessimistic as explained throughout this section.

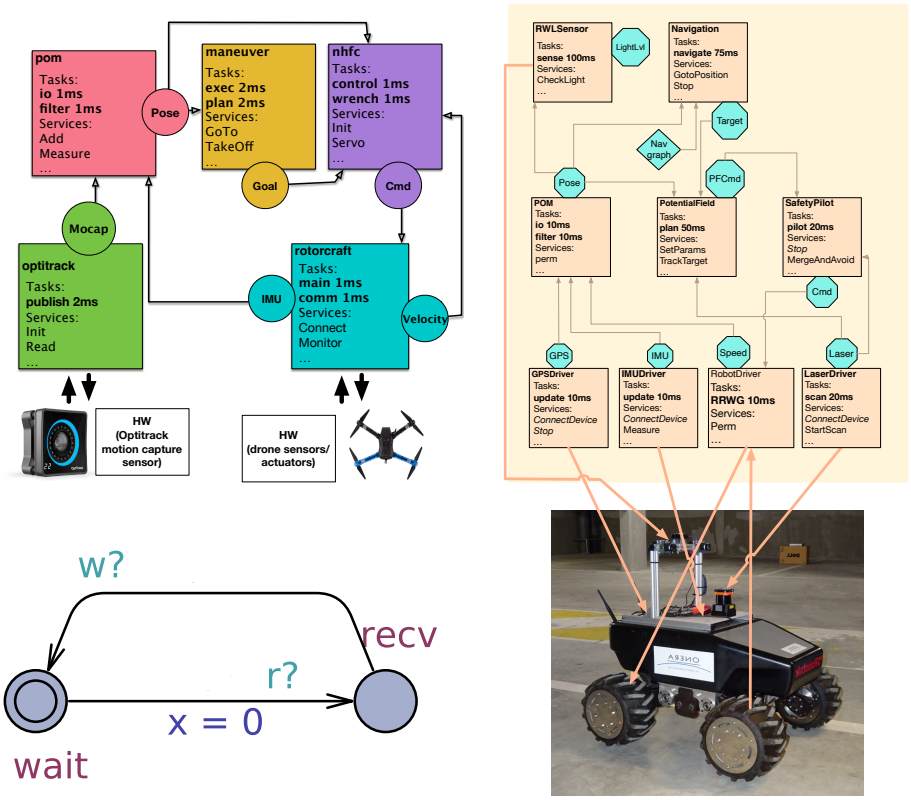


Figure 5: Case studies and observer

series of random runs [33]. We refer the reader to [33] for in-depth details of both robots. We give in the following examples of properties we verified for each application.

Drone. Table 2 gives a summary of tasks in the drone with their priorities and periods, and whether they are HRT tasks or not (we refer the reader to [34] for more details on HRT/non-HRT decomposition of tasks in an older version of this case study). Since the smallest non-zero timing constraint is $10\mu s$, time values are given with a resolution of $10^{-5}s$, e.g., 100 time units is equal to $1ms$.

We start the experiments by Step 1, which fails to schedule all tasks, drops the non-HRT task plan and fails again, then drops another non-HRT task exec and generates a partial affinity satisfying the schedulability of the remaining (seven) tasks. We use this partial affinity as the grounds to try and achieve a total one. We notice that cores 2 and 4 have each only one task, respectively control and comm (both HRT), so we allocate the dropped tasks plan and exec to cores 2 (together with control) and 4 (together with comm) respectively. The

Table 2: Drone case study: task information, affinity, and resulting task WCRTs.

Task	Period	Priority	HRT	Affinity	$optW$	tW	$pessW$
main	100	2	Yes	3	65	69	69
comm	100	2	Yes	4	60	66	81
io	100	2	Yes	1	35	43	63
filter	100	2	Yes	1	35	43	63
wo	100	2	Yes	3	65	69	69
control	100	2	Yes	2	64	74	**102**
publish	200	1	No	1	90	98	165
plan	200	0	No	2	177	187	**209**
exec	200	0	No	4	170	182	**208**

total affinity that we obtain accordingly is given in Table 2. We then generate an UPPAAL model and investigate this further. We verify, one file_c.xta at a time, the properties explained in Sect. 6. We ensure that no *error* location is reachable, then, that no task ever starves using the leadsto properties as explained in Sect. 6, then, for all tasks allocated to cores different than cores 2 and 4, that *overshoot* is not reachable. Afterwards, we focus on cores 2 and 4 (drone_2.xta and drone_4.xta). We query, on drone_2.xta, whether location *overshoot* is reachable in any of both tasks assigned to core 2 using the formula $E \diamond (TA_\tau.overshoot)$ with $TA_\tau \in \{control, plan\}$. Surprisingly, it is not, which means that both tasks are schedulable. The same thing happens with drone_4.xta; both tasks assigned to core 4 are schedulable. Therefore, all tasks are schedulable provided the partial affinity given by the ILP algorithm that we completed manually (Table 2). This is an intriguing result since the ILP step has not found a total solution. As explained in Sect. 5, 6, this should be due to the pessimism in linear schedulability tests for periodic workloads, a known issue in real-time scheduling.

To investigate this further, we fix the affinity to the one given in Table 2, since our compositional model checking results show that it is a valid solution. Then, we compute using UPPAAL the WCRTs for all tasks and recompute the WCRTs using Inequality 8, i.e., we compute, respectively, $tW(\tau)$ and $pesW(\tau)$ for each task τ in Table 2. This will give us insights on the reason behind the ILP pass failing to find a solution. The results are given in Table 2.

As we can see under the column $pesW$, the schedulability ratio using Inequality 8 is $\frac{6}{9}$ with tasks control, plan and exec deemed unschedulable because their (pessimistic) WCRTs are larger than their periods (marked in **bold**). This actually explains why the ILP algorithm refrained from e.g., assigning both control and plan to the same core: in doing so, according to the ILP's computations, none of them would be schedulable[17]. On the other hand, thanks to the scalable and tight computations via compositional model checking (column

[17]This is more thoroughly investigated in the artefact, see the provided links under Sect. 7.3.

tW), the schedulability ratio is equal to 1. To further tune the WCRTs, we compute, using UPPAAL, the optimistic WCRT of each task (where data sharing overheads are assumed null, column $optW$). Accordingly, we can situate each task's exact WCRT within the interval $[optW, tW]$. Notice how these intervals are typically tight due to the fine-grained computation of data sharing overheads. In Sect. 7.2.2, we will see how these intervals become even tighter when WCETs and data penalties estimation is more elaborate.

Finally, we report on the scalability of our approach on this case study. Both the ILP pass and the UPPAAL verification of properties (including WCRTs computation) were extremely fast, making the whole process last barely a few seconds per property. For model checking alone, and thanks to our compositional approach, the verification time was upper-bounded by *0.01s* and the memory consumption by merely *15MB*.

Terrestrial Robot. For the terrestrial robot, Step 1 succeeds, and the schedulability is confirmed with UPPAAL. We now try to quantify the maximum time separating perceiving the environment (reading the laser sensor, task *scan*) and using that latest perception to compute a new speed (task *plan*). Since Step 1 allocated these tasks to two different cores c and c', we need to take both UPPAAL networks N_c and $N_{c'}$ into consideration, and this is of course interesting scalability-wise. We identify segments s and s' in charge of, respectively, reading the laser (task *scan*) and computing the speed (task *plan*) and replace the labels of their outgoing edges with $r!$ and $w!$, respectively (r and w are broadcast channels). Then, we compose N_c and $N_{c'}$ with the observer shown in Fig. 5 (bottom left). Finally, we query the maximum value of clock x while the observer is at location *recv*. Note that this observer is not suitable for "minimum time" properties (r events at *recv* are ignored) and that here also we verify beforehand $wait \rightsquigarrow recv$ and vice versa. We quantify the tight upper bound as equal to *29* ms. This is an important information to take into account in order to compute e.g., the robot's reactivity to appearing obstacles.

Scalability-wise, the ILP pass was again extremely fast lasting barely a second. The model checking phase lasted roughly one minute with less than *100MB* of RAM consumption.

7.2.2. WATERS Industrial Challenge

To test further the resilience of our approach notably w.r.t. scalability, we apply it to the APP4MC specification of the automotive case study from the WATERS 2017 industrial challenge [41] featuring a four-core hardware. Since our approach is suitable for periodic tasks only, we apply it to core 2 that has seven tasks with 710 segments (giving over 37,000 segments executed in a hyperperiod). The tasks with their periods, priorities and number of segments are given in Table 3. Note that due to the length of tasks names in the original description, we have simplified them for readability.

This example is of a great interest to us especially because it presents two major threats to model checking scalability. First, the number of segments (called runnables in the automotive jargon) is huge. Second, the WCETs and data

Table 3: WATERS challenge: task information (core 2), and resulting task WCRTs.

Task	Period ($\times 10^6$)	Priority	# Segments	$optW$	tW
T_2	2	13	28	357,269	357,362
T_5	5	12	23	823,170	823,635
T_20	20	9	307	6,974,083	6,980,293
T_50	50	8	46	8,718,349	8,725,762
T_100	100	7	247	14,499,273	14,514,138
T_200	200	6	15	14,568,511	14,584,180
T_1000	1,000	5	44	14,637,078	14,653,884

penalties estimation is quite elaborate, which results in a nanosecond resolution (one time unit is equal to $10^{-9}s$). With timing constraint varying from 0 to 1 billion ($1s$), scalability is threatened as the construction of zone graphs is quite sensitive to such large variations.

The full description is downloadable in an XML format from the challenge official forum[18]. The first thing we have done is changing the frequency from *200MHz* to *400MHz*; this is because tasks are inherently non schedulable (resp. schedulable) with the former (resp. the latter), as shown in [73]. As explained at the beginning of this section, we fully automatized Step 0 for this case study as well: a *waters_c.xta* is automatically generated from the XML description for each core c, given the affinity provided within the same description. Then, we focus on *waters_2.xta* comprising only periodic tasks (Table 3). Given that (i) the affinity is given and (ii) the model is quite interesting to evaluate scalability even further, our experiments for this case study will focus on compositional model checking at Step 3: its cost (verification time and memory consumption) and its preciseness (tightness of exact WCRTs intervals). In particular, the pessimistic WCRTs will not be considered since (i) they are out of focus for this case study (see above) and (ii) tasks schedule quite comfortably at *400MHz*.

The tight and optimistic WCRT of each task on core 2 is computed using UPPAAL and provided in Table 3. Notice how the intervals containing the exact WCRTs are quite narrow, thanks to the fine-grained computation of data sharing overheads. Indeed, the largest difference between an optimistic and a tight WCRT is barely 17,000, i.e., $17\mu s$ for task T_1000, whose deadline is $1s$. We can also precisely quantify the "part" of data sharing overheads of a task's tight WCRT by computing the ratio $\frac{tW - optW}{tW}$. The largest such ratio is equal to 3×10^{-4}, obtained for task T_20. This means that the "effect" of data sharing overheads on a task's WCRT in this example is no higher than 0.03 percent.

Now, scalability-wise, the verification time and memory consumption had a

[18] https://www.ecrts.org/forum/viewtopic865d.html?f=32&t=85&sid=d74079af129d5480a5ac4fd1778eecc1

peak of, respectively, *102s* and *3GB*. This means that we manage to compute a WCRT in less than two minutes on average, and the total memory consumption after all computations is less than half of the available RAM of the mid-range computer used for the experiments. These results are quite promising, given that the real-time model of this challenge is remarkably complex and features known threats to verification scalability.

7.3. Artefacts

The experiments described in this section are publicly available and fully reproducible. The source code is available at the public gitlab repository https://gitlab.math.univ-paris-diderot.fr/foughali/fhz_aeic-jsa. The README file contains details on how to reproduce the experiments and/or use the toolchain on the user's own examples. The user has the choice between downloading the sources and using the toolchain on their own machine (in which case they should install the dependencies themselves), or download a ready virtual machine. Also, we provide a WATCHME video, lasting roughly twenty minutes, that demonstrates the experiments and provides a walk-through to reproduce them in a fully automatic manner. Links to both the video (that can be directly played without prior download) and the VM, hosted permanently on our institutional cloud, are provided within the README.

7.4. Discussion

We manage to verify important properties compositionally. For the robotics case study, both the ILP pass and the UPPAAL verification of properties local to one core (WCRT and schedulability) were extremely fast, making the whole process last barely a few seconds per property. For the other property where we had to consider two cores, the verification lasted roughly one minute. The results on WCRT are a typical example of model checking giving tight bounds contrary to schedulability tests. Dually, the ILP pass is crucial as it managed to solve the affinity problem for the terrestrial robot, and provided grounds to achieve such solution for the drone. This interplay between the ILP pass and model checking, a key trait of our interdisciplinary approach, proved to be of great advantage through combining the strengths of these techniques. This advantage is made possible through enabling compositionality of verification: we could not have obtained any such interesting results on a global model (with all cores at the same time), which we have already tried in previous work [33, 34].

For the automotive industrial challenge, our UPPAAL model proved resilient to large clock values, with timing constraints varying from 0 to 1 billion time units. The authors of [73] verified the same case study (also on core 2) but their approach does not support precedence constraints between runnables (i.e., the successor relation that we have in our FSM, Sect. 3.1). Our understanding is that the authors kept the priority levels between tasks intact, then put each runnable in a new task and enforced strict ascending priorities between runnables with strictly ascending identifiers [73, Sect. V.A]; in this case, their model becomes equivalent to ours in the absence of *release jitters*. Further, their approach does

not support data sharing. Therefore, we can provide an indicative, preliminary comparison, only for the optimistic WCRT computations, under the above equivalence assumption and in the absence of release jitters. We notice that our UPPAAL model, albeit evaluated on a mid-range computer that is much less powerful than the one used in [73], is seven times faster than the Scheduling Abstract Graph (SAG) approach [73, Fig. 4(h), Runnable-level, No jitters], and provides exact results contrary to the SAG-POR extension (optimistic WCRTs are exact under the assumption of total absence of overheads). These comparisons put aside, our approach is quite different than SAG-related approaches as we will explain in Sect. 8. Further, the experiments on this automotive case study showed the benefit of our fine-grained overheads computation, leading to narrow intervals of exact WCRTs with data sharing taken into account. In the presence of data sharing, the choice of computing overheads analytically in a tight way proved to be an excellent tradeoff between scalability and exact WCRTs computation.

However, our model is, so far, suitable for periodic tasks only. For the robotic case studies, there are two aperiodic tasks in charge of reading sensors, which we transformed into periodic based on the frequencies of such sensors; and for the industrial challenge we only verified core 2 due to the presence of sporadic tasks elsewhere. Also, our model, so far, does not consider release jitters. While sporadic tasks and release jitters are hard to handle with model checking because of the possibility to "activate a task at any time" within a possibly large interval, some combination with real-time techniques is worth investigating. Finally, for the time being, we have no systematic approach to "complete" the affinity when the ILP pass provides a partial one. In our experiments, this is done manually, in a rather ad-hoc manner. It is worth investigating a more methodical way to do this based on the results of model checking. This could be quite challenging, though, given that the model checking phase itself depends on the affinity.

8. Related Work

Model checking. The literature is rich in model checking real-time systems, e.g., in automotive systems and robotics, but most works either abstract away important hardware-software settings [46, 61] and/or are carried out on small examples [25, 51]. For instance, in [25, 61], hardware is not considered, which restricts the validity of results to the unrealistic assumption that there are enough cores to run all tasks in parallel. When the models are realistic, scalability issues resurface quickly. Indeed, among the many works that use model checking for multicore systems, such issues are reported [8, 55, 22] or preemption is disabled [51, 79, 82].

Schedulability analysis. There exists a solid body of research on schedulability analysis in ERTS under different assumptions of scheduling [10, 15] and data sharing [17, 68]. Many works also integrate cache interference in the derived bounds [2, 28]. More recent works even provide mechanized proofs for the accuracy of WCRTs' upper bounds in schedulability tests [11, 56]. These techniques are mature and efficient, but they remain restricted w.r.t. tasks models and other

properties than schedulability. For instance, as argued in [40] and as we have shown in [37, 34], schedulability tests on simplified models are hard to generalize to real-world ERTS, at least in the robotic context with periodic workloads. Further, scalable schedulability tests (e.g., linear, purely analytical ones) suffer from pessimism as exemplified in Sect. 7.

Hybrid methods. One trend in this category is the use of model checking for schedulability analysis, leading to the development of frameworks such as TIMES [8] and POLA [71]. Unfortunately, such frameworks also rely on simple task models and face scalability concerns with large applications. A more promising trend is to combine model checking and schedulability analysis. An interesting approach based on *timed automata tasks* (TATs) is proposed recently in [76], but focuses on single-core scheduling only. The authors of [39] present a realistic ERTS model with sequences of non-preemptible job segments. They propose a scalable method, to reduce the state space size, that relies on computing an *arrival curve* to abstract memory interferences [70] prior to model checking, a costly procedure after which an overapproximation is required to embed the curve in an UPPAAL model. In contrast, our approach provide tight intervals for exact WCRTs, and, once at the model checking stage, requires no computations external to UPPAAL. Another related work under this category is the digraph model introduced in [75], a powerful formalism presenting a trade-off between expressivity and scalability. However, digraphs are destined to uniprocessor problems. The Scheduling Abstract Graph (SAG) [64] and its extensions [65, 73] is another line of related work. It leverages techniques known in model checking (state-space construction, partial order reduction) to devise algorithms specific to the schedulability property, making SAG and its extensions a state-of-the-art family of techniques suitable for exhaustive schedulability analysis, especially in the presence of jitters and offsets. However, contrary to our approach, the exact version of SAG (i.e., providing exact WCRTs without data sharing, corresponding to our optimistic WCRTs in Sect. 7) is restricted to non-preemptive jobs on uniprocessor platforms [64], which is also the case for the more recent, non-exact POR extension [73]. The multiprocessor version is reserved for global scheduling, and is either non exact with independent tasks (in the limited preemtive setting [65]), or fully non-preemptive with FIFO-spinlock-based data sharing [68]. Besides, our approach is quite different from both digraph and SAG-related approaches. First, our method provides a solution to a wider problem where the affinity is not known beforehand and the ERTS is generic with an arbitrary number of preemptible jobs (each comprising an arbitrary number of non-preemptible segments) and fine-grained data sharing constraints. Second, one of our aims is to be able also to verify other properties besides schedulability, such as bounded response between events occurring on different cores. For this, we strive to keep the generic expressiveness of TA and their model checking tools instead of tailoring constructions specific to schedulability.

Compositional verification. Compositional verification is well-grounded for untimed systems [7, 14]. However, for ERTS, compositional reasoning on a for-

malism (e.g., TA) in a generic sense is a hard research problem, leading to frameworks supporting small fragments of the underlying formalisms. For instance, ECDAR [26] is restricted to deterministic I/O automata, and only a subset of the RT-BIP language [1] (with e.g., no priorities, no broadcast channels and no data variables) is supported by the RTD-Finder tool [12]. As we have shown in [32], such restrictions may render compositional verification even less scalable than classical model checking in real ERTS. While our approach is not generic w.r.t. any TA model, our framework is pragmatic enough to verify important properties in ERTS under state-of-the-art scheduling assumptions.

Our previous work. In [37], we proposed model checking for schedulability and other properties, but it does not scale for the drone application (Sect. 7). More recently, we proposed a hybrid approach that combines SMC and schedulability analysis [34, 38]. However, the schedulability analysis step was overpessimistic and not automated, and the use of SMC is problematic in safety-critical settings (Sect. 1). Moreover, works in [37, 34, 38] were specific to the robotic case.

9. Conclusion

In this paper, we proposed a scalable approach to verify ERTS compositionally. The approach can be used on any ERTS under P-FP scheduling with limited preemption, two state-of-the-art scheduling assumptions. Through bridging the gap between the real-time systems and the formal methods communities, we provide a fully automated framework for compositional verification, and therefore contribute to the fight against state-space explosion. The results, on two real robots that we failed to verify exhaustively for a number of years, as well as on an automotive system from an industrial challenge, are promising.

Although we have very good results on real applications with complex tasks, the extent to which our approach is scalable is still under scrutiny. We aim to confront our framework with larger numbers of tasks, e.g., using synthetic benchmarks. Also, our approach is restricted to periodic tasks and does not support release jitters, so we plan to investigate its extension to sporadic task models and/or jitters. Moreover, we are still sometimes forced to consider more than one core at a time during verification. Abstractions, as to be able to replace cores/tasks with smaller models, are another line of future efforts in which works like [39] can be helpful. Our model also allows to define a tolerance constraint for non-HRT tasks allowing them to miss their deadlines, which makes it more flexible than purely schedulability-driven verification approaches. In the future, it will be interesting to extend this to the (m, k)-firm constraint model [72] where in any sequence of k jobs m deadlines can be missed. Besides, our approach relies on tight analytical computation of data sharing overheads that do not influence schedulability in our case studies. One possible future work is to try to tighten them further through ILP encoding, relying on additional behavioral information on the taskset in the style of [81]. Finally, while our approach treats job segments as black boxes executed between their best-case and worst-case

execution times, verifying the code itself is important. Approaches such as the ones in [31, 30] can be thus complementary to ours.

Acknowledgement

Many thanks to Björn Brandenburg for his valuable comments and judicious remarks that helped us improve this paper.

Funding

MF was supported by the ANR project MAVeriQ (ANR-20-CE25-0012) and the joint ANR-JST project CyPhAI. AZ was supported by the Chair for Cyber-Physical Systems in Production Engineering at TUM and the Alexander von Humboldt Foundation.

References

[1] Abdellatif, T., Combaz, J., Sifakis, J.: Model-based implementation of real-time applications. In: Proc. of the ACM International Conference on Embedded Software. pp. 229–238 (2010). https://doi.org/10.1145/1879021.1879052

[2] Altmeyer, S., Davis, R.I., Indrusiak, L., Maiza, C., Nelis, V., Reineke, J.: A generic and compositional framework for multicore response time analysis. In: Proc. of the International Conference on Real Time and Networks Systems. pp. 129–138 (2015). https://doi.org/10.1145/2834848.2834862

[3] Alur, R.: Timed automata. In: Proc. of the International Conference on Computer Aided Verification. pp. 8–22 (1999). https://doi.org/10.1007/3-540-48683-6_3

[4] Alur, R., Courcoubetis, C., Dill, D.: Model-checking for real-time systems. In: Proc. of the IEEE Symposium on Logic in Computer Science. pp. 414–425 (1990). https://doi.org/10.1109/LICS.1990.113766

[5] Alur, R., Courcoubetis, C., Dill, D.: Model-checking in dense real-time. Information and computation **104**(1), 2–34 (1993). https://doi.org/10.1006/inco.1993.1024

[6] Alur, R., Dill, D.: The theory of timed automata. In: Proc. of the REX Workshop Real-Time: Theory in Practice. pp. 45–73 (1991). https://doi.org/10.1007/BFb0031987

[7] Alur, R., Madhusudan, P., Nam, W.: Symbolic compositional verification by learning assumptions. In: Proc. of the International Conference on Computer Aided Verification. pp. 548–562 (2005). https://doi.org/10.1007/11513988_52

[8] Amnell, T., Fersman, E., Mokrushin, L., Pettersson, P., Yi, W.: TIMES: A tool for schedulability analysis and code generation of real-time systems. In: Proc. of the International Conference on Formal Modeling and Analysis of Timed Systems. pp. 60–72 (2003). https://doi.org/10.1007/978-3-540-40903-8_6

[9] Bartocci, E., Falcone, Y., Francalanza, A., Reger, G.: Introduction to runtime verification. Lectures on Runtime Verification: Introductory and Advanced Topics pp. 1–33 (2018)

[10] Baruah, S.: Techniques for multiprocessor global schedulability analysis. In: Proc. of the IEEE International Real-Time Systems Symposium. pp. 119–128 (2007). https://doi.org/10.1109/RTSS.2007.35

[11] Bedarkar, K., Vardishvili, M., Bozhko, S., Maida, M., Brandenburg, B.B.: From intuition to coq: A case study in verified response-time analysis 1 of FIFO scheduling. In: Proc. of the IEEE International Real-Time Systems Symposium. pp. 197–210 (2022). https://doi.org/10.1109/RTSS55097.2022.00026

[12] Ben-Rayana, S., Bozga, M., Bensalem, S., Combaz, J.: RTD-Finder: A tool for compositional verification of real-time component-based systems. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 394–406 (2016). https://doi.org/10.1007/978-3-662-49674-9_23

[13] Bengtsson, J., Yi, W.: Timed automata: Semantics, algorithms and tools. In: Lectures on Concurrency and Petri Nets. pp. 87–124 (2003). https://doi.org/10.1007/978-3-540-27755-2_3

[14] Bensalem, S., Bozga, M., Sifakis, J., Nguyen, T.H.: Compositional verification for component-based systems and application. In: Proc. of the International Symposium on Automated Technology for Verification and Analysis. pp. 64–79 (2008). https://doi.org/10.1007/978-3-540-88387-6_7

[15] Bertogna, M., Baruah, S.: Limited preemption EDF scheduling of sporadic task systems. IEEE Transactions on Industrial Informatics **6**(4), 579–591 (2010). https://doi.org/10.1109/TII.2010.2049654

[16] Bowman, H.: Time and action lock freedom properties for timed automata. In: Proc. of the International Conference on Formal Techniques for Networked and Distributed Systems. pp. 119–134 (2001). https://doi.org/https://doi.org/10.1007/0-306-47003-9_8

[17] Brandenburg, B.B.: Scheduling and locking in multiprocessor real-time operating systems. Ph.D. thesis, The University of North Carolina at Chapel Hill (2011). https://doi.org/10.17615/x1zq-v169

[18] Brandenburg, B.B.: Multiprocessor real-time locking protocols. In: Handbook of Real-Time Computing, pp. 347–446. Springer (2022)

[19] Brandenburg, B.B., Anderson, J.H.: Spin-based reader-writer synchronization for multiprocessor real-time systems. Real Time Systems **46**(1), 25–87 (2010). https://doi.org/10.1007/s11241-010-9097-2

[20] Buttazzo, G.C.: Hard real-time computing systems: predictable scheduling algorithms and applications, Real-Time Systems Series, vol. 24. Springer (2011). https://doi.org/10.1007/978-1-4614-0676-1

[21] Buttazzo, G.C., Bertogna, M., Yao, G.: Limited preemptive scheduling for real-time systems. a survey. IEEE transactions on Industrial Informatics **9**(1), 3–15 (2012). https://doi.org/10.1109/TII.2012.2188805

[22] Cicirelli, F., Furfaro, A., Nigro, L., Pupo, F.: Development of a schedulability analysis framework based on pTPN and UPPAAL with stopwatches. In: Proc. of the IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications. pp. 57–64 (2012). https://doi.org/10.1109/DS-RT.2012.16

[23] Clarke, E.M.: Model checking. In: Proc. of the International Conference on Foundations of Software Technology and Theoretical Computer Science. pp. 54–56 (1997). https://doi.org/https://doi.org/10.1007/BFb0058022

[24] Clarke, E.M., Klieber, W., Nováček, M., Zuliani, P.: Model checking and the state explosion problem. In: Proc. of the International Summer School Tools for Practical Software Verification. pp. 1–30 (2011). https://doi.org/10.1007/978-3-642-35746-6_1

[25] Cortés, L.A., Eles, P., Peng, Z.: Modeling and formal verification of embedded systems based on a Petri net representation. Journal of Systems Architecture **49**(12-15), 571–598 (2003). https://doi.org/10.1016/S1383-7621(03)00096-1

[26] David, A., Larsen, K., Legay, A., Møller, M.H., Nyman, U., Ravn, A.P., Skou, A., Wąsowski, A., et al.: Compositional verification of real-time systems using ECDAR. International Journal on Software Tools for Technology Transfer **14**(6), 703–720 (2012). https://doi.org/10.1007/s10009-012-0237-y

[27] Davis, R.I., Burns, A.: Response time upper bounds for fixed priority real-time systems. In: Proc. of the Real-Time Systems Symposium (2008). https://doi.org/10.1109/RTSS.2008.18

[28] Davis, R.I., Altmeyer, S., Indrusiak, L.S., Maiza, C., Nelis, V., Reineke, J.: An extensible framework for multicore response time analysis. Real-Time Systems **54**(3), 607–661 (2018). https://doi.org/10.1007/s11241-017-9285-4

[29] Dechev, D., Pirkelbauer, P., Stroustrup, B.: Understanding and effectively preventing the ABA problem in descriptor-based lock-free designs. In: Proc. of the 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing. pp. 185–192 (2010). https://doi.org/10.1109/ISORC.2010.10

[30] Devi, S., Nalini, C., Kumar, N.: An efficient software verification using multi-layered software verification tool. International Journal of Engineering & Technology **7**(2.21), 454–457 (2018). https://doi.org/https://doi.org/10.14419/ijet.v7i2.21.12465

[31] Erbsen, A., Gruetter, S., Choi, J., Wood, C., Chlipala, A.: Integration verification across software and hardware for a simple embedded system. In: Proc. of the ACM SIGPLAN International Conference on Programming Language Design and Implementation. pp. 604–619 (2021). https://doi.org/10.1145/3453483.3454065

[32] Foughali, M.: Toward a correct-and-scalable verification of concurrent robotic systems: insights on formalisms and tools. In: Proc. of the International Conference on Application of Concurrency to System Design. pp. 29–38 (2017). https://doi.org/10.1109/ACSD.2017.10

[33] Foughali, M.: Formal verification of the functionnal layer of robotic and autonomous systems. Ph.D. thesis, INSA de Toulouse (2018)

[34] Foughali, M.: A two-step hybrid approach for verifying real-time robotic systems. In: Proc. of the IEEE International Conference on Embedded and Real-Time Computing Systems and Applications. pp. 1–10 (2020). https://doi.org/10.1109/RTCSA50079.2020.9203687

[35] Foughali, M., Bensalem, S., Combaz, J., Ingrand, F.: Runtime verification of timed properties in autonomous robots. In: 18th ACM-IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE). pp. 1–12. IEEE (2020)

[36] Foughali, M., Dal Zilio, S., Ingrand, F.: On the semantics of the genom3 framework. Tech. rep. (2019)

[37] Foughali, M., Hladik, P.E.: Bridging the gap between formal verification and schedulability analysis: The case of robotics. Journal of Systems Architecture **111**, 101817 (2020). https://doi.org/10.1016/j.sysarc.2020.101817

[38] Foughali, M., Zuepke, A.: Formal verification of real-time autonomous robots: An interdisciplinary approach. Frontiers in Robotics and AI p. 1 (2022). https://doi.org/10.3389/frobt.2022.791757

[39] Giannopoulou, G., Lampka, K., Stoimenov, N., Thiele, L.: Timed model checking with abstractions: Towards worst-case response time analysis in resource-sharing manycore systems. In: Proc. of the ACM international conference on Embedded software. pp. 63–72 (2012). https://doi.org/10.1145/2380356.2380372

[40] Gobillot, N., Lesire, C., Doose, D.: A design and analysis methodology for component-based real-time architectures of autonomous systems. Journal of Intelligent & Robotic Systems **96**(1), 123–138 (2019). https://doi.org/10.1007/s10846-018-0967-5

[41] Hamann, A., Dasari, D., Kramer, S., Pressler, M., Wurst, F., Ziegenbein, D.: Waters industrial challenge 2017. In: International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS) (2017)

[42] Hemminger, S.: fast reader/writer lock for gettimeofday 2.5.30 (8 2002), https://lwn.net/Articles/7388/

[43] Henzinger, T.A., Nicollin, X., Sifakis, J., Yovine, S.: Symbolic model checking for real-time systems. Information and computation **111**(2), 193–244 (1994). https://doi.org/10.1006/inco.1994.1045

[44] Herlihy, M.: Wait-free synchronization. ACM Trans. Program. Lang. Syst. **13**(1), 124–149 (1991). https://doi.org/10.1145/114005.102808

[45] Karp, R.M.: Reducibility among combinatorial problems. In: Proc. of a symposium on the Complexity of Computer Computations. pp. 85–103. Springer (1972). https://doi.org/10.1007/978-1-4684-2001-2_9

[46] Kim, J.H., Larsen, K.G., Nielsen, B., Mikučionis, M., Olsen, P.: Formal analysis and testing of real-time automotive systems using UPPAAL tools. In: Proc. of the International Workshop on Formal Methods for Industrial Critical Systems. pp. 47–61 (2015). https://doi.org/10.1007/978-3-319-19458-5_4

[47] Klotzbücher, M., Soetens, P., Bruyninckx, H.: OROCOS RTT-Lua: an execution environment for building real-time robotic domain specific languages. In: Proc. of the International Workshop on Dynamic languages for RObotic and Sensors. vol. 8 (2010)

[48] Koopman, P.: A case study of toyota unintended acceleration and software safety. Presentation. Sept (2014), https://users.ece.cmu.edu/~koopman/pubs/koopman14_toyota_ua_slides.pdf

[49] Kopetz, H., Reisinger, J.: The non-blocking write protocol NBW: A solution to a real-time synchronisation problem. In: Proc. of the Real-Time Systems Symposium. pp. 131–137 (1993). https://doi.org/10.1109/REAL.1993.393507

[50] Lameter, C.: Effective synchronization on Linux/NUMA systems. In: Proc. of the Gelato Federation Meeting (2005), http://www.lameter.com/gelato2005.pdf

[51] Lampka, K., Perathoner, S., Thiele, L.: Analytic real-time analysis and timed automata: a hybrid method for analyzing embedded real-time systems. In: Proc. of the ACM international conference on Embedded software. pp. 107–116 (2009). https://doi.org/10.1145/1629335.1629351

[52] Lamport, L.: Concurrent reading and writing. Commun. ACM **20**(11), 806–811 (1977). https://doi.org/10.1145/359863.359878

[53] Larsen, K.G., Pettersson, P., Yi, W.: UPPAAL in a nutshell. International journal on software tools for technology transfer **1**(1), 134–152 (1997). https://doi.org/10.1007/s100090050010

[54] Legay, A., Delahaye, B., Bensalem, S.: Statistical model checking: An overview. In: Proc. of the International conference on Runtime Verification. pp. 122–135 (2010). https://doi.org/10.1007/978-3-642-16612-9_11

[55] Lime, D., Roux, O.: Formal verification of real-time systems with preemptive scheduling. Real-Time Systems **41**(2), 118–151 (2009). https://doi.org/10.1007/s11241-008-9059-0

[56] Maida, M., Bozhko, S., Brandenburg, B.B.: Foundational response-time analysis as explainable evidence of timeliness. In: Maggio, M. (ed.) Proc. of the 34th Euromicro Conference on Real-Time Systems. vol. 231, pp. 19:1–19:25 (2022). https://doi.org/10.4230/LIPIcs.ECRTS.2022.19

[57] Mallet, A., Pasteur, C., Herrb, M., Lemaignan, S., Ingrand, F.: Genom3: Building middleware-independent robotic components. In: Proc. of the IEEE International Conference on Robotics and Automation. pp. 4627–4632 (2010). https://doi.org/10.1109/ROBOT.2010.5509539

[58] McKenney, P.E.: Is parallel programming hard, and, if so, what can you do about it? CoRR **abs/1701.00854** (2017), http://arxiv.org/abs/1701.00854

[59] Mellor-Crummey, J.M., Scott, M.L.: Algorithms for scalable synchronization on shared-memory multiprocessors. ACM Transactions on Computer Systems **9**(1), 21–65 (1991). https://doi.org/10.1145/103727.103729

[60] Mellor-Crummey, J.M., Scott, M.L.: Scalable reader-writer synchronization for shared-memory multiprocessors. In: Proc. of the ACM Symposium on Principles & Practice of Parallel Programming (PPOPP). pp. 106–113 (1991). https://doi.org/10.1145/109625.109637

[61] Miyazawa, A., Ribeiro, P., Li, W., Cavalcanti, A., Timmis, J.: Automatic property checking of robotic applications. In: Proc. of the IEEE/RSJ International Conference on Intelligent Robots and Systems. pp. 3869–3876 (2017). https://doi.org/10.1109/IROS.2017.8206238

[62] Monot, A., Navet, N., Bavoux, B., Simonot-Lion, F.: Multicore scheduling in automotive ECUs. In: Proc. of the Embedded Real Time Software and Systems (2010), https://hal.inria.fr/inria-00543179

[63] Monot, A., Navet, N., Bavoux, B., Simonot-Lion, F.: Multisource software on multicore automotive ECUs - combining runnable sequencing with task scheduling. IEEE Transactions on Industrial Electronics **59**(10), 3934–3942 (2012). https://doi.org/10.1109/TIE.2012.2185913

[64] Nasri, M., Brandenburg, B.B.: An exact and sustainable analysis of non-preemptive scheduling. In: 2017 IEEE Real-Time Systems Symposium (RTSS). pp. 12–23 (2017)

[65] Nasri, M., Nelissen, G., Brandenburg, B.B.: Response-time analysis of limited-preemptive parallel dag tasks under global scheduling. In: 31st Conference on Real-Time Systems. pp. 21–1 (2019)

[66] Nelder, J.A., Mead, R.: A simplex method for function minimization. The computer journal **7**(4), 308–313 (1965). https://doi.org/10.1093/comjnl/7.4.308

[67] Nemitz, C.E., Amert, T., Goyal, M., Anderson, J.H.: Concurrency groups: a new way to look at real-time multiprocessor lock nesting. In: Proc. of the International Conference on Real-Time Networks and Systems. pp. 187–197 (2019). https://doi.org/10.1145/3356401.3356404

[68] Nogd, S., Nelissen, G., Nasri, M., Brandenburg, B.B.: Response-time analysis for non-preemptive global scheduling with FIFO spin locks. In: Proc. of the IEEE Real-Time Systems Symposium. pp. 115–127 (2020). https://doi.org/10.1109/RTSS49844.2020.00021

[69] Ocón, J., Dragomir, I., Coles, A., Green, A., Kunze, L., Marc, R., Perez, C., Germa, T., Bissonnette, V., Scalise, G., et al.: Ade: Autonomous decision making in very long traverses (2020)

[70] Pellizzoni, R., Schranzhofer, A., Chen, J.J., Caccamo, M., Thiele, L.: Worst case delay analysis for memory interference in multicore systems. In: Proc. of the Design, Automation & Test in Europe Conference & Exhibition. pp. 741–746 (2010). https://doi.org/10.1109/DATE.2010.5456952

[71] Peres, F., Hladik, P.E., Vernadat, F.: Specification and verification of real-time systems using POLA. International Journal of Critical Computer-Based Systems **2**(3-4), 332–351 (2011). https://doi.org/10.1504/IJCCBS.2011.042332

[72] Ramanathan, P.: Overload management in real-time control applications using (m, k)-firm guarantee. IEEE Transactions on parallel and distributed systems **10**(6), 549–559 (1999). https://doi.org/10.1109/71.774906

[73] Ranjha, S., Nelissen, G., Nasri, M.: Partial-order reduction for schedule-abstraction-based response-time analyses of non-preemptive tasks. In: Proc. of the IEEE 28th Real-Time and Embedded Technology and Applications Symposium. pp. 121–132 (2022). https://doi.org/10.1109/RTAS54340.2022.00018

[74] Simpson, H.: Four-slot fully asynchronous communication mechanism. IEEE Proceedings E (Computers and Digital Techniques) **137**(1), 17–30 (1990). https://doi.org/10.1049/ip-e.1990.0002

[75] Stigge, M., Ekberg, P., Guan, N., Yi, W.: The digraph real-time task model. In: Proc of the 17th IEEE Real-Time and Embedded Technology and Applications Symposium. pp. 71–80 (2011). https://doi.org/10.1109/RTAS.2011.15

[76] Sun, J., Guan, N., Shi, R., Tan, G., Yi, W.: Schedulability analysis for timed automata with tasks. ACM Transactions on Embedded Computing Systems **20**(5s), 1–26 (2021). https://doi.org/10.1145/3477020

[77] Tipaldi, M., Bruenjes, B.: Survey on fault detection, isolation, and recovery strategies in the space domain. Journal of Aerospace Information Systems **12**(2), 235–256 (2015)

[78] Tripakis, S.: Verifying progress in timed systems. In: Proc. of the International Workshop on Aspects of Real-Time Systems and Concurrent and Distributed Software. pp. 299–314 (1999). https://doi.org/10.1007/3-540-48778-6_18

[79] Waszniowski, L., Hanzálek, Z.: Formal verification of multitasking applications based on timed automata model. Real-Time Systems **38**(1), 39–65 (2008). https://doi.org/10.1007/s11241-007-9036-z

[80] Wieder, A., Brandenburg, B.B.: Efficient partitioning of sporadic real-time tasks with shared resources and spin locks. In: Proc. of the IEEE International Symposium on Industrial Embedded Systems. pp. 49–58 (2013). https://doi.org/10.1109/SIES.2013.6601470

[81] Wieder, A., Brandenburg, B.B.: On spin locks in AUTOSAR: Blocking analysis of FIFO, unordered, and priority-ordered spin locks. In: 2013 IEEE 34th Real-Time Systems Symposium. pp. 45–56 (2013). https://doi.org/10.1109/RTSS.2013.13

[82] Yalcinkaya, B., Nasri, M., Brandenburg, B.B.: An exact schedulability test for non-preemptive self-suspending real-time tasks. In: Proc. of the Design, Automation Test in Europe Conference Exhibition. pp. 1228–1233 (2019). https://doi.org/10.23919/DATE.2019.8715111

Appendix A. Implementation Considerations for the ILP Problem

In Section 5, we have not mentioned the chosen optimization function and have only approached the affinity problem as a constraint satisfaction problem (CSP), i.e., finding a solution that satisfies all the constraints without optimizing any criteria. For our implementation, we introduce an optimization criterion that seeks to balance the load on the cores as best as possible:

$$\text{minimize } \max_{c \in 1..|C|} \left\{ \sum_{t \in 1..|T|} \alpha[t,c] \frac{wt[t]}{p[t]} \right\}$$

and with shared data:

$$\text{minimize } \max_{c \in 1..|C|} \left\{ \sum_{t \in 1..|T|} \frac{\delta[t,c]}{p[t]} \right\}$$

Appendix B. Proof of Proposition 2 (Sketch)

To prove that N_c is timelock free, we need to prove that the underlying TS has no timelock states. Let n be the number of tasks TA in N_c. A state of the underlying TS is therefore of the form $S = ((l_s, l_1, \ldots, l_n), v)$ where l_s is the current location of the scheduler O_c, $l_1 \ldots l_n$ the current locations of each task TA_τ and v a vector of clock valuations. The intuition of this very high-level sketch[19] is to show that S is not a timelock state no matter the values of l_s, $l_1 \ldots l_n$. We use the notation $-_n$ to denote any values of $l_1 \ldots l_n$.

If $l_s = wait$, then any $S = ((l_s, -_n), -)$ is timelock free. Indeed, since *wait* is invariant free, either a delay transition is taken from S, or a termination or an activation of some task happens. In any case, a finite number of discrete transitions takes place before reaching a state S' where l_s is equal to *wait* or a state S'' where $l_s = preempt$. This is because the (i) number of tasks is finite and (ii) periods are strictly positive integers (Sect. 3.1), so sequences of zero-time terminations/activations are finite, meaning that committed locations on the paths leading from *wait* to *wait* or *preempt* in the scheduler are eventually left.

From S', a delay transition will inevitably happen. This is because no time elapsed since last activating all tasks that had to be activated (all paths that lead back to *wait* pass through committed locations at which time may not elapse) and therefore no other task will be activated or terminated immediately. Here, delay transitions take place as tasks execute[20], and such execution is guaranteed to be (non-zeno) timelock free because the intervals $[bcs(s), wcs(s)]$ are non empty by definition for any segment s (Sect. 3.1).

From S'' (with $l_s = preempt$), either a delay transition happens (invariant-free location) or a finite succession of discrete transitions takes place (again because the number of tasks is finite) and a new TS state with $l_s = wait$ and

[19] More guarantees on absence of timelocks are explained in the public artefact (link in Sect. 7.3).

[20] If BCETs are set to zero, the possible zero-time sequence corresponding to tasks execution one after another until there is no more task to execute (finite number of activated tasks) or activate immediately (periods are strictly positive integers, Sect. 3.1), leads to a new state with $l_s = wait$ from which delay transitions are inevitable.

inevitable delay transitions is reached (at this new state, delay transitions are inevitable exactly for the same reasons they are inevitable in S' above).

Summarized, from any state S with $l_s = \mathit{wait}$ or $l_s = \mathit{preempt}$, either delay transitions take place or another state $\mathcal{S}$ with $l_s = \mathit{wait}$ or $l_s = \mathit{preempt}$ where a delay transition takes place is reached, and inductively from $\mathcal{S}$ a delay transition will eventually happen. Therefore, time diverges from S through cumulating these delay transitions in an infinite manner, and thus S is not a timelock state.

For the remaining configurations, if l_s in S is equal anything but *wait* or *preempt*, then a new TS state $\mathcal{S}$ with $l_s = \mathit{preempt}$ or $l_s = \mathit{wait}$ is eventually reached. Indeed, for instance, if $l_s = \mathit{decide}$, then a TS state with either $l_s = \mathit{wait}$, $l_s = \mathit{preempt}$ or $l_s = \mathit{release}$ is eventually reached (the disjunction of outgoing edges $\mathit{decide} \rightarrow$ is a tautology), and if $l_s = \mathit{release}$ in the new TS state, then a TS state with $l_s = \mathit{wait}$ is eventually reached (the synchronization over the edge $\mathit{decide} \rightarrow \mathit{wait}$ is guaranteed to happen in zero time). As shown above, time will diverge from $\mathcal{S}$ and therefore there is an infinite path from S such that time diverges (because $\mathcal{S}$ is reachable from S). It follows that S is not a timelock state.

In conclusion, any state of the underlying TS of N_c is timelock free, and therefore N_c is timelock free.