Non-Simultaneity as a Design Constraint

Jean Guyomarc'h
Krono-Safe, Massy, France
Université Paris-Saclay, CNRS,
Systèmes et Applications des Technologies de l'Information et de l'Energie, Orsay, France
jean.guyomarch@krono-safe.com

François Guerret
Krono-Safe, Massy, France
francois.guerret@krono-safe.com

Bilal El Mejjati
Krono-Safe, Massy, France
bilal.elmejjati@krono-safe.com

Emmanuel Ohayon
Krono-Safe, Massy, France
emmanuel.ohayon@krono-safe.com

Bastien Vincke
Université Paris-Saclay, CNRS,
Systèmes et Applications des Technologies de l'Information et de l'Energie, Orsay, France
bastien.vincke@universite-paris-saclay.fr

Alain Mérigot
Université Paris-Saclay, CNRS,
Systèmes et Applications des Technologies de l'Information et de l'Energie, Orsay, France
alain.merigot@universite-paris-saclay.fr

Abstract

Whether one or multiple hardware execution units are activated (*i.e.* CPU cores), invalid resource sharing, notably due to simultaneous accesses, proves to be problematic as it can yield to unexpected runtime behaviors with negative implications such as security or safety issues. The growing interest for off-the-shelf multi-core architectures in sensitive applications motivates the need for safe resources sharing. If critical sections are a well-known solution from imperative and non-temporized programming models, they fail to provide safety guarantees. By leveraging the time-triggered programming model, this paper aims at enforcing that identified critical windows of computations can never be simultaneously executed. We achieve this result by determining, before an application is compiled, the exact dates during which a task accesses a shared resource, which enables the off-line validation of non-simultaneity constraints.

2012 ACM Subject Classification Theory of computation → Models of computation

Keywords and phrases Temporal reasoning, Temporal constraints, Specification and verification of systems

Digital Object Identifier 10.4230/LIPIcs.TIME.2020.13

Supplementary Material The implementation of algorithms is available at `https://github.com/krono-safe/mcti-detect/`

Funding This research is supported by the company Krono-Safe.

Acknowledgements We would like to thank Fabien Siron, Matthieu Texier for their interesting and constructive discussions and other engineers from Krono-Safe who helped contributing to this paper. We would also like to thank the anonymous reviewers for their feedback and suggestions.

27th International Symposium on Temporal Representation and Reasoning (TIME 2020).
Editors: Emilio Muñoz-Velasco, Ana Ozaki, and Martin Theobald; Article No. 13; pp. 13:1–13:15
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Resources sharing is a topic of particular interest, notably in safety-critical real-time research, which is challenging for multi-core architectures. These systems are usually bound to stringent timing constraints: failure to perform a computation within a well-specified time interval contributes to the system failure [16]. As failure is not an option, industrials usually rely on a strategy of time provisioning, where predefined slices of time are dedicated to computations, with an additional safety margin. For example, this concept is described as a system of time frames in the ARINC-653P1 specification, which is used in the avionic industry [11]. Determining a strict upper bound of the computation time is known as the Worst-Case Execution Time (WCET) problem: the execution times of a sequence of computations may vary between multiple runs. This variability of execution times is caused by multiple factors, such as the hardware implementation [30, 13], the physical environment in which the hardware operates or the implementation of the software and the interactions software-hardware [32].

To reduce the development and production costs of their systems, as well as the time-to-market, industrials usually rely on commercially available Components Off-The-Shelf (COTS) instead of designing and manufacturing their own hardware [6]. Hardware COTS are produced by a different industry that targets a wider audience. As a result, most architectures are designed in order to minimize average execution times, rather than worst-case execution times. In addition to time-interferences induced by a single core, simultaneous accesses to a same hardware resource (*e.g.* the shared memory or a peripheral) made by multiple cores causes the hardware to arbitrate these concurrent accesses and to serialize them, effectively introducing additional time-interferences [31]. It is estimated that the current WCET analysis techniques would yield the WCET to be multiplied by a value close to the number of cores activated [24, 22, 8]. Such pessimistic estimates lead to over-constrained systems, wasting computing resources, causing higher development and production costs with an unnecessarily increased power consumption.

Contributions. This paper proposes a technique for safe multi-core systems design that is based on an offline temporal partitioning. It allows a system designer to specify windows of computations that shall never be executed simultaneously. Such property would be of great importance for safety-critical avionics systems [1, 29]. After reviewing related work in Section 2, we detail the model of computation our work is based on in Section 3. We then improve this model in Section 4 to express *simultaneity*, and in Section 5 we devise state-of-the-art algorithms to verify that non-simultaneity constraints always hold. An illustrative proof-of-concept is then provided in Section 6 before we conclude in Section 7.

2 Related Work

As summarized in [28], resources sharing can either be limited or avoided by design to ensure the absence of interferences, or controlled during the execution of the system through dedicated services. We advocate for the first proposition, however other interesting research has been conducted in different directions and are worth mentioning.

2.1 Hardware Design

In this paper, we focus only on off-the-shelf processors because they are intensively used by industrials. However, it should be noted that hardware solutions have been devised, notably with PRET machines [13] or the MERASA project [30], with the goal to design specific hardware that are better suited towards time-sensitive applications. For example, Reineke et al. [26] have designed a DRAM controller that aims at eliminating contention for shared resources.

2.2 Runtime Mitigations

Mancuso et al. [20] have proposed the Single Core Equivalence framework, that can be applied on COTS platforms to partition shared resources and, as a result dynamically provide isolation between the different cores. To achieve this goal, the authors rely mainly on three techniques: colored cache lockdown [19], MemGuard [34] and PALLOC [33]. These have been implemented on a Linux kernel and are well suited for dynamic systems by assigning portions of cache to tasks, regulating memory bandwidth and allocating memory pages based on the affinity of DRAM banks with tasks. Bak et al. [5] build on the PREM model of execution [23] by taking advantage of predictable intervals that distinguish memory and execution phases. Memory phases are dedicated to access shared memories, while execution phases shall not (by contract) access these. This allows to dynamically schedule tasks so two memory phases do not execute simultaneously, effectively removing sources of inter-core interferences. If these approaches effectively contribute to improving resources sharing, they do not provide strict design guarantees, because the resolution of resources sharing is determined at runtime.

2.3 Time-Division Multiplexing

Time-Division Multiplexing (TDM) has been extensively studied because of its inherent predictability and improved composability [16, 4]. Because immutable time slices are statically reserved in TDM, this time-division scheme presents the downside to cause underutilization of resources [14]. This is however a useful safety guarantee for safety-critical systems, because it offers greater failure detection capabilities [15].

TDM are enforced at run-time by an *execution model*, which usually consists in a tasks scheduler based on a source of time. Because they are difficult to build by hand, multiple solutions have been devised to generate them. Boniol et al. [9] propose an approach in which they instantiate a scheduling plan in which time slices are dedicated either to access the shared memory or to execute code that does not use shared resources. Their system is generated from a model of the hardware and a static analysis of WCET. Similar works have been conducted by Becker et al. [7].

David et al. [12], Chabrol et al. [10] and Lemerre et al. [18] rely on a model of computation that can be instantiated to express temporal constraints. From instances of this model of computation, data configuring an execution model are produced. This execution model ensures that the specified temporal constraints are enforced at run-time. This model has been formalized as a *time-constrained automata* [17]. It has also been explicitly used by Jan et al. [15] to automatically generate a TDM scheme allowing the control of a real-time network bus from communication specifications that were expressed in the model of computation. Our contribution follows the same path, by improving their model of computation with *non-simultaneity* semantics; effectively enabling to design critical sections driven by the time-triggered paradigm. It differs from critical sections used in imperative and non-temporized programming models [25] in that the dates at which each critical section start and end are precisely known at compile-time, offering additional safety properties, such as the guaranteed absence of deadlocks.

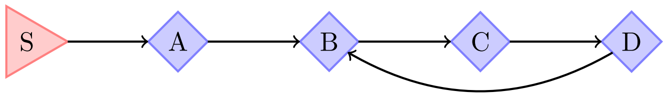


Figure 1 Example of a trivial time-constrained automaton that describes a periodic behavior. After starting at node S, the node A is accepted. Then the sequence of nodes B, C and D is periodically repeated.

3 Time-Constrained Automata

The model proposed in this paper is based on the model of computation formalized by Lemerre et al.: *time-constrained automata* [17]. We extend it later in Section 4.2, but we start by explaining briefly its foundations. This formalism defines a block as a sequence of computations that are time-bounded by at least one of the following constraints:

- *after* that indicates that a block may only start from a given date; and
- *before* that indicates that a block must end before a given date.

They respectively define the *earliest start date* and *deadline* of a batch of blocks, with homogeneous time units. Such automata are formalized as directed graphs, where arcs represent the blocks and nodes represent the temporal constraints that are applied to the arcs joining them (hence constraining the blocks). A node may carry both constraints, but only one constraint for each type. Therefore, three types of time-constrained node exist. They can either be a representation of:

- a single *after* constraint, denoted by ▷, which can be seen in Figure 1 as the node S.
- a single *before* constraint, usually denoted by ◁, but not represented in this paper as it is never used as the sole constraint of a node;
- both a *before* and an *after* constraints, denoted by ◇, which can be seen in Figure 1 as the nodes A, B, C and D. This particular node is named *synchronization*.

▶ **Definition 1** (Trivial time-constrained automaton). *A time-constrained automaton is trivial if and only if every node of the automaton has exactly one output arc. Otherwise, the automaton is said non-trivial.*

There exist several graph simplification techniques that allow to detect impossible graphs or to remove redundant constraints. They are formally defined in the original paper, and we only assume their existence and that graphs can possibly be re-written to a simpler form or proved impossible. In the following of this paper, we assume that all time-constrained automata are valid and reduced to their most simplified form.

An interesting application of time-constrained automata is the ability to derive execution models (*i.e.* scheduling schemes) that preserves the temporal constraints that bound computation blocks. The ability to transform a mathematical model to a concrete result that can be embedded on a hardware target asserted our choice to build on top of this model. The authors of the original paper designed and implemented a variation of the EDF (Earliest Deadline First) algorithm, called *EDF-dyn*, which has been proved optimal for time-constrained sequences of blocks on single processors. However, our approach is not limited to one specific scheduling algorithm, since verification algorithms are applied on the model of computation, and not on the model of execution.

4 System Model

The model of computation we propose is based on time-constrained automata described in Section 3. We insist on the separation of *model of computation* that embodies the design space and the *model of execution* that embodies the run-time of the designed application on a specific execution platform (*e.g.* an embedded COTS system).

4.1 Non-Simultaneity as a Design Constraint

In this paper, we define the *simultaneity* as applied to windows of computations that execute within a known and bounded time span. Simultaneity between two windows of computations describes that their execution may overlap in time.

In Section 2.3, it has been shown that scheduling plans implementing critical sections driven by the time-triggered paradigm can be generated from constraints deduced from characteristics of the system. In approaches that do not rely on a model of computation, there is no guarantee that a feasible schedule exists, because simultaneity is yet another parameter involved in scheduling algorithms. In such cases, it is necessary to tweak multiple parameters of the scheduling algorithm to hope for a viable solution to be found. This process is not guaranteed to converge towards a solution.

Considering a *model of computation* during the design phase that is implemented by a *model of execution* allows to divide the global scheduling problem into independent ones. As the model of computation deals with temporal constraints, simultaneity can be verified regardless of the actual execution times of the tasks. If the application does not respect these new design constraints, then only the original design has to be modified. On the contrary, if such errors were detected later, fixing them would jeopardize the whole application: both its design and implementation.

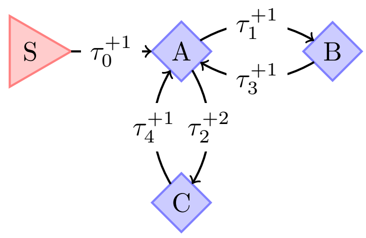
To the best of our knowledge, there exist no methodology in time-triggered resource sharing that allows to model *simultaneity* as an explicit design constraint integrated to a model of computation. We think that addressing this early in the design phase contributes to safer and more robust multi-core applications.

4.2 Augmenting Time-Constrained Automata

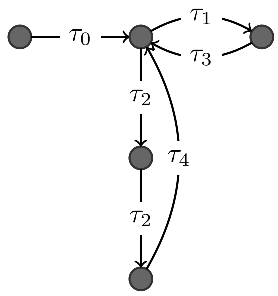
This paper claims to add a new semantic to time-constrained automata, which is detailed in this section.

Temporal transitions. Let a *clock* be a structure that causes the global time to advance; a time-constrained automaton is bound to exactly one clock. We define a *temporal transition* as the ordered set of blocks encompassed within exactly one *after* and one *before* constraints. It is associated with the time span of the computations, which corresponds to the time difference between the deadline (carried by the before constraint) and the earliest start date (carried by the after constraint). This time span, denoted by t may only be strictly positive and is expressed as a finite number of clock ticks. As such, a temporal transition is formally written as the time interval τ^{+t}. The time span can be omitted for brevity; in this case a temporal transition is only denoted by its name (*e.g.* τ).

Isochronous Time-Constrained Automata. Let us consider time-constrained automata where every sequence of blocks is bounded by exactly one *after* and one *before* constraints. They are composed of an *entry* node and a connected graph of *synchronization nodes*, in



(a) Non-trivial time-constrained automata with each temporal transition is bounded by a before and an after node.



(b) Isochronous equivalent of the automaton in Figure 2a.

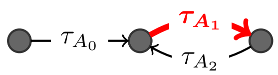
Figure 2 Representations of a non-trivial time-constrained automaton (Figure 2a) and its isochronous equivalent (Figure 2b). From the start node S, only one temporal transition is allowed: τ_0, which is performed in one clock tick. After A is reached, either B or C is reachable, respectively through τ_1 in one tick and τ_2 in two ticks. A is then activated from either B or C in one tick through either τ_3 or τ_4, depending on the previous transition. This behavior is then infinitely repeated.

which each node has at least one output arc. As a result, there exists at least one cycle in this graph. The entry node is an *after* node, which represents the unique entry point of the automaton. It is connected to the graph of synchronization nodes by at least one output arc, and it accepts no input arc. Such automata can be made *isochronous* by splitting each temporal transition into a sequence of successive transitions of unitary length, such that the sum of lengths of the resulting transitions equals the time span of the original transition. In the underlying graphical representations, these additional nodes are denoted by ●. We define such automata as *isochronous time-constrained automata*. Figure 2 illustrates how the non-trivial time-constrained automaton with labeled temporal transitions shown in Figure 2a can be represented as an isochronous time-constrained automaton in Figure 2b.

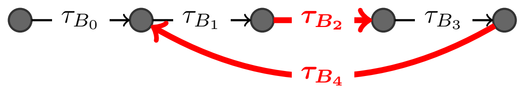
Time-Constrained Applications. A *time-constrained application* is defined as a fixed set of isochronous time-constrained automata that share a same unique base clock. More specifically, at each clock tick a new temporal transition is simultaneously completed by all the automata that compose the application: because they share the same clock, they are *synchronous*. A software implementation of time-constrained applications is required to implement *bound multi-processing*: each task described by an isochronous time-constrained automaton must be statically assigned to one execution unit (*i.e.* a CPU core).

An application is associated with a set of *exclusion groups*, an exclusion group being a fixed set of temporal transitions that shall not overlap in time. These are specified by the designer of the application after a preliminary analysis. The property that temporal transitions of a given exclusion group do not overlap in time is a *safety property* (“bad things do not happen during execution of a program” [2]). For a given exclusion group, this property must be verified on the result of the composition of every automata that has at least one temporal transition belonging to this exclusion group.

Exclusion groups model the *non-simultaneity* within a system. When part of a set, they translate the requirement that the simultaneous execution of their associated windows of computation is *forbidden*.



(a) Automaton A, allocated to core c_A, which describes a periodic behavior: after τ_{A_0} has been taken, the sequence τ_{A_1} and τ_{A_2} is repeated.



(b) Automaton B, allocated to core c_B, which describes a periodic behavior: after τ_{B_0} has been taken, the sequence τ_{B_1}, τ_{B_2}, τ_{B_3} and τ_{B_4} is repeated.

Figure 3 Example of time-constrained application composed of two trivial isochronous time-constrained automata A and B respectively allocated to cores c_A and c_B such that $c_A \neq c_B$. The temporal transitions τ_{A_1}, τ_{B_2} and τ_{B_4} shall not overlap in time.

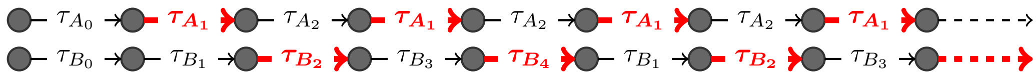


Figure 4 Infinite "unfolding" of automata A (above) and B (below). It hints towards a periodic pattern where temporal transition in the exclusion group $G = \{\tau_{A_1}, \tau_{B_2}, \tau_{B_4}\}$ cannot overlap in time, because of the temporal specfication of A and B.

4.3 Example

Figure 3 shows an example of a simple time-constrained application that consists in two trivial time-constrained automata A and B that are allocated to two different CPU cores. Each automaton defines its own set of temporal transitions: τ_{A_0}, τ_{A_1} and τ_{A_2} for A and τ_{B_0}, τ_{B_1}, τ_{B_2}, τ_{B_3} and τ_{B_4} for B. One exclusion group is arbitrarily defined here: $G = \{\tau_{A_1}, \tau_{B_2}, \tau_{B_4}\}$: these temporal transitions shall not overlap in time.

In this example, the temporal design of automata A and B allows for the exclusion group G to hold: since isochronous time-constrained automata within a time-constrained application are synchronous and since temporal transitions are isochronous, one can observe that when A runs τ_{A_1}, B simultaneously runs either τ_{B_1} or τ_{B_3}, but never τ_{B_2} nor τ_{B_4}. This is illustrated by Figure 4, which shows that "unfolding" A and B hints towards thinking that temporal transitions listed in the exclusion group G cannot overlap in time. In the next section, we show how this problem can be automatically verified.

5 Validating the simultaneity constraints

We have introduced in Section 4 the notions of *time-constrained applications* and of *exclusion groups*, that specify the property that the temporal transitions they contain must not execute simultaneously. In this section, we propose algorithms that verify this property.

5.1 Formalization of the problem

Time-constrained automata may exhibit an infinite possibility of temporal behaviors, because a task embodying the software implementation of an automaton virtually does not have an upper bound of running time. The dates at which a transition can be activated may result from all the infinite possible sequences of these cycles. As an illustration of this complexity, Figure 5 shows all the possible temporal behaviors of the time-constrained automaton shown in Figure 2b between clock ticks zero and seven.

Because a time-constrained application is composed of isochronous time-constrained automata and because they all share the same clock, they are also *synchronous*. As a result, each clock tick causes a temporal transition to be activated in each automata. This implies

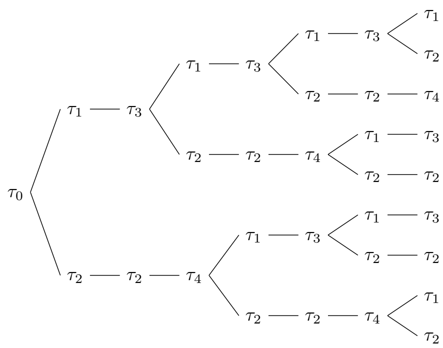


Figure 5 Tree that results from the "unfolding" of temporal behaviors of the non-trivial time-constrained automaton shown in Figure 2b after seven clock ticks. The transition τ_0 is activated at date zero and each arc represents the occurrence of a clock tick.

that a temporal transition can be activated for a possibly infinite set of dates, where a date is represented by a natural number. For example, in Figure 3a, τ_{A_0} can only be activated at date zero, whereas τ_{A_1} can be activated for all dates that are odd. An isochronous time-constrained automaton can therefore be understood as a finite automaton, where:

- each state but the initial one can be marked as accepting;
- the increment of time, associated to all the temporal transitions can be seen as the symbol of a unary alphabet (isochronous property);
- the set of dates at which a state can be reached is given by set of the lengths of the words that lead to this state. Note that this set may be infinite, if the state is included in a cycle.

The set of dates at which a state can be reached can therefore be expressed as the regular language over a unary alphabet accepted by the automaton where only this state is marked as accepting. It is known that each regular unary language can be represented as the union of a finite number of arithmetic progressions of the form $\{c + dk | k \in \mathbb{N}\}$ where c and d are positive constants specifying their offset and period [27]. They can also be written as the pair (c, d).

Temporal transitions that originate from a state are reachable at this set of dates. Therefore, the set of dates at which a temporal transition can be activated is the union of set of dates at which their respective states are reachable.

5.2 Determination of dates of reachability for every transitions

Notations. Let a unary, non-deterministic finite automaton (UNFA) $\mathcal{A}$ with $n \geq 2$ states and m transitions, such that $\mathcal{A} = (Q, \delta, I, F)$ where Q is the finite set of states ($|Q| = n$), $\delta \subseteq Q \times Q$ is a transition relation, $I \subseteq Q$ is the set of initial states of the automaton and $F \subseteq Q$ is the set of *accepting states*. Using the notations defined in [27], $q \xrightarrow{x} q'$ denotes that there exist a path of length x from $q \in Q$ to $q' \in Q$. On a UNFA, a path of length x can be seen as a word x; as such, a word of length x is accepted by $\mathcal{A}$ if there exists a path of length x from $q_i \in I$ to $q_f \in F$, and the language $\mathcal{L}(\mathcal{A})$ accepted by $\mathcal{A}$ is the set of all the words accepted by $\mathcal{A}$.

Expressing $\mathcal{L}(\mathcal{A})$. Sawa proposes in [27] the algorithm *UNFA-Arith-Progressions* that processes a UNFA $\mathcal{A}$ to construct a finite set of arithmetic progressions $\mathcal{R}$ describing the language $\mathcal{L}(\mathcal{A})$, with a space complexity in $O(n+m)$ and a time complexity in $O(n^2(n+m))$.

Applied to isochronous time-constrained automata, the result of this algorithm consists in the exhaustive set of dates at which a given state can be reached. The essence of the algorithm relies on expressing a path α from $q_i \in I$ to $q_f \in F$ *via* $q \in Q$ so that $q_i \xrightarrow{c_1} q \xrightarrow{c_2} q_f$. If q belongs to a cycle of length d, then the length of α can be expressed as the pair $(c_1 + c_2, d)$; otherwise it is simply $(c_1 + c_2, 0)$. As such, $\mathcal{R} = \mathcal{R}_1 \cup \mathcal{R}_2$ with $|\mathcal{R}_1| \leq n^2$ and $|\mathcal{R}_2| \leq n$. $\mathcal{R}_1$ contains every word of length $x < n^2$ written $(x, 0)$ whereas $\mathcal{R}_2$ contains all the other words of $\mathcal{L}(\mathcal{A})$ (with $x \geq n^2$) expressed as arithmetic progressions (at most n).

Tailoring the algorithm. Running the algorithm unmodified for each of the $n-1$ states that can be marked as accepting[1] would yield a total time complexity of $O(n^3(n+m))$. We propose a modified version of this algorithm to specifically determine the set of reachability dates of temporal transitions without degrading the time complexity:

- For $q \in Q$, the value $sl(q)$ is defined as the length of the shortest loop that can be done in q. If q is not part of a loop, then $sl(q)$ is undefined.
- A state q is called *important* if q belongs to a nontrivial strongly connected component C (implying that $sl(q)$ is defined) and the value $sl(q)$ is minimal for all states in C.
- The sets S_i are computed so each set contains all states reachable from the initial state s by i steps: $S_i = \{q \in Q : s \xrightarrow{i} q\}$ for $i \in [0, n^2)$.
- Let Imp the set of important states of $\mathcal{A}$.
- Let $Q_{imp} = S_{n-1} \cap Imp$ the important states that can be reached after exactly $n-1$ steps from the initial state s.
- Let $D = \{sl(q) | q \in Q_{imp}\}$ the set of the shortest loop lengths among the important states in Q_{imp}.
- Since there is only one initial state s to isochronous time-constrained automata, I can be written as $I = \{s\}$.
- We re-define F as the set of states that can be marked as accepting. By definition, $F = Q \backslash \{s\}$.
- We define $\mathbb{T}_q$ the set of temporal transitions that can be activated at state q, that is the outgoing vertices.

From the definition of isochronous time-constrained automata, we can propose a new formulation of the set $\mathcal{R}_1$, such that $\mathcal{R}_1 = \{(i, 0) | i \in [1, n^2)\}$. This allows to build a first set of dates at which states are reachable. In this case, we can re-use this formula to determine an initial set of dates for each temporal transition $\mathcal{D}_{1,\tau}$ as shown in Algorithm 1. Because the original formula excludes the initial state, we add that the transitions reachable from the initial state are all reachable at date zero (by definition). We just associate the temporal transitions activated at a state q with the date at which q is reached. This is possible because each state is associated with a date.

The second set of dates $\mathcal{R}_2$ is built around the sets T_i that contain all states from which some final state can be reached by i steps. They are defined as in Equation (1). Then the pair $(c' + n - 1, d)$ is added to $\mathcal{R}_2$ for $c' \in [n^2 - 2n, n^2 - n - 1]$ and each $d \in D$ such that $c' \geq n^2 - n - d$, if there exists some $q \in Q_{imp}$ with $sl(q) = d$ such that $q \in T_{c'}$.

$$T_i = \{q \in Q | \exists q_f \in F : q \xrightarrow{i} q_f\} \text{ for } i \in [0, n^2 - n - 1] \tag{1}$$

A consequence of this formulation in the original algorithm is that the different temporal transitions leading to $q_f \in F$ are entangled in the construction of the sets T_i in Equation (1).

[1] the initial state cannot be reached from another state

Algorithm 1 Construction of the first set of dates $\mathcal{D}_{1,\tau}$ that contains dates at which each temporal transition τ is activated.

```
for τ ∈ 𝕋_s do
    𝒟_{1,τ} = {(0,0)}
for i ∈ [1, n²) do
    for q ∈ S_i do
        for τ ∈ 𝕋_q do
            𝒟_{1,τ} = 𝒟_{1,τ} ∪ {(i,0)}
```

To preserve dates specific to temporal transitions, we can instead propose the creation of the sets that discriminate temporal transitions, as written in Equation (2).

$$T_{i,q_f} = \{q \in Q : q \xrightarrow{i} q_f\} \text{ for } i \in [0, n^2 - n - 1] \text{ and } q_f \in F \tag{2}$$

Considering each $q_f \in F$, and each $\tau \in \mathbb{T}_{q_f}$, the same algorithm can be re-used to compute $\mathcal{D}_{2,\tau}$ as in Algorithm 2 by substituting T_i with T_{i,q_f}. Finally the set of dates $\mathcal{D}_\tau$ for which each temporal transition τ is activated can be computed as $\mathcal{D}_\tau = \mathcal{D}_{1,\tau} \cup \mathcal{D}_{2,\tau}$.

Instead of running the original algorithm for each of the $n-1$ accepting states, we build the sets T_{i,q_f} once. Furthermore, constructing the sets T_{i,q_f} requires the same operations than the sets T_i as only data organization changes. Thus, we can preserve the overall time complexity of the original algorithm ($O(n^2(n+m))$) since the application of Algorithm 2 does not increase it.

Algorithm 2 Construction of the second set of dates $\mathcal{D}_{2,\tau}$ that contains dates at which each temporal transition τ is activated.

```
for q ∈ Q_imp do
    for c' ∈ [n² − 2n, n² − n − 1] do
        for q_f ∈ F do
            if q ∈ T_{c',q_f} and c' ≥ n² − n − sl(q) then
                for τ ∈ 𝕋_{q_f} do
                    𝒟_{2,τ} = 𝒟_{2,τ} ∪ {(c' + n − 1, sl(q))}
```

Special case of trivial time-constrained automata. The structure of trivial time-constrained automata allows major simplifications of this algorithm. The dates at which a state can be reached can be written as a single arithmetic progression. If we consider the graph representation of the automaton, nodes that are not part of a cycle can be written as $(i, 0)$ where i can be trivially found by exploring the automaton until the node is reached. Nodes that are part of a cycle can be written as (c, d) where c is the first date at which the node is reached and d is the length of the loop. For example, in Figure 3a, dates at which the temporal transition τ_{A_1} is activated are $\{1 + 2k | k \in \mathbb{N}\}$. Similarly, they are $\{2 + 2k | k \in \mathbb{N}\}$ for τ_{A_2} and $\{0\}$ for τ_{A_0}.

5.3 Intersection of dates

We have shown earlier how to compute the set of dates $\mathcal{D}_\tau$ at which each temporal transition τ is activated. This set can be defined as a union of:

- singletons; and
- arithmetic progressions expressed as $\{c + dk | k \in \mathbb{N}\}$.

For a given exclusion group G, verifying that the intersection of the dates that characterize temporal transitions belonging to different automata is empty is equivalent to verify the safety property that temporal transitions within G cannot overlap in time, and as a result cannot be executed simultaneously. We now show the conditions that apply on two dates D_a and D_b for them to intersect. Different techniques may be used depending on whether D_a and D_b represent constants or arithmetic progressions.

Intersection of dates for two constants: let $D_a = h_a$ and $D_b = h_b$ two constant dates. In this trivial case, they share a date in common if and only if $h_a = h_b$.

Intersection of dates for arithmetic progressions: let $D_a = \{c_a + d_a k | k \in \mathbb{N}\}$ and $D_b = \{c_b + d_b k | k \in \mathbb{N}\}$ two arithmetic progressions. Their intersection is not empty if and only if the following linear diophantine equation has a solution: $\alpha x + \beta y = \gamma$ for $x \in \mathbb{Z}$ and $y \in \mathbb{Z}$, with $\alpha = d_a$, $\beta = -d_b$ and $\gamma = c_b - c_a$. Linear diophantine equations are well-known structures that have been extensively studied; the problem of testing the existence of solutions as well as finding them has long been solved [3]. This linear diophantine equation admits a solution in $\mathbb{Z}^2$ if and only if the greatest common denominator of α and β divides γ. If this equation has no solution then the intersection of D_a and D_b is empty. Otherwise, if there exists a solution in $\mathbb{Z}^2$ then D_a and D_b have in common an infinite set of dates since for any solution (x_0, y_0) of the equation, the set of solutions $\{(x_0 + d_b k, y_0 + d_a k) | k \in \mathbb{Z}\}$ can always be built (this set of solutions in $\mathbb{Z}^2$ contains an infinite number of pairs where both members are natural integers).

Intersection of dates for a constant and an arithmetic progression: if $D_a = \{h_a\}$ is a singleton and $D_b = \{c_b + d_b k | k \in \mathbb{N}\}$ is an infinite set representing an arithmetic progression, they may intersect at most once, if $D_a \subset D_b$, that is when they exist $x \in \mathbb{N}$ such that $h_a = c_b + d_b x$. We find that this is true when $h_a \geq c_b$ and d_b divides $h_a - c_b$.

6 Proof of Concept

Implementation and Reproducibility. The model defined in Section 4 has been integrated to the ASTERIOS suite, developed by the Krono-Safe company. It relies on a programming model detailed by Methni et al. in [21] to instantiate a model of computation, in which support for simultaneity has been added. The algorithms presented in Section 5 and the method to validate the intersection of dates have been implemented in a standalone executable that has been open-sourced[2] under the Apache-2.0 license. It takes as inputs a specification of the different tasks that compose an application with the list of exclusion groups to be checked, and generates a report containing the dates at which each temporal transition can be activated, as well as a graphical representation of the time-constrained application and either the validation of exclusion groups or a counter-example. The proof-of-concept in this section is based on the open-source version.

[2] `https://github.com/krono-safe/mcti-detect/`

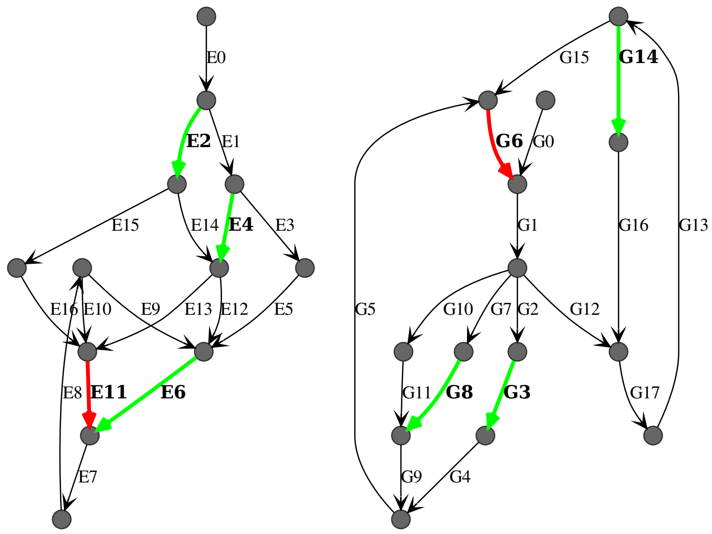


Figure 6 Design in which E_2, E_4, E_6, E_{11}, G_3, G_6, G_8 and G_{14} are used to access the shared resource. However it is found that E_{11} and G_6 can be simultaneously reached at the same date. This is therefore an example of a design that does not guarantee safe resources sharing.

Table 1 Counter-example showing traces leading for E_{11} and G_6 to overlap in time at tick 12.

Tick	0	1	2	3	4	5	6	7	8	9	10	11	**12**
Task E	E_0	E_1	E_4	E_{12}	E_6	E_7	E_8	E_{10}	E_{11}	E_7	E_8	E_{10}	$\boldsymbol{E_{11}}$
Task G	G_0	G_1	G_{10}	G_{11}	G_9	G_5	G_6	G_1	G_{12}	G_{17}	G_{13}	G_{15}	$\boldsymbol{G_6}$

Sharing resources between two parallel tasks. For this illustrative proof-of-concept, let's consider a simple application that uses two non-trivial tasks E and G, each implanted on a different CPU core. The requirements of this application impose they exchange data through a shared resource (*e.g.* shared memory). In this specific use case, we assume the temporal constraints are fixed: nodes cannot be added nor removed. When considering an incremental design, this may not be the case. The end goal is to guarantee that accesses to the shared resources are performed during temporal transitions that never overlap in time. The occurrence of unwanted simultaneous accesses may result in data corruption (*e.g.* the two tasks write at the same memory address) or in increased execution times caused by additional contention.

Exposing an invalid design. A first design can be seen in Figure 6, which represents a time-constrained application composed of two non-trivial tasks where accesses to the shared resource are performed during the temporal transitions E_2, E_4, E_6, E_{11}, G_3, G_6, G_8 and G_{14}. However, we find that temporal transitions E_{11} and G_6 may overlap in time, as shown in Table 1. This small example showcases that checking for absence of simultaneity is not a trivial process and highlights the importance of automated validation.

Towards a safe design. If the design in Figure 6 does not guarantee safe resources sharing, it is possible to try other design candidates. If the functional requirements of the application allow it, the shared resource could be accessed from G_5 and G_{15} and the retrieved data made available to G_6. This modified design is checked as in Figure 7 and the new set of temporal

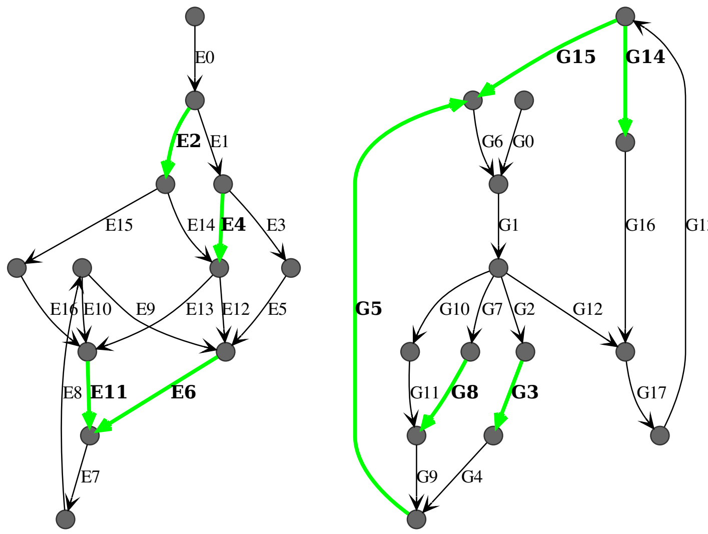


Figure 7 Design in which E_2, E_4, E_6, E_{11}, G_3, G_5, G_8, G_{14} and G_{15} are used to access the shared resource. It is found that they never overlap in time. This is therefore an example of a design that guarantees safe resources sharing, given that accesses to the shared resource happen only during these temporal transitions.

transitions that access the shared resource (E_2, E_4, E_6, E_{11}, G_3, G_5, G_8, G_{14} and G_{15}) have been found to never overlap in time. Implementing such design removes entire classes of problems that could comprise data integrity or negatively impact execution times, while allowing for a better use of overall computing resources.

7 Conclusion and Perspectives

We have presented a model of computation based on time-constrained automata, that can be used to express *non-simultaneity* as a design constraint in a model of computation. This allows to express a safety property over parallel systems, which, if verified, ensures that litigious sequences of computations can never run simultaneously. Designing such systems with non-simultaneity as a constraint from the ground-up is believed to bring significant safety benefits, notably for safety-critical real-time systems. We have then shown that this safety property could be automatically verified, with reasonable complexity, by standalone and open-sourced algorithms that extend the state of the art. As for future work, it would be interesting to propose more advanced techniques to help the designer to interactively explore the traces leading to a violation of its design constraints, for a more efficient convergence towards a safe design. It seems also important to explore techniques to determine the sources of time-interferences when they occur.

References

1 Irune Agirre, Jaume Abella, Mikel Azkarate-Askasua, and Francisco J Cazorla. On the tailoring of cast-32a certification guidance to real cots multicore architectures. In *2017 12th IEEE International Symposium on Industrial Embedded Systems (SIES)*, pages 1–8. IEEE, 2017. doi:10.1109/SIES.2017.7993376.

2 Bowen Alpern and Fred B Schneider. Recognizing safety and liveness. *Distributed computing*, 2(3):117–126, 1987. doi:10.1007/BF01782772.

3 George E Andrews. *Number theory.* Courier Corporation, 1994.

4 Christophe Aussagues, Damien Chabrol, and Vincent David. Method for the deterministic execution and synchronisation of an information processing system comprising a plurality of processing cores executing system tasks, April 2010. Patent WO 2010/043706 A2.

5 Stanley Bak, Gang Yao, Rodolfo Pellizzoni, and Marco Caccamo. Memory-aware scheduling of multicore task sets for real-time systems. In *2012 IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 300–309. IEEE, 2012. doi:10.1109/RTCSA.2012.48.

6 Thomas G Baker. Lessons learned integrating cots into systems. In *International Conference on COTS-Based Software Systems*, pages 21–30. Springer, 2002. doi:10.1007/3-540-45588-4_3.

7 Matthias Becker, Dakshina Dasari, Borislav Nicolic, Benny Akesson, Vincent Nélis, and Thomas Nolte. Contention-free execution of automotive applications on a clustered many-core platform. In *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 14–24. IEEE, 2016. doi:10.1109/ECRTS.2016.14.

8 Jingyi Bin, Sylvain Girbal, Daniel Gracia Pérez, Arnaud Grasset, and Alain Mérigot. Studying co-running avionic real-time applications on multi-core COTS architectures. In *Embedded Real Time Software and Systems (ERTS2014)*, Toulouse, France, February 2014.

9 Frédéric Boniol, Hugues Cassé, Eric Noulard, and Claire Pagetti. Deterministic execution model on cots hardware. In *International Conference on Architecture of Computing Systems*, pages 98–110. Springer, 2012. doi:10.1007/978-3-642-28293-5_9.

10 Damien Chabrol, Vincent David, Patrice Oudin, Gilles Zeppa, and Mathieu Jan. Freedom from interference among time-triggered and angle-triggered tasks: a powertrain case study. In *Embedded Real Time Software and Systems (ERTS2014)*, Toulouse, France, February 2014.

11 Airlines Electronic Committee. Avionics application software standard interface - part 1: Required services. Arinc 653p1, Airlines Electronic Committee, August 2015.

12 Vincent David, Christophe Aussaguès, Stéphane Louise, Philippe Hilsenkopf, Bertrand Ortolo, and Christophe Hessler. The oasis based qualified display system. In *Fourth American Nuclear Society International Topical Meeting on Nuclear Plant Instrumentation, Controls and Human-Machine Interface Technologies (NPIC&HMIT 2004), Columbus, Ohio, USA*, page 11, 2004.

13 Stephen A Edwards and Edward A Lee. The case for the precision timed (pret) machine. In *Proceedings of the 44th annual Design Automation Conference*, pages 264–265. ACM, 2007. doi:10.1145/1278480.1278545.

14 Farouk Hebbache, Mathieu Jan, Florian Brandner, and Laurent Pautet. Shedding the shackles of time-division multiplexing. In *2018 IEEE Real-Time Systems Symposium (RTSS)*, pages 456–468. IEEE, 2018. doi:10.1109/RTSS.2018.00059.

15 Mathieu Jan, Jean-Sylvain Camier, and Vincent David. Scheduling safety-critical real-time bus accesses using time-constrained automata. In *RTNS*, pages 87–96. Citeseer, 2011.

16 Hermann Kopetz. *Real-time systems: design principles for distributed embedded applications*. Real-Time Systems Series. Springer, 2011. doi:10.1007/978-1-4419-8237-7.

17 Matthieu Lemerre, Vincent David, Christophe Aussagues, and Guy Vidal-Naquet. An introduction to time-constrained automata. In *Proceedings of the 3rd Interaction and Concurrency Experience Workshop (ICE'10)*, volume 38, pages 83–98, June 2010. doi:10.4204/EPTCS.38.9.

18 Matthieu Lemerre and Emmanuel Ohayon. A model of parallel deterministic real-time computation. In *2012 IEEE 33rd Real-Time Systems Symposium*, pages 273–282. IEEE, 2012. doi:10.1109/RTSS.2012.78.

19 Renato Mancuso, Roman Dudko, Emiliano Betti, Marco Cesati, Marco Caccamo, and Rodolfo Pellizzoni. Real-time cache management framework for multi-core architectures. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 45–54. IEEE, 2013. doi:10.1109/RTAS.2013.6531078.

20 Renato Mancuso, Rodolfo Pellizzoni, Marco Caccamo, Lui Sha, and Heechul Yun. Wcet (m) estimation in multi-core systems using single core equivalence. In *2015 27th Euromicro Conference on Real-Time Systems*, pages 174–183. IEEE, 2015. doi:10.1109/ECRTS.2015.23.

21 Amira Methni, Emmanuel Ohayon, and François Thurieau. ASTERIOS Checker : A Verification Tool for Certifying Airborne Software. In *10th European Congress on Embedded Real Time Systems (ERTS 2020)*, Toulouse, France, January 2020. URL: https://hal.archives-ouvertes.fr/hal-02508852.

22 Jan Nowotsch and Michael Paulitsch. Leveraging multi-core computing architectures in avionics. In *2012 Ninth European Dependable Computing Conference*, pages 132–143. IEEE, 2012. doi:10.1109/EDCC.2012.27.

23 Rodolfo Pellizzoni, Emiliano Betti, Stanley Bak, Gang Yao, John Criswell, Marco Caccamo, and Russell Kegley. A predictable execution model for cots-based embedded systems. In *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 269–279. IEEE, 2011. doi:10.1109/RTAS.2011.33.

24 Rodolfo Pellizzoni, Andreas Schranzhofer, Jian-Jia Chen, Marco Caccamo, and Lothar Thiele. Worst case delay analysis for memory interference in multicore systems. In *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*, pages 741–746. IEEE, 2010. doi:10.1109/DATE.2010.5456952.

25 Michel Raynal. *Concurrent programming: algorithms, principles, and foundations*. Springer Science, 2013. doi:10.1007/978-3-642-32027-9.

26 Jan Reineke, Isaac Liu, Hiren D Patel, Sungjun Kim, and Edward A Lee. Pret dram controller: Bank privatization for predictability and temporal isolation. In *2011 Proceedings of the Ninth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS)*, pages 99–108. IEEE, 2011. doi:10.1145/2039370.2039388.

27 Zdeněk Sawa. Efficient construction of semilinear representations of languages accepted by unary nondeterministic finite automata. *Fundamenta Informaticae*, 123(1):97–106, 2013. doi:10.3233/FI-2013-802.

28 Nathanaël Sensfelder, Julien Brunel, and Claire Pagetti. Modeling cache coherence to expose interference. In *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019. doi:10.4230/LIPIcs.ECRTS.2019.18.

29 Certification Authorities Software Team. Multi-core processors - position paper. Cast-32a, Certification Authorities Software Team, November 2016.

30 Theo Ungerer, Francisco Cazorla, Pascal Sainrat, Guillem Bernat, Zlatko Petrov, Christine Rochange, Eduardo Quinones, Mike Gerdes, Marco Paolieri, Julian Wolf, et al. Merasa: Multicore execution of hard real-time applications supporting analyzability. *IEEE Micro*, 30(5):66–75, 2010. doi:10.1109/MM.2010.78.

31 Stephen C Vestal, Pamela Binns, Aaron Larson, Murali Rangarajan, and Ryan Roffelsen. Safe partition scheduling on multi-core processors, 2012. US Patent 8,316,368.

32 Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, et al. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):36, 2008. doi:10.1145/1347375.1347389.

33 Heechul Yun, Renato Mancuso, Zheng-Pei Wu, and Rodolfo Pellizzoni. Palloc: Dram bank-aware memory allocator for performance isolation on multicore platforms. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 155–166. IEEE, 2014. doi:10.1109/RTAS.2014.6925999.

34 Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 55–64. IEEE, 2013. doi:10.1109/RTAS.2013.6531079.