Scalable Computation of Inter-Core Bounds Through Exact Abstractions

Mohammed Foughali, Marius Mikučionis, Maryline Zhang

To cite this version:

Mohammed Foughali, Marius Mikučionis, Maryline Zhang. Scalable Computation of Inter-Core Bounds Through Exact Abstractions. International Conference on Computers, Software, and Applications (COMPSAC), Jul 2024, Osaka, Japan. hal-04571414v1

HAL Id: hal-04571414

https://hal.science/hal-04571414v1

Submitted on 7 May 2024 (v1), last revised 13 May 2024 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Scalable Computation of Inter-Core Bounds Through Exact Abstractions

Mohammed Aristide Foughali[1,3], Marius Mikučionis[2], and Maryline Zhang[1]

[1] Université Paris Cité, CNRS, IRIF, F-75013, Paris, France
[2] Department of Computer Science, Aalborg University, Denmark
[3] Email foughali@irif.fr

Abstract. Real-time systems (RTSs) are at the heart of numerous safety-critical applications. An RTS typically consists of a set of real-time tasks (the software) that execute on a multicore shared-memory platform (the hardware) following a *scheduling policy*. In an RTS, computing *inter-core bounds*, i.e., bounds separating events produced by tasks on different cores, is crucial. While efficient techniques to over-approximate such bounds exist, little has been proposed to compute their exact values. Given an RTS with a set of cores C and a set of tasks T, under partitioned fixed-priority scheduling with limited preemption, a recent work by Foughali, Hladik and Zuepke (FHZ) models tasks with affinity c (i.e., allocated to core $c \in C$) as a Uppaal timed automata (TA) network N_c. For each core c in C, N_c integrates blocking (due to data sharing) using tight analytical formulae. Through compositional model checking, FHZ achieved a substantial gain in scalability for bounds *local to a core*. However, computing inter-core bounds for some events of interest E, produced by a subset of tasks $T_E \subseteq T$ with different affinities $C_E \subseteq C$, requires model checking $N_E = ||_{c \in C_E} N_c$, i.e., the parallel composition of all TA networks N_c for each $c \in C_E$, which produces a large, often intractable, state space. In this paper, we present a new scalable approach based on *exact abstractions* to compute exact inter-core bounds in a *schedulable* RTS, under the assumption that tasks in T_E have distinct affinities, i.e., $|T_E| = |C_E|$. We develop a novel algorithm, leveraging a new query that we implement in Uppaal, that computes for each TA network N_c in N_E an abstraction $\mathcal{A}(N_c)$ preserving the *exact* intervals within which events occur on c. Then, we model check $\mathcal{A}(N_E) = ||_{c \in C_E} \mathcal{A}(N_c)$ (instead of N_E), therefore drastically reducing the state space. The scalability of our approach is demonstrated on the WATERS 2017 industrial challenge, for which we efficiently compute various types of inter-core bounds where FHZ fails to scale.

Keywords: Timed automata · Model checking · Real-time systems

1 Introduction

Real-time systems (RTSs) underly many safety-critical applications, spanning areas such as robotics and automotive industry. An RTS typically boils down

to a set of complex real-time *tasks* (the software) that execute on a *multicore shared-memory* platform (the hardware). Tasks, that must obey stringent timing constraints (*deadlines*) under limited resources, i.e., a small number of cores, and concurrency protocols, execute following a scheduling policy. Verifying that an RTS meets the *schedulability* requirement is crucial. An RTS is schedulable, under a given scheduling policy, if each of its tasks is schedulable, i.e., it always finishes its execution before its deadline [5]. Response-time analysis (RTA) [4,7,18,21] is a popular approach to verify schedulability, through computing the worst-case response time (WCRT) of each task.

Schedulability is, however, not all what an RTS is about. In a schedulable RTS, one still needs to compute bounds separating some events of interest. To put this in the most generic way, let T_E be the set of tasks producing some events in the set E. The goal is to compute, in a scalable way, exact bounds separating the production of events in E following some requirement. For example, $E = \{e_1, e_2, e_3, e_4\}$ and we want to quantify precisely the minimum and maximum amount of time, in all possible scenarios, between each production of e_1 and each production of e_4, with e_2 and e_3 occurring concurrently in between. Solving this problem is hard, in particular if tasks in T_E execute on different cores (i.e., computing *inter-core* bounds). Indeed, RTA-based techniques, specific to schedulability, are not suitable for inter-core bounds. In contrast, model checking is a natural candidate but suffers from state-space explosion. To illustrate this, consider a recent work by Foughali, Hladik and Zuepke [9] (FHZ hereafter). Given an RTS with *periodic* tasks and *partitioned fixed-priority* (P-FP) scheduling with *limited preemption* (Sect. 2), the behavior of tasks with affinity c (i.e., allocated to core c), is modeled as N_c, a UPPAAL timed automata (TA) network [16]. Since N_c analytically integrates *blocking* due to data sharing, WCRTs of tasks with affinity c are computed efficiently on the state space of N_c only (with a computation time more than seven times faster than the Schedule Abstract Graph [21] for the WATERS 2017 case study [13]). However, for an inter-core bound, FHZ suggest to compute it on the parallel composition $N_E = ||_{c \in C_E} N_c$, where C_E is the smallest subset of cores covering the affinities of tasks in T_E. Though this remains compositional, building the state space of N_E does not scale in practice (Sect. 4). Many works focus on *end-to-end latencies* [8,12,20], a special case of the problem above, for which efficient yet non-exact procedures are proposed (more in Sect. 5).

Contributions. We present in this paper a scalable approach to compute exact inter-core bounds in a schedulable RTS following the FHZ model, under the assumption that tasks in T_E have distinct affinities (i.e., $|T_E| = |C_E|$). We devise a novel algorithm, using a new query that we implement in UPPAAL, that computes from every core network N_c, $c \in C_E$, an *exact abstraction* $\mathcal{A}(N_c)$ that preserves the exact intervals within which events in E occur on c. Accordingly, we compute exact inter-core bounds on the state space of $\mathcal{A}(N_E) = ||_{c \in C_E} \mathcal{A}(N_c)$, instead of $N_E = ||_{c \in C_E} N_c$. $\mathcal{A}(N_E)$ underlies a drastically smaller state space compared to N_E, and therefore scalability is significantly improved. Our approach is successfully evaluated on the WATERS 2017 industrial challenge [13].

Outline. In Sect. 2, we introduce models and tools used in this paper, namely TA, Uppaal and the FHZ model. Sect. 3 presents the core of our contribution, i.e., a novel algorithm to compute exact abstractions from the TA networks of FHZ. We demonstrate the scalability of our approach in Sect. 4. Finally, we compare with related work (Sect. 5) and wrap up with concluding remarks (Sect. 6).

2 Preliminaries

2.1 Uppaal

Uppaal [16] is a state-of-the-art model checker. It features a modeling language, based on an extension of timed automata (TA) [1], and a query (property) language, based on a subset of the Computation Tree Logic (CTL) [6].

2.1.1 Modeling Language

TA syntax. a timed automaton A is a tuple $\langle \Sigma, L, \ell_0, X, E, I\rangle$, where:

actions Σ is a finite set of actions (including the *silent* action ϵ),
locations L is a finite set of locations,
initial location $\ell_0 \in L$ is the initial location,
clocks X a finite set of real-valued clocks,
edges E a set of edges $(\ell_i, g, a, u, \ell_j)$ between locations $\ell_i, \ell_j \in L$, with guard $g \in G$, action label $a \in \Sigma$ and $u \subseteq X$ reset expression over clocks,
invariants I a location invariant $\forall \ell \in L$: $I(\ell) \in G$.

Constraints G are conjunctions of the atomic form $\{x_i - x_j \prec c_{i,j} \mid 0 \leq i, j \leq |X|\}$ over clock variables $x_i, x_j \in X$, with x_0 a special variable that is always equal to zero, $c_{i,j} \in \mathbb{Z}$ an integer bound and $\prec \in \{<, \leq\}$. Constraints of the form $x_i - x_0 \leq c_{i,0}$ (resp. $x_0 - x_i \leq c_{0,i}$) are written simply as $x_i \leq c_{i,0}$ (resp. $x_i \geq -c_{0,i}$) for some $i \neq 0$. Similarly, a tautology $\top$ can be expressed as $x_0 - x_i \leq 0$ for some $i \in 0 \mathinner{..} |X|$. A location ℓ with an invariant equivalent to $\top$ is referred to as *invariant free*. For simplicity, guards and invariants equivalent to $\top$ are not explicitly represented in this paper.

TA semantics. $A = \langle \Sigma, L, \ell_0, X, E, I\rangle$ is defined by a Timed Labeled Transition System $TLTS = \langle \Sigma, S, s_0, T\rangle$:

state $\langle \ell, \overline{v}\rangle \in S$ consists of a location $\ell \in L$ and a valuation vector $\overline{v}$ assigning a non-negative real $\overline{v}[x] \in \mathbb{R}_{\geq 0}$ to each clock variable $x \in X$,
initial state $s_0 = \langle \ell_0, \overline{0}\rangle \in S$ with $\overline{0}[x] = 0$ for every clock variable $x \in X$,
delay transition $\langle \ell, \overline{v_i}\rangle \xrightarrow{\delta} \langle \ell, \overline{v_j}\rangle \in T$ for a delay $\delta \in \mathbb{R}_{\geq 0}$, where $\overline{v_j}$ is a clock valuation obtained by incrementing each clock $x \in X$ value by δ: $\overline{v_j}[x] = \overline{v_i}[x] + \delta$ and the location invariant is satisfied $\overline{v_j} \models I(\ell)$,

action transition $\langle \ell_i, \overline{v_i} \rangle \xrightarrow{a} \langle \ell_j, \overline{v_j} \rangle \in T$ for an edge $(\ell_i, g, a, u, \ell_j) \in E$, clock valuations $\overline{v_i}$ satisfy g, and target valuation $\overline{v_j}$, obtained by applying edge reset $u[\overline{v_i}]$, satisfies $I(\ell_j)$.

A Uppaal TA is extended with discrete variables and C-like functions to ease modeling complex systems. Tests (resp. updates) on discrete variables and functions can be used in conjunction with guards (resp. performed together with clock resets). Uppaal TA can be composed in parallel into a *network* $||_{i \in 1..n} A_i$, $n \in \mathbb{N}_{>1}$. Uppaal uses *channels* to allow different A_i to synchronize on actions. An exclamation mark (!) denotes the *emitter* and question marks (?) *receivers*. In a *handshake* (resp. *broadcast*) channel, an emitter synchronizes with one receiver at a time in a blocking manner (resp. with as many receivers as available; if none, it proceeds alone). We refer in this paper to broadcast channels without receivers as *singleton channels*. Channels *priorities* may be used to prioritize one concurrent transition over another. *Committed locations* are useful to model atomic action sequences. A state (in the TLTS) comprising at least one committed location must be left immediately (no delay transition possible) and the transition taken to leave it must include an edge emanating from a committed location.

The underlying TLTS of a TA network being typically infinite, Uppaal uses symbolic semantics where each state is represented using a vector of locations (and discrete-variable values), and a symbolic *zone*, defined by a conjunction of constraints of the form G. Uppaal uses Difference Bound Matrices (DBMs) to maintain the constraint system in a compact canonical form by storing $c_{i,j}$ values in a matrix. The matrix values are extended with a special value ∞: $c_{i,j} \in \mathbb{Z} \cup \{\infty\}$ meaning that a particular constraint over $x_i - x_j \prec c_{i,j}$ is absent. In practice, removing upper constraints $\{x_i - x_0 \leq c_{i,0}\}$ is achieved by setting the corresponding entries in the matrix to the special value $c_{i,0} := \infty$ and resetting to zero is achieved by assignments $c_{i,0} := 0$ and $c_{0,i} := 0$, and the constraints are propagated by computing a canonical form. Details on TA symbolic reachability algorithms are published in [3,17].

2.1.2 Query Language

A state formula is a propositional formula over locations and discrete and clock variables, to be evaluated on a symbolic state. For example, $A.l$ *and* $A.x \geq 2$ *and* $A.x \leq 3$ is a state formula that evaluates to true in all symbolic states where A is at location l and the value of clock x in A is comprised between 2 and 3. In this paper, we use a compact notation when possible, e.g., we write the above formula as $A.l$ *and* $2 \leq A.x \leq 3$. Uppaal supports a subset of CTL to query properties, written as state formulae quantified over using *path formulae*, on TA networks. In particular, Uppaal supports *sup* and *inf* queries for a specified set of clocks $\{x_i\}$ and a state formula F, which explore the entire state space using symbolic constraint solving, record the constraint bounds $c_{i,0}$ (resp. $-c_{0,i}$), and report the maximum (resp. minimum) value of clocks $\{x_i\}$ when F holds. For example $sup\{A_1.l$ *and* $A_1.x \leq 2\} : A_2.x$ reports the maximum value of clock x of A_2 observed when A_1 is in location l and the value of its clock x is less than 2. Concrete examples are given in Sect. 3.

2.2 RTS Model

We present a simpler version of FHZ's model for a schedulable RTS, i.e., an affinity that guarantees the schedulability of all tasks is known beforehand (obtained e.g., using the approach in FHZ [9, Section 5]).

2.2.1 Syntax

An RTS R is a tuple $\langle T, C \rangle$ with T a set of *periodic* tasks and C a set of cores.

Each $\tau \in T$ is a tuple $\langle FSM_\tau, P_\tau, \pi_\tau, aff_\tau \rangle$ where $P_\tau \in \mathbb{N}_{>0}$ is τ's *period* (also its deadline), $\pi_\tau \in \mathbb{N}$ is τ's *fixed priority*, and $aff_\tau \in C$ is τ's affinity, i.e., the only core it can execute on (*partitioned allocation*). Each core $c \in C$ is associated with a *partition* $prt_c = \{\tau \in T | aff_\tau = c\}$, the affinity's dual. FSM_τ is a finite-state machine $\langle S_\tau, act, tr_\tau \rangle$ representing τ's behavior where:

- $S_\tau = JS_\tau \cup \{act, end\}$ is the set of states, with act and end the special *activation* and *termination* states and JS_τ the set of *segments*,
- act is the initial state,
- $tr_\tau \subset S_\tau \times S_\tau$ is the transition relation satisfying (i) act (resp. end) has no predecessors (resp. successors) and (ii) each *maximal path* from act to end is finite.

We obtain accordingly J_τ, the set of *jobs* in τ, mapping each maximal path to the ordered set of segments appearing in it.

At a lower level, each segment s in the set $\mathcal{JS} = \bigcup_{\tau \in T} JS_\tau$, is associated with $bt_s \in \mathbb{N}_{>0}$ (resp. $wt_s \in \mathbb{N}_{>0}$), its best-case execution time (BCET) (resp. worst-case execution time (WCET)).

Two important remarks are worth emphasizing. First, the overheads due to data sharing are taken into account, e.g., following the approach in FHZ [9, Sect. 5]. That is, for any segment s, wt_s includes *blocking* due to data sharing, if any. Second, the model above is more expressive than many standard periodic models in the literature. For instance, allowing multiple paths within a task enables handling control flows, therefore widening the model's suitability to e.g., robotic applications [9, Sect. 3].

Fig. 1 (top) illustrates a schedulable RTS with four tasks $\tau_1 = \langle FSM_{\tau_1}, 20, 1, c_1 \rangle$ and $\tau_2 = \langle FSM_{\tau_2}, 30, 0, c_1 \rangle$ (allocated to core c_1), and $\tau_3 = \langle FSM_{\tau_3}, 20, 1, c_2 \rangle$ and $\tau_4 = \langle FSM_{\tau_4}, 40, 0, c_2 \rangle$ (allocated to c_2); the timing constraints of a segment s are indicated on the corresponding state using a couple (bt_s, wt_s). For instance, τ_2 contains three segments $JS_{\tau_2} = \{s2, s3, s4\}$, with $bt_{s2} = 1$, $bt_{s3} = 3$, $bt_{s4} = 2$, $wt_{s2} = 3$, $wt_{s3} = 6$, $wt_{s4} = 5$, and three jobs $J_{\tau_2} = \{\{s_2, s_3\}, \{s_4, s_3\}, \{s_4\}\}$. This RTS example will be used throughout the paper.

The semantics of an RTS, thoroughly discussed in FHZ, relies on P-FP scheduling with limited preemption and is exemplified using Uppaal models next.

2.2.2 Uppaal Model

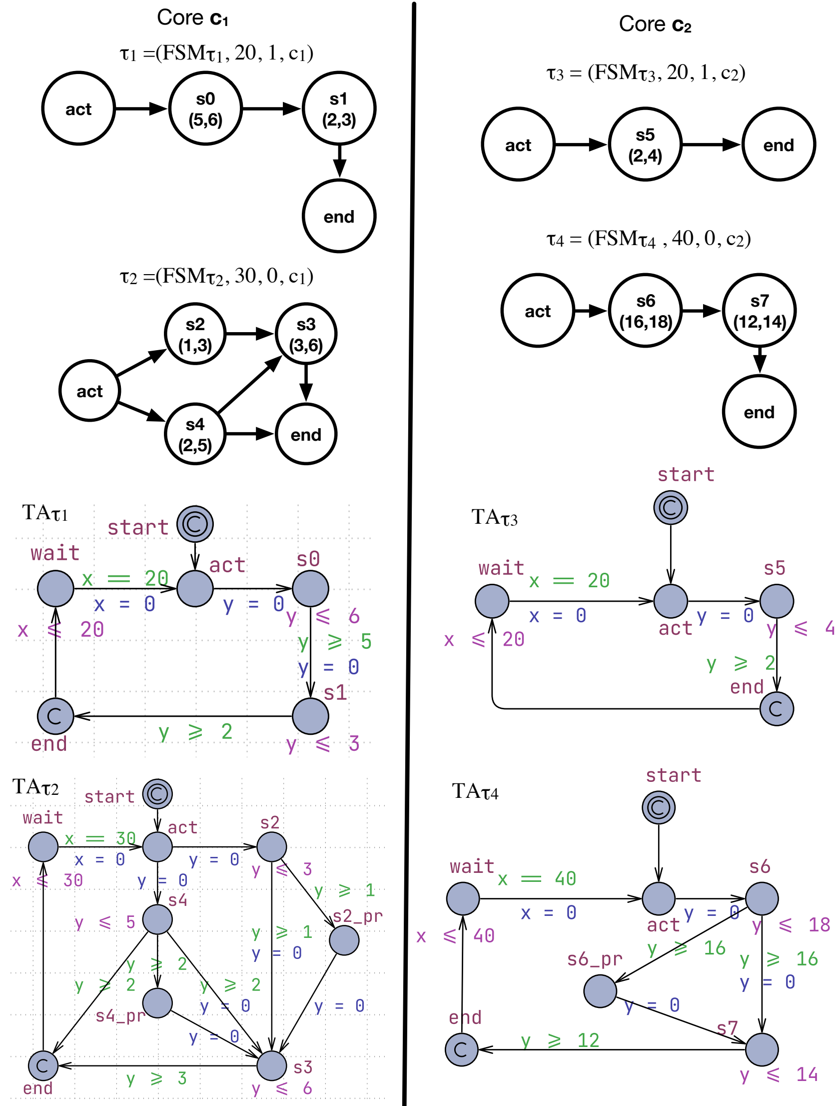

Fig. 1: An example of four tasks and their UPPAAL models.

Tasks. FHZ provide definitions to generate a UPPAAL TA TA_τ from any task τ with the above syntax [9, Sect. 6]. We illustrate what such definitions give for the RTS example in Fig 1 (top). Fig 1 (bottom) depicts accordingly TA_τ of tasks $\tau \in \{\tau_1, \tau_2, \tau_3, \tau_4\}$. We explain using TA_{τ_2}. We use the notation $s \to s'$ to denote an edge from s to s', and $\to s$ (resp. $s \to$) to denote all incoming (resp. outgoing) edges of s.

τ_2 is *activated* at each multiple of its period (multiples of 30). This is ensured, in TA_{τ_2}, through the invariant at *wait*, and the guard and reset of clock x on

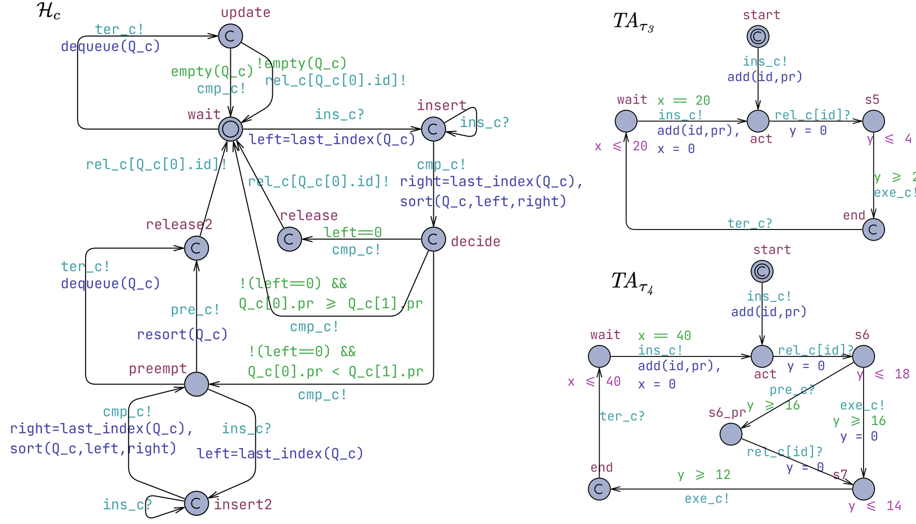

Fig. 2: Network N_{c_2} corresponding to core c_2 from Fig. 1. To simplify notations, we rename c_2 as c.

wait $\to$ *act*. The activation at 0 is handled through the initial, committed location *start*. Edges *act* $\to$ correspond to *releasing* τ_2, i.e., allowing it to execute. Since time until such release is unknown, *act* is invariant free. Executing a job in τ_2 corresponds to traversing a path in TA_{τ_2} from *act* to *end*. Executing a segment s between bt_s and wt_s is ensured by the invariant at location s, the guards on edges $s \to$, and the resets of clock y on edges $\to s$. Preemption may happen at the end of a segment, by taking an edge $s \to s_{pr}$. Since the time until τ_2 is next released (after being preempted) is unknown, locations s_{pr} are invariant free. Notice the absence of preemption location $s3_{pr}$ (since τ_2 can only terminate after executing *s3*, its preemption at the end of *s3* coincides with its termination). On the other tasks TA, notice how e.g., TA_{τ_1} does not contain preemption locations (since τ_1 has the highest priority among tasks allocated to c_1). Also, since tasks are schedulable, it is impossible to reach location *end* with the value of clock x larger than P_τ in some TA_τ. That is, locations used to handle deadline violation in FHZ are eliminated after verifying that the RTS is schedulable.

RTS. TA_τ, for each τ allocated to some core c, are composed with H_c, the (P-FP with limited preemption) scheduler of c, to obtain $N_c = H_c||_{\tau \in prt_c} TA_\tau$, the network of core c. The RTS network is therefore $N = ||_{c \in C} N_c$. In N_c, the behavior of tasks is influenced by the scheduler H_c as their respective TA synchronize through channels and variables (see below), whereas in N, each N_c is independent (since blocking, if any, is integrated in segments WCETs).

We explain the behavior of $N_{c_2} = H_{c_2}||TA_{\tau_3}||TA_{\tau_4}$ (Fig. 2), the TA network corresponding to core c_2 and tasks τ_3 and τ_4 in Fig. 1. To simplify notations, we rename c_2 as c. We analyze the behaviors of N_c between times 0 and 40. At time 0, H_c synchronizes, on handshake channel ins_c, with TA_{τ_3} and TA_{τ_4} atomically one after another. This is ensured through committed location *insert* and the

priority $ins_c > cmp_c$ (cmp_c is a singleton channel). At each synchronization on ins_c, a task inserts "itself", i.e., its unique identifier id and its priority pr in Q_c, a queue of records of two integers, using the function $add(id, pr)$. From location *insert*, H_c sorts all tasks in Q_c according to their priorities (to their order of arrival if priorities are equal) and reaches committed location *decide*. This sorting does not involve the task at $Q_c[0]$ (the head of Q_c) if it is already executing. Here, Q_c was empty before the insertion started, so TA_{τ_3} is now at $Q_c[0]$. H_c immediately releases TA_{τ_3}, using its id, through the handshake channel $rel_c[id]$ (edge *decide* → *release* then *release* → *wait*). TA_{τ_3} then executes *s5* and reaches committed location *end* at a time between 2 and 4. Then, it atomically synchronizes with H_c (handshake channel ter_c) as the latter removes it from Q_c (function *dequeue*, edge *wait* → *update*), then immediately releases TA_{τ_4} which starts executing *s6* accordingly. Such execution ends between times $2 + 16 = 18$ and $4 + 18 = 22$.

If *s6* is left between 18 and 20, TA_{τ_4} moves to *s7* immediately, the execution of which ends between times $18 + 12 = 30$ and $20 + 14 = 34$. Meanwhile, at time 20, TA_{τ_3} is activated again, H_c moves to location *insert* and TA_{τ_3} to location *act*. Now, H_c sorts only tasks behind TA_{τ_4}, already executing, in Q_c. Then, from location *decide*, since $\pi_{\tau_4} < \pi_{\tau_3}$, TA_{τ_4} must be preempted; H_c moves immediately to location *preempt*. Since the only successor of *s7* is *end*, the preemption of TA_{τ_4} coincides with its termination. Therefore, between times 30 and 34, TA_{τ_4} synchronizes on ter_c with H_c as the latter removes it from Q_c and moves to committed location *release2*. Immediately, H_c releases TA_{τ_3}, now at $Q_c[0]$, and transits back to *wait*. TA_{τ_3} then executes *s5* between times $30 + 2 = 32$ and $34 + 4 = 38$ and terminates.

If, instead, *s6* is left between 20 and 22, TA_{τ_4} is forced to preempt while at location *s6* (edge $s6 \rightarrow s6_{pr}$) as H_c transits to *release2*, synchronizing on channel pre_c (ensured by the priority $pre_c > exe_c$, exe_c is a singleton channel). Together with this transition, H_c re-sorts Q_c by putting TA_{τ_4} behind TA_{τ_3} (function *resort*) and releases TA_{τ_3}, which then executes *s5* between times $20 + 2 = 22$ and $22 + 4 = 26$ and terminates. H_c then removes TA_{τ_3} from Q_c (transitioning to *update*) then immediately releases TA_{τ_4} (transitioning back to *wait*) which will accordingly resume its execution at *s7* (moving from $s6_{pr}$ to *s7*). TA_{τ_4} terminates then between $22 + 12 = 34$ and $26 + 14 = 40$.

We conclude with three remarks. First, FHZ explores all possible orderings of simultaneous events. For instance, if TA_{τ_4} finishes executing *s6* at the same time TA_{τ_3} is activated (i.e., 20), the subsequent behavior depends on which of these two events happened *first* at this particular instant. This explains why this border case belongs to both scenarios above. Second, the description above is example-oriented and high level, whereas H_c follows in fact a universal model for P-FP scheduling with limited preemption (we refer the interested reader to [9, Sect. 6] for low-level details). Finally, the set of behaviors in our example between times 40 and 80, 80 and 120, etc. is identical to the one depicted between times 0 and 40. This is because, in a schedulable network N_c, the behaviors

repeat at each *hyperperiod* hp_c, given by the least common multiple of periods of tasks allocated to c, i.e., $hp_c = lcm(\{P_\tau | \tau \in prt_c\})$.

3 Approach

In this section, we present the core of our contribution. Note that the presentation throughout this section, including the used examples and the proposed algorithms, favors simplicity over technical details. The latter are discussed in Sect. 4.1.

3.1 Problem Statement & Motivation

Let $R = \langle T, C \rangle$ be a schedulable RTS, $N = ||_{c \in C} N_c$ its UPPAAL model, and E a non-empty set of events of interest. Let $T_E \subseteq T$ be the smallest subset of tasks producing events in E. A task τ in T_E produces events through segments in the set JS^E_τ. For simplicity, we assume that such segments belong to different jobs in J_τ, i.e., $\forall s, s' \in JS^E_\tau, J \in J_\tau : s \in J \wedge s' \in J \Rightarrow s = s'$. The set of all event-producing segments in R is therefore $\mathcal{JS}_E = \cup_{\tau \in T_E} JS^E_\tau$. Let $C_E \subseteq C$ be the smallest subset of cores covering the affinities of tasks in T_E ($C_E = \{c \in C | T_E \cap prt_c \neq \varnothing\}$). Events occur on at least two cores, i.e., $|C_E| \geq 2$. We require that tasks in T_E have *distinct affinities*, i.e., $|T_E| = |C_E|$. The reason behind this restriction is detailed in Sect. 3.5.3. If $s \in JS^E_\tau$ produces event e, we say that τ produces e, and that e occurs on $c = aff_\tau$. There is no restriction on the number of cores an event can occur on.

A segment s in $\mathcal{JS}_E$ may produce a sequence of an arbitrary number of events. This is handy in practice, e.g., when a segment reads and writes some data (Sect. 4). Let $\mathbb{I}_s$ be the set of non-empty closed intervals of reals with natural bounds within the relative execution time of segment $s \in \mathcal{JS}_E$. That is, $[a, b] \in \mathbb{I}_s,\ a, b \in \mathbb{N}, a \leq b \leq wt_s$, equals the set of reals $\{x \in \mathbb{R} | a \leq x \leq b\}$. We associate each segment s in $\mathcal{JS}_E$ with a set of couples $s_E \subset E \times \mathbb{I}_s$. Each ith element of s_E represents an event e that s produces as well as the interval within which s may produce e, relative to the execution of s. For example, if s produces one event e at the end of its execution then $s_E = \{(e, [bt_s, wt_s])\}$. The set s_E is *ordered*. An event belonging to a couple at position i of s_E occurs *always* before an event appearing in a couple at position $j > i$. For intervals, we impose an order on their bounds: any two elements $[a, b]$ and $[a', b']$ appearing in couples at positions i and j, $j > i$, respectively, obey the inequalities $a \leq a'$ and $b \leq b'$. That is, intervals may overlap, but they must respect the strict ordering of events occurrences. For instance, if $s_E = \{(e, [a, b]), (e', [a', b'])\}$, and $[a, b] \cap [a', b'] = [c, d] \neq \varnothing$, then e still occurs before e' in the interval $[c, d]$ (i.e., the possible occurrence of e' in $[c, d]$ is conditioned by a prior occurrence of e). In brief, we remain the most generic possible as we impose no constraints on *when* a segment produces an event within its execution, since elements of $\mathbb{I}_s$ can be any interval included in $[0, wt_s]$.

The goal is to compute *exact* inter-core bounds following some requirement. For example, given $E = \{e_1, e_2, e_3\}$, what is the exact maximum amount of

time separating each production of e_1 and the next production of e_2, with e_3 happening in between? One motivation behind exactness is the fact that existing RTS (multicore) models, on which analyses are carried out, typically include over-approximations due to blocking. FHZ, for instance, integrate blocking in segments WCETs using an analytical approach. Though the latter is particularly tight, the resulting WCETs may be slightly over-approximated [9, Sect. 5]. Consequently, it is important to be exact w.r.t. to a model of R (in our case N) that inevitably over-approximates the effects of blocking in one way or another, therefore avoiding further pessimism. The *direct method* to achieve this is to compose $N_E = ||_{c \in C_E} N_c$ with an *observer* Obs, as suggested in FHZ. However, given the complexity of each N_c, building the state-space of $N_E||Obs$ does not scale for large systems. For example, if we try to compute an inter-core bound on the 2017 WATERS industrial challenge (for which FHZ successfully computed WCRTs on one core), the direct method leads to memory exhaustion (Sect. 4). We need therefore to use *abstractions* to reduce the state-space size.

3.2 Core Idea

Our main idea is the following. As long as building the state space of each N_c apart is scalable, we use this to our advantage as we compute a (much smaller) abstraction $\mathcal{A}(N_c)$ that preserves the *absolute intervals* within which events occur on c, i.e., produced by the only task, say τ, in $T_E \cap prt_c$. This abstraction is based on the original N_c behavior, and therefore takes into account all the delays event-producing segments in τ may incur due to other tasks allocated to c, i.e., in $prt_c \backslash \{\tau\}$ (and blocking due to tasks allocated to other cores, already integrated in N_c). Then, instead of computing the bounds on $N_E||Obs$, we do so on $\mathcal{A}(N_E)||Obs$, where $\mathcal{A}(N_E) = ||_{c \in C_E} \mathcal{A}(N_c)$, therefore drastically reducing the state space. We first present a coarse abstraction and explain its weakness (Sect. 3.4). Then, we devise an exact abstraction, relying on a new query that we implement in Uppaal (Sect. 3.5).

3.3 Examples

For illustration, we make use of simple[4] examples, all relying on the schedulable RTS in Sect. 2.2 (Figures. 1, 2). The hyperperiods are $hp_{c_1} = 60$ and $hp_{c_2} = 40$.

3.3.1 Example 1

$E = \{e_1, e_2\}$ with e_1 produced by segment $s5$ and e_2 by $s1$. Both events are produced at the end of execution of such segments, that is at any instant comprised between their best- and worst-case execution times. Therefore, $C_E = \{c_1, c_2\}$, $T_E = \{\tau_1, \tau_3\}$, $JS^E_{\tau_1} = \{s1\}$, $JS^E_{\tau_3} = \{s5\}$, $s5_E = \{(e_1, [2, 4])\}$ and $s1_E = \{(e_2, [2, 3])\}$.

[4] We demonstrate our approach's scalability on the real WATERS 2017 industrial challenge in Sect. 4.

3.3.2 Example 2

Example 1 with a third event e_3 produced by $s5$ in the relative interval $[0, 1]$. $E = \{e_1, e_2, e_3\}$, $C_E = \{c_1, c_2\}$, $T_E = \{\tau_1, \tau_3\}$, $JS^E_{\tau_1} = \{s1\}$, $JS^E_{\tau_3} = \{s5\}$, $s5_E = \{(e_3, [0, 1]), (e_1, [2, 4])\}$, and $s1_E = \{(e_2, [2, 3])\}$.

3.3.3 Example 3

Example 2 with τ_1 no longer producing events. Instead, τ_2 produces event e_4 (resp. e_2) through $s2$ in the relative interval $[0, 3]$ (resp. $s4$ in the relative interval $[2, 4]$). $E = \{e_1, e_2, e_3, e_4\}$, $C_E = \{c_1, c_2\}$, $T_E = \{\tau_3, \tau_2\}$, $JS^E_{\tau_2} = \{s2, s4\}$, $JS^E_{\tau_3} = \{s5\}$, $s5_E = \{(e_3, [0, 1]), (e_1, [2, 4])\}$, $s2_E = \{(e_4, [0, 3])\}$ and $s4_E = \{(e_2, [2, 4])\}$.

3.4 Coarse Abstractions

Consider Example 1 (Sect. 3.3.1). We want to compute $B^{max}_{e_1,e_2}$, the *maximal* inter-core bound between each production of e_1 and the next production of e_2.

For N_{c_1}, we build a coarse abstraction $\mathcal{CA}(N_{c_1})$ as follows:

- Add a clock x to H_{c_1},
- Run, on N_{c_1} separately, Uppaal with the queries:
  $min_{s1,e_2,1} = inf\{\phi_{s1,e_2}$ *and* $H_{c_1}.x \leq 20\} : H_{c_1}.x$
  $max_{s1,e_2,1} = sup\{\phi_{s1,e_2}$ *and* $H_{c_1}.x \leq 20\} : H_{c_1}.x$
  $min_{s1,e_2,2} = inf\{\phi_{s1,e_2}$ *and* $20 \leq H_{c_1}.x \leq 40\} : H_{c_1}.x$
  $max_{s1,e_2,2} = sup\{\phi_{s1,e_2}$ *and* $20 \leq H_{c_1}.x \leq 40\} : H_{c_1}.x$
  $min_{s1,e_2,3} = inf\{\phi_{s1,e_2}$ *and* $40 \leq H_{c_1}.x \leq 60\} : H_{c_1}.x$
  $max_{s1,e_2,3} = sup\{\phi_{s1,e_2}$ *and* $40 \leq H_{c_1}.x \leq 60\} : H_{c_1}.x$
  with $\phi_{s1,e_2} = TA_{\tau_1}.s1$ *and* $2 \leq TA_{\tau_1}.y \leq 3$,
- Obtain $min_{s1,e_2,1} = 7$, $max_{s1,e_2,1} = 9$, $min_{s1,e_2,2} = 27$, $max_{s1,e_2,2} = 29$, $min_{s1,e_2,3} = 47$ and $max_{s1,e_2,3} = 50$,
- Build $\mathcal{CA}(N_{c_1})$ accordingly (Fig. 3, bottom left).

And for N_{c_2}:

- Add a clock x to H_{c_2},
- Run, on N_{c_2} separately, Uppaal with the queries:
  $min_{s5,e_1,1} = inf\{\phi_{s5,e_1}$ *and* $H_{c_2}.x \leq 20\} : H_{c_2}.x$
  $max_{s5,e_1,1} = sup\{\phi_{s5,e_1}$ *and* $H_{c_2}.x \leq 20\} : H_{c_2}.x$
  $min_{s5,e_1,2} = inf\{\phi_{s5,e_1}$ *and* $20 \leq H_{c_2}.x \leq 40\} : H_{c_2}.x$
  $max_{s5,e_1,2} = sup\{\phi_{s5,e_1}$ *and* $20 \leq H_{c_2}.x \leq 40\} : H_{c_2}.x$
  with $\phi_{s5,e_1} = TA_{\tau_3}.s5$ *and* $2 \leq TA_{\tau_3}.y \leq 4$,
- Obtain $min_{s5,e_1,1} = 2$, $max_{s5,e_1,1} = 4$, $min_{s5,e_1,2} = 22$, and $max_{s5,e_1,2} = 38$,
- Build $\mathcal{CA}(N_{c_2})$ accordingly (Fig. 3, top left).

Let us explain how this works, using the example of core c_2. Adding clock x to the scheduler allows to have an *absolute reference* of time (this clock is never reset, and all clocks start synchronously at 0 in a TA network, Sect. 2.1). Relying on this reference, we compute, in each period of τ_3, the task producing e_1 through segment $s5$, and up to the hyperperiod hp_{c_2}, the exact *earliest* and

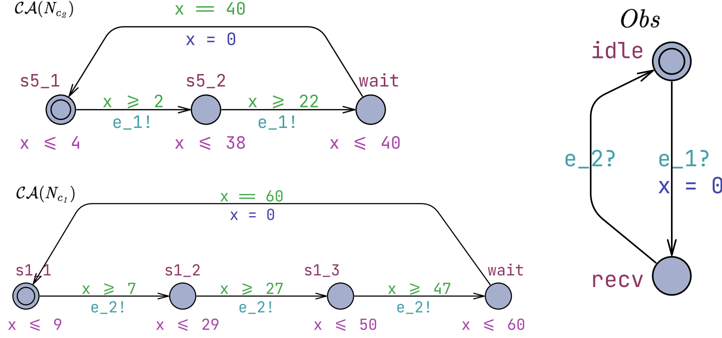

Fig. 3: Coarse abstractions for cores c_1 (bottom left) and c_2 (top left) and observer *Obs* (right), for Example 1 (Sect. 3.3.1).

latest production times of e_1, using the original (non-abstracted) network N_{c_2}. For example, $max_{s5,e_1,2}$ stores the latest time at which e_1 is produced by *s5* within the second period of τ_3. This corresponds to the largest value of the scheduler's clock x when τ_3 is at location *s5* and the value of τ_3's clock y is comprised between bt_{s5} and wt_{s5} (state formula ϕ_{s5,e_1}) *and* the absolute time is greater than the first multiple of P_{τ_3} and less than the second multiple of P_{τ_3} ($20 \leq H_{c_2}.x \leq 40$). Once all such times are acquired, we simply abstract N_{c_2} into $\mathcal{CA}(N_{c_2})$ where, in each kth period ($k \in 1 \, .. \, \frac{hp_{c_2}}{P_{\tau_3}}$) of τ_3, e_1 is produced between $min_{s5,e_1,k}$ and $max_{s5,e_1,k}$ (recall that τ_3 is the only task in T_E with affinity c_2). When the multiple of the period coincides with the hyperperiod, location *wait* ensures waiting for the hyperperiod to be reached, after which the behavior repeats by transiting back to the initial location of $\mathcal{CA}(N_{c_2})$ and resetting its clock x. Before going any further, we recall that the earliest and latest production times are ensured to be *exact* w.r.t. the model of the RTS, since we compute them on the *original core network* N_c (Sect. 3.2). In the case of core c_2, for instance, we obtain $min_{s5,e_1,2} = 22$ and $max_{s5,e_1,2} = 38$, which coincide with the analysis we carried out on N_{c_2} within the second period of τ_3 (Sect. 2.2.2).

We compose then both core abstractions to obtain a new (smaller) network $\mathcal{CA}(N_E) = \mathcal{CA}(N_{c_1}) || \mathcal{CA}(N_{c_2})$, which we compose in parallel with the observer *Obs* in Fig. 3 (right). Here, *Obs* receives e_1 and e_2 on broadcast channels having the same name. From its initial location *idle*, when an e_1 occurs, it transits to location *recv* and resets its clock x. At location *recv*, it awaits the subsequent occurrence of e_2. Therefore, to compute the maximum inter-core bound $B^{max}_{e_1,e_2}$ separating each e_1 and the next e_2, it suffices to query $sup\{Obs.recv\} : Obs.x$ on $\mathcal{CA}(N_E) || Obs$. Note that *Obs* is the simplest possible to obtain $B^{max}_{e_1,e_2}$, e.g., it is not suitable for computing minimal bounds. In Sect. 4, we show how we leverage the power of observers to compute various types of bounds.

The Hole Phenomenon. Unfortunately, inter-core bounds computed using coarse abstractions may be not exact. For instance, the procedure above gives $B^{max}_{e_1,e_2} = 23$. This value is in fact over-approximated. We illustrate this in Fig. 4, where we

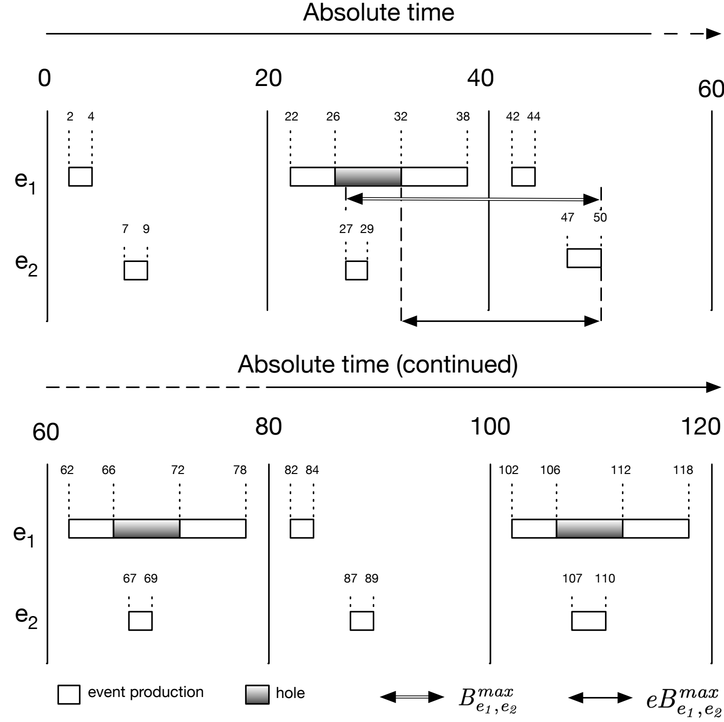

Fig. 4: The hole phenomenon (Example 1, Sect. 3.3.1.)

depict all intervals within which e_1 and e_2 may be produced on a scale of absolute time, up to the least common multiple of both hyperperiods hp_{c_1} and hp_{c_2}, that is $hp = 120$. The behavior repeats afterwards, i.e., in Fig. 4, when reaching time 120, the behavior continues by going back to 0 and shifting the time scale to the right by hp, then again when reaching 240 and so on. The white rectangles represent the exact absolute intervals within which an event may occur. For e_1, they come from the analysis we carried out in Sect. 2.2.2 (and for e_2 through a similar analysis). We can easily see that $B^{max}_{e_1,e_2} = 23$ (double-stroke double-headed arrow), is larger than $eB^{max}_{e_1,e_2} = 18$, the exact one (single-stroke double-headed arrow). In the worst case (in the sense of maximizing the amount of time between any e_1 and a subsequent e_2), e_1 occurs at absolute time $32 + n \cdot hp$ and e_2 occurs at $50 + n \cdot hp$, where n is a positive integer. For $n = 0$, the worst case corresponds to e_1 occurring at the very beginning of the second absolute production interval within the second period of τ_3, i.e., at exactly 32. Now, the next e_2 production will not happen before the third period of τ_1, because e_2 has already been produced in its second period between 27 and 29. Therefore, the subsequent e_2 will be produced at 50 in the worst-case scenario. The exact maximal bound is therefore equal to $50 - 32 = 18$.

The reason behind this weakness of coarse abstractions is the possible existence of *holes*, illustrated using gray rectangles in Fig. 4. In a nutshell,

coarse abstractions capture the exact earliest and latest production times of an event within a given period, but do not take into account non-empty intervals in between (holes) within which the event may never occur, i.e., both white and gray rectangles are treated as white ones. Therefore, in the coarse abstraction shown in Fig. 3, e_1 may occur at any point between 22 and 38, including within the hole $]26, 32[$. Consequently, the worst case scenario corresponds to both events happening at 27 in the order e_2 then e_1, which explains the obtained maximal bound $50 - 27 = 23$.

3.5 Exact Abstractions

To overcome the hole phenomenon, we devise a new, exact abstraction. For this, a new UPPAAL query is needed to extract the "white rectangles" in the sense of Fig. 4.

3.5.1 A new UPPAAL query In a general sense, the new query must be able to compute all the contiguous intervals of values of a clock x within which some state formulae F holds. We implemented this query, called *bounds*, in UPPAAL 5.1.0. In brief, $bounds\{F\} : A.x$ generalizes *sup* and *inf* queries (Section 2.1.2) in a sense that it reports both the minimum and maximum value of x, and in addition reports a precise union of possible values intervals of x when F holds. Such collection of value intervals does not pose a significant overhead besides exploring the full symbolic state space, which is already the case for *sup* and *inf* (Sect. 4).

3.5.2 Using the bounds Query Back to Example 1 (Sect. 3.3.1). To compute exact abstractions $\mathcal{A}(N_{c_2})$ and $\mathcal{A}(N_{c_1})$, we use a similar procedure as for coarse abstractions, with two main differences. First, instead of querying with *inf* and *sup*, we query with *bounds* to compute the exact absolute production intervals. For example, for event e_1, produced by segment $s5$ (task τ_3) on core c_2, we compute:
$Iv_{s5,e_1,1} = bounds\{\phi_{s5,e_1}\ and\ H_{c_2}.x \leq 20\} : H_{c_2}.x$
$Iv_{s5,e_1,2} = bounds\{\phi_{s5,e_1}\ and\ 20 \leq H_{c_2}.x \leq 40\} : H_{c_2}.x$
where ϕ_{s5,e_1} is the same formula as in Sect. 3.4.
We obtain accordingly $Iv_{s5,e_1,1} = \{[2, 4]\}$ and $Iv_{s5,e_1,2} = \{[22, 26], [32, 38]\}$, which correspond to white rectangles in Fig. 4. Second, instead of allowing events to be produced at any moment between their earliest and latest production times, we constrain such production to the exact absolute intervals computed above as follows. For each production of an event e by segment $s \in JS_\tau$ within the kth period of τ, we will have as many outgoing edges as $|Iv_{s,e,k}|$ from location s_k. The guards on these edges will restrict the production of e to the time intervals allowed by the elements of $Iv_{s,e,k}$ (i.e., the white rectangles). For example, in $\mathcal{A}(N_{c_2})$, the exact abstraction of c_2, we will have two outgoing edges from $s5_2$, one guarded with $x \geq 22 \wedge x \leq 26$ and the other with $x \geq 32 \wedge x \leq 38$, as illustrated in Fig. 5.

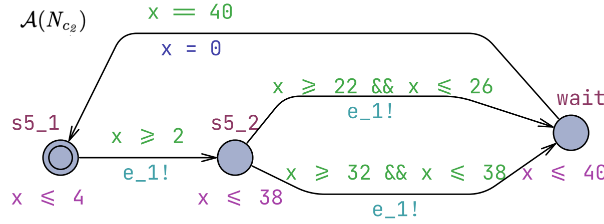

Fig. 5: Exact abstraction of core c_2 for Example 1 (Sect. 3.3.1). The exact abstraction of core c_1 is the same as its coarse abstraction in Fig. 3.

Now, we simply compose $\mathcal{A}(N_{c_2})$ (Fig. 5) with $\mathcal{A}(N_{c_1})$ and the observer in Fig. 3 (right). $\mathcal{A}(N_{c_1})$, the exact abstraction of core c_1, is identical to its coarse abstraction shown in Fig. 3 (bottom left) since there are no holes within the earliest and latest production times of e_2 (Fig. 4). We obtain, using the query $sup\{Obs.recv\} : Obs.x$, the exact bound $eB^{max}_{e_1,e_2} = 18$ which corresponds to the value discussed earlier.

3.5.3 On The Distinct-Affinity Assumption As said in Sect. 3.1, our approach requires tasks in T_E to have distinct affinities ($|T_E| = |C_E|$). Under this assumption, $\mathcal{A}(N_c)$ is a faithful representative of c w.r.t. the exact absolute intervals within which τ, the only task in $T_E \cap prt_c$, produces events. This is because such intervals are (i) obtained on the original N_c, i.e., they include all delays due to scheduling decisions on c, and (ii) not influenced by tasks having a different affinity than τ (since blocking, if any, is integrated in segments WCETs). However, if we lift this assumption, the above is no longer true. Let us illustrate this with an example. Consider a variant of Example 1 (Sect. 3.3.1) breaking the distinct-affinity assumption: now segment $s6$, in task τ_4, also produces event e_2 within the relative interval $[bt_{s6}, wt_{s6}]$. Therefore, two tasks producing events (i.e., τ_3 and τ_4) have the same affinity (c_2). The intuitive way to build $\mathcal{A}(N_{c_2})$ would be to compose in parallel $\mathcal{A}(TA_{\tau_4})$ with $\mathcal{A}(TA_{\tau_3})$, two abstractions with the exact production intervals for e_2 and e_1 by τ_4 and τ_3, respectively. $\mathcal{A}(TA_{\tau_3})$ is what we already illustrated in Fig. 5. $\mathcal{A}(TA_{\tau_4})$ will consist of a TA that produces e_2 between 18 and 22 in each hyperperiod hp_{c_2} (see the analysis in Sect. 2.2.2). Now, this naive $\mathcal{A}(N_{c_2})$ is not an exact abstraction of N_{c_2}: it over-approximates the set of behaviors of N_{c_2} w.r.t. the production of e_1 and e_2. For example, within the first hyperperiod, $\mathcal{A}(N_{c_2})$ contains the behavior "e_1 produced at 38 and e_2 produced at 21". This behavior does not exist in N_c: if e_2 is produced at 21, then e_1 is necessarily produced between 23 and 26 (because in this case, τ_2 is preempted at the end of $s6$ and $s5$ executes between $21+2 = 23$ and $22+4 = 26$, Sect. 2.2.2). As said in Sect. 1, solving the generic problem of computing exact inter-core bounds is particularly hard. We pay the price of exactness through the distinct-affinity restriction.

3.5.4 Automatic Generation We now present algorithms to automatically generate exact abstractions, and begin with setting up some notation. We use

(i) to denote an element at position i in an ordered set, and $.i$ to access the ith element in a couple. The function lb (resp. rb) returns the left (resp. right) bound of an interval in its domain $\cup_{s \in \mathcal{JS}_E} \mathbb{I}_s$. A compact notation is used for imbricated loops. For instance, the *for* loop in Algorithm 1 (lines $5-8$) unfolds into three imbricated *for* loops, from the outermost to the innermost: *for* τ *in* T_E, *for* k *in* $1 \mathinner{..} \frac{hp_c}{P_\tau}$ and *for* s *in* JS^E_τ. Finally, the statements between quotes (" ") are interpreted as is, with variables replaced by their values. For the readability of figures, we sometimes omit clauses in guards that are superfluous. For example, if a guard on $s \rightarrow s'$ generated by an algorithm is $x \geq a \wedge x \leq b$ and s has an invariant $x \leq b$, then the second clause of such guard is sometimes dropped in the figures.

Computing exact intervals. We propose in Algorithm 1 a generic procedure to compute Iv variables, which will be used to generate the abstractions. First, we generate original N_c networks for each $c \in C_E$ using the automatic generator from FHZ, simplified after verifying schedulability (Sect. 2.2), and with an additional reference clock for the scheduler; then build SS_c, the state space of N_c (lines 1-4). Afterwards, we compute, for each $\tau \in T_E$ (with distinct affinity c), for each period of τ in the hyperperiod hp_c and for each segment s in JS^E_τ, the exact intervals of the *first event* produced by s, stored in the variables $Iv_{s,s_E(1).1,k}$ (lines 5-8). This is done through querying the model checker (line 8), on the state space SS_c, already computed in line 4, with *bounds* properties built using the values of c, τ, k, s and s_E (line 7). For Example 1 (Sect. 3.3.1), we obtain the values of Iv variables discussed in Sect. 3.5.2.

Algorithm 1: Computing exact intervals of event production

1 **for** c *in* C_E **do**
2 | generate $N_c = H_c ||_{\tau \in prt_c} TA_\tau$ ◁ generate original N_c network (FHZ)
3 | add clock x to H_c
4 | $SS_c \leftarrow$ buildSS(N_c) ◁ call the model checker
5 **for** $\langle \tau, k, s \rangle$ *in* $T_E \times 1 \mathinner{..} \frac{hp_c}{P_\tau} \times JS^E_\tau$ **do**
6 | $c \leftarrow aff_\tau$
7 | $\theta \leftarrow$ "$TA_\tau.s$ *and* $lb(s_E(1).2) \leq TA_\tau.y \leq rb(s_E(1).2)$ *and* $(k-1) \cdot P_\tau \leq H_c.x \leq k \cdot P_\tau$"
8 | $Iv_{s,s_E(1).1,k} \leftarrow$ compute(SS_c, "$bounds\{\theta\} : H_c.x$") ◁ call the model checker

Generating exact abstractions. The algorithm to generate exact abstractions is presented in a gradual manner. First, we provide an algorithm under the assumption of *single-job tasks*: each task producing an event has only one job, i.e., $\forall \tau \in T_E : |J_\tau| = 1$ (and therefore $|JS^E_\tau| = 1$). The procedure under the above assumption is given in Algorithm 2.
Single-event segments: Let us begin by considering that each segment in $\mathcal{JS}_E$ produces only one event (which is the case of Example 1, Sect. 3.3.1); we ignore therefore for now grayed lines in Algorithm 2. To construct $\mathcal{A}(N_c)$ for some core c, the exact abstraction of the only task in T_E allocated to c, say τ, we need to generate its clocks, locations and edges. The clock is x (line 5). Since τ is single job, we store the only segment in JS^E_τ that produces an event, say e, in

variable s (line 4). Locations s_k and their invariants are generated for each period of τ in the hyperperiod hp_c (lines 9-10). The constraint over x is driven by the right bound of the last interval in the corresponding Iv variable (line 10). Location *wait*, with invariant $x \leq hp_c$, is added (line 13). The initial location is s_1 (line 14). Afterwards, edges are generated (lines 16-26, 37). For each kth period of τ (up to the hyperperiod) and i^{th} interval (among the possible intervals within which s may produce e), the statements in lines 22 and 24 determine the successor of s_k depending on whether there are more events within the hyperperiod. Then, the auxiliary function *edge1*() (Algorithm 3) creates an edge accordingly. Each ith outgoing edge of location s_k, producing event e in the kth period of τ, has:

- a guard with clock x larger than the left bound of the ith interval in $Iv_{s,e,k}$ and less than its right bound,
- an emission (!) on broadcast channel e,
- an empty set of clocks to reset.

Line 37 generates an edge from location *wait* to the event-producing location in the first period, with the guard $x = hp_c$ and a reset over clock x. For Example 1, Algorithm 2 will give accordingly $\mathcal{A}(N_{c_2})$ that we obtained in Fig. 5, and $\mathcal{A}(N_{c_1})$, illustrated in Fig. 3, bottom left (we recall that $\mathcal{A}(N_{c_1})$ is identical to $\mathcal{CA}(N_{c_1})$ because of the absence of holes w.r.t. the production of e_2).

Multiple-event segments: Now, segments may generate multiple events. This part requires a lot of care not to lose the exactness of the abstraction, as we will explain in detail.

We reason as follows. If s, the only segment in JS_τ^E, produces more than one event ($|s_E| > 1$), we need the exact intervals corresponding to the production of the *first* event in each kth period in the hyperperiod, i.e., $Iv_{s,s_E(1).1,k}$, already obtained from the generic Algorithm 1. The absolute intervals for producing subsequent events, within the same period, in each *branch* corresponding to each interval in $Iv_{s,s_E(1).1,k}$, are deduced accordingly.

Let us illustrate this through building the abstraction of core c_2 from Example 2 (Sect. 3.3.2). The exact absolute intervals are computed for the production of e_3 by $s5$. We obtain accordingly $Iv_{s5,e_3,1} = \{[0, 1]\}$ and $Iv_{s5,e_3,2} = \{[20, 23], [30, 35]\}$. For the subsequent events, Algorithm 2 creates locations $s_k_i_j$ corresponding to "producing the jth event in s_E, on the ith branch, in the kth period" (lines 11-12). For example, location $s5_1_1_2$ corresponds to producing the second event (e_1) on the first branch (here there is only one branch since $|Iv_{s5,e_3,1}| = 1$) in the first period. Using Algorithm 2, we obtain $\mathcal{A}(N_{c_2})$, illustrated in Fig. 6 ($\mathcal{A}(N_{c_1})$ remains the same as in Fig. 3, bottom left). Obtaining the exact absolute interval of producing an event by $s_k_i_j$ is the hard part.

First, let us ignore constraints over clock y in Fig. 6. Let s be a segment and $s_E = \{(e, [a, b]), (e', [a', b'])\}$. We know that, in any execution scenario, if e is produced by s in the absolute interval $[f, h]$, e' will follow within $[f + c, h + d]$ where $c = a' - a$ and $d = b' - b$. This is guaranteed because segments may not be preempted (Sect. 2.2) and $[c, d]$ is necessarily non-empty (Sect. 3.1). Constraints over x will therefore model $[f + c, h + d]$, where each $[f, h]$ interval within the

Algorithm 2: exact abstractions (single-job tasks)

```
1  Algorithm 1                                           ◁ Compute exact intervals
   ◁ Compute abstractions                                                        ▷
2  create exact abstraction A(N_E) with broadcast channel "e" for each e ∈ E
3  for τ ∈ T_E do
4      c ← aff_τ, s ← JS^E_τ(1)
5      create exact abstraction A(N_c) with clock set X = {x}
6      if |s_E| ≠ 1 then
7          X ← X ∪ {y}
8      generate locations:
9          for k in 1 .. hp_c/P_τ do
10             create location "s_k" with invariant "x ≤ rb(Iv_{s,s_E(1).1,k}(|Iv_{s,s_E(1).1,k}|))"
11             for ⟨i, j⟩ in 1 .. |Iv_{s,s_E(1).1,k}| × 2 .. |s_E| do
12                 create location "s_k_i_j" with invariant
                     "x ≤ rb(Iv_{s,s_E(1).1,k}(i)) + rb(s_E(j).2) − rb(s_E(1).2) &&
                      y ≤ rb(s_E(j).2) − lb(s_E(j − 1).2)"
13         create location "wait" with invariant "x ≤ hp_c"
14         make location "s_1" initial
15     generate edges:
16         for ⟨k, i⟩ in 1 .. hp_c/P_τ × 1 .. |Iv_{s,s_E(1).1,k}| do
17             iv ← Iv_{s,s_E(1).1,k}(i)
18             if |s_E| ≠ 1 then
19                 succ ← s_k_i_2, reset ← {y}
20             else
21                 if k ≠ hp_c/P_τ then
22                     succ ← s_(k + 1)
23                 else
24                     succ ← wait
25                 reset ← ∅
26             edge1(s,k,iv,succ,reset)
27             for j in 2 .. |s_E| do
28                 if j ≠ |s_E| then
29                     succ ← s_k_i_(j + 1), reset ← {y}
30                 else
31                     if k ≠ hp_c/P_τ then
32                         succ ← s_(k + 1)
33                     else
34                         succ ← wait
35                     reset ← ∅
36                 edge2(s,k,i,j,iv,succ,reset)
37         create edge "(wait, x == hp_c, ϵ, {x}, s_1)"
38     add A(N_c) to A(N_E)
```

Algorithm 3: Auxiliary functions

```
1  Function edge1(s,k,iv,s′,r) is
2      create edge "(s_k, x ≥ lb(iv) && x ≤ rb(iv), s_E(1).1!, r, s′)"
3  Function edge2(s,k,i,j,iv,s′,r) is
4      create edge "(s_k_i_j, x ≥ lb(iv) + lb(s_E(j).2) − lb(s_E(1).2) &&
         y ≥ lb(s_E(j).2) − rb(s_E(j − 1).2), s_E(j).1!, r, s′)"
```

k^{th} period is an interval in $Iv_{s,e,k}$, which corresponds to a *branch*, as introduced above. In Fig. 6, the invariant constraint of $s5_2_1_2$ over x, for instance, is $x \leq 26$, where, using the notation above, $h + d = 26$, with $h = 23$ (the right bound of the first element in $Iv_{s5,e_3,2}$) and $d = b' - b = 4 - 1 = 3$, 4 (resp. 1)

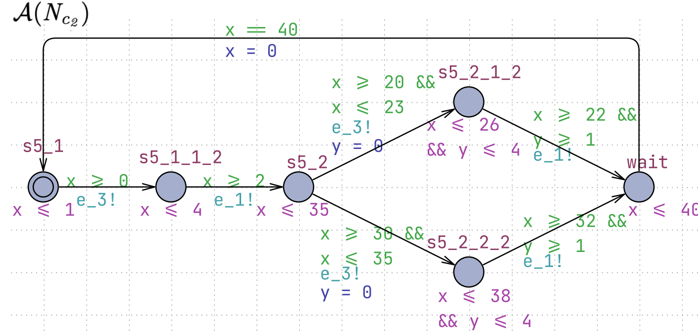

Fig. 6: Exact abstraction of core c_2 for Example 2 (Sect. 3.3.2), obtained using Algorithm 2. The exact abstraction obtained for core c_1 is the same as its coarse abstraction in Fig. 3, bottom left.

being the right bound of $s5_E(2).2$ (resp. $s5_E(1).2$). The guard $x \geq 22$ on the outgoing edge of *s5_2_1_2* is obtained similarly. We get the same constraints over x for the production of e_1 as in Fig. 5, which is expected since the producer and relative production intervals of e_1 are the same as in Example 1 (Sect. 3.3.1).

However, these constraints are not enough to guarantee an exact abstraction. In our example, following only the constraints on x in Fig. 6, *s5_2* (resp. *s5_2_1_2*) can produce e_3 (resp. e_1) at any instant between 20 and 23 (resp. 22 and 26). In the original network, however, it is not possible to produce e_3 then e_1, both at e.g., 22, because their relative production intervals $[0, 1]$ and $[2, 4]$ are separated by at least one time unit. This difficulty arises from the fact that a range within an absolute interval corresponds to a *sliding* relative interval: values in $[20, 22]$ may all correspond to the relative 0 for e_3. To remedy this, we conjunct constraints on x with constraints on clock y (created in line 7). The latter is reset when an event e is produced, and constrains the production of a subsequent event e' using information from their relative production intervals. For example, *s5_2_1_2* may produce e_1 within its absolute interval iff it obeys the minimum delay, i.e., $2 - 1 = 1$, and the maximum delay, i.e., $4 - 0 = 4$ separating e_1 and e_3. The conjunction of constraints works nicely, preventing *s5_2_1_2* to produce e_1 too early or too late. On the one hand, constraints over x forbid e_1 to occur 1 (resp. 4) time units after e_3 if the production of e_3 corresponds exclusively to the relative instant 0 (resp. 1). That is, if e_3 occurs at 20 (resp. 23), which corresponds necessarily to the relative instant 0 (resp. 1), the guard $x \geq 22$ (resp. the invariant $x \leq 26$) will prevent e_1 to be produced at 21, i.e., too early (resp. 27, i.e., too late). On the other hand, constraints on y will prevent *s5_2_1_2* to produce e_1 e.g., at 26 if e_3 is produced at 21 (too late) or at 22 when e_3 is produced at the same time (too early). Note that, for the readability of Fig. 6, constraints on y are removed when superfluous.

General case: Finally, we lift the single-job assumption in Algorithm 4. The main difference is when task τ is multiple job (lines 6-28). The difficulty here is that multiple segments (in different jobs) may produce events, but we may only have one initial location in a TA. For this, we use an initial, committed location *act* (line 14) that allows to atomically transit to one of the event producers

in the first period, through edges $act \rightarrow s_1$, where s is an element of JS_τ^E (line 20). For the remaining edges, it suffices to iterate over JS_τ^E and reuse the corresponding instructions from Algorithm 2 (line 21). Then, we need to consider paths corresponding to producing an event within one job in one period, and producing an event within a different job in the next period (lines 22-27). For Example 3 (Sect. 3.3.3), Algorithm 4 generates $\mathcal{A}(N_{c_1})$ illustrated in Fig. 7. For single-job tasks, we take a shortcut reusing lines from Algorithm 2, therefore generating, for Example 3, the same $\mathcal{A}(N_{c_2})$ illustrated in Fig. 6.

Algorithm 4: exact abstractions (general case)

```
1  Algorithm 1                                              ◁ Compute exact intervals
   ◁  Compute abstractions                                                       ▷
2  create exact abstraction A(N_E) with broadcast channel "e" for each e ∈ E
3  for τ ∈ T_E do
4      if |JS_τ^E| = 1 then
5          Algorithm 2, lines 4-37
6      else
7          c ← aff_τ
8          Create exact abstraction A(N_c) with clock set X ← {x}
9          for s ∈ JS_τ^E do
10             if |s_E| > 1 then
11                 X ← X ∪ {y}
12                 break
13         generate locations:
14             create committed initial location "act"
15             create location wait with invariant "x ≤ hp_c"
16             for s in JS_τ^E do
17                 Algorithm 2, lines 9-12
18         generate edges:
19             for s in JS_τ^E do
20                 create edge "(act, ⊤, ϵ, ∅, s_1)"
21                 Algorithm 2, lines 16-36
22             for ⟨s, k, iv, s' ≠ s⟩ in JS_τ^E × 1 .. hp_c/P_τ − 1 × Iv_{s,s_E(1).1,k} × JS_τ^E do
23                 if |s_E| = 1 then
24                     edge1(s,k,iv,s'_(k + 1), ∅)
25                 else
26                     i ← index(iv)
27                     edge2(s,k,i,|s_E|,iv,s'_(k + 1), ∅)
28             create edge "(wait, x == hp_c, ϵ, {x}, act)"
29     add A(N_c) to A(N_E)
```

Note that, in the particular case of multiple-job tasks producing events, the validity of requirements is very important. Consider e.g., Example 3 (Sect. 3.3.3) with the requirement Req_1 "compute the maximal bound between each production of e_4 and the next production of e_3". Req_1 is valid, because it seeks the maximum amount of time between any production of e_4 (given that it takes place) and the subsequent e_3 (which always happens). In contrast, Req_2, corresponding to Req_1 where e_3 and e_4 are switched, is invalid. Indeed, such bound does not exist: there are infinite executions where e_4 does not occur at all, i.e., following the path where τ_2 never executes the job $\{s2, s3\}$. This is not a verification issue, but a design issue, and does not have an influence on the abstraction's exactness. The

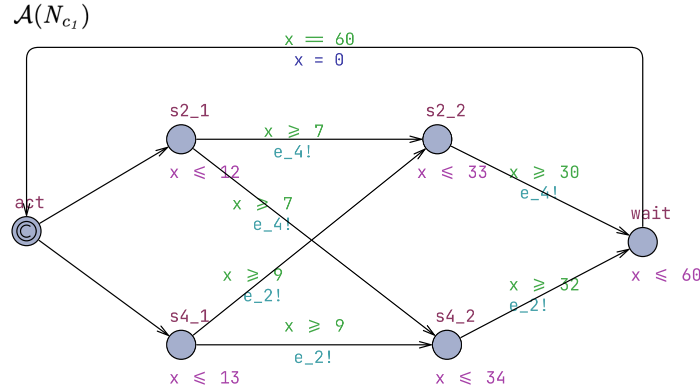

Fig. 7: Exact abstraction of core c_1 for Example 3 (Sect. 3.3.3), obtained using Algorithm 4. The exact abstraction obtained for core c_2 is the same as in Fig. 6.

trickier, and more dangerous case, is when the user has an invalid requirement in mind, and tries to build an exact abstraction for a multiple-job task where some jobs produce no events at all. Our implementation of Algorithm 4 anticipates this case (details in Sect. 4.1).

4 Evaluation

4.1 Technicalities

We automate our approach through a full implementation of the algorithms presented in Sect. 3. The code is written in OCaml and is publicly available. Several technicalities are addressed.

Improving scalability. Algorithm 1 contains two potential sources of costly computation: (i) the heavy use of clock constraints in the *bounds* query (line 3) and (ii) the multiple calls to the model checker (lines 7-8). Point (i) stems from the fact that in *sup* and *inf*, and their *bounds* generalization, it is recommended to use as few clock constraints as possible (in the enclosed state formula) for a better scalability. In our implementation of Algorithm 1, we drastically minimize the number of clock constraints in *bounds* queries as well as the number of invocations of the model checker using the following trick. Instead of adding a clock to the scheduler H_c to get an absolute reference of time, we compose N_c with a TA *Ref* with one clock x and one location *hper* with the invariant $x \leq hp_c$. This creates an artificial *timelock* at exactly hp_c, therefore allowing to stop the exploration when the reference clock hits the hyperperiod limit. Then, instead of querying the model checker k times for each event-producer within each task, we simply replace clock constraints over x in H_c with the proposition *Ref.hper* and compute only one time the bounds on *Ref.x* per event-producing segment. We get therefore all the absolute production intervals of the first event in each segment within the hyperperiod at once, then simply group them according to

the period to which they belong. For Example 1 (Sect. 3.3.1), for instance, in order to build the exact abstraction $\mathcal{A}(N_{c_2})$, we query only once:

$$Iv = bounds\{\phi_{s5,e_1} \text{ and } Ref.hper\} : Ref.x$$

To obtain $Iv = \{[2,4],[22,26],[32,38]\}$, then group the intervals per period of τ_3 to get $Iv_{s5,e_1,1} = \{[2,4]\}$ and $Iv_{s5,e_1,2} = \{[22,26],[32,38]\}$.

Guaranteeing exactness under invalid requirements. In our implementation of Algorithm 4, in the case of multiple-job tasks producing events, we always issue a warning, asking the user to make sure their requirement is valid (i.e., they are not trying to compute a bound that does not exist). Moreover, we check that, in such multiple-job tasks, each job has a segment producing an event. If this is not the case, the procedure is aborted. To explain the reason behind this, let us consider a variation of Example 3 (Sect. 3.3.3) where τ_2 produces no longer e_2 and the requirement *Req* "compute the maximal bound between each production of e_3 and the next production of e_4". This requirement is clearly invalid, for the same reasons depicted at the end of Sect. 3. The problem here is, if the user chooses to generate the abstraction anyway, it will be no longer exact given the requirement *Req*: the path corresponding to the job $\{s2, s3\}$ will no longer exist and the verification would give a value for the non-existent bound defined by *Req*. Therefore, we generate no abstraction in this case and ask the user to provide an event producer for every job. This choice is deliberately conservative, as we seek to guarantee the exactness of the abstraction regardless of whether the requirement is valid or not. It is still possible to ignore this safeguard and generate the abstraction anyway (see below).

Options. The -xta option allows to use existing TA files for N_c networks (i.e., skipping line 2 in Algorithm 1), and the -verbose option to visualize the computation of intervals and their grouping within periods. The -force option allows to generate the abstraction if at least one multiple-job task $\tau \in T_E$ fails the test above (at least one of its jobs contains no event-producing segment). In this case, it is up to the user to make sure that their requirement guarantees the exactness of the generated abstraction.

4.2 Experiments

To demonstrate the scalability of our approach, we compute exact inter-core bounds of various types on the WATERS 2017 industrial challenge [13]. The reported results have been obtained using Uppaal 5.1.0 (with the state-space representation option set to DBM) on a mid-range computer with an Intel Core i9 processor, 10 cores and 32 GiB of RAM. The results are reproducible in a fully automatic manner[5].

The challenge underlies a real automotive quad-core RTS. Our approach is applicable to the subsystem with cores c_1 and c_2, all tasks allocated to which are periodic. Model checking this subsystem is particularly challenging. First, the

[5] https://gitlab.math.univ-paris-diderot.fr/mzhang/exact-abstraction

Task	Period ($\times 10^{-6} s$)	Priority	# Segments	Affinity
T_2	2 000	6	28	c_2
T_5	5 000	5	23	c_2
T_20	20 000	4	307	c_2
T_50	50 000	3	46	c_2
T_100	100 000	2	247	c_2
T_200	200 000	1	15	c_2
T_1000	1 000 000	0	44	c_2
Angle_Sync	6 660	1	146	c_1
T_1	1 000	0	41	c_1

Table 1: WATERS 2017 industrial challenge information (core c_1 and c_2).

number of interleavings is extremely high: nine tasks, with 897 segments overall (Table 1). Second, WCETs and BCETs of segments (not reported in Table 1) have a nanosecond resolution, and the difference between the smallest and the largest timing constraint is about 1 billion time units. Tasks segments use data *labels* to communicate. N_{c_1} and N_{c_2} are both schedulable given a frequency of *400 MHz*. FHZ verified N_{c_2} apart, computing precise WCRTs of tasks in prt_{c_2} with blocking due to labels sharing taken into account. All tasks in the challenge are single job.

We want to compute various minimum and maximum exact bounds between writing a label lbl_x, by a segment s in a task allocated to one core, and writing another label lbl_y by a segment s', in a task allocated to the other core, such that s' writes lbl_y based on a prior reading of lbl_x. We discuss the results for labels $x = 1327$ and $y = 4164$, where segment *Runnable_6660_us_1* in task *Angle_Sync* (core c_1) writes lbl_{1327}, and segment *Runnable_50_ms_2* in task *T_50* (core c_2) reads lbl_{1327} and writes lbl_{4164} accordingly. For simplicity, we use 1 instead of 1327 (for x), 2 instead of 4164 (for y), $s1$ instead of *Runnable_6660_us_1* and $s2$ instead of *Runnable_50_ms_2*. Using *implicit communication semantics*, a segment reads (resp. writes) labels at the beginning (resp. end) of its execution. Accordingly, $E = \{w_1, r_1, w_2\}$, $C_E = \{c_1, c_2\}$, $T_E = \{Angle_Sync, T_50\}$, $JS^E_{Angle_Sync} = \{s1\}$, $JS^E_{T_50} = \{s2\}$, $s1_E = \{(w_1, [bt_{s1}, wt_{s1}])\}$ and $s2_E = \{(r_1, [0, 0]), (w_2, [bt_{s2}, wt_{s2}])\}$, and the requirement is "compute exact bounds between each w_1 and the next w_2 with r_1 happening in between". Note the relative interval associated with r_1, equal to $[0, 0]$. Since r_1 occurs before w_2 within the same segment, and given the requirement above, the latest relative instant corresponding to reading lbl_1 is unimportant: what matters is that r_1 actually happens between w_1 and w_2.

Leveraging the power of observers, we answer the requirement above, following different latency semantics [8]. The *first-to-first (FF)* semantics corresponds simply to "between each production of w_1 and the next w_2 with r_1 happening in between" (this semantics is what we used in Sect. 3.4, with two events, to illustrate the hole phenomenon). The *last-to-first (LF)* semantics exclude overwritten w_1 events. That is, FF excluding executions where at least one w_1 occurs between w_1 and r_1. The exact minimum and maximum FF and LF bounds are denoted, respectively, $eBF^{min}_{w_1,r_1,w_2}$ and $eBF^{max}_{w_1,r_1,w_2}$, $eBL^{min}_{w_1,r_1,w_2}$ and $eBL^{max}_{w_1,r_1,w_2}$.

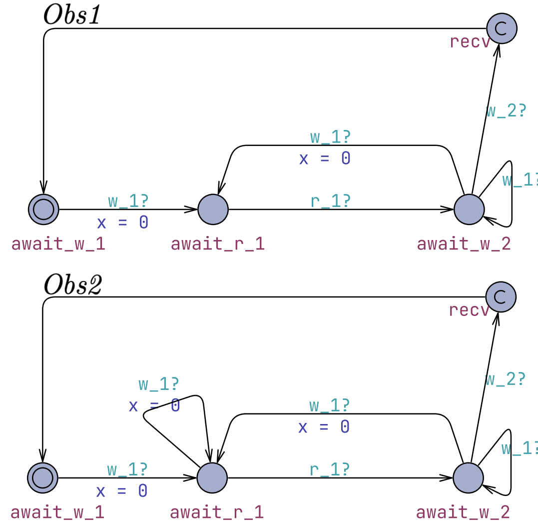

Fig. 8: Observers for FF and LF bounds.

To compute FF bounds, we use *Obs1* (Fig. 8, top). It awaits for a chain of events w_1 (edge $await_w_1 \to await_r_1$, with a reset over clock x) followed by r_1 (edge $await_r_1 \to await_w_2$) followed by w_2 (edge $await_w_2 \to recv$). Events w_1 that may occur while awaiting r_1 are safely ignored. In contrast, events w_1 that may occur while awaiting w_2 must be considered. The non determinism over the edges $await_w_2 \to$ synchronized on w_1 allows to either take such w_1 events into account or ignore them. Therefore, all the chains $w_1 \rightarrowtail r_1 \rightarrowtail w_2$, such that further occurrences of w_1 between w_1 and r_1 are ignored, are taken into account (FF semantics). For LF bounds, *Obs2* (Fig. 8, bottom) is similar to *Obs1*, with the difference that it takes into account overwriting w_1 through resetting clock x whenever an event w_1 occurs while waiting for r_1. Therefore, to get exact FF and LF bounds, i.e., respectively, $eBF^{min}_{w_1,r_1,w_2}$, $eBF^{max}_{w_1,r_1,w_2}$, and $eBL^{min}_{w_1,r_1,w_2}$ and $eBL^{max}_{w_1,r_1,w_2}$, it suffices to query $inf\{Obs1.recv\} : Obs1.x$, $sup\{Obs1.recv\} : Obs1.x$, and $inf\{Obs2.recv\} : Obs2.x$, $sup\{Obs2.recv\} : Obs2.x$.

Abstraction	Cost (s)	Cost (GiB)
$\mathcal{A}(N_{c_1})$	109	0.56
$\mathcal{A}(N_{c_2})$	122	3.3

Table 2: Cost (time and memory) of abstractions' computation

Bound	Value (10^{-9} s)
$eBF^{min}_{w_1,r_1,w_2}$	1 766 230
$eBF^{max}_{w_1,r_1,w_2}$	56 808 228
$eBL^{min}_{w_1,r_1,w_2}$	14 080
$eBL^{max}_{w_1,r_1,w_2}$	7 074 911

Table 3: Exact FF and LF bounds

We first apply the direct method from FHZ, i.e., try to compute the bounds on the parallel composition $N_E||Obs$ with $N_E = N_{c_1}||N_{c_2}$ and $Obs \in \{Obs1, Obs2\}$. The verification fails with all 32 GiB of RAM exhausted in less than 15 minutes. Then, we use our approach to generate $\mathcal{A}(N_E) = \mathcal{A}(N_{c_1})||\mathcal{A}(N_{c_2})$. The cost of computing $\mathcal{A}(N_{c_1})$ and $\mathcal{A}(N_{c_2})$, mainly driven by model checking N_{c_1} and N_{c_2}

apart (to compute the exact intervals), is reported in Table 2: less than four minutes and less than 4 GiB of RAM overall. We compose $\mathcal{A}(N_E)$ with each observer and compute FF and LF bounds (Table 3); the cost of such computation is negligible (less than 1 second, and less than 30 MB of RAM).

4.3 Discussion

We efficiently compute, in a fully automatic manner, various exact (to the nanosecond) inter-core bounds on the real WATERS 2017 industrial challenge. To the best of our knowledge, this is the first work that does so. These exact bounds significantly reduce pessimism. For e.g., $eBF^{max}_{w_1,r_1,w_2}$, its value using LET semantics, a popular choice in automotive industry (Sect. 5), is 96 920 000, i.e., > 40 ms larger than the one we computed (Table 3). Yet, our technique is restricted to a scheduling policy, and by the distinct-affinity assumption. LET-based approaches are therefore more generic, but more pessimistic than ours (Sect. 5).

5 Related Work

Due to the state-space explosion problem, the literature related to inter-core bounds is dominated by analytical approaches, mostly specific to end-to-end latencies. The latter typically refer to maximum bounds in cause-effect chains (Becker et al. [2]), separating sensor reading and actuation. Feiertag et al. [8] propose in a seminal paper a number of algorithms for end-to-end latencies on single-core platforms, for which they define precise semantics. The pessimism of upper bounds in cause-effect chains is reduced in Kloda et al. [15] and Girault et al. [11], under specific workload and scheduling assumptions (P-FP preemptive scheduling for the former, and event-triggered workloads on single-core platforms for the latter). Martinez et al. [20] and Günzel et al. [12] provide upper bounds for end-to-end latencies with different semantics under different communication models for automotive systems. Due to the dependency of existing algorithms (to over-approximate end-to-end latencies) on scheduling assumptions, communication models based on the Logical Execution Time (LET) [14] gained significant popularity [19]. Using *publishing points* of read/write events [19, 20], they are oblivious to scheduling assumptions. Therefore, the computed bounds remain valid regardless of tasks execution order. The price to pay is however larger bounds, as we have exemplified in Sect. 4.

Our approach shares a central assumption with the related work above: they all consider a schedulable RTS. Besides, all such works, excluding the ones on LET, assume that a faithful model, that e.g., integrates blocking, exists. However, we compute exact bounds, whereas the works above over-approximate them. Also, our model is not restricted to cause-effect chains. In particular, in a cause-effect chain, a task consumes one event and produces another one, which is consumed by the next task and so on. Our model does not make such assumption, and is therefore suitable for bounds where e.g., two tasks in parallel produce events that

are used by a subsequent task (see e.g., the drone application in [10]). Moreover, we preserve the generality of model checking, therefore allowing for extensions to verify other properties. On the downside, the distinct-affinity assumption is a strong one. Further, compared to LET-based approaches, our model is restricted to a specific, even though popular, scheduling model. The last two points are the price we pay for exactness.

6 Conclusion

We presented in this paper a novel scalable approach to compute exact inter-core bounds in a schedulable RTS. Our technique, fully automated, is based on an algorithm that computes exact abstractions through a new query that we implemented in the state-of-the-art model checker Uppaal. The scalability of our algorithms is demonstrated on the WATERS 2017 industrial challenge, on which we successfully computed various exact inter-core bounds in less than four minutes and with less than 4 GiB of memory. However, our work is restricted to periodic tasks under P-FP scheduling with limited preemption, and more importantly constrained by the distinct-affinity assumption. In future work, we plan to extend our method to classical cause-effect chains, therefore relaxing the distinct-affinity assumption. This will allow us to provide engineers with two different solutions based on their needs.

7 Acknowledgements

This work has received support under the program “Investissement d’Avenir” launched by the French Government and implemented by ANR, with the reference “ANR-18-IdEx-000” as part of its program “Emergence”. MAF, the main author, thanks ANR MAVeriQ (ANR-20-CE25-0012) and ANR-JST CyPhAI (ANR-20-JSTM-0001/JPMJCR2012).

References

1. Alur, R., Dill, D.L.: A theory of timed automata. Theor. Comp. Sci. **126**(2), 183–235 (1994). https://doi.org/10.1016/0304-3975(94)90010-8
2. Becker, M., Dasari, D., Mubeen, S., Behnam, M., Nolte, T.: End-to-end timing analysis of cause-effect chains in automotive embedded systems. Journal of Systems Architecture **80**, 104–113 (2017). https://doi.org/10.1016/j.sysarc.2017.09.004
3. 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
4. Bini, E., Buttazzo, G.C.: Schedulability analysis of periodic fixed priority systems. IEEE Trans. Computers **53**(11), 1462–1473 (2004). https://doi.org/10.1109/TC.2004.103
5. Buttazzo, G.C.: Hard real-time computing systems: predictable scheduling algorithms and applications, Real-Time Systems Series, vol. 24 (2011). https://doi.org/10.1007/978-1-4614-0676-1

6. Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Transactions on Programming Languages and Systems (TOPLAS) **8**(2), 244–263 (1986). https://doi.org/10.1145/5397.5399
7. Davis, R.I., Burns, A.: Response time upper bounds for fixed priority real-time systems. In: Real-Time Systems Symposium (RTSS). pp. 407–418. IEEE (2008). https://doi.org/10.1109/RTSS.2008.18
8. Feiertag, N., Richter, K., Nordlander, J., Jonsson, J.: A compositional framework for end-to-end path delay calculation of automotive systems under different path semantics. In: Real-Time Systems Symposium (RTSS). IEEE (2009)
9. Foughali, M., Hladik, P., Zuepke, A.: Compositional verification of embedded real-time systems. J. Syst. Archit. **142**, 102928 (2023). https://doi.org/10.1016/j.sysarc.2023.102928
10. Foughali, M., Zuepke, A.: Formal verification of real-time autonomous robots: An interdisciplinary approach. Frontiers Robotics AI **9**, 791757 (2022). https://doi.org/10.3389/FROBT.2022.791757
11. Girault, A., Prévot, C., Quinton, S., Henia, R., Sordon, N.: Improving and estimating the precision of bounds on the worst-case latency of task chains. IEEE transactions on computer-aided design of integrated circuits and systems (TCAD) **37**(11), 2578–2589 (2018). https://doi.org/10.1109/TCAD.2018.2861016
12. Günzel, M., Chen, K.H., Ueter, N., Brüggen, G.v.d., Dürr, M., Chen, J.J.: Compositional timing analysis of asynchronized distributed cause-effect chains. ACM Transactions on Embedded Computing Systems **22**(4), 1–34 (2023). https://doi.org/10.1145/3587036
13. Hamann, A., Dasari, D., Kramer, S., Pressler, M., Wurst, F., Ziegenbein, D.: Waters industrial challenge. In: International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS) (2017)
14. Henzinger, T.A., Horowitz, B., Kirsch, C.M.: Embedded control systems development with Giotto. In: Workshop on Languages, compilers and tools for embedded systems (LCTES). pp. 64–72 (2001). https://doi.org/10.1145/384197.384208
15. Kloda, T., Bertout, A., Sorel, Y.: Latency analysis for data chains of real-time periodic tasks. In: International conference on emerging technologies and factory automation (ETFA). vol. 1, pp. 360–367. IEEE (2018). https://doi.org/10.1109/ETFA.2018.8502498
16. Larsen, K.G., Pettersson, P., Yi, W.: UPPAAL in a nutshell. International journal on software tools for technology transfer (STTT) **1**(1), 134–152 (1997). https://doi.org/10.1007/s100090050010
17. Larsen, K.G., Larsson, F., Pettersson, P., Yi, W.: Efficient verification of real-time systems: Compact data structure and state-space reduction. In: Real-Time Systems Symposium (RTSS). pp. 14–24. IEEE (1997). https://doi.org/10.1109/REAL.1997.641265
18. Maida, M., Bozhko, S., Brandenburg, B.B.: Foundational response-time analysis as explainable evidence of timeliness. In: Euromicro Conference on Real-Time Systems (ECRTS). pp. 1–25 (2022). https://doi.org/10.4230/LIPIcs.ECRTS.2022.19
19. Martinez, J., Sañudo, I., Bertogna, M.: Analytical characterization of end-to-end communication delays with logical execution time. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD) **37**(11), 2244–2254 (2018). https://doi.org/10.1109/TCAD.2018.2857398

20. Martinez, J., Sañudo, I., Bertogna, M.: End-to-end latency characterization of task communication models for automotive systems. Real-Time Systems **56**, 315–347 (2020). https://doi.org/10.1007/s11241-020-09350-3
21. Nasri, M., Brandenburg, B.B.: An exact and sustainable analysis of non-preemptive scheduling. In: RTSS. pp. 12–23 (2017). https://doi.org/10.1109/RTSS.2017.00009